Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομεας Τεχνολογιας Πληροφορικης και
Υπολογιστων

Αναλυτική Επεξεργασία και Βελτιστοποίηση Ερωτημάτων σε Ροές Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΓΕΩΡΓΙΟΥ ΡΑΦΑΗΛ Ι. ΘΕΟΔΩΡΑΚΗ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2017

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογιας Πληροφορικης και
Υπολογιστων

Αναλυτική Επεξεργασία και Βελτιστοποίηση Ερωτημάτων σε Ροές Δεδομένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΓΕΩΡΓΙΟΥ ΡΑΦΑΗΛ Ι. ΘΕΟΔΩΡΑΚΗ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 12η Ιουλίου 2017.

.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Δημήτριος Τσουμάκος Επ. Καθηγητής Ι.Π.	Γεώργιος Γκούμας Επ. Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2017

.....

ΓΕΩΡΓΙΟΣ ΡΑΦΑΗΛ Ι. ΘΕΟΔΩΡΑΚΗΣ

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

Περίληψη

Στην εποχή μας, εφαρμογές δεδομένων μεγάλων σε όγκο και πολυπλοκότητα παράγουν δεδομένα σε κολοσσιαίες τιμές. Για να είναι εφικτή η επεξεργασία τέτοιων δεδομένων ροής και να παραχθούν γρήγορα αποτελέσματα, χρησιμοποιείται η τεχνολογία της επεξεργασίας δεδομένων ροής. Ωστόσο, οι σύγχρονες πλατφόρμες επεξεργασίας ροών δεδομένων, όπως το Apache Storm, το Spark Streaming και το Apache Samza, είτε δεν διαθέτουν είτε έχουν περιορισμένη υποστήριξη SQL για τη δήλωση ερωτημάτων συνεχούς ροής και απαιτούν καλή γνώση προστακτικού προγραμματισμού και κατανεμημένων συστημάτων για να χρησιμοποιηθούν αποτελεσματικά. Επιπλέον, τέτοια συστήματα υποστηρίζουν τυπικούς κανόνες βελτιστοποίησης που βασίζονται σε ευριστικούς κανόνες (Spark Catalyst) και τεχνικές βελτιστοποίησης που στοχεύουν σε μοντέλα κόστους βασισμένα σε Συστήματα Διαχείρισης Σχεσιακών Βάσεων Δεδομένων, χωρίς να λαμβάνονται υπόψη οι διαφορετικές απαιτήσεις που έχουν τα συστήματα ροής. Ενώ υπάρχουν πολλά διαφορετικά μοντέλα και αλγόριθμοι που έχουν προταθεί για τη βελτιστοποίηση της εκτέλεσης σύνθετων ερωτημάτων συνεχούς ροής, δεν χρησιμοποιούνται από τα σύγχρονα συστήματα ροής δεδομένων, καθώς απαιτείται μεγάλη σχεδιαστική και προγραμματική προσπάθεια για την υλοποίηση τους. Το σύστημά μας, το RBStream, είναι χτισμένο πάνω στο Apache Calcite, ένα σύγχρονο πλαίσιο ανοιχτού κώδικα για την ανάλυση, την επικύρωση και τη βελτιστοποίηση των ερωτημάτων και εισάγει ένα λογικό εργαλείο βελτιστοποίησης βάσει κόστους στην υβριδική μηχανή επεξεργασίας ροής SABER. Παρουσιάζουμε τεχνικές βελτιστοποίησης βασισμένες στο ρυθμό εισροής δεδομένων, χρησιμοποιώντας ένα συνδυασμό του Volcano μαζί με τη υλοποίηση ενός Heuristic βελτιστοποιητή στο Calcite, παρόμοιο με το Spark Catalyst. Η διαδικασία βελτιστοποίησης χωρίζεται σε φάσεις χρησιμοποιώντας ενσωματωμένους και προσαρμοσμένους κανόνες, που εφαρμόζουν ισοδύναμους αλγεβρικούς μετασχηματισμούς για να παραχθεί ένα πλάνο βελτιστοποιημένο σε επίπεδο throughput, latency και χρήσης της CPU. Εφαρμόσαμε τα ευρήματά μας στο SABER και αξιολογήσαμε πειραματικά το RBStream με συνθετικά δεδομένα σε διαφορετικά configuration.

Λέξεις Κλειδιά: Calcite, SABER, RBStream, επεξεργασία δεδομένων ροής, στατική βελτιστοποίηση ερωτημάτων, βελτιστοποίηση με βάση το ρυθμό εισροής δεδομένων, συρόμενα παράθυρα, σταθερά παράθυρα, Volcano, Cost Based Logical Optimizer.

Abstract

Nowadays, Big Data applications produce data at colossal rates and enterprises have to extract valuable information in acceptable time from these massive data volumes, as to react quickly when problems arise or to detect new trends. In order to process this rapidly flowing data and produce fast results, the technology of stream processing is being used. However, modern data stream processing platforms like Apache Storm, Spark Streaming and Apache Samza either lack or have limited support for SQL like declarative query languages and require sound knowledge on imperative style programming and distributed systems to effectively utilize them. In addition, such systems support typical optimization rules based on heuristics (Spark Catalyst) and optimization techniques targeted towards Relational Database Management Systems' cost models, without taking into account the different requirements that streaming systems have, such as, for instance, stream rates or the size of the windows used. Although there are many different models and algorithms that have been proposed to optimize the execution of complex streaming queries, they are not utilized by modern streaming systems, as it requires a lot of design and programmatic effort to remodel them. Our system, RBStream, is built on top of Apache Calcite, a state of the art open-source framework for query parsing, validation, and optimization, and introduces a Cost Based Logical Optimizer to SABER, a hybrid relational stream processing engine. With our integration, we present rate-based optimization techniques by utilizing a combination of Volcano planner along with a Heuristic optimizer implementation of Calcite, similar to Spark Catalyst. The optimization procedure is divided into separate phases using built-in and custom transformer rules, that create equivalent algebraic transformations and result in considerable improvements as far as throughput, latency and Cpu utilization are concerned. We applied our findings to SABER and experimentally evaluated RBStream with synthetic data on different configurations.

Keywords: Calcite, SABER, RBStream, stream processing, static query optimization, rate-based optimization, sliding windows, tumbling windows, Volcano, Cost Based Logical Optimizer

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον καθηγητή κ. Νεκτάριο Κοζύρη για τη δυνατότητα που μου έδωσε να ασχοληθώ με το σύγχρονο και ενδιαφέρον αντικείμενο της παρούσης διπλωματικής. Επίσης, θα ήθελα να ευχαριστήσω τους μεταδιδακτορικούς ερευνητές Ιωάννη Κωνσταντίνου και Αλέξανδρο Κολιούση για τη συνεχή παρακολούθηση, την υποστήριξη και το χρόνο που αφιέρωσαν για την εκπόνηση της εργασίας αυτής.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς και τους ανθρώπους που ήταν κοντά μου, για την απεριόριστη στήριξη που μου παρείχαν σε καλές και δύσκολες στιγμές κατά την διάρκεια των σπουδών μου.

Γεώργιος Ραφαήλ Ι. Θεοδωράκης,

Αθήνα, 12η Ιουλίου 2017

Contents

List of Figures & Tables

0

Εισαγωγή

0.1 Συστήματα Διαχείρισης Ροής δεδομένων

Τις τελευταίες δεκαετίες, τα συστήματα διαχείρισης σχεσιακών βάσεων δεδομένων μπορούσαν να αντεπεξέλθουν στις ανάγκες των παραδοσιακών επιχειρηματικών εφαρμογών. Τα δεδομένα είχαν φορτωθεί εκ των προτέρων στη βάση όταν οι χρήστες υπέβαλαν σύνθετα ερωτήματα, τα οποία στη συνέχεια υπολογίζονταν αποτελεσματικά από τη σχεσιακή βάση δεδομένων. Αυτά τα αποθηκευμένα αρχεία δεδομένων μπορούν να θεωρηθούν σχετικά στατικά και εξακολουθούν να είναι έγκυρα μέχρις ότου τροποποιηθούν ή αφαιρεθούν. Επομένως, υποθέτουμε ότι ο αριθμός των ερωτημάτων που υποβάλλονται στο σύστημα υπερβαίνει τον αριθμό των τροποποιήσεων των δεδομένων. Ο στόχος μιας σχεσιακής βάσης δεδομένων είναι να παρέχει ανθεκτικότητα, συνέπεια, και διαθεσιμότητα, μαζί με την αποτελεσματική εκτέλεση του ερωτήματος.

Ωστόσο, σήμερα έχουμε την εμφάνιση ενός νέου τύπου εφαρμογών με μεγάλη ένταση δεδομένων που απαιτεί την επεξεργασία δεδομένων που συνεχώς φτάνουν στο σύστημα από πηγές ροής. Οι web εφαρμογές, τα δίκτυα αισθητήρων, οι υπηρεσίες που βασίζονται στην τοποθεσία, η παρακολούθηση της κυκλοφορίας του δικτύου, οι ηλεκτρονικές συναλλαγές και η καταγραφή συναλλαγών είναι μερικά παραδείγματα που ανήκουν σε αυτή την νέα κατηγορία εφαρμογών και παράγουν δεδομένα σε πολύ ταχείς ρυθμούς. Η αυξανόμενη χρήση των ασύρματων συσκευών πληροφορικής, ο αυξανόμενος αριθμός των web εφαρμογών και οι εξελίξεις στους αισθητήρες συμβάλλουν στην ανάπτυξη τέτοιων εφαρμογών ροής δεδομένων. Παρόλο που η διεκπεραιωτική ικανότητα αυξάνεται, οι επιχειρήσεις και οι χρήστες επιθυμούν όλο και πιο έγκαιρα αποτελέσματα. Μπορεί οι τελευταίες εξελίξεις στο hardware να βοηθούν προς αυτήν την κατεύθυνση, αλλά αυτό δεν είναι αρκετό για να συμβαδίσουμε με τα προβλήματα απόδοσης και την λανθάνουσα περίοδο. Είναι εξαιρετικά ακριβό να εξαγάγουμε πολύτιμες πληροφορίες σε αποδεκτό χρόνο από πολύ μεγάλους όγκους δεδομένων που έχουν συσσωρευτεί πάνω από μήνες ή ακόμη και χρόνια. Ως εκ τούτου, κάποια δεδομένα πρέπει να απορριφθούν, χωρίς να αναλυθούν, για να απελευθερωθεί χώρος για νέα δεδομένα που έρχονται.

Για να επεξεργαστούμε αυτές τις γρήγορες ροές δεδομένων και να παραχθούν άμεσα αποτελέσματα, χρησιμοποιείται η τεχνολογία της επεξεργασίας ροών δεδομένων. Οι βασικές έννοιες πίσω από αυτή την τεχνολογία προέρχονται από την ερευνητική κοινότητα των βάσεων δεδομένων. Αν και σχεσιακές βάσεις δεδομένων και μηχανές συνεχούς ροής μοιράζονται πολλά κοινά χαρακτηριστικά, δεν είναι ίδιες. Σε μια βάση δεδομένων, οι χρήστες κάνουν ερωτήματα σχετικά με δεδομένα που έχουν φτάσει και αποθηκευτεί εκ των προτέρων. Οι συμβατικές σχεσιακές βάσεις δεδομένων δεν έχουν σχεδιαστεί για τη συνεχή αξιολόγηση των ερωτημάτων σε δεδομένα συνεχούς ροής. Αντιθέτως, σε μια μηχανή ερωτημάτων συνεχούς ροής, τα ερωτήματα παρουσιάζονται πριν από τα δεδομένα.

Τα Συστήματα Διαχείρισης Ροής δεδομένων παρέχουν την απαιτούμενη υποστήριξη για ερωτήματα που πρέπει να υπολογιστούν σε δεδομένα που έρχονται συνεχώς ως δυνητικά απεριόριστη, ταχεία και χρονικά μεταβαλλόμενη ροή δεδομένων. Οι ροές δεδομένων περνάνε μέσα από μια σειρά συνεχώς εκτελούμενων ερωτημάτων και τελεστών, από τα οποία παίρνουμε ως αποτέλεσμα τους μετασχηματισμούς τους. Θα μπορούσε κανείς να πει ότι μια σχεσιακή βάση δεδομένων επεξεργάζεται τα δεδομένα σε κατάσταση ηρεμίας με μεταβλητά ερωτήματα, ενώ ένα Σύστημα Διαχείρισης Ροής δεδομένων επεξεργάζεται δεδομένα που έρχονται συνεχόμενα [17] με σταθερά ερωτήματα.

0.2 Κίνητρο

Σύγχρονες πλατφόρμες επεξεργασίας ροών δεδομένων όπως το Apache Storm [40], το Spark Streaming [41] και το Apache Samza [43] δεν έχουν υποστήριξη για SQL ερωτήματα και απαιτούν καλή γνώση προστακτικού προγραμματισμού και κατανεμημένων συστημάτων για να χρησιμοποιηθούν αποτελεσματικά [4]. Τα συστήματα αυτά παρέχουν API χαμηλού επιπέδου και απαιτούνται σημαντικές προσπάθειες για την παραγωγή αξιοπρεπούς κώδικα, λόγω της πολυπλοκότητας τους και της επιβάρυνσης λόγω συντήρησης [20]. Επιπλέον, αρκετές γνωστές λύσεις που βασίζονται σε Hadoop χρησιμοποιούν υποστήριξη SQL, όπως το Hive [56], Impala [74] ή Presto [75]. Έτσι, η ερώτηση που τίθεται είναι αν μπορούμε να επεκτείνουμε αυτές τις μηχανές ροής μεγάλων δεδομένων για να υποστηρίξουμε SQL με σαφή σημασιολογία ή όχι.

Γνωρίζουμε ήδη από τις παραδοσιακές βάσεις δεδομένων, ότι η SQL σαν γλώσσα είναι ένας μαθηματικά ορθός τρόπος έκφρασης μιας ερώτησης σε μια βάση δεδομένων.

Προκειμένου να καταστούν τα συστήματα συνεχούς ροής πιο εύκολα προσβάσιμα για τους χρήστες που δεν είναι εξοικειωμένοι με τον προγραμματισμό και τα κατανεμημένα συστήματα, θα πρέπει να εισαχθεί η υποστήριξη SQL συνεχούς γλώσσας ερωτήσεων ή SQL με επεκτάσεις συνεχούς ροής. Οι λόγοι για τη χρήση της SQL σαν γλώσσα ερωτημάτων είναι [52, 76, 77, 80]:

- Η SQL είναι δηλωτική. Επομένως, οι χρήστες καθορίζουν ποιες πληροφορίες θέλουν να εξάγουν, αλλά όχι πώς θα υπολογιστούν.
- Επιτρέπει τις αναλύσεις σε πραγματικό χρόνο να εκφραστούν χρησιμοποιώντας απλά ερωτήματα, λόγω της εκφραστικότητά της. Οι περισσότεροι προγραμματιστές κατανοούν την SQL και είναι ευκολότερο για αυτούς να τη μάθουν. Μόνο στην περίπτωση που δεν καλύπτει μια συγκεκριμένη περίπτωση χρήσης, πρέπει να ασχοληθούν με πιο σύνθετες λύσεις, όπως ο προστακτικός προγραμματισμός. Η SQL είναι ικανή να εμπεριέχει την υποκείμενη πολυπλοκότητα των εργασιών διαχείρισης δεδομένων, και έτσι να απλοποιεί τις αναλύσεις σε πραγματικό χρόνο. Επίσης, οι αναλύσεις σε πραγματικό χρόνο μπορούν να βελτιστοποιηθούν καλύτερα με την SQL. Με τη χρήση της υποστήριξης SQL στο σύστημά μας, δίνουμε τη δυνατότητα βελτιστοποίησης με μια δομημένη διαδικασία, η οποία προέρχεται από τις παραδοσιακές σχεσιακές βάσεις δεδομένων. Οι τεχνικές βελτιστοποίησης και οι αλγόριθμοι που χρησιμοποιούνται στις σχεσιακές βάσεις δεδομένων μπορούν να προσαρμοστούν και να επεκταθούν αναλόγως, προκειμένου να επιλύσουν τα ζητήματα που προκύπτουν από τις εφαρμογές ροής δεδομένων.
- *"Η SQL μπορεί να εφαρμοστεί για το δυναμικό σχεδιασμό και βελτιστοποίηση ερωτήσεων, αφού είναι μαθηματικά σχεδιασμένη για το σκοπό αυτό."* [52]
- Τα ερωτήματα σε SQL μπορούν να βελτιστοποιούνται αυτόματα είτε σε κεντρικά είτε σε κατανεμημένα συστήματα. Ως εκ τούτου, μπορούμε να επιτύχουμε σημαντικά καλύτερα αποτελέσματα με λιγότερη προσπάθεια, εξαλείφοντας την ανάγκη για χειροκίνητη και συχνά κακής ποιότητας βελτιστοποίηση των ερωτημάτων. *"Η SQL προσφέρει ισχυρά και κομψά στοιχεία που έχουν γενική χρήση, μπορεί να βελτιστοποιηθεί με δυναμικό τρόπο για βέλτιστη απόδοση με δεδομένη την τρέχουσα διαμόρφωση υλικού, και μπορεί να συνδυαστεί με άλλες λειτουργίες SQL"* [52]. Οι εφαρμογές SQL Streaming αποδίδουν καλύτερα από τις αντίστοιχες εφαρμογές Storm ή Spark, καθώς η ανάπτυξη μιας εφαρμογής ροής δεδομένων σε αυτές τις πλατφόρμες απαιτεί γραφή κώδικα σε Java ή Scala, οι οποίες είναι λιγότερο συμπαγείς από την SQL. Αυτές οι γλώσσες προγραμματισμού

"απαιτούν σημαντική συλλογή σκουπιδιών, η οποία είναι ιδιαίτερα ανεπαρκής και ενοχλητική για την επεξεργασία εντός της μνήμης" [79].

- Μπορούμε να αποσυνδέσουμε την υλοποίηση εφαρμογών SQL από το υπόλοιπο σύστημα, δεδομένου ότι μπορούν να κατασκευαστούν σε λιγότερο χρόνο σε σχέση με τις χαμηλού επιπέδου πλατφόρμες ανοικτού κώδικα και τις SQL πλατφόρμες.
- Σε αντίθεση με τα Συστήματα Επεξεργασίας Σύνθετων Γεγονότων (CEP [55]) και πολλές από τις σημερινές πλατφόρμες ανοικτού κώδικα, οι πλατφόρμες SQL μπορούν να ενημερωθούν κατά την λειτουργία τους χωρίς να χρειάζεται να μεταγλωττιστεί ο κώδικας, χαρακτηριστικό το οποίο είναι απαραίτητο σε πολλές εφαρμογές. Για παράδειγμα, όταν χρησιμοποιούμε stored procedures οι αλλαγές γίνονται άμεσα εμφανείς χωρίς μεταγλώττιση του κώδικα.
- Επίσης, μπορούμε να ορίσουμε λειτουργίες από το χρήστη γραμμένες σε μια γλώσσα υψηλού επιπέδου και να αναπτυχθούν σε SQL, οι οποίες θα λύνουν εν μέρει το πρόβλημα που δεν υποστηρίζει όλες τις περιπτώσεις χρήσης πραγματικών εφαρμογών.
- Τέλος, μπορούμε να χρησιμοποιήσουμε πλατφόρμες για ανάλυση ροών δεδομένων και οπτικοποίησης τους με γραφικό περιβάλλον, αφού κάτι τέτοιο είναι εύκολο να δημιουργηθεί αυτόματα στην SQL.

Ωστόσο, η παραδοσιακή SQL, τα σχεσιακά μοντέλα δεδομένων και η άλγεβρα [73] δεν σχεδιάστηκαν για την επεξεργασία συνεχούς ροής. Τα συστήματα σχεσιακής βάσης δεδομένων ασχολούνται με πεπερασμένες ακολουθίες πλειάδων, οι οποίες είναι άμεσα διαθέσιμες και παράγουν αποτελέσματα σταθερού μεγέθους. Ορισμένα από αυτά τα συστήματα χαρακτηρίζονται από έντονη διατήρηση των υλοποιημένων προβολών (materialized views), που ορίζονται σε ένα ερώτημα SQL σαν μια κανονική προβολή. Αυτός ο τύπος επεξεργασίας είναι παρόμοιος με την αξιολόγηση ερωτημάτων SQL σε ροές δεδομένων, με τη διαφορά ότι στα συστήματα ροής πρέπει να ενημερώσουμε την προβολή όταν τροποποιηθούν οι βασικές σχέσεις.

Επομένως, είναι προφανές ότι υπάρχει ανάγκη για τις μηχανές επεξεργασίας συνεχούς ροής να διαθέτουν μια δηλωτική γλώσσα SQL και ως εκ τούτου υποστήριξη βελτιστοποίησης ερωτημάτων [80]. Η βελτιστοποίηση των ερωτημάτων αποτελεί βασικό στοιχείο της τεχνολογίας των βάσεων δεδομένων επειδή λειτουργεί ως γέφυρα μεταξύ των ερωτημάτων SQL και της αποτελεσματικής εκτέλεσης τους. Επιπλέον, στις περισσότερες περιπτώσεις, οι βελτιστοποιητές καθιστούν τα δηλωτικά ερωτήματα SQL μια πολύ καλύτερη επιλογή

από τον επιτακτικό προγραμματισμό. Ωστόσο, *"αν και οι περισσότερες πλατφόρμες Big Data επανεξετάζουν πολλές από τις ιδέες που πρωτοστάτησαν στην κοινότητα των βάσεων δεδομένων κατά τη δεκαετία του 2000, λειτουργικότητες, όπως οι τυπικοί βελτιστοποιητές ερωτημάτων σε αυτά τα πλαίσια, είναι στοιχειώδεις σε σύγκριση με πολλές ώριμες εμπορικές βάσεις δεδομένων, αλλά εξελίσσονται γρήγορα"* [78]. Ορισμένα συστήματα, όπως το Spark [41] ή το Storm [40], προσφέρουν ορισμένες βελτιστοποιήσεις ερωτημάτων, αλλά δεν βασίζονται σε χαρακτηριστικά ροής. Προκειμένου να βελτιστοποιήσουμε την απόδοσή τους, πρέπει είτε να γράψουμε προσαρμοσμένο κώδικα είτε να συντονίσουμε τις εφαρμογές μας με αλλαγές διαμόρφωσης. Από την άλλη πλευρά, παρόλο που το Apache Flink [42] υιοθετεί streaming SQL μαζί με αλγεβρικούς μετασχηματισμούς στην τελευταία έκδοση (1.3.0), χρησιμοποιώντας το Apache Calcite, δεν χρησιμοποιεί μεταβλητές από το περιβάλλον των δεδομένων ροής για να βρει το βέλτιστο σχέδιο εκτέλεσης. Επομένως, πρέπει να δοθεί μια πιο συγκεκριμένη λύση που εστιάζει στη σημασιολογία των ροών, συμπεριλαμβανομένων των τεχνικών βελτιστοποίησης βάσει κόστους, και όχι μόνο σε μετασχηματισμούς σε επίπεδο τοπολογίας.

Αυτές οι δύο πτυχές των συστημάτων διαχείρισης ροών δεδομένων δεν προσφέρουν μόνο ευκολία στη χρήση, αλλά βελτιώνουν σημαντικά την απόδοση ενός συστήματος. Κατά την υποβολή ενός πολύπλοκου ερωτήματος, ο χρήστης δεν χρειάζεται να ξοδεύει χρόνο για να το γράψει με κατάλληλο τρόπο ώστε να εκτελεστεί βελτιστοποιημένα. Για παράδειγμα, δεν απαιτείται να καθορίσει τη σειρά με την οποία θα γίνει η συνένωση των ροών δεδομένων, καθώς είναι ευθύνη της βελτιστοποίησης να χειριστεί τέτοιες περιπτώσεις. Απλές τεχνικές βελτιστοποίησης, όπως το κλάδεμα ιδιοτήτων κατά την προβολή, μπορούν να επηρεάσουν τόσο τη συνολική απόδοση όσο και τους πόρους που χρησιμοποιούνται, καθώς τα δεδομένα που μεταφέρονται στη διοχετευμένη διαδικασία εκτέλεσης ενός τέτοιου συστήματος μειώνονται.

0.3 Αντικείμενο Διπλωματικής και Συνεισφορά

Η διπλωματική εισάγει την υποστήριξη μιας δηλωτικής γλώσσας επερώτησης, αξιοποιώντας μια σύγχρονη πλατφόρμα ανοιχτού κώδικα για την ανάλυση, την επικύρωση και τη βελτιστοποίηση ερωτημάτων SQL. Σας παρουσιάζουμε τεχνικές βελτιστοποίησης που βασίζονται στο ρυθμό παραγωγής αποτελεσμάτων πάνω σε ροές δεδομένων και εφαρμόζουμε τα ευρήματά μας στη μηχανή επεξεργασίας ροών που έχουμε επιλέξει, το

SABER [2]. Οι τεχνικές αυτές ταξινομούνται στη στατική βελτιστοποίηση και έχουν τεράστιο αντίκτυπο στην απόδοση του συστήματος μας, RBStream.

Οι συνεισφορές μας συνοψίζονται ως εξής:

1. Βελτιστοποίηση ερωτημάτων: Χρησιμοποιούμε και διαμορφώνουμε μια σειρά από τεχνικές βελτιστοποίησης ερωτημάτων, όπως την ώθηση κατηγορημάτων προς τα κάτω, το κλάδεμα προβολών, τη συγχώνευση τελεστών και την επιλογή σειράς ενώσεων των συνεχών ροών δεδομένων, βασισμένες σε κανόνες και μοντέλα κόστους που προτείνονται στη βιβλιογραφία, για τη βελτιστοποίηση της πολύπλοκης επεξεργασίας SQL ερωτημάτων σε ροές δεδομένων, λαμβάνοντας υπόψη το ρυθμό εισροής, το μέγεθος των παραθύρων, τη χρήση των πόρων του συστήματος και την κατανάλωση μνήμης.
2. Μοντέλο Κόστους: Τροποποιήσαμε και επεκτείναμε τα χρησιμοποιούμενα μοντέλα, προκειμένου να βελτιστοποιήσουμε την απόδοση της εκτέλεσης των ερωτημάτων, ενώ ταυτόχρονα ελαχιστοποιούμε την κατανάλωση πόρων, όπως τη χρήση μνήμης και CPU. Εισάγουμε ένα μοντέλο κόστους που βασίζεται στο ρυθμό παραγωγής αποτελεσμάτων πάνω σε χαρακτηριστικά ροών δεδομένων. Το μοντέλο αυτό επιτρέπει την εκτίμηση της χρησιμοποίησης πόρων και βοηθά τον optimizer να επιλέξει το καλύτερο σχέδιο εκτέλεσης.
3. SQL υποστήριξη: Έχουμε προσθέσει υποστήριξη SQL για το SABER, μια υβριδική σχεσιακή μηχανή επεξεργασίας ροών. Σε αυτή τη διπλωματική, εισάγουμε μια δηλωτική γλώσσα επερώτησης, για να εκφράσουμε συνεχείς ερωτήσεις πάνω στις πηγές δεδομένων συνεχούς ροής, που βασίζεται σε ένα υποσύνολο στοιχείων παραθύρου οριζόμενων στην SQL-99 για τις συναρτήσεις OLAP [64]. Ένα ερώτημα SQL μεταφράζεται σε δεκάδες γραμμές κώδικα, έτοιμες για εκτέλεση στο SABER.
4. Υλοποίηση συστήματος: Δημιουργήσαμε το σύστημα μας χρησιμοποιώντας ως βάση το Apache Calcite [1], μια πλατφόρμα βελτιστοποίησης ερωτημάτων, και δημιουργήσαμε έναν μετατροπέα, για να μετατρέψουμε τους λογικούς τελεστές στους αντίστοιχους φυσικούς της μηχανής εκτέλεσης μας. Στην υλοποίηση μας, επεκτείνουμε το Apache Calcite για να χρησιμοποιήσουμε τις προαναφερόμενες τεχνικές, επιτρέποντας ταυτόχρονα στον χρήστη να εκτελεί εκφραστικές επερωτήσεις SQL ακολουθώντας το πρότυπο SQL-99, και προσφέρουμε ένα σύστημα end-to-end, όπου έχουμε ενσωματώσει το τροποποιημένο Apache Calcite με το SABER, μια υβριδική μηχανή επεξεργασίας ροών.

5. Αξιολόγηση αποτελεσμάτων: Έχουμε αναλύσει και αξιολογήσει την αποτελεσματικότητα του RBStream υπό διαφορετικές περιπτώσεις φόρτου εργασίας και διαφορετικές διαμορφώσεις του συστήματος πάνω σε Benchmark με συνθετικά δεδομένα, που αφορούσε συναλλαγές σε Αγορές και Παραγγελίες προϊόντων, και θα δείξουμε ότι η υλοποίηση μας μπορεί να ξεπεράσει την απλή προσέγγιση σε ένα διαδραστικό web UI. Σε ορισμένες περιπτώσεις χωρίς τελεστές συνένωσης, το βελτιστοποιημένο πλάνο είχε ως αποτέλεσμα ακόμη και 2,7 φορές μεγαλύτερη απόδοση, χρησιμοποιώντας σχεδόν τους μισούς πόρων σε σύγκριση με το μη βελτιστοποιημένο πλάνο. Όταν χρησιμοποιήθηκαν ένας ή περισσότεροι τελεστές συνένωσης στο ερώτημα, παρατηρήθηκε λιγότερο από το 70% του latency ενός συμβατικού πλάνου.

0.4 Δομή Εργασίας

Το υπόλοιπο αυτής της εργασίας είναι οργανωμένο ως εξής:

- Στην Ενότητα 0.5, θα επικεντρωθούμε στο θεωρητικό υπόβαθρο που απαιτείται για την κατανόηση των βασικών εννοιών για τις ροές δεδομένων, τη βελτιστοποίηση ερωτημάτων και τα Συστήματα Διαχείρισης ροών δεδομένων, μαζί με τις υπάρχουσες λύσεις σε αυτούς τους τομείς. Παρουσιάζουμε επίσης την πλατφόρμα βελτιστοποίησης, Apache Calcite, και τη μηχανή επεξεργασίας ροών δεδομένων, SABER, τα οποία χρησιμοποιούνται στην υλοποίηση μας.
- Στην Ενότητα 0.6, περιγράφουμε τη διαδικασία βελτιστοποίησης του Calcite. Σας παρουσιάζουμε κάποιους από τους ήδη υπάρχοντες κανόνες βελτιστοποίησης μαζί με δικούς μας κανόνες, οι οποίοι χρησιμοποιούνται για την υλοποίηση του συστήματός μας.
- Στην Ενότητα 0.7, ορίζουμε το μοντέλο του κόστους που χρησιμοποιείται για τη βελτιστοποίηση ερωτημάτων στο σύστημά μας. Καθορίζουμε τις παραμέτρους και το στόχο βελτιστοποίησης του μοντέλου, προκειμένου να καλύπτουν σε σημασιολογία τις ροές δεδομένων.
- Στην Ενότητα 0.8, παρουσιάζουμε την υλοποίηση του RBStream μαζί με ένα διαδραστικό web interface για να παρουσιάσουμε τα δεδομένα. Οι σχεδιαστικές επιλογές και τα χαρακτηριστικά της υλοποίησης μας εξηγούνται αναλυτικά.

- Στην Ενότητα 0.9, παρουσιάζουμε τα αποτελέσματα και τα διαγράμματα από τα πειράματα που τρέξαμε για το σύστημά μας με τα διάφορα μοντέλα κόστους που χρησιμοποιήθηκαν.
- Τέλος στην Ενότητα 0.10, συνοψίζουμε τα συμπεράσματά μας από τα αποτελέσματα, και αναφερόμαστε σε θέματα που δεν είχαμε την ευκαιρία να εργαστούμε και τα οποία θα μπορούσαν να θεωρηθούν ως μελλοντικές κατευθύνσεις.

0.5 Υπόβαθρο

Στο πλαίσιο αυτής της διπλωματικής, πρόκειται να χρησιμοποιήσουμε το Apache Calcite [1], μια δυναμική πλατφόρμα δεδομένων, και το SABER [2], μια κεντρική, υβριδική, υψηλών επιδόσεων μηχανή επεξεργασίας σχεσιακών ροών δεδομένων για επεξεργαστές και κάρτες γραφικών. Για να δικαιολογήσουμε την επιλογή μας και για την εξοικείωση του αναγνώστη με τις βασικές έννοιες που αφορούν τα συστήματα συνεχών ροών και τη βελτιστοποίηση ερωτημάτων, σε αυτό το κεφάλαιο πρόκειται να παρουσιάσουμε το αναγκαίο υπόβαθρο, που περιγράφει τους βασικούς μηχανισμούς και τα προγραμματιστικά μοντέλα που χρησιμοποιούνται στις μέρες μας, καθώς και μια σύντομη αναφορά σε σχετικές εργασίες.

0.5.1 Βασικές έννοιες για τις ροές δεδομένων

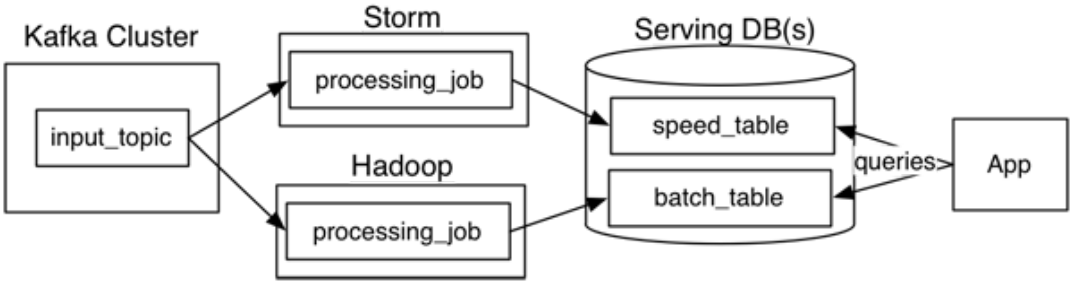
Πρώτα απ' όλα, θα πρέπει να καλύψουμε κάποιες βασικές έννοιες για τις ροές δεδομένων [5]. Αυτές οι έννοιες είναι χωρισμένες σε τρεις κατηγορίες:

- Ορολογία: για να εμπλακεί κάποιος με πιο πολύπλοκα ζητήματα, πρέπει να δοθούν ακριβείς ορισμοί.
- Δυνατότητες: αυτό το τμήμα είναι σχετικά με τις συχνά αντιληπτές αδυναμίες των συστημάτων συνεχών ροών.
- Χρονικά πεδία ορισμού: υπάρχουν δύο κύρια πεδία ορισμού του χρόνου που σχετίζονται με την επεξεργασία των δεδομένων.

Μια μηχανή συνεχών ροών είναι μια μηχανή εκτέλεσης σχεδιασμένη για απεριόριστα (συνεχώς αυξανόμενα, ουσιαστικά άπειρα) σύνολα δεδομένων, όπως αναφέρει ο Tyler Akidau στο post του για τα streaming συστήματα [5]. Ο όρος «επεξεργασία ροών

δεδομένων» δεν σημαίνει κατά προσέγγιση ή υποθετικά αποτελέσματα. Ένα καλά σχεδιασμένο σύστημα συνεχών ροών μπορεί να παράγει σωστά, συνεπή, επαναλήψιμα αποτελέσματα όπως και κάθε υπάρχουσα μηχανή επεξεργασίας με δέσμες (batch engine - Lambda Architecture [72]). Στην Lambda Αρχιτεκτονική τρέχουμε ένα σύστημα συνεχούς ροής, παράλληλα με ένα σύστημα δέσμης. Παρά το γεγονός ότι και τα δύο εκτελούν τον ίδιο υπολογισμό, το σύστημα συνεχούς ροής επιστρέφει με χαμηλό latency ανακριβή αποτελέσματα, ενώ το σύστημα δέσμης παρέχει τα σωστά αποτελέσματα κάποια στιγμή αργότερα.

Ωστόσο, η διατήρηση ενός τέτοιου συστήματος δεν είναι πρακτική, αφού έχουμε υπολογισμό του αποτελέσματος δύο φορές χρησιμοποιώντας δύο υποσυστήματα [35]. Επομένως, τα συστήματα επεξεργασίας ροών δεδομένων πρέπει να διασφαλίζουν την ορθότητα (ισχυρή συνέπεια απαιτείται για την ακριβώς μία φορά επεξεργασία) και τα εργαλεία για την αντιμετώπιση προβλημάτων που σχετίζονται με το χρόνο (απεριόριστα, μη ταξινομημένα δεδομένα με χρονικής απόκλισης μεταξύ του χρόνου εκδήλωσης του γεγονότος και της επεξεργασίας του από το σύστημα). Ο σχεδιασμός και η υλοποίηση ισχυρών συστημάτων επεξεργασίας συνεχούς ροής παραμένει ένα δύσκολο έργο, όχι μόνο λόγω της τεράστιας κλίμακας τους, της πολυπλοκότητας και της μη προβλεψιμότητας των εκτελέσεων του συστήματος, αλλά και λόγω της πρόκλησης τόσο του όγκου όσο και της ταχύτητας των μεγάλων δεδομένων. Η μετάβαση από τα συστήματα Αρχιτεκτονικής Lambda σε Kappa [34, 35] απαιτεί έναν ισχυρό επεξεργαστή ροής για να επεξεργάζεται δεδομένα με υψηλούς ρυθμούς εισροής και να διασφαλίζει την ασφαλή διατήρηση τους.

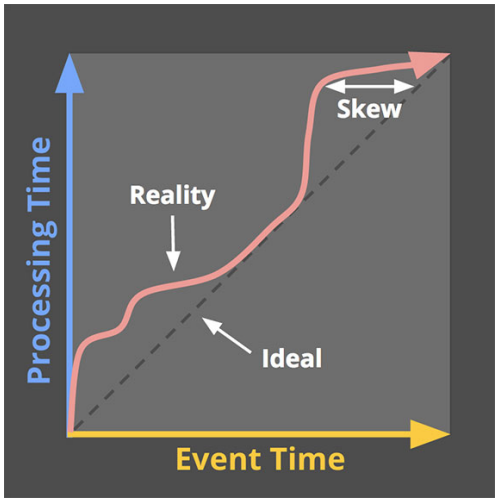


Εικ. 0.5.1.a, Λάμδα Αρχιτεκτονική [35].

Στη συνέχεια, πρέπει να καταλάβουμε τα δύο κύρια πεδία χρόνου που εμπλέκονται στην επεξεργασία ροών δεδομένων:

- ώρα του συμβάντος, που είναι ο χρόνος κατά τον οποίο συνέβη το γεγονός.
- χρόνος επεξεργασίας, που είναι ο χρόνος κατά τον οποίο τα γεγονότα παρατηρήθηκαν στο σύστημα.

Σε έναν ιδανικό κόσμο, οι δύο αυτοί χρόνοι θα είναι πάντα ίσοι. Ωστόσο, η απόκλιση που υπάρχει μεταξύ τους, δεν μπορεί να περάσει απαρατήρητη. Υπάρχουν περιπτώσεις όπου νοιαζόμαστε για μόνο έναν από αυτούς και θα δημιουργήσουμε τα παράθυρα σύμφωνα με αυτό το συγκεκριμένο χρονικό πεδίο, και περιπτώσεις όπου μας χρειάζονται και οι δύο, στις οποίες προκύπτουν διάφορα προβλήματα.



Εικ. 0.5.1.b, Χρόνος που συνέβη το γεγονός σε σύγκριση με το Χρόνο Επεξεργασίας [5].

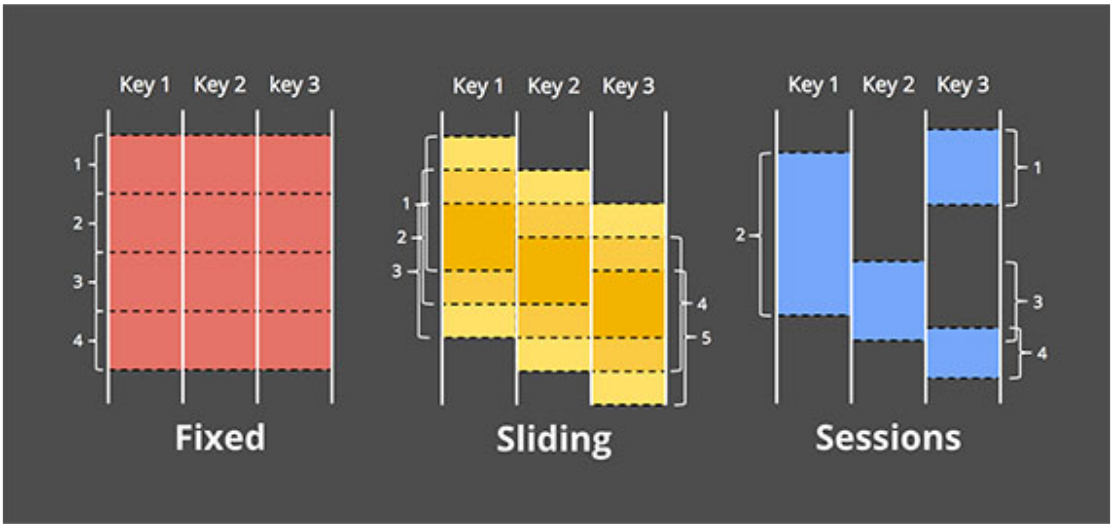
Στις παραδοσιακές βάσεις δεδομένων, ξέρουμε πώς να ασχοληθούμε με την επεξεργασία περιορισμένων δεδομένων. Το πρόβλημα προκύπτει όταν έχουμε να αντιμετωπίσουμε απεριόριστα σύνολα δεδομένων. Στις μηχανές συνεχούς ροής έχουμε πέντε διαφορετικές προσεγγίσεις για την αντιμετώπιση των απεριόριστων αυτών δεδομένων:

- Χρόνο-αγνωστικιστικός: Αυτό το είδος της επεξεργασίας που χρησιμοποιείται όταν ο χρόνος είναι ουσιαστικά άνευ σημασίας (π.χ. η σχετική λογική είναι με γνώμονα τα δεδομένα). Τα συστήματα συνεχούς ροής θα πρέπει να υποστηρίζουν αυτές τις περιπτώσεις χρήσης. Τέτοιες ενέργειες είναι το Φιλτράρισμα και η Προβολή. Οι ενέργειες αυτές δεν έχουν κατάσταση και δεν χρειάζεται μνήμη για να τις υπολογίσουμε.
- Προσέγγιση: Ο δεύτερος τύπος επεξεργασίας είναι οι προσεγγιστικοί αλγόριθμοι, όπως το προσεγγιστικό Top-N [65] και streaming K-means [66]. Αυτοί οι

αλγόριθμοι έχουν σχεδιαστεί για απεριόριστα δεδομένα και χαμηλή επιβάρυνση. Ωστόσο, είναι περίπλοκοι και η κατά προσέγγιση φύση τους περιορίζει τη χρήση τους.

- Παράθυρα σύμφωνα με το χρόνο επεξεργασίας.
- Παράθυρα σύμφωνα με το χρόνο εκδήλωσης.
- Παράθυρα από πλειάδες.

Για να προχωρήσουμε με τα τελευταία τρία είδη της επεξεργασίας, πρέπει να δοθεί ο ορισμός των παραθύρων. *"Η διαδικασία δημιουργίας παραθύρων είναι ο χωρισμός σε πεπερασμένα τμήματα μιας πηγής δεδομένων (είτε απεριόριστης είτε περιορισμένης) σύμφωνα με σταθερά όρια για την επεξεργασία της [5]."*



Εικ. 0.5.1.c, Τύποι Παραθύρων [5].

Όπως φαίνεται στο παραπάνω σχήμα, υπάρχουν τρεις διαφορετικοί τύποι παραθύρων:

- Σταθερά Παράθυρα: Αυτά τα παράθυρα "κόβουν" το χρόνο σε τμήματα με μια χρονική διάρκεια σταθερού μεγέθους, τα οποία εφαρμόζονται είτε ομοιόμορφα σε ολόκληρο το σύνολο δεδομένων (ευθυγραμμισμένα παράθυρα όπως στο σχήμα παραπάνω.) ή σε διαφορετικά υποσύνολα των δεδομένων (μη ευθυγραμμισμένα παράθυρα). Καλούνται, επίσης, tumbling windows.
- Συρόμενα παράθυρα: Πρόκειται για μια γενίκευση των σταθερών παραθύρων και ορίζονται από δύο παραμέτρους, ένα σταθερό μήκος και ένα σταθερό χρονικό διάστημα. Αν η περίοδος είναι μικρότερη από το μήκος, τα παράθυρα επικαλύπτονται. Αν είναι ίσα, έχουμε σταθερά παράθυρα. Τέλος, όταν η περίοδος

είναι μεγαλύτερη από το μήκος, έχουμε κάτι σαν ένα παράθυρο δειγματοληψίας, που εξετάζει μόνο υποσύνολα των δεδομένων με την πάροδο του χρόνου.

- Συνεδρίες: Οι συνεδρίες είναι δυναμικά παράθυρα, που αποτελούνται από ακολουθίες γεγονότων που τερματίζονται με ένα κενό αδράνειας μεγαλύτερο από κάποιο χρονικό όριο. Το μήκος των συνεδριών δεν μπορεί να προσδιοριστεί εκ των προτέρων και εξαρτάται από τα πραγματικά δεδομένα που εμπλέκονται.

Στα παράθυρα σύμφωνα με το χρόνο επεξεργασίας, αποθηκεύουμε προσωρινά τα εισερχόμενα δεδομένα για ένα συγκεκριμένο περιθώριο του χρόνου επεξεργασίας. Αυτό το είδος της επεξεργασίας είναι απλό, διευκολύνει τον έλεγχο πληρότητας του παραθύρου και ταιριάζει σε περιπτώσεις όπου θέλουμε να συμπεράνουμε πληροφορίες σχετικά με την πηγή δεδομένων που παρατηρούμε.

Αντίθετα, σε περιπτώσεις όπου τα δεδομένα είναι συναφή με το χρόνο εκδήλωσης, είναι δύσκολο να τα επεξεργαστούμε, καθώς μπορεί να φτάσουν εκτός σειράς. Σε τέτοιες περιπτώσεις, διαχωρίζουμε τις πηγές δεδομένων σε πεπερασμένα κομμάτια που αντικατοπτρίζουν τον χρόνο που συνέβησαν τα γεγονότα αυτά. Ωστόσο, είναι δύσκολο να διατηρηθεί η ορθότητα του χρόνου εκδήλωσης (σωστή σειρά) και χρειάζεται περισσότερη προσωρινή αποθήκευση δεδομένων, αλλά μπορούμε να δημιουργήσουμε συνεδρίες.

Στα παράθυρα από πλειάδες, το μέγεθος των παραθύρων μετράται σε αριθμό στοιχείων. Τα παράθυρα με βάση τις πλειάδες είναι μια μορφή των παραθύρων χρόνου επεξεργασίας, όπου τα στοιχεία έχουν μονοτονικά αυξανόμενες χρονικές σημάνσεις.

Λαμβάνοντας υπόψιν ότι νέα δεδομένα φθάνουν κατά τη διάρκεια που παλιά δεδομένα υποβάλλονται σε επεξεργασία, θα πρέπει να χρησιμοποιηθούν διαφορετικοί αλγόριθμοι για τους τελεστές επεξεργασίας συνεχούς ροής δεδομένων. Οι αλγόριθμοι αυτοί θα πρέπει να είναι σε θέση να συμβαδίσουν με την ρυθμό εισροής των ροών δεδομένων, έχοντας χαμηλό χρόνο υπολογισμού ανά στοιχείο. Επομένως, ο υπολογισμός προτιμάται να περιορίζεται στην κύρια μνήμη χωρίς πρόσβαση στο δίσκο. Επίσης, η είσοδος των τελεστών που "μπλοκάρουν" την επεξεργασία (όπως Join, Aggregate και Sort) πρέπει να υπάρχει πριν ξεκινήσει η επεξεργασία τους. Για παράδειγμα, χρειαζόμαστε το σύνολο των δεδομένων να προϋπάρχει πριν από την εφαρμογή της λειτουργίας της ταξινόμησης. Αυτοί οι τελεστές θεωρούμε ότι κρατάνε απαραίτητες πληροφορίες για την κατάσταση της επεξεργασίας,

καθώς θα πρέπει να χρησιμοποιήσουν τη μνήμη για να υπολογίσουν το αποτέλεσμά τους. Επεκτάσεις των ήδη γνωστών αλγορίθμων χρησιμοποιούνται, προκειμένου να αλλάξουμε τη συμπεριφορά "μπλοκαρίσματος" γνωστών αλγορίθμων αυτών των τελεστών σε συνεχόμενη επεξεργασία (e.g. non-blocking join [36]).

Προκειμένου να επιταχυνθεί η εκτέλεση του ερωτήματος και να κρατήσουμε το χρόνο υπολογισμού χαμηλό, υπάρχουν πολλές τεχνικές που μπορούν να χρησιμοποιηθούν [54]. Τα παράθυρα χρησιμοποιούνται για να ορίσουν όρια πάνω σε απεριόριστα δεδομένα και να βοηθήσουν το σύστημα συνεχούς ροής να υπολογίσει το αποτέλεσμα τελεστών που "μπλοκάρουν" την επεξεργασία. Επιπλέον, υπάρχουν ορισμένες εφαρμογές στις οποίες υψηλής ποιότητας προσεγγιστικές απαντήσεις είναι αποδεκτές. Σε αυτές τις περιπτώσεις, χρησιμοποιούμε αλγόριθμους προσέγγισης για τον υπολογισμό του αποτελέσματος μας.

0.5.2 CQL: Continuous Query Language

Η CQL [3] δεν είναι SQL, αλλά μια δηλωτική γλώσσα SQL για ερωτήματα σε συνεχείς ροές δεδομένων και αποθηκευμένες σχέσεις. Η σημασιολογία της CQL στηρίζεται σε τρεις τύπους ενεργειών: stream-to-relations, relation-to-relation και τη relation-to-stream σε δύο τύπους δεδομένων, τις ροές και τις σχέσεις.

Ροές και Σχέσεις

Η κύρια διαφορά μεταξύ των σχέσεων στην CQL και τις τυπικές σχέσεις είναι η πρόσθετη έννοια του χρόνου στην σημασιολογία των σχέσεων R της CQL.

- Ροή Δεδομένων - Μια ροή S είναι ένα (πιθανώς άπειρο) σύνολο στοιχείων (s, t), όπου το s είναι μια πλειάδα του σχήματος που περιγράφει το S και τ είναι η χρονική σήμανση που συνδέεται με το στοιχείο. Η χρονική σήμανση δεν θεωρείται να μέρος του σχήματος της ροής, και θα μπορούσε να υπάρξουν μηδέν, ένα ή πολλαπλά στοιχεία που μοιράζονται την ίδια χρονική σήμανση στη ροή μας. Η μόνη απαίτηση που έχουμε είναι ότι υπάρχει ένας πεπερασμένος (αλλά απεριόριστος) αριθμός στοιχείων με την ίδια χρονική σήμανση. Υπάρχουν δύο "είδη" ροών: οι ροές δεδομένων πηγής που φτάνουν στο σύστημα επεξεργασίας συνεχούς ροής, οι οποίες ονομάζονται ροές βάσης, και οι ροές που παράγονται από τους ενδιάμεσους τελεστές, οι οποίες ονομάζονται παραγόμενες ροές.

- Σχέση - Μια σχέση R είναι μια απεικόνιση από ένα διακριτό, διατεταγμένο πεδίο χρόνου T σε ένα πεπερασμένο, αλλά απεριόριστο σύνολο πλειάδων του σχήματος αυτής της σχέσης. Για μια συγκεκριμένη χρονική στιγμή $\tau \in T$, έχουμε R(τ), το οποίο είναι ένα μη ταξινομημένο σύνολο πλειάδων για την χρονική στιγμή αυτήν.

Τελεστές

Μερικοί συχνοί τελεστές SQL από σχεσιακές βάσεις δεδομένων, όπως η συνένωση (join) και οι συναθροιστικές συναρτήσεις (aggregation), μπλοκάρουν την επεξεργασία και δεν μπορούν να χρησιμοποιηθούν για τον υπολογισμό αποτελεσμάτων σε ένα πλαίσιο συνεχούς ροής. Για να ξεπεραστεί αυτό το εμπόδιο, χρησιμοποιούμε τελεστές παραθύρων, για να διαιρέσουμε τις ροές σε πιθανώς επικαλυπτόμενα υποσύνολα, μετά από τη σάρωση τους, προκειμένου να μειωθεί το πεδίο υπολογισμού του ερωτήματος σε επίπεδο παραθύρου.

Η έννοια του παραθύρου είναι ενσωματωμένη στην CQL σημασιολογία, με τη χρήση της έννοιας της στιγμιαίας σχέση, όπως συζητήθηκε παραπάνω. Αυτό επιτρέπει τις μηχανές επεξεργασίας συνεχούς ροής να υλοποιήσουν τελεστές που μπλοκάρουν την επεξεργασία και την ενσωμάτωση των αποθηκευμένων σχέσεων στα ερωτήματα του συστήματος. Όταν μια ροή μετατρέπεται σε στιγμιαία σχέση μπορούμε να εργαστούμε όπως κάναμε για τις σχέσεις.

- **Stream-to-relation** - Η είσοδος αυτού του τελεστή είναι μια ροή S και η έξοδος είναι μια σχέση R με τον ίδιο σχήμα όπως το S. Κάθε σχέση R(τ) θα πρέπει να είναι υπολογίσιμη από το S έως το χρονικό σημείο τ, για κάθε τ. Στη CQL έχουμε μόνο ένα stream-to-relation τελεστή, τον τελεστή παραθύρου (Window). Υπάρχουν τρεις κατηγορίες συρόμενων παραθύρων στην CQL, οι οποίες ορίζονται ως εξής: με βάση το χρόνο, με βάση τις πλειάδες, και κατανεμημένα. Ο τελεστής συρόμενου παραθύρου βασίζεται στις προδιαγραφές της SQL-99.
    - **Time-based sliding windows**: Αυτός ο τύπος παραθύρου καθορίζεται από ένα χρονικό διάστημα T και ορίζει μια υπολογίσιμη χρονική περίοδο εφαρμογής. Εξ ορισμού, το αποτέλεσμα του συρόμενου παραθύρου υπολογίζεται από όλες τις τελευταίες πλειάδες μιας διατεταγμένης ροής που ανήκουν σε ένα χρονικό διάστημα μεγέθους T. Η σχέση εξόδου R του "S [Range T]" ορίζεται ως εξής:

$$R(\tau) = \{s \mid \langle s, \tau' \rangle \in S \ \wedge \ (\tau' \leq \tau) \ \wedge \ (\tau' \geq \max\{\tau - T, 0\})\}$$

Υπάρχουν δύο ειδικές περιπτώσεις: T = 0 και T = ∞. Στην πρώτη περίπτωση, η σύνταξη "S [Now]" εισάγεται, και το R(τ) αποτελείται από πλειάδες που προέρχονται από στοιχεία του S με σφραγίδα χρόνου τ. Στη δεύτερη περίπτωση, η σύνταξη "S [Range Unbounded]" χρησιμοποιείται, και το R(τ) αποτελείται από πλειάδες που προέρχονται από όλα τα στοιχεία της S μέχρι και το τ.

- **Tuple-based windows**: Ένα συρόμενο παράθυρο πλειάδων που βασίζεται σε μια ροή S καθορίζεται από έναν αριθμό N γραμμών. Η σχέση της εξόδου υπολογίζεται από το συρόμενο παράθυρο πλειάδων περιλαμβάνει τις τελευταίες N πλειάδες μιας διατεταγμένης ροής ("S [Rows N]"). Μια σχέση R(τ) αποτελείται από τις N πλειάδες της S με τη μεγαλύτερη χρονική σφραγίδα που είναι μικρότερη ή ίση του τ. Εάν υπάρχουν αρκετές πλειάδες με την νιοστή πιο πρόσφατη χρονική σφραγίδα (ενώ για λόγους σαφήνειας, ας υποθέσουμε ότι οι άλλες N-1 πιο πρόσφατες χρονικές σφραγίδες είναι μοναδικές), θα πρέπει να "σπάσουμε την ισοπαλία" με κάποιο τρόπο για να δημιουργήσει ακριβώς N πλειάδες στο παράθυρο. Για το λόγο αυτό, τα παράθυρα πλειάδων μπορεί να μην είναι κατάλληλα για τις περιπτώσεις που οι χρονικές σφραγίδες δεν είναι μοναδικές. Υπάρχει μια ειδική περίπτωση, όταν N = ∞ και καθορίζεται από [Rows Unbounded].
- **Partitioned Windows**: Αυτό το είδος συρόμενου παραθύρου παίρνει ένα θετικό ακέραιο N και ένα υποσύνολο {A1,. . . , Ak} γνωρισμάτων S ως παραμέτρους. Θα μπορούσε να καθοριστεί από [Partition By A1,...,Ak Rows N]. Αυτό το παράθυρο χωρίζει λογικά το S σε διαφορετικές υπο-ροές βασιζόμενο στην ισότητα των χαρακτηριστικών A1,. . . , Ak, υπολογίζει μια πλειάδα με βάση το συρόμενο παράθυρο του μεγέθους N ανεξάρτητα και τελικά υπολογίζει το αποτέλεσμα από την συνένωση αυτών των παραθύρων.

- **Relation-to-relation** - Οι relation-to-relation τελεστές προέρχονται από τους παραδοσιακούς σχεσιακούς τελεστές της SQL. Η κύρια ιδέα πίσω από αυτούς τους τελεστές είναι ότι από τη στιγμή που έχουμε χρησιμοποιήσει έναν τελεστή Stream-to-relation πάνω σε μια ροή και την μετατρέψαμε σε παράθυρο, μπορούμε να την αντιμετωπίσουμε ως μια σχέση. Αυτοί οι τελεστές έχουν απλή

σημασιολογική αντιστοίχηση σε χρονικά μεταβαλλόμενα σχέσεις. Όλες οι παραδοσιακές σχέσεις ενός ερωτήματος SQL μπορούν να περιγραφούν με την CQL σημασιολογία.

- **Relation-to-stream** - Ένας relation-to-stream τελεστής παίρνει ως είσοδο μια σχέση R και δίνει ως έξοδο μια ροή S με το ίδιο σχήμα όπως η R. Κάθε ροή S πρέπει να είναι υπολογίσιμη από την R έως το χρονικό διάστημα τ, για κάθε τ. Χρησιμοποιούμε τη διαφορά μεταξύ της προηγούμενης και της τρέχουσας στιγμιαίας σχέσης για να μετατρέψουμε μια σχέση σε μια ροή. Με τους φορείς αυτούς μπορούμε να μετατρέψουμε το αποτέλεσμα της πράξης stream-to-relation σε μια ροή για περαιτέρω επεξεργασία, η οποία είναι απαραίτητη για την διαδικασία της εκτέλεσης του ερωτήματος σε μια μηχανή επεξεργασίας ροών. Στην CQL, υπάρχουν τρεις relation-to-stream τελεστές: IStream, Dstream και Rstream. Υποθέτουμε ότι οι τελεστές ∪, ×, και - καλύπτονται από τους ορισμούς που ακολουθούν.
    - 1. Όταν ο Istream (“insert stream”) εφαρμόζεται σε μια σχέση R περιέχει ένα στοιχείο ροής (s, τ), κάθε φορά που μια πλειάδα s ανήκει στην διαφορά R(τ) - R(τ-1). Ο εν λόγω τελεστής χρησιμοποιείται για να δημιουργήσει μια ροή δεδομένων από μια σχέση. Κάθε φορά που μια πλειάδα εισάγεται στη σχέση, ένα αντίγραφο αποστέλλεται στη ροή δεδομένων, αλλά μόνο αν οι πλειάδες που εισάγονται δεν είναι διπλότυπα. Ουσιαστικά έχουμε:

$$\mathtt{Istream}(R) = \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \{\tau\})$$

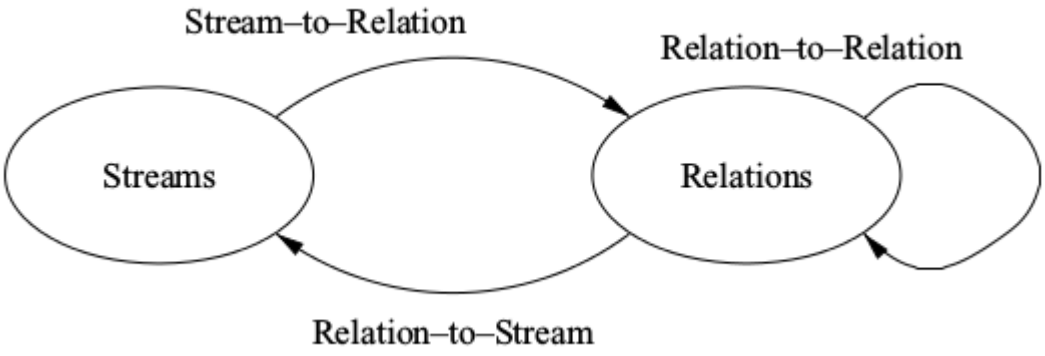
    - 2. Όταν ο Dstream (“delete stream”) εφαρμόζεται σε μια σχέση R περιέχει ένα στοιχείο ροής (s, τ), όποτε μια πλειάδα s ανήκει στη διαφορά R(τ - 1) - R(τ). Χρησιμοποιώντας τον τελεστή DSTREAM, μια ροή δεδομένων δημιουργείται από μια σχέση, όπου για κάθε πλειάδα που αφαιρείται από μια σχέση, αυτή η πλειάδα στέλνεται στο ροή δεδομένων. Ουσιαστικά έχουμε:

$$\mathtt{Dstream}(R) = \bigcup_{\tau > 0} ((R(\tau - 1) - R(\tau)) \times \{\tau\})$$

    - 3 Όταν ο Rstream (“relation stream”) εφαρμόζεται σε μια σχέση R περιέχει ένα στοιχείο ροής (s, τ), όποτε μια πλειάδα s είναι στην σχέση R σε χρόνο τ. Ο RSTREAM τελεστής μετατρέπει μια ολόκληρη σχέση σε μια ροή

δεδομένων. Δηλαδή, όλες οι πλειάδες παρόντες αυτή τη στιγμή στη σχέση αποστέλλεται στη ροή δεδομένων. Ουσιαστικά έχουμε:

$$\texttt{Rstream}(R) = \bigcup_{\tau \geq 0} (R(\tau) \times \{\tau\})$$



Εικ. 0.5.2.a , CQL Τελεστές [3].

Ας χρησιμοποιήσουμε μερικά παραδείγματα από το paper για την CQL [3], προκειμένου να διευκρινιστεί η χρήση των ανωτέρω τελεστών:

- ```
  Select Distinct vehicleId
  From PosSpeedStr [Range 30 Seconds]
  ```

  Αυτό το ερώτημα είναι κατασκευασμένο από έναν τελεστή stream-to-relation (ένα συρόμενο παράθυρο μεγέθους 30 δευτερόλεπτα και ολίσθησης 1) και έναν τελεστή relation-to-relation (distinct), που εκτελεί προβολή και εξάλειψη διπλότυπων και είναι γνωστός σε μας από τις παραδοσιακές βάσεις δεδομένων. Το ερώτημα έχει ληφθεί από Linear Road Benchmark, το οποίο χρησιμοποιείται από το paper της CQL και εξάγει μια σχέση με όλα τα ενεργά οχήματα που έχουν μεταδώσει μια μέτρηση της ταχύτητας κατά τη διάρκεια των τελευταίων 30 δευτερολέπτων.

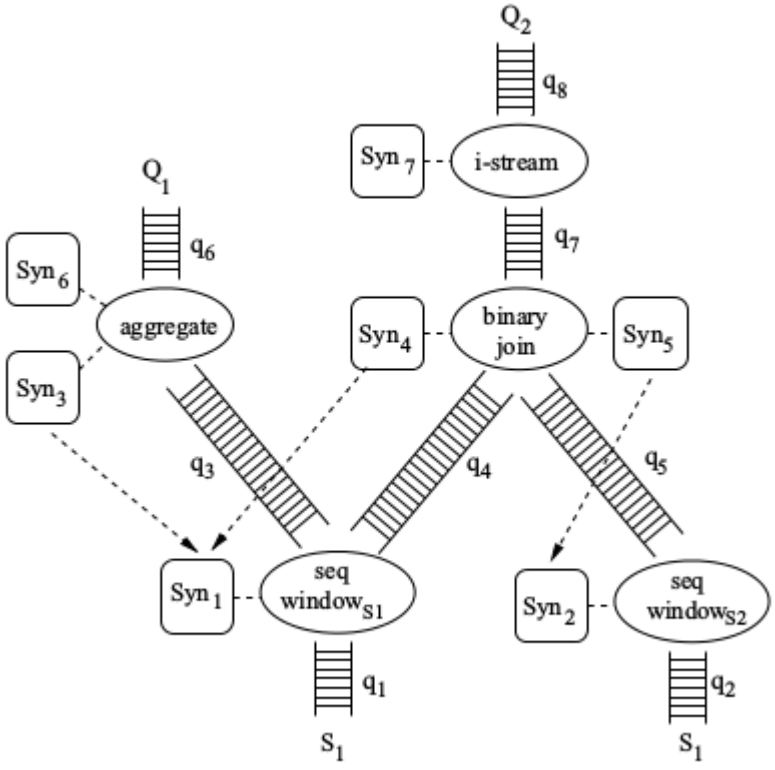
- ```
  Select Istream(*)
  From PosSpeedStr [Range Unbounded]
  Where speed > 65
  ```

Σε αυτό το ερώτημα μπορούμε να διακρίνουμε τρεις τελεστές: ένα stream-to-relation τελεστή απεριόριστου παραθύρου, που περιέχει σε ένα χρόνο τ όλες τις μετρήσεις θέσης ταχύτητα μέχρι τ, έναν τελεστή filter relation-to-relation και έναν τελεστή IStream relation-to-stream, ο οποίος προωθεί νέες τιμές ροών στο filter.

Πλάνα ερωτημάτων στη μηχανή STREAM

Εκτός από τα παραπάνω σημασιολογικά χαρακτηριστικά της CQL, είναι ενδιαφέρον να εξηγήσουμε τη στρατηγική της εκτέλεσης ερωτημάτων στη μηχανή επεξεργασίας STREAM, που χρησιμοποιεί CQL, για να κατανοηθούν καλύτερα τα βασικά τμήματα ενός συστήματος συνεχών ροών. Μία ροή αναπαρίσταται ως μία ακολουθία εισερχόμενων πλειάδων με χρονικές σφραγίδες. Αυτή η ακολουθία μπορεί να είναι άπειρη ή μπορεί να περιοριστεί με δύο διαφορετικές προσεγγίσεις, είτε με τη χρήση τεχνικών συμπίεσης, που προσπαθούν να συνοψίσουν τα δεδομένα, ή με τεχνικές παραθύρων, τα οποία δεσμεύουν τα δεδομένα σε πεπερασμένα κομμάτια. Μια σχέση αναπαρίσταται ως εισαγωγή και διαγραφή πλειάδων με χρονικές σφραγίδες, που δείχνουν την αλλαγή κατάστασης αυτής της σχέσης. Λαμβάνοντας υπόψη τους παραπάνω ορισμούς, είναι ευκολότερο να εφαρμοστεί αυξανόμενη επεξεργασία (incremental processing) και βελτιστοποίηση ερωτημάτων στις ροές. Η αυξανόμενη επεξεργασία προκύπτει από την παρατήρηση ότι πολλοί υπολογισμοί σχετικά με τις εισερχόμενες ροές μπορούν να εκτιμηθούν μόνο μία φορά και να επαναχρησιμοποιηθούν για επερχόμενους υπολογισμούς. Επίσης, οι βασικές έννοιες για τη βελτιστοποίηση συνεχών ερωτημάτων είναι ανάλογες με εκείνες από τις παραδοσιακές βάσεις δεδομένων. Αν το σύστημά μας υποστηρίζει σχεσιακές ροές δεδομένων και το λογικό πλάνο μας βασίζεται σε σχεσιακούς τελεστές, αλγεβρικές ισοδυναμίες μπορούν να παραχθούν από έναν βελτιστοποιητή ερωτημάτων.

Εδώ είναι ένα παράδειγμα για το πώς εκτελούνται δύο πλάνα ερωτημάτων στο STREAM:



Εικ. 0.5.2.b, πλάνα ερωτημάτων για δύο ερωτήματα στο STREAM [3].

Q1:
```
Select B, max(A)
From S1 [Rows 50,000]
Group By B
```

Q2:
```
Select Istream(*)
From
S1 [Rows 40,000],
S2 [Range 600 Seconds]
Where S1.A = S2.A
```

Στο προηγούμενο παράδειγμα, παρατηρούμε ότι το αποτέλεσμα υπολογίζεται με τρόπο διοχέτευσης. Εμείς δεσμεύουμε τα δεδομένα εισόδου με παράθυρα και κρατάμε την κατάσταση των τελεστών, προκειμένου να προχωρήσουμε στους υπολογισμούς μας. Οι ουρές q1 μέχρι q8 αντιπροσωπεύουν την κίνηση δεδομένων του συστήματος και οι συνόψεις S1 έως S7 χρησιμοποιούνται για να διατηρήσουν την κατάσταση των τελεστών. Μπορούμε επίσης να δούμε τους τελεστές παραθύρων που χρησιμοποιούνται στις εισόδους

των ροών δεδομένων για να δεσμεύονται τα δεδομένα σε πεπερασμένα κομμάτια. Πιο συγκεκριμένα, στο πρώτο ερώτημα, έχουμε το συρόμενο παράθυρο με βάση πλειάδες εύρους 50.000 γραμμών και ολίσθηση ένα, ενώ στο Q2 ερώτημα που έχουμε παράθυρα με βάση πλειάδες και χρόνο με ολίσθηση ένα και εύρος 40.000 γραμμές και 600 δευτερόλεπτα αντίστοιχα.

Έχουμε ήδη συζητήσει για τους τελεστές που κρατάνε ή δεν κρατάνε την πληροφορία της κατάστασης. Οι τελεστές που απαιτούν μνήμη για τον υπολογισμό τους, όπως οι συναθροιστικές συναρτήσεις (aggregate) ή η συνένωση (Join), παίζουν κρίσιμο ρόλο στο ρυθμό παραγωγής αποτελεσμάτων και την αξιοποίηση των πόρων ενός συστήματος συνεχών ροών. Αντίθετα, οι τελεστές που δεν κρατάνε την κατάσταση δεν επηρεάζονται από τα όρια του παραθύρου και έτσι μπορούν να δώσουν το ίδιο αποτέλεσμα, ανεξάρτητα από την επιλογή του παραθύρου. Ωστόσο, τοποθετώντας ένα filter ή ένα projection νωρίς μπορεί να μειώσει σημαντικά την ποσότητα των δεδομένων που παράγονται και μεταφέρονται σε μεταγενέστερα στάδια της εκτέλεσης. Από τα παραπάνω, είναι προφανές ότι η σωστή τοποθέτηση των τελεστών μπορεί να μεγιστοποιήσει το ρυθμό παραγωγής αποτελεσμάτων και να μειώσει τους απαραίτητους υπολογιστικούς πόρους. Αυτό είναι αυτό που προσπαθούμε να επιτύχουμε με τη βελτιστοποίηση ερωτημάτων.

0.5.3 Βελτιστοποίηση Ερωτημάτων

Αυτή η ενότητα ασχολείται με τις βασικές έννοιες της βελτιστοποίησης ερωτημάτων, που προέρχονται από τις παραδοσιακές βάσεις δεδομένων. Η βελτιστοποίηση ερωτημάτων είναι ένα από τα βασικά συστατικά του πυρήνα των συστημάτων διαχείρισης δεδομένων, που ενώνει μια δηλωτική γλώσσα ερωτημάτων με την αποδοτική εκτέλεση τους. Στις παραδοσιακές βάσεις δεδομένων, τα ερωτήματα SQL μπορεί να έχουν περισσότερα από ένα αντίστοιχα πλάνα εκτέλεσης. Παρά το γεγονός ότι όλα αυτά τα σχέδια δίνουν το ίδιο αποτέλεσμα όταν εκτελούνται, ενδέχεται να διαφέρουν ως προς το χρόνο εκτέλεσης ή την χρήση απαιτούμενων πόρων. Σε αυτήν την περίπτωση, η βελτιστοποίηση πρέπει να βρει το βέλτιστο ή ένα σχεδόν βέλτιστο πλάνο που θα εκτελεστεί.

Κάθε λογικό ή φυσικό πλάνο αντιπροσωπεύεται από ένα δέντρο τελεστών, στον οποίο οι κόμβοι αντιπροσωπεύουν τους τελεστές και οι ακμές αντιπροσωπεύουν τη ροή δεδομένων. Όταν έχουμε σύνθετα ερωτήματα πάνω σε πολλαπλές πηγές εισόδου, οι διαφορετικοί

τρόποι υλοποίησης τους μπορεί να είναι εκατομμύρια ή και περισσότεροι. Οι αποφάσεις για τη σειρά των ενώσεων, τη σειρά των διαδοχικών τελεστών (π.χ. εάν ένα φίλτρο θα πρέπει να ωθείται κάτω από μια προβολή) ή για το ποιον αλγόριθμο θα χρησιμοποιήσουμε για την υλοποίηση ενός τελεστή (π.χ. natural join or hash join)) παράγουν ένα μεγάλο χώρο αναζήτησης. Η ανίχνευση του βέλτιστου σχεδίου, ακόμη και σε απλά ερωτήματα έχει τεράστιο αντίκτυπο στο χρόνο που δαπανάται για να υπολογιστεί το αποτέλεσμα και στην αξιοποίηση των πόρων.

Η βελτιστοποίηση ερωτήσεων μπορεί να θεωρηθεί ότι είναι ένα πρόβλημα αναζήτησης και χρειάζεται:

- Ένα χώρος αναζήτησης των πιθανών πλάνων.
- Ένα μοντέλο κόστους, που θα χρησιμοποιηθεί για να υπολογιστεί το κόστος του κάθε υποψηφίου πλάνου.
- Ένα αλγόριθμος απαρίθμησης για να ψάξουμε μέσα από τα πιθανά σχέδια και να βρούμε το αυτό με το βέλτιστο κόστος. Αυτός ο αλγόριθμος μπορεί να συνδυαστεί με ένα μηχανισμό κλαδέματος, για να μειώσει το μέγεθος του χώρου αναζήτησης.

Ένας optimizer θα πρέπει να έχει τα ακόλουθα χαρακτηριστικά:

- Να περιλαμβάνει τα πλάνα με το χαμηλότερο κόστος στο χώρο αναζήτησης του.
- Το μοντέλο κόστους που χρησιμοποιεί να παρέχει ακριβή εκτίμηση του κόστους.
- Ο αλγόριθμος απαρίθμησης να είναι υπολογιστικά αποδοτικός.

Υπάρχουν δύο διαφορετικές αρχιτεκτονικές βελτιστοποίησης ερωτήσεων που χρησιμοποιούνται κατά τη διάρκεια των τελευταίων δεκαετιών: System R-style bottom-up dynamic programming optimizers [50] και Volcano-style top-down memoization with branch-and-bound pruning optimizers [11, 12]. Θα μπορούσαμε να εφαρμόσουμε πάντα ευριστικούς κανόνες για ένα συγκεκριμένο υποσύνολο του χώρου αναζήτησης, αλλά σε γενικές γραμμές αυτοί οι μετασχηματισμοί δεν εξασφαλίζουν ότι το κόστος θα μειωθεί σε κάθε περίπτωση. Ως εκ τούτου, οι κανόνες βελτιστοποίησης πρέπει να εφαρμόζονται με τρόπο που βασίζεται στο κόστος. Και οι δύο επιτυγχάνουν την εύρεση του βέλτιστου πλάνου σύμφωνα με ένα μοντέλο κόστους. Η ποιότητα του πλάνου εξαρτάται από τους κανόνες μετατροπής που χρησιμοποιούνται για την παραγωγή του και την ορθότητα του μοντέλου του κόστους και όχι από το είδος της βελτιστοποίησης. Ωστόσο, οι top-down optimizers έχουν το πλεονέκτημα ότι μπορούν να κλαδεύουν το χώρο αναζήτησης νωρίς.

Το κλάδεμα θα είναι αποτελεσματικό ανάλογα με τη σειρά αναζήτησης και πόσο γρήγορα φτάνει στο καλύτερο σχέδιο. Θα συζητήσουμε αργότερα την αρχιτεκτονική του Volcano πιο λεπτομερώς.

Οι ίδιες αρχές της βελτιστοποίησης ερωτημάτων μπορούν να εφαρμοστούν σε συστήματα διαχείρισης δεδομένων συνεχούς ροής, εισάγοντας τους stream-to-relation και relation-to-stream τελεστές. Αν προσθέσουμε τους κατάλληλους κανόνες για τους τελεστές αυτούς και τους συνδυάσουμε με τους κανόνες relation-to-relation, μπορούμε να βελτιστοποιήσουμε τα ερωτήματα ροών μας, όπως κάνουμε με τα παραδοσιακά ερωτήματα των βάσεων δεδομένων. Δύο πλάνα ερωτημάτων σε ροές είναι ισοδύναμα εφόσον παράγουν ισοδύναμα στιγμιότυπα αποτελεσμάτων. Αυτό σημαίνει ότι για κάθε χρονική στιγμή t στιγμιότυπα των δύο πλάνων είναι ίσα.

Μετρικές Κόστους και Στατιστικά

Η βελτιστοποίηση ερωτημάτων σε παραδοσιακές σχεσιακές βάσεις δεδομένων χρησιμοποιεί τις πληροφορίες της επιλεκτικότητας (selectivity) και των διαθέσιμων πόρων για να διαλέξει το αποτελεσματικό πλάνο εκτέλεσης για το ερώτημα (για παράδειγμα πλάνα με τους λιγότερους κύκλους CPU ή τις λιγότερες προσπελάσεις στο δίσκο). Ωστόσο, αυτές οι μετρικές και οι στατιστικές δεν μπορούν να χρησιμοποιηθούν για τη βελτιστοποίηση ερωτημάτων συνεχούς ροής, όπου το κόστος επεξεργασίας ανά μονάδα χρόνου είναι πιο κατάλληλο [24, 25, 26]. Η αλλαγή του στόχου βελτιστοποίησης για το υψηλότερο ποσοστό αποτελεσμάτων [24], θα μπορούσε να μας δώσει μια αξιόπιστη λύση. Υπό την προϋπόθεση ότι έχουμε το ρυθμό άφιξης ροών και το ρυθμό παραγωγής αποτελεσμάτων από τους τελεστές, θα μπορούσαμε επίσης να ψάξουμε για ένα πλάνο που παίρνει το λιγότερο χρόνο για την παραγωγή αποτελεσμάτων για ένα δεδομένο αριθμό πλειάδων (rate-based model [25]).

Μερικές φορές, οι τελεστές μπορεί να χρειαστεί να αλλάξουν on-the-fly σύμφωνα με αλλαγές στις συνθήκες του συστήματος κατά το χρόνο εκτέλεσης, λόγω της φύσης των πηγών δεδομένων εισόδου (προσαρμοστική βελτιστοποίηση). Υπάρχουν τρεις λόγοι που μπορεί να αλλάξει το αρχικό κόστος του πλάνου ενός ερωτήματος: αλλαγή στο χρόνο επεξεργασίας ενός τελεστή, αλλαγή της επιλεκτικότητας (selectivity) των κατηγορημάτων και αλλαγή του ρυθμού άφιξης μιας ροής. Έχουν υπάρξει σχετικές εργασίες για

βελτιστοποίηση ερωτημάτων, είτε στατική [24, 25, 26] είτε προσαρμοστική [27, 28], σε ερωτήματα επεξεργασίας ροών δεδομένων.

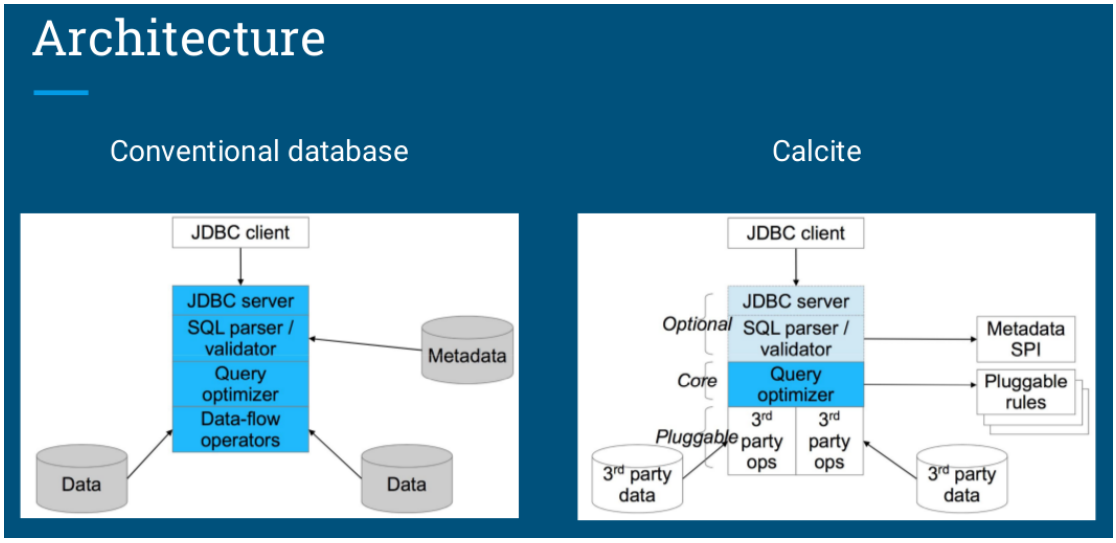
Ορισμένες πιθανές μετρικές κόστους για την επεξεργασία ροών δεδομένων μαζί με τα απαραίτητα στατιστικά στοιχεία που απαιτούνται για να υλοποιηθεί τόσο στατική όσο και προσαρμοστική βελτιστοποίηση είναι [51]:

- Η ακρίβεια και η καθυστέρηση των αποτελεσμάτων έναντι της χρήσης μνήμης και CPU: Σε ορισμένες περιπτώσεις, θα πρέπει να αποφασίσουμε σχετικά με την ακρίβεια και την υποβολή αποτελεσμάτων με καθυστέρηση έναντι της χρήσης μνήμης και CPU, ανάλογα με τη φύση της εφαρμογής μας. Μπορούμε να χρησιμοποιήσουμε τεχνικές δειγματοληψίας και διαγραφής δεδομένων για να μειώσουμε τη χρήση της μνήμης, αυξάνοντας όμως την πιθανότητα λάθους στο αποτέλεσμα.
- Ο ρυθμός παραγωγής αποτελεσμάτων: Αν ο ρυθμός άφιξης των ροών και ο ρυθμός παραγωγής αποτελεσμάτων των τελεστών είναι γνωστά, είναι δυνατόν να βελτιστοποιηθεί το σύστημα για υψηλότερο ρυθμό αποτελεσμάτων ή να βρεθεί ένα πλάνο που παίρνει το λιγότερο χρόνο για την παραγωγή αποτελεσμάτων για ένα δεδομένο αριθμό πλειάδων [25].
- Η χρήση ισχύος: Η κατανάλωση ενέργειας διαδραματίζει έναν κρίσιμο ρόλο σε ορισμένου τύπου εφαρμογές (π.χ. σε ένα ασύρματο δίκτυο αισθητήρων που λειτουργούν με μπαταρίες)

0.5.4 Calcite

Το Apache Calcite [1] είναι μια δυναμική πλατφόρμα διαχείρισης δεδομένων γραμμένη σε Java, παλαιότερα γνωστό ως Optic, εμπνευσμένο από τον Volcano Optimizer [7, 8]. Υπάρχουν τέσσερα στάδια στη βελτιστοποίησης ερωτημάτων: α) η ανάλυση του ερωτήματος με ένα JavaCC αναλυτή, β) η επικύρωση του ερωτήματος με τα γνωστά μεταδεδομένα βάσης δεδομένων, γ) η βελτιστοποίηση του λογικού πλάνου και η μετατροπή του σε φυσικούς τελεστές και δ) η μετατροπή του φυσικού πλάνου για την εκτέλεση σε συγκεκριμένο σύστημα. Το Calcite είναι *"μια συλλογή από βιβλιοθήκες λογισμικού και εργαλείων που μπορούν να χρησιμοποιηθούν"* [20] για να υλοποιήσουμε και να προσθέσουμε προσαρμοσμένη λογική για κάθε ένα από τα προηγούμενα τέσσερα στάδια. Οι χρήστες μπορούν να προσθέσουν τους κανόνες βελτιστοποίησης που επιθυμούν και να επεκτείνουν την ενσωματωμένη λογική και φυσική άλγεβρα, σύμφωνα με τον σύστημα που

χρησιμοποιείται για την εκτέλεση των ερωτημάτων, προκειμένου να επιτευχθεί προσαρμοσμένη βελτιστοποιήση. Για παράδειγμα, μπορούν να συνδέσουν τους κανόνες και τους τελεστές που είναι ήδη υλοποιημένοι σε ένα συγκεκριμένο μοντέλο κόστους, που αντιστοιχεί στη φυσική υλοποίηση των τελεστών. Πολλές σύγχρονες λύσεις για εφαρμογές δεδομένων σε μεγάλη κλίμακα, όπως το Apache Hive [56], το Apache Kylin [57], το Apache Drill [58] και το Apache Phoenix [59], χρησιμοποιούν το Calcite για το σχεδιασμό πλάνων και για τη βελτιστοποίηση τους. Ένα άλλο ενδιαφέρον χαρακτηριστικό των βιβλιοθηκών που χρησιμοποιεί το Calcite, είναι ότι βοηθούν τη φάση παραγωγής κώδικα του query planner. Για όλους αυτούς τους λόγους, είναι η προτιμώμενη λύση για την ανάλυση, την επικύρωση, τη βελτιστοποίηση και τη μετατροπή ερωτημάτων SQL σε ένα πλάνο εκτέλεσης στη μηχανή ροών μας.



Εικ. 0.5.4, Η αρχιτεκτονική με το Calcite [13].

"Το Calcite περιέχει πολλά από τα κομμάτια που συνθέτουν ένα τυπικό σύστημα διαχείρισης βάσης δεδομένων, αλλά παραλείπει κάποιες βασικές λειτουργίες: την αποθήκευση των δεδομένων, τους αλγορίθμους για την επεξεργασία των δεδομένων, καθώς και μια αποθήκη για την αποθήκευση μεταδεδομένων [1]" Αυτή η επιλογή είναι σκόπιμη, καθώς καθιστά το Calcite μια πολύ καλή επιλογή για τη διαμεσολάβηση μεταξύ των εφαρμογών και ένα ή περισσότερα αποθηκευτικά μέσα δεδομένων και μηχανές επεξεργασίας δεδομένων. Μπορεί επίσης να χρησιμοποιηθεί ως βάση για τη δημιουργία μιας βάσης δεδομένων, απλά προσθέτοντας δεδομένα και δείχνοντας στο Calcite πού θα βρει αυτή τη μορφή δεδομένων και πώς θα την χειριστεί. Για να προσθέσετε ένα αρχείο προέλευσης δεδομένων, θα πρέπει

να γράψετε ένα σωστό προσαρμογέα για να περιγράψει στο Calcite τι συλλογές από την πηγή δεδομένων θα πρέπει να θεωρήσει ως "πίνακες".

Επιπλέον, μπορούμε να προσθέσουμε προσαρμοσμένους στις ανάγκες μας κανόνες βελτιστοποίησης και να ορίσουμε τη δική μας άλγεβρα. Οι κανόνες βελτιστοποίησης χρησιμοποιούνται για τη βελτιστοποίηση του λογικού πλάνου, τη μετατροπή του σε φυσικούς τελεστές, την πρόσβαση σε δεδομένα σε νέα μορφή και τη δήλωση νέων προσαρμοσμένων τελεστών. Ο χρήστης μπορεί να συνδυάσει την προσαρμοσμένη λογική του με τους ενσωματωμένους κανόνες και τελεστές, να εφαρμόσει βελτιστοποίηση με βάση το κόστος (με βάση προσαρμοσμένο ή ενσωματωμένο μοντέλο κόστους), και να δημιουργήσει ένα αποτελεσματικό πλάνο. Οι κανόνες αυτοί λειτουργούν αναζητώντας μοτίβα στο δέντρο του ερωτήματος. Για παράδειγμα, όταν ο FilterProjectTransposeRule κανόνας βρίσκει ένα τελεστή Φίλτρου πάνω από ένα τελεστή Προβολής, θα τους αλλάξει τη σειρά. Ένα άλλο παράδειγμα της χρήσης κανόνων είναι όταν θέλουμε να ορίσουμε ένα προσαρμοσμένο πίνακα ή σχήμα, προκειμένου να καταστεί η πρόσβαση πιο αποτελεσματική.

"Το Calcite δεν πυροδοτεί κανόνες σε προκαθορισμένη σειρά. Η διαδικασία βελτιστοποίησης ερωτημάτων ακολουθεί πολλά κλαδιά ενός δέντρου διακλάδωσης, και ακριβώς όπως ένα πρόγραμμα σκακιού που παίζει, εξετάζει πολλές πιθανές ακολουθίες κινήσεων. Αν οι κανόνες Α και Β και αντιστοιχούν σε ένα δεδομένο τμήμα του δέντρου τελεστών του ερωτήματος, τότε το Calcite μπορεί να πυροδοτήσει και τους δύο. Επίσης, το Calcite χρησιμοποιεί το μοντέλο κόστους για την επιλογή μεταξύ των πλάνων, αλλά το μοντέλο κόστους δεν εμποδίζει τους κανόνες που μπορεί να φαίνεται ότι οδηγούν σε πιο ακριβά πλάνα σε σύντομο χρονικό διάστημα να πυροδοτηθούν"[1]

Υπάρχουν optimizers που χρησιμοποιούν ένα γραμμικό σύστημα βελτιστοποίησης. Αυτό σημαίνει ότι, όταν η βελτιστοποίηση έχει να κάνει μια επιλογή μεταξύ δύο κανόνων σε ένα ορισμένο χρονικό διάστημα, θα πρέπει να επιλέξει αμέσως ποιον κανόνα θα επιβάλει, ανάλογα με ορισμένη πολιτική. Παρ' όλα αυτά, *το Calcite* δεν αντιμετωπίζει αυτούς τους περιορισμούς, όταν προσπαθεί να συνδυάσει κανόνες.

Όπως έχει ήδη αναφερθεί, *το Calcite* επιλέγει το φθηνότερο πλάνο σύμφωνα με ένα μοντέλο κόστους. Αυτό το μοντέλο κόστους καθορίζει τους τύπους κόστους που

χρησιμοποιήθηκαν για να περιγράψουν τους τελεστές (στατιστικά) και το στόχο της βελτιστοποίησης. Με τη χρήση ενός καλά ορισμένου μοντέλου του κόστους μπορούμε να κλαδέψουμε το δέντρο αναζήτησης, για να αποτρέψουμε την δραματική αύξηση του χώρου αναζήτησης. Ωστόσο, η μέθοδος του κόστους δεν αναγκάζει τον optimizer να επιλέξει ανάμεσα σε δύο κανόνες και αποφεύγει να πέφτει σε τοπικά ελάχιστα στο χώρο αναζήτησης και σε αποτελέσματα που δεν είναι στην πραγματικότητα βέλτιστα.

Volcano Optimizer

Ο Volcano Optimizer [11, 12] είναι ένα top-down style πλαίσιο βελτιστοποίησης. Αυτή η οικογένεια βελτιστοποίησης διαθέτει μια από πάνω προς τα κάτω αναζήτηση στόχων με branch-and-bound pruning. Αυτή η ενότητα ασχολείται με κάποια χαρακτηριστικά σχεδιασμού υψηλού επιπέδου του Volcano που σχετίζονται με την προηγούμενη συζήτηση για τη βελτιστοποίηση ερωτημάτων.

1. Χώρος αναζήτησης: Η Top-down βελτιστοποίηση χρησιμοποιεί δύο τύπους των κανόνων: τους κανόνες μετασχηματισμού και τους κανόνες εφαρμογής [11,12]. Οι κανόνες μετασχηματισμού αντιστοιχίζουν μια αλγεβρική έκφραση σε μια άλλη. Οι κανόνες εφαρμογής αντιστοιχίζουν μια αλγεβρική έκφραση σε ένα δέντρο τελεστών. Ο Volcano Optimizer χρησιμοποιεί μόνο μία φάση βελτιστοποίησης, επειδή όλοι οι μετασχηματισμοί είναι αλγεβρικοί και με βάση το κόστος, η αντιστοίχηση από το αλγεβρικούς σε φυσικούς τελεστές συμβαίνει σε ένα μόνο βήμα και εκτελεί την εφαρμογή των κανόνων με γνώμονα τους στόχους. Στο Volcano, οι κανόνες μεταφράζονται ανεξάρτητα ο ένας από τον άλλο και συνδυάζονται μόνο με τη μηχανή αναζήτησης κατά τη βελτιστοποίηση ενός ερωτήματος. Οι κανόνες αυτοί αντιπροσωπεύουν τη γνώση του χώρου αναζήτησης στους top-down optimizers.
2. Εκτίμηση του κόστους: Η εκτίμηση του κόστους πρέπει να υπολογίζεται από κάτω προς τα πάνω. Αυτό συμβαίνει επειδή το σωρευτικό κόστος του κάθε ενδιάμεσου επίπεδου του πλάνου εξαρτάται από το κόστος και τα στατιστικά στοιχεία των προηγούμενων επιμέρους πλάνων.
3. Αλγόριθμος απαρίθμησης (enumeration)/ αλγόριθμος κλαδέματος: Κατά τη διάρκεια της φάσης βελτιστοποίησης, όταν η βελτιστοποίηση συναντά μια έκφραση ερωτήματος, ελέγχει αν έχει ήδη υπολογιστεί αναζητώντας σε ένα πίνακα αποθήκευσης πλάνων που έχουν βελτιστοποιηθεί στο παρελθόν. Εάν δεν βρεθεί,

εφαρμόζει ένα λογικό κανόνα μετασχηματισμού, έναν κανόνα εφαρμογής, ή χρησιμοποιεί έναν εφαρμοστή για να τροποποιήσει τις ιδιότητες της ροής δεδομένων. Τα πλάνα αποθηκεύονται σε ένα πίνακα κατακερματισμού των εκφράσεων και κλάσεων ισοδυναμίας, προκειμένου να εντοπίσει περιττά παράγωγα των ίδιων εκφράσεων και των πλάνων κατά τη διάρκεια της βελτιστοποίησης, για να μειώσει την προσπάθεια βελτιστοποίησης. Μια κλάση ισοδυναμίας αντιπροσωπεύει δύο συλλογές, μία με ισοδύναμες λογικές και μία με φυσικές εκφράσεις. Όπως προαναφέρθηκε, στη βελτιστοποίηση top-down χρησιμοποιείται branch-and-bounding για να κλαδέψουμε το χώρο αναζήτησης (directed dynamic programming). Εάν ένα υπο-πλάνο υπερβαίνει ένα ορισμένο όριο κόστους μπορεί να κλαδευτεί με ασφάλεια νωρίς στη διαδικασία βελτιστοποίησης και εξοικονομήσουμε χρόνο με το να μην διατρέξουμε τα υπόδενδρα του. Το κλάδεμα είναι ενσωματωμένο στην απαρίθμηση και ως εκ τούτου πρέπει να γίνει με μια συγκεκριμένη σειρά από πάνω προς τα κάτω.

Calcite Streaming

Το Calcite έχει επεκτείνει την SQL και τη σχεσιακή άλγεβρα, προκειμένου να υποστηρίξει τα ερωτήματα ροών δεδομένων [14]. Υπάρχουν περιπτώσεις όπου οι επιχειρήσεις ή οι χρήστες θέλουν να κάνουν ερωτήματα πάνω σε ροές όπως κάνουν με σχεσιακούς πίνακες. Η ανάγκη αυτή έρχεται με την απαίτηση να έχουν μερικά από τα πλεονεκτήματα των συνηθισμένων βάσεων δεδομένων: να χρησιμοποιούν μια γλώσσα υψηλού επιπέδου που βασίζεται (αν είναι δυνατόν) στη σχεσιακή SQL, που επικυρώνεται σύμφωνα με ένα σχήμα και μπορεί να βελτιστοποιηθεί για να επωφεληθούμε από τους πόρους του συστήματος μας και τους αλγόριθμους που χρησιμοποιούνται. Η SQL του Calcite είναι μια επέκταση της πρότυπης SQL και έχει τα ακόλουθα χαρακτηριστικά:

- Είναι εύκολο να τη μάθουμε, γιατί είναι μια επέκταση της πρότυπης SQL.
- Η σημασιολογία είναι σαφής, επειδή κληρονομείται από την αντίστοιχη της SQL. Τα αποτελέσματα των ροών πρέπει να είναι τα ίδια όπως αν είχαμε τα ίδια δεδομένα σε έναν πίνακα.
- Ο χρήστης έχει τη δυνατότητα να συνδυάσει τις ροές και τους πίνακες (ή την ιστορία μιας ροής, η οποίο είναι ουσιαστικά ένας πίνακα στη μνήμη) και να δημιουργήσει πιο πολύπλοκα ερωτήματα.

- Υπάρχουν ήδη πολλά υπάρχοντα εργαλεία που μπορούν να χρησιμοποιηθούν για τη δημιουργία SQL.

Εάν ο χρήστης δεν χρησιμοποιεί τη λέξη-κλειδί STREAM, τότε κάνει χρήση της κανονικής SQL. Η συνεχούς ροής SQL ακολουθεί την προσέγγιση που είδαμε στη CQL [3]. Αυτό σημαίνει ότι κάθε σχεσιακό ερώτημα μετατρέπεται σε μια ροή χρησιμοποιώντας τη λέξη-κλειδί STREAM στο κορυφαίο SELECT. Κατά τη διάρκεια της προετοιμασίας του ερωτήματος, το Calcite διακρίνει αν αναφερόμαστε σε μια ροή ή μια ιστορική σχέση. Σε ορισμένες περιπτώσεις, δεν υπάρχει διαθέσιμη η ιστορία μιας ροής (για παράδειγμα, τις τελευταίες 24 ώρες των δεδομένων σε ένα Apache Kafka topic [60]). Κατά το χρόνο εκτέλεσης, το Calcite υπολογίζει εάν υπάρχει επαρκής ιστορικό για να εκτελέσει ένα ερώτημα που απαιτεί τη χρήση μιας ιστορικής σχέσης.

Για τη σημασιολογία του Calcite streaming, υπάρχει μια δυαδικότητα μεταξύ ροής και βάσης δεδομένων [13]:

- *"Η βάση δεδομένων είναι μόνο μια cache της ροής"*
- *"Η ροή είναι απλά μια σύλληψη αλλαγής της βάσης δεδομένων"*

Ένας πίνακας μπορεί να χρησιμοποιηθεί ως μια ροή και μια ροή, όπως ένας πίνακας, έχοντας το σύστημα για να καθορίσει πού θα βρει τα αρχεία. Επιπλέον, οι ροές δεδομένων μπορούν να θεωρηθούν η παράγωγος των πινάκων, ενώ οι πίνακες μπορούν να θεωρηθούν αντίστοιχα τα ολοκληρώματα των ροών. Με τη λέξη-κλειδί STREAM αναφερόμαστε στα αρχεία από τώρα έως +∞. Χωρίς αυτό, αναφερόμαστε στα αρχεία από -∞ εώς τώρα, όπως θα κάναμε αν είχαμε έναν κανονικό πίνακα. Οι ροές συμπληρώνουν τους πίνακες και αντιπροσωπεύουν ό,τι συμβαίνει στο παρόν και το μέλλον των εγγραφών. Οι πίνακες αντιπροσωπεύουν το παρελθόν (ιστορικό) των αρχείων και είναι πολύ κοινό για μια ροή να αποθηκευτεί σε έναν πίνακα. Επομένως, ο χρήστης μπορεί να συνδυάσει το παρελθόν και το μέλλον σε σύνθετα ερωτήματα και να επιτύχει υψηλή εκφραστικότητα.

Η αρχή της επανάληψης ισχύει σε αυτήν τη σημασιολογία: «Ένα ερώτημα ροής παράγει το ίδιο αποτέλεσμα με το αντίστοιχο ερώτημα μη συνεχούς ροής εάν τα ίδια δεδομένα ήταν αποθηκευμένα σε έναν πίνακα" [13].

Ένα απλό παράδειγμα το ερώτημα που προβάλλει όλες τις στήλες του μιας ροής που ονομάζεται Orders:

```
SELECT STREAM *
FROM Orders;
```

Σημεία Στίξης (Punctuation)

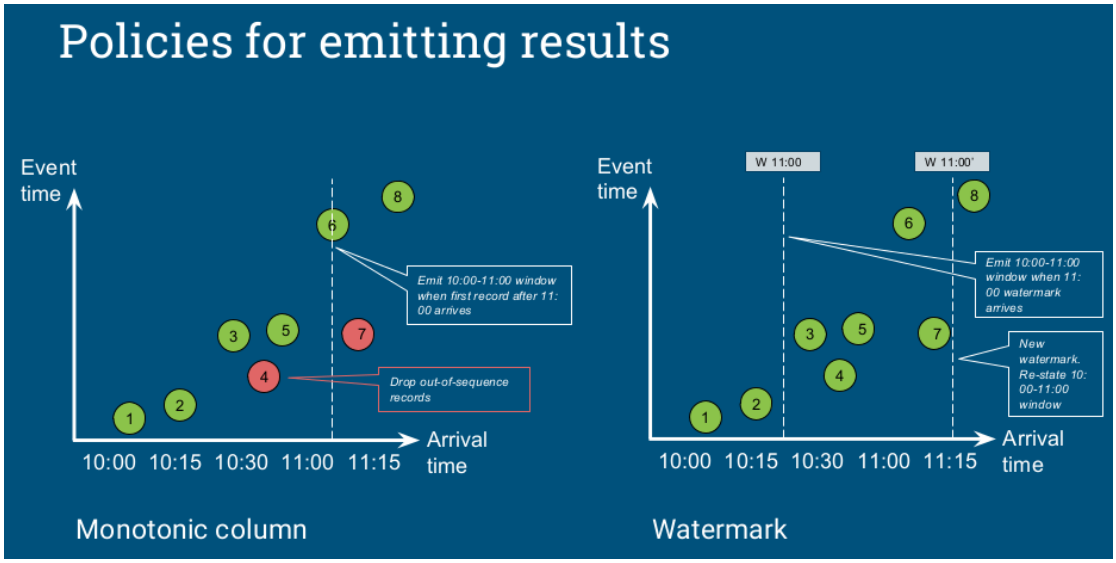
Μια άλλη βασική έννοια του Calcite streaming είναι να σημειώνει πρόοδο στους υπολογισμούς του. Σε εφαρμογές πραγματικού χρόνου δεν αρκεί μόνο να πάρουμε τα σωστά αποτελέσματα. Πρέπει να τα υπολογίζουμε και στη σωστή χρονική στιγμή, ώστε να έχουν αξία. Αυτοί είναι οι τρόποι για να επιτευχθεί πρόοδος χωρίς συμβιβασμούς στην ασφάλεια:

- Μονοτονικά αυξανόμενες στήλες (π.χ. rowtime) και εκφράσεις (π.χ. floor(rowtime to hour))
- Σημεία στίξης (π.χ. watermarks)
- Ή συνδυασμός και των δύο

Μπορούμε να χρησιμοποιήσουμε σημεία στίξης [15, 16], για να διασφαλιστεί ότι το ερώτημα μας θα σημειώσει πρόοδο ακόμη και αν δεν υπάρχουν αρκετές τιμές στο μονοτονικό κλειδί για να παραχθούν τα αποτελέσματα. Όταν μια ροή επιτρέπει σημεία στίξης, μπορεί να ταξινομηθεί ακόμα και αν είναι μη ταξινομημένη αρχικά. Αυτό μας βοηθά να χρησιμοποιούμε τη σημασιολογία μας, σαν να ήταν ταξινομημένη η ροή. Επίσης, μια μη ταξινομημένη ροή μπορεί επίσης να ταξινομηθεί εάν είναι t-ταξινομημένη ή k-ταξινομημένη και τα ερωτήματα μπορεί να προγραμματιστούν με παρόμοιο τρόπο όπως εάν είχαμε σημεία στίξης.

Μερικές φορές θέλουμε να συγκεντρώσουμε αποτελέσματα (aggregate) πάνω σε χαρακτηριστικά που δεν είναι με βάση το χρόνο, αλλά είναι παρ' όλα αυτά μονοτονικά. *"Ο αριθμός των φορών που μια ομάδα έχει μετατοπιστεί μεταξύ της κατάστασης νίκης και της κατάστασης ήττας"* θα μπορούσε να είναι μια μονότονη ιδιότητα. Σε αυτήν την περίπτωση, το σύστημα θα πρέπει να καταλάβει μόνο του ότι είναι ασφαλές να εκτελέσει υπολογισμούς πάνω από μια τέτοια ιδιότητα και τα σημεία στίξης δεν προσθέτουν καμία επιπλέον πληροφορία.

Ορισμένες μετρήσεις κόστους (μεταδεδομένα) πρέπει να υλοποιηθούν για τον planner, προκειμένου να υποστηρίξει τα σημεία στίξης, αφού μέχρι τώρα υποστηρίζει μόνο μονοτονικές στήλες με την κλάση BuiltInMetadata.Collation.



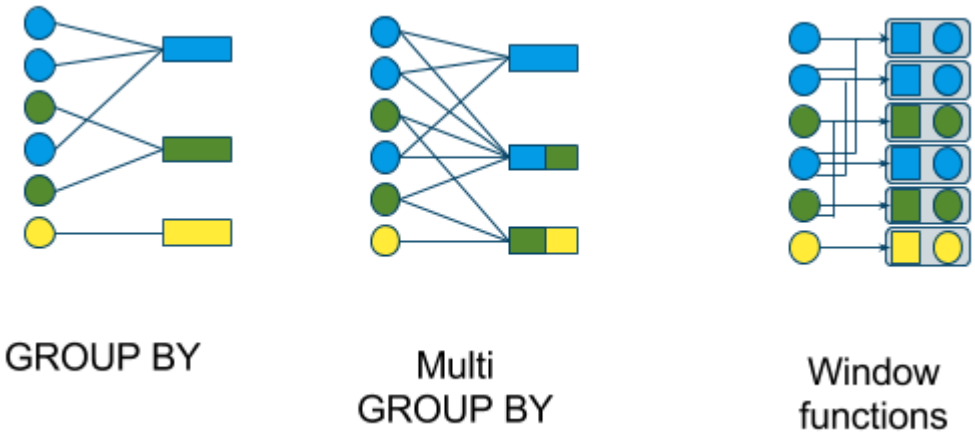
Εικ. 0.5.4.1, Παράδειγμα με watermarks [13].

Ορισμοί Παραθύρων

Στο Calcite, μπορούμε να υπολογίσουμε συναθροιστικές συναρτήσεις σε ροές με πολλούς διαφορετικούς τρόπους. Υπάρχουν διάφοροι τύποι παραθύρων, που ποικίλουν ανάλογα με [14]:

- *"Πόσες σειρές παράγονται για κάθε σειρά που εισέρχεται στη συνάρτηση;"*
- *"Κάθε εισερχόμενη τιμή εμφανίζεται σε ένα σύνολο ή σε περισσότερα;"*
- *"Αυτό που καθορίζει το παράθυρο, το σύνολο των σειρών που συμβάλλουν σε μια συγκεκριμένη σειρά που παράγεται."*
- *"Είναι το αποτέλεσμα μια ροή ή μια σχέση;"*

Το απλό **Group By** συγκεντρώνει πολλαπλές σειρές σε υποσύνολα, αφού κάθε σειρά συμβάλλει σε ακριβώς ένα υποσύνολο. Αντιθέτως, όταν έχουμε ένα **Multi-Group By** (π.χ. HOP ή Grouping Sets) μια σειρά μπορεί να συμβάλει σε περισσότερα από ένα υποσύνολα. Έχουμε επίσης συναρτήσεις παραθύρων, χρησιμοποιώντας το **OVER**, οι οποίες αφήνουν τον αριθμό των γραμμών αμετάβλητο και υπολογίζουν επιπλέον εκφράσεις για κάθε γραμμή.



Εικ. 0.5.4.2, Διαφορετικές Συναθροιστικές συναρτήσεις [14].

Τα παραπάνω παράθυρα μπορούν να κατηγοριοποιηθούν στους ακόλουθους τύπους παραθύρων:

- σταθερά παράθυρα (GROUP BY)
- hopping window (multi GROUP BY)
- συρόμενα παράθυρα (window functions)
- επικαλυπτόμενα παράθυρα (window functions)

Σταθερά Παράθυρα

"Τα σταθερά παράθυρα είναι μια σειρά σταθερού μεγέθους, μη-επικαλυπτόμενα και συνεχόμενα χρονικά διαστήματα, που εκφράζονται με το group by" [22]. Προκειμένου να επιτραπεί στο ερώτημα να εκτελεστεί στο Calcite, θα πρέπει να χρησιμοποιήσουμε την μονοτονική έκφραση στο GROUP BY (βλέπε σημεία στίξης παραπάνω). Εκτός από τη χρήση του group by, το Calcite εισήγαγε τις TUMBLE, TUMBLE_START και TUMBLE_END συναρτήσεις για να δώσει στο χρήστη τη δυνατότητα να ορίσει πιο πολύπλοκες εκφράσεις.

Hopping Windows

"Τα Hopping παράθυρα είναι μια γενίκευση των σταθερών παραθύρων που επιτρέπουν σε δεδομένα να διατηρούνται σε ένα παράθυρο για ένα μεγαλύτερο διάστημα από το εκπέμπουν διάστημα" [14]. Μπορούμε να ορίσουμε hopping παράθυρα, χρησιμοποιώντας τις συναρτήσεις HOP, HOP_START και HOP_END. Σε αντίθεση με τα σταθερά, είναι

επικαλυπτόμενα παράθυρα, στα οποία κάθε εγγραφή εισόδου συμβάλλει στον υπολογισμό τουλάχιστον μίας εγγραφή εξόδου.

Grouping Sets

Χρησιμοποιούμε Grouping Sets για τη δημιουργία μιας εξόδου ισοδύναμης με εκείνη που παράγεται από ένα UNION ALL πολλαπλών απλών GROUP BY. Για παράδειγμα τα ακόλουθα ερωτήματα περιέχουν ένα ενιαίο σύνολο ομαδοποίησης και δίνουν το ίδιο αποτέλεσμα [48]:

q1:
```
SELECT a, b, c, SUM(x)
FROM v53
GROUP BY GROUPING SETS ((a, b, c));
```

q2:
```
SELECT a, b, c, SUM(x)
FROM v53
GROUP BY a, b, c;
```

Αν και τα Grouping Sets μπορεί να εκφραστούν από τις συναρτήσεις ROLLUP και CUBE [47], οι πράξεις αυτές δεν ισχύουν για τα ερωτήματα συνεχούς ροής, καθώς θα δημιουργήσουν ένα σύνολο που συγκεντρώνει τα πάντα (GROUP BY ()) [14]. Από την άλλη πλευρά, Grouping Sets που περιέχουν μονοτονικές εκφράσεις μπορούν να χρησιμοποιηθούν σε υπολογισμό ροών δεδομένων.

Συρόμενα Παράθυρα

Τα συρόμενα παράθυρα προέρχονται από τα βασικά χαρακτηριστικά της SQL για "αναλυτικές συναρτήσεις" που μπορούν να χρησιμοποιηθούν στο SELECT. Για κάθε εγγραφή που έρχεται, μια εγγραφή βγαίνει, σε αντίθεση με το group by που οποία ομαδοποιεί τα στοιχεία εισόδου. Μερικά άλλα χαρακτηριστικά του συρόμενου παραθύρου είναι:

- Μπορούν να οριστούν παράθυρα πλειάδων, πέρα από χρόνου.
- Μπορούμε να αναφερθούμε σε γραμμές που δεν έχουν έρθει ακόμα.
- Συναρτήσεις που βασίζονται στην ταξινόμηση όπως RANK και median μπορούν να υπολογιστούν.

Επικαλυπτόμενα Παράθυρα

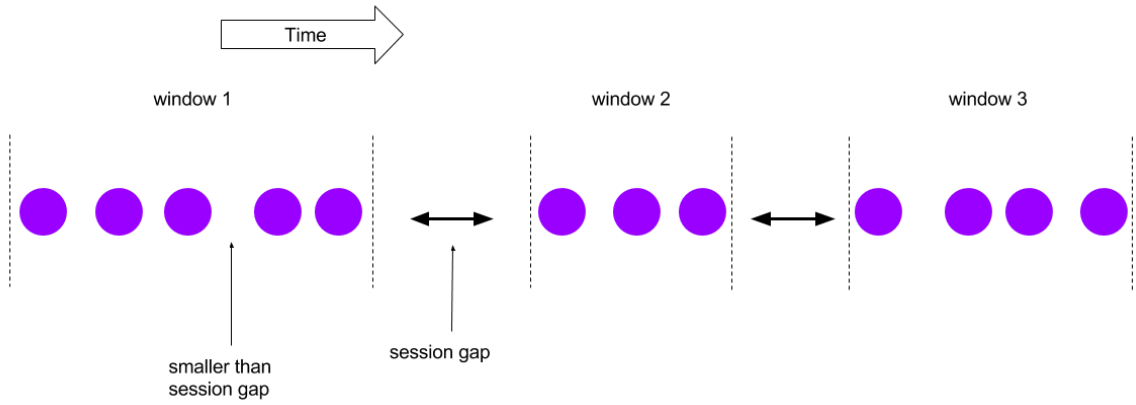
"Αν θέλουμε ένα ερώτημα που επιστρέφει ένα αποτέλεσμα για κάθε στοιχείο, όπως ένα συρόμενο παράθυρο, αλλά επαναφέρει τα σύνολα για ένα σταθερό χρονικό διάστημα, όπως ένα σταθερό παράθυρο" [14], μπορούμε να χρησιμοποιήσουμε ένα επικαλυπτόμενο παράθυρο. Για παράδειγμα:

```
SELECT STREAM rowtime,
productId,
units,
SUM(units) OVER (PARTITION BY FLOOR(rowtime TO HOUR)) AS
      unitsSinceTopOfHour
FROM Orders;
```

Η διαφορά με τον προηγούμενο ορισμό του συρόμενου παραθύρου είναι ότι η μονοτονική έκφραση βρίσκεται στο PARTITION BY του παραθύρου. Κάθε φορά που μια ώρα περνάει στο παράδειγμά μας, η τιμή του FLOOR(rowtime TO HOUR) αλλάζει, ξεκινώντας ένα νέο partition. Λαμβάνοντας υπόψιν ότι παλιά partition δεν θα χρησιμοποιηθούν σε επόμενους υπολογισμούς, όλα τα επιμέρους σύνολα μπορούν να απομακρυνθούν από την εσωτερική μνήμη του Calcite. Ως εκ τούτου, μπορούμε να υποθέσουμε ότι τα επικαλυπτόμενα παράθυρα είναι συρόμενα παράθυρα με ολίσθηση μεγαλύτερη από 1.

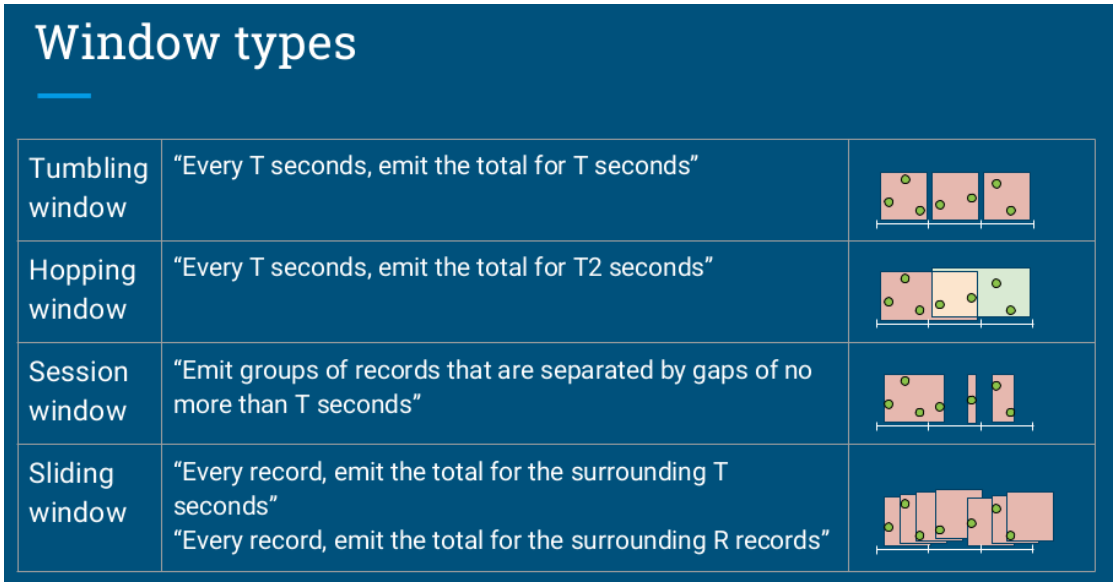
Παράθυρα Συνεδρίας

Εκπέμπουν ομάδες στοιχείων που χωρίζονται από κενά που δεν υπερβαίνουν Τ δευτερόλεπτα. Όταν χρησιμοποιούμε τα παράθυρα συνεδρίας, τα στοιχεία θα πρέπει να διαιρεθούν μεταξύ τους με βάση την χρονική σήμανση τους. Αν δύο στοιχεία έχουν χρονική απόσταση μικρότερη από ό,τι το κενό Τ δευτερολέπτων, θα είναι στην ίδια περίοδο. Στοιχεία με μεγαλύτερη απόσταση του χρόνου θα είναι σε διαφορετικές περιόδους, αν δεν υπάρχει στοιχείο που να "κλείσει" το κενό συνεδρίας μεταξύ τους [49]:

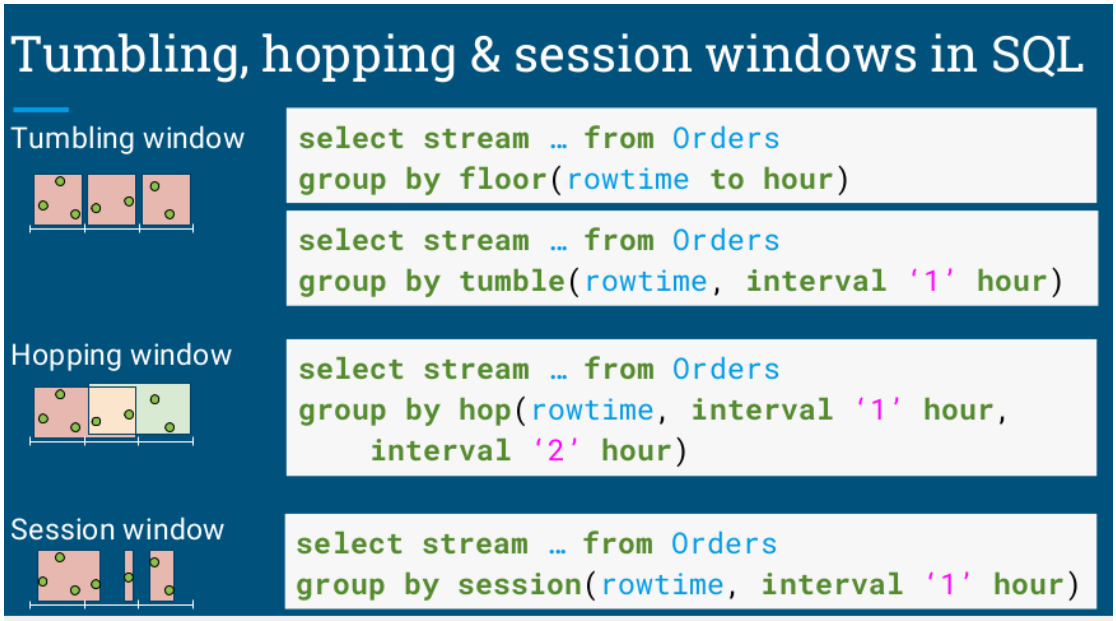


Εικ. 0.5.4.6.a, Παράθυρα Συνεδρίας [49].

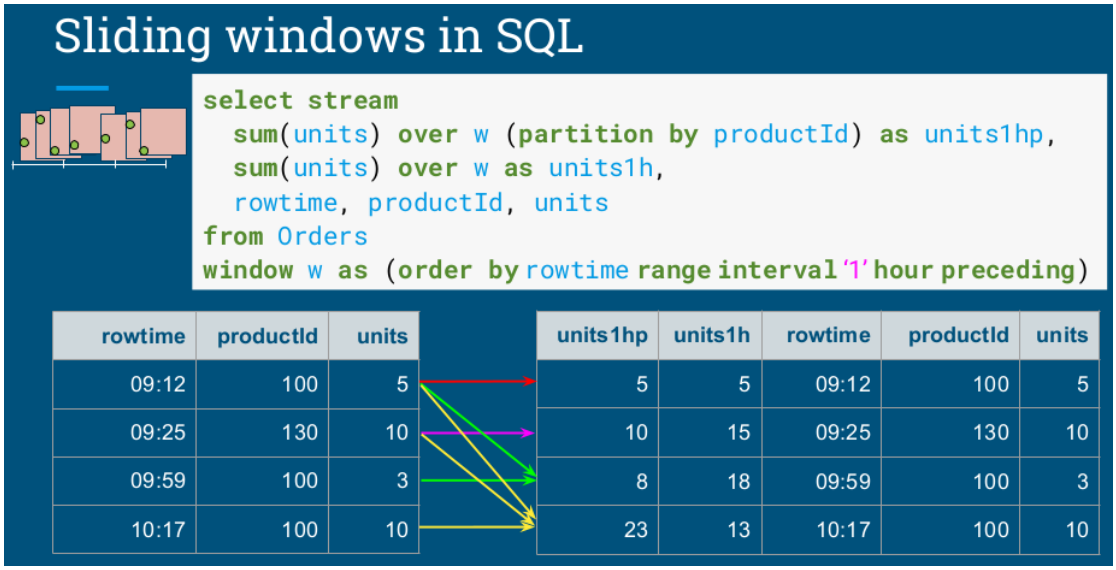
Μερικοί από τους παραπάνω ορισμούς μπορεί να διευκρινιστούν από τα επόμενα στοιχεία:



Εικ. 0.5.4.6.b, Τύποι παραθύρων [13].



Εικ. 0.5.4.6.c, Σταθερά, Hopping παράθυρα και παράθυρα συνεδρίας [13].

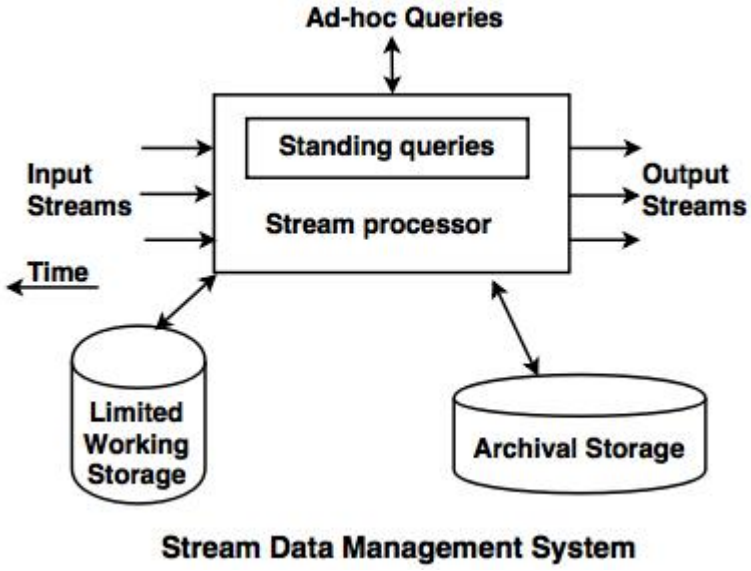


Εικ. 0.5.4.6.d, Συρόμενα παράθυρα [13].

0.5.5 Συστήματα Διαχείρισης Ροών Δεδομένων

Οι μηχανές επεξεργασίας ροών δεδομένων είναι σχεδιασμένες για να εκτελούν την επεξεργασία με SQL τρόπο για τα εισερχόμενα στοιχεία καθώς φτάνουν, χωρίς κατ 'ανάγκη την αποθήκευση τους. Σε περιπτώσεις όταν είναι απαραίτητη η αποθήκευση, συμβατικές βάσεις δεδομένων SQL, ενσωματωμένες στο σύστημα, μπορούν να χρησιμοποιηθούν για την αποδοτικότητα. Οι μηχανές αυτές υποστηρίζουν μεγάλου όγκου και χαμηλής

καθυστέρησης επεξεργασία των ροών δεδομένων που φτάνουν. Πολλά τέτοια συστήματα έχουν προκύψει τις τελευταίες δεκαετίες, τα οποία εξυπηρετούν διαφορετικούς τύπους εφαρμογών και εισάγουν νέα χαρακτηριστικά στην επεξεργασία ροής.



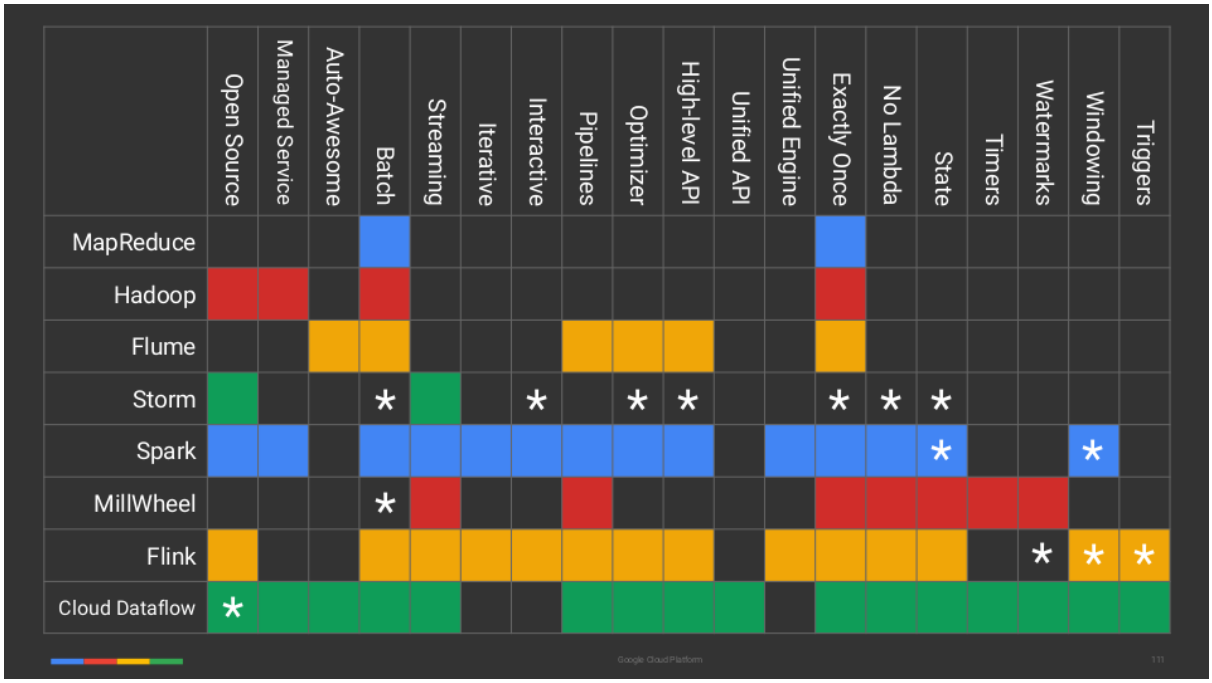
Εικ. 0.5.5.a, Σύστημα Διαχείρισης Ροών Δεδομένων.

Συστήματα όπως το Niagara [33] (μια μηχανή επεξεργασίας XML), το Borealis [29] (ένα κατανεμημένο σύστημα επεξεργασίας ροής που κληρονομεί τη λειτουργικότητα επεξεργασίας του από το Aurora και την κατανεμημένη λειτουργικότητα από το Medusa), STREAM [30] (γενικής χρήσης πρωτότυπο σύστημα διαχείρισης δεδομένων ροής) και TelegraphCQ [31] (ένα σύστημα για την εφαρμογή συνεχώς προσαρμοστικής επεξεργασίας ερωτημάτων με βάση το Eddy) έθεσαν τα θεμέλια για τις σύγχρονες πλατφόρμες επεξεργασίας ροής δεδομένων. Κάποια άλλα γνωστά συστήματα, που έχουν υιοθετηθεί ευρέως και χρησιμοποιούνται σε εφαρμογές σήμερα είναι:

- Apache Storm [40]: Το Storm είναι ένα ανοικτό πλαίσιο κώδικα που παρέχει μεγάλης κλίμακας συλλογή γεγονότων, που δημιουργήθηκε από το Twitter.
- Apache Spark [41]: Ένα γενικό πλαίσιο για την επεξεργασία δεδομένων μεγάλης κλίμακας που υποστηρίζει πολλές διαφορετικές γλώσσες προγραμματισμού και έννοιες όπως την επεξεργασία ροών.
- Apache Flink [42]: Ένα ανοικτό πλαίσιο κώδικα για κατανεμημένα analytics μεγάλων δεδομένων.
- Apache Samza [43]: Ένα κατανεμημένο πλαίσιο επεξεργασίας ροής, που πρόσφατα έγινε ανοιχτού κώδικα από το LinkedIn.

- Esper [44]: Esper είναι ένα ανοιχτού κώδικα προϊόν βασισμένο σε Java για thn επεξεργασία περίπλοκων γεγονότων (CEP [55]) και την επεξεργασία ροών γεγονότων (ESP), που αναλύει σειρά γεγονότων για την εξαγωγή συμπερασμάτων από αυτά.

Ωστόσο, τα περισσότερα από αυτά τα συστήματα αντιμετωπίζουν δυσκολίες στην εκπλήρωση των απαιτήσεων που πρέπει να πληρεί μια μηχανή επεξεργασίας ροής [32]. Τα συστήματα που βασίζονται στην επεξεργασία με δέσμες, όπως τα πλαίσια MapReduce, FlumeJava [45] και Spark [41], πάσχουν από προβλήματα latency. Πολλά συστήματα δεν έχουν ανοχή σε σφάλματα και δεν είναι κλιμακώσιμα με εισόδους μεγάλου μεγέθους (π.χ. Aurora [29], TelegraphCQ [31], Niagara [33], Esper [44]). Κάποια άλλα συστήματα, παρέχουν ανοχή σε σφάλματα και κλιμάκωση με κόστος στην εκφραστικότητα ή στην ορθότητα. Υπάρχουν επίσης περιπτώσεις, όπου τα συστήματα δεν έχουν την ικανότητα να παρέχουν σημασιολογία για ακριβώς μία φορά επεξεργασία, που επηρεάζει την ορθότητα της εξόδου (Storm [40], Samza [43]). Άλλα παρέχουν περιορισμένη σημασιολογία παραθύρου (μόνο παράθυρα με πλειάδες ή παράθυρα χρόνου), όπως το Spark Streaming [41] ή το Trident [40]. Τα περισσότερα από τα υπάρχοντα συστήματα συνεχούς ροής δεν υποστηρίζουν το συνδυασμό παραθύρων με βάση το χρόνο εκδήλωσης και το το χρόνο επεξεργασίας στη σημασιολογία τους. *"Το MillWheel και το Spark Streaming δεν διαθέτουν τα μοντέλα προγραμματισμού υψηλού επιπέδου που κάνουν τον υπολογισμό συνεδρίων με χρόνο εκδήλωσης απλό"* [32]. Επίσης, τα συστήματα Lambda Αρχιτεκτονικής μπορούν να επιτύχουν πολλές από τις επιθυμητές απαιτήσεις, αλλά αντιμετωπίζουν αρκετούς περιορισμούς, και ίσως θα πρέπει να αντικατασταθούν από συστήματα Kappa Αρχιτεκτονικής [34].



Εικ. 0.5.5.b, Σύγκριση διαφορετικών συστημάτων ροών δεδομένων [69].

Ένα άλλο πολύ σημαντικό μειονέκτημα, που το μεγαλύτερο μέρος των παραπάνω συστημάτων συνεχούς ροής έχει, είναι η έλλειψη ή η περιορισμένη υποστήριξη για μια SQL δηλωτική γλώσσα επερώτησης (π.χ. Storm, Spark, Samza). Ως αποτέλεσμα, είναι αναγκαία η καλή γνώση προστακτικού προγραμματισμού και κατανεμημένων συστημάτων για να χρησιμοποιηθούν αυτά τα συστήματα αποτελεσματικά. Με τη χρήση μιας SQL γλώσσας, που επεκτείνει την παραδοσιακή SQL για να εξυπηρετήσει τη σημασιολογία ροής, παρέχουμε στο σύστημα ένα δομημένο τρόπο έκφρασης ερωτημάτων και επιτρέπουμε τη βελτιστοποίηση τους με γνωστές τεχνικές βάσεων δεδομένων που χρησιμοποιούνται στα παραδοσιακά συστήματα. Η βελτιστοποίηση δεν περιορίζεται μόνο σε γνωστούς κανόνες και αλγορίθμους των σχεσιακών βάσεων δεδομένων, αλλά μπορεί να επεκταθεί για να καλύψει τα ζητήματα ροής δεδομένων. Αυτή είναι η λειτουργικότητα που προσπαθούμε να εισάγουμε στη μηχανή επεξεργασίας ροής που έχουμε επιλέξει, το SABER [2].

SABER

Στην ενότητα αυτή, θα συζητήσουμε κάποιες έννοιες υψηλού επιπέδου για το SABER [2]. Προηγούμενες ερευνητικές προσπάθειες για μηχανές επεξεργασίας ροής δεδομένων, προσπαθούσαν να εκμεταλλευτούν είτε τον παραλληλισμό εργασιών είτε τον

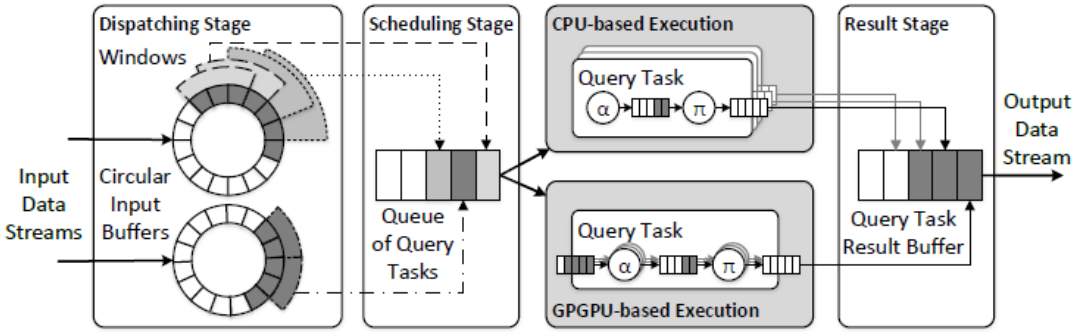
παραλληλισμό δεδομένων στα κατανεμημένα ή κεντρικά περιβάλλοντα, χωρίς να εξεταστεί η δυνατότητα ετερογενών αρχιτεκτονικών. Ωστόσο, στις μέρες μας οι servers με ετερογενείς αρχιτεκτονικές είναι διαθέσιμοι σε κέντρα δεδομένων και υπηρεσίες cloud, εισάγοντας μια νέα πηγή παραλληλισμού για την επεξεργασία ροής δεδομένων. Οι GPGPUs προσφέρουν την ευκαιρία της εκτέλεσης ορισμένων τύπων πολύπλοκων υπολογισμών σε υψηλούς βαθμούς παραλληλισμού, με αποτέλεσμα τη μεγιστοποίηση της απόδοσης συνολικού συστήματος. Το SABER είναι μια υβριδική σχεσιακή μηχανή επεξεργασίας ροών, που εκτελεί ερωτήματα SQL σε τέτοια ετερογενή συστήματα, με έναν παράλληλο για τα δεδομένα τρόπο. Χρησιμοποιεί όλους τους διαθέσιμους CPU και GPGPU πυρήνες για την επίτευξη υψηλής απόδοσης επεξεργασίας.

Οι βασικές τεχνικές συνεισφορές του SABER είναι:

- Το SABER χρησιμοποιεί ένα υβριδικό μοντέλο επεξεργασίας ροής, στο οποίο οι εργασίες τρέχουν παράλληλα σε CPU και GPGPU και παραλληλοποιούνται περαιτέρω σε όλους τους πυρήνες της GPGPU. Ο χρονοδρομολογητής χρησιμοποιεί έναν heterogeneous lookahead scheduling (HLS) αλγόριθμο, για να εκχωρήσει κάθε εργασία ερωτήματος στον κατάλληλο τύπο επεξεργαστή. Αυτός ο αλγόριθμος αντιστοιχίζει ένα συγκεκριμένο ερώτημα σε έναν ετερογενή επεξεργαστή με βάση την προηγούμενη συμπεριφορά (υψηλότερη απόδοση) και επιλύει πιθανές συγκρούσεις, προκειμένου να βελτιστοποιηθεί η αξιοποίηση των πόρων του συστήματος και να αυξηθεί η απόδοση.
- Υποστηρίζει σημασιολογία συρόμενων παραθύρων και διατηρεί υψηλή απόδοση για μικρά μεγέθη παραθύρου και ολίσθησης. Σε αντίθεση με άλλες μηχανές, το SABER αποσυνδέει τη σημασιολογία παραθύρου από την απόδοση του συστήματος. Αυτό έχει επίδραση στην απόδοση, ειδικά όταν η ολίσθηση παραθύρου είναι μικρή.
- Υποστηρίζει αυξανόμενη επεξεργασία (incremental processing) κατά την επεξεργασία ενός ερωτήματος με ένα συρόμενο παράθυρο, χρησιμοποιώντας τα αποτελέσματα που υπολογίστηκαν για προηγούμενα τμήματα παραθύρου στην ίδια δέσμη.
- Προκειμένου να αποφευχθούν οι καθυστερήσεις που προκαλούνται από την κίνηση των δεδομένων, το SABER εισάγει έναν μηχανισμό σωλήνωσης με πέντε στάδια για να “κρύψει” αυτό το κόστος στη GPGPU.
- Η διαχείριση μνήμης του SABER ελαχιστοποιεί την εκχώρηση μνήμης με:

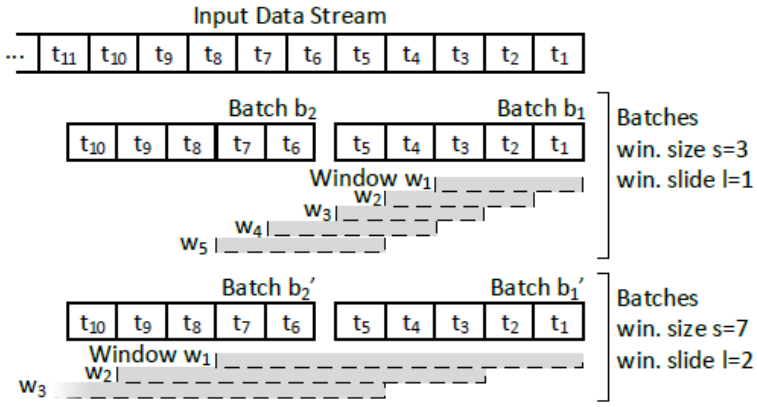
- Lazy deserialization: Οι πλειάδες αποσειριοποιούνται μόνο όταν χρειάζεται και αποθηκεύονται σε αναπαράσταση byte. Επίσης, η αποσειριοποίηση παράγει μόνο πρωτόγονους τύπους.
- Object pooling: το SABER χρησιμοποιεί στατικά κατανεμημένα σύνολα αντικειμένων (ένα ανά νήμα) για όλα τις εργασίες, και πίνακες byte, για την αποθήκευση των ενδιάμεσων αποτελεσμάτων των τμημάτων παραθύρων, προκειμένου να αποφευχθεί η δυναμική κατανομή μνήμης.

- Η αρχιτεκτονική του SABER αποτελείται από τέσσερα στάδια επεξεργασίας:
  - Στάδιο Αποστολής: δημιουργεί σταθερό μέγεθος εργασιών για τα ερωτήματα, οι οποίες μπορούν να εκτελεστούν τόσο σε CPUs όσο και GPUs, και τα τοποθετεί σε μια ουρά για ολόκληρο το σύστημα.
  - Στάδιο Προγραμματισμού (scheduling): αποφασίζει ποια εργασία θα εκτελεστεί επόμενη από κάθε επεξεργαστή χρησιμοποιώντας τον HLS.
  - Στάδιο Εκτέλεσης: τρέχει μια εργασία, είτε σε πυρήνες CPU ή GPGPU, και αξιολογεί το αποτέλεσμα με εφαρμογή της συνάρτησης του τελεστή επί των τμημάτων του παράθυρου εισαγωγής.
  - Στάδιο Αποτελεσμάτων: αναδιατάσσει τα αποτελέσματα των εργασιών και συναρμολογεί τα αποτελέσματα παραθύρου με αποτελέσματα από το προηγούμενο στάδιο.



Εικ. 0.5.5.1.a, Επισκόπηση της αρχιτεκτονικής του SABER [2].

Χρησιμοποιήσαμε το SABER ως μηχανή επεξεργασίας ροής για το σύστημα μας, επειδή είναι κεντρική (η οποία βοηθά στη διεξαγωγή πειραμάτων σε ένα μοναδικό μηχάνημα) και υποστηρίζει την σημασιολογία ολίσθησης παραθύρου, αλλά δεν έχει SQL υποστήριξη και βελτιστοποίηση ερωτημάτων.



Εικ. 0.5.5.1.b , Δέσμες ροών δεδομένων με δύο διαφορετικούς τύπους παραθύρων [2].

0.6 Κανόνες Βελτιστοποίησης στο Calcite

Πως χρησιμοποιούμε τον RelOptPlanner

Αυτά είναι τα βήματα που πρέπει να ακολουθηθούν για την βελτιστοποίηση μιας σχεσιακής έκφρασης R :

- Επιλέγουμε την επιθυμητή υλοποίηση του RelOptPlanner. Μπορούμε να επιλέξουμε ανάμεσα στον HepPlanner (ευριστικός βελτιστοποιητής παρόμοιος με τον Catalyst του Spark [12]) και στον VolcanoPlanner (βελτιστοποιητής δυναμικού προγραμματισμού) ή σε ένα συνδυασμό αυτών.
    - Καταχωρούμε την σχεσιακή έκφραση R στον planner.
    - Δημιουργούμε τις κλήσεις για όλους τους κανόνες που θέλουμε να εφαρμόσουμε.
    - Κατατάσσουμε τις κλήσεις των κανόνων ανάλογα με τη σημασία τους. Η σημασία του κανόνα καθορίζεται από την πιθανότητα του να παράγει την καλύτερη υλοποίηση μιας σχεσιακής έκφρασης στην επιλογή του σχεδίου. Ως αποτέλεσμα, παράγει ένα μέλος από μια σημαντική σχέση με φθηνά παιδιά.
        - Καλούμε τον πιο σημαντικό κανόνα.
        - Επαναλαμβάνουμε.

Ένας κανόνας μπορεί να πυροδοτηθεί από όποιους τελεστές ταιριάζουν σε αυτόν στην σχεσιακή μας έκφραση. Για παράδειγμα έχουμε του εξής κανόνες (το παράδειγμα πάρθηκε από [18]):

1. Ο PushFilterThroughProjectRule, που εφαρμόζεται στους τελεστές (Filter (Project(...)))
2. Ο CombineProjectsRule, που εφαρμόζεται στους τελεστές (Project (Project(...)))

Ας θεωρήσουμε το επόμενο πλάνο:

```
Project (deptno)                          [exp 1, subset A]
  Filter (gender='F')                     [exp 2, subset B]
    Project (deptno, gender, empno)       [exp 3, subset C]
      Project (deptno, gender, empno, salary)  [exp 4, subset D]
        TableScan (emp)                   [exp 0, subset X]
```

Όταν εφαρμόζουμε τον κανόνα PushFilterThroughProjectRule στις εκφράσεις 2 και 3 παίρνουμε το ακόλουθο πλάνο:

```
Project (deptno)                          [exp 1, subset A]
  Project (deptno, gender, empno)         [exp 5, subset B]
    Filter (gender='F')                   [exp 6, subset E]
      Project (deptno, gender, empno, salary)  [exp 4, subset D]
        TableScan (emp)                   [exp 0, subset X]
```

Παρατηρούμε ότι δημιουργήθηκαν δύο νέες σχέσεις από την εφαρμογή αυτού του κανόνα. Η έκφραση 5 ανήκει στο υποσύνολο B (επειδή είναι αντίστοιχη της έκφρασης 2) και η έκφραση 6, αντίστοιχα, στο υποσύνολο E. Στο νέο αυτό παραγόμενο πλάνο, παρατηρούμε ότι μπορούμε να εφαρμόσουμε νέους κανόνες που δεν μπορούσαμε πριν (CombineProjectsRule), λόγω της μετατόπισης των αρχικών κανόνων. Επομένως, δύο ακόμα κανόνες πυροδοτούνται:

Η έκφραση 5 πυροδοτεί τον κανόνα CombineProjectsRule στις εκφράσεις 1 και 5, που πυροδοτεί το επόμενο πλάνο:

```
Project (deptno)                          [exp 7, subset A]
  Filter (gender='F')                     [exp 6, subset E]
```

```
Project (deptno, gender, empno, salary)      [exp 4, subset D]
  TableScan (emp)                            [exp 0, subset X]
```

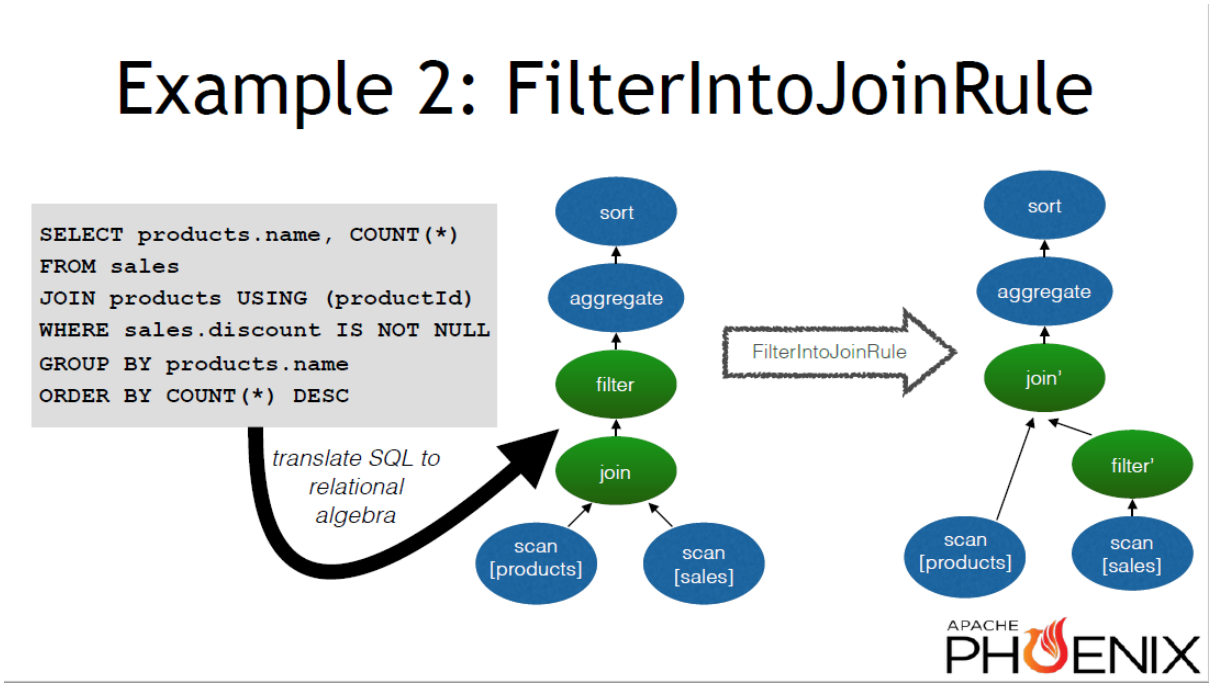
Η νέα καταχωρημένη έκφραση 6 πυροδοτεί τον κανόνα PushFilterThroughProjectRule στις εκφράσεις 6 και 4, που δημιουργεί το επόμενο πλάνο:

```
Project (deptno)                             [exp 1, subset A]
  Project (deptno, gender, empno)            [exp 5, subset B]
    Project (deptno, gender, empno, salary)  [exp 8, subset E]
      Filter (gender='F')                    [exp 9, subset F]
        TableScan (emp)                      [exp 0, subset X]
```

Τελικά, ο κανόνας CombineProjectsRule στις έκφρασις 7 και 8 μειώνει ακόμα περισσότερο το βάθος του δέντρου στο τελικό μας πλάνο:

```
Project (deptno)                             [exp 10, subset A]
  Filter (gender='F')                        [exp 9, subset F]
    TableScan (emp)                          [exp 0, subset X]
```

Αντιστοιχίσεις κανόνων μπορούν να προκληθούν από ένα παιδί υποσύνολο για οποιοδήποτε από τους γονείς του. Κατά τη διαδικασία της εφαρμογής των κανόνων, νέες relexps θα μπορούσαν να καταχωρηθούν, οι οποίες μπορούν να προκαλέσουν διάφορους κανόνες, συμπεριλαμβανομένου του κανόνα που τις δημιούργησε. Επίσης, κατά την επιβολή των κανόνων, είναι δυνατόν να συγχωνευθούν υποσύνολα.



Εικ. 0.6.1, Η εφαρμογή του FilterJoinRule [53].

Προσαρμοσμένοι Κανόνες

Στο RBStream, χωρίσαμε τη διαδικασία βελτιστοποίησης σε ξεχωριστές φάσεις, για να γίνει πιο αποτελεσματική και πιο εύκολα διαχειρίσιμη (θα συζητηθεί με περισσότερες λεπτομέρειες στην ενότητα της υλοποίησης). Αυτές οι φάσεις χρησιμοποιούν είτε τον VolcanoPlanner ή τον HepPlanner. Ο VolcanoPlanner χρησιμοποιείται σε μία ενδιάμεση φάση βελτιστοποίησης, με ένα υποσύνολο των κανόνων, για να ψάξουμε δυναμικά σε ένα ορισμένο χώρο αναζήτησης και να βρούμε τη βέλτιστη λύση. Επίσης, μόνο με τον VolcanoPlanner μπορούμε να χρησιμοποιήσουμε τους κανόνες εφαρμογής, που μετατρέπουν τους λογικούς τελεστές μας σε φυσικούς τελεστές με συγκεκριμένη σύμβαση (convention) και χαρακτηριστικά (π.χ. enumerable).

Επειδή όμως η φάση βελτιστοποίησης με τον VolcanoPlanner ακολουθείται από φάσεις που εφαρμόζουμε ευριστικούς κανόνες μετασχηματισμού και οι περισσότεροι από αυτούς ισχύουν μόνο για τους απλούς λογικούς τελεστές (δεν μπορούν να χρησιμοποιηθούν σε φυσικούς τελεστές), θα πρέπει να εφαρμόζονται προσαρμοσμένοι κανόνες μετασχηματισμού, που δεν επηρεάζουν τη σύμβαση και τα χαρακτηριστικά των τελεστών και μετατρέπουν τους φυσικούς τελεστές σε λογικούς πάλι, κρατώντας ανέπαφα τα χαρακτηριστικά τους. Με αυτή την απλή μετατροπή, είμαστε σε θέση να

επαναχρησιμοποιήσουμε τους ενσωματωμένους κανόνες του Calcite μας σε μεταγενέστερες φάσεις, χωρίς να χρειάζεται να τους ξαναγράψουμε από την αρχή.

Δημιουργήσαμε επίσης κανόνες εφαρμογής, προκειμένου να εισάγουμε τους δικούς μας τελεστές, που ακολουθούν το μοντέλο κόστους με βάση το ρυθμό εισροής των δεδομένων που χρησιμοποιήθηκε στο RBStream. Τέλος, δημιουργήσαμε κάποιους κανόνες μετασχηματισμού που χρησιμοποιούνται είτε για τη βελτιστοποίηση, σε ορισμένες περιπτώσεις, είτε για την παρασκευή του τελικού λογικού σχεδίου μας, που θα είναι κατάλληλο για τον μετασχηματισμό του στο αντίστοιχο φυσικό σχέδιο του SABER.

Κανόνες εφαρμογής SaberRel

Υπάρχουν 6 κανόνες εφαρμογής, ένας για κάθε τελεστή του συστήματος μας. Αυτοί οι κανόνες μετατρέπουν τους ενσωματωμένους λογικούς τελεστές του Calcite σε SaberRel τελεστές, που προσθέτουν την λογική που θέλουμε για τον υπολογισμό του κόστους των πλάνων.

Νέοι Κανόνες Μετασχηματισμού

FilterPushThroughFilterRule

Αυτός ο κανόνας χρησιμοποιείται για να μεταθέσει δύο συνεχόμενους τελεστές φίλτρου, ανάλογα με την επιλεκτικότητα (selectivity) τους. Το φίλτρο με την μικρότερη επιλεκτικότητα μεταφέρεται χαμηλότερα, για να μειώσουμε το μέγεθος των δεδομένων που προωθείται στα επόμενα στάδια της εκτέλεσης του ερωτήματος. Αυτός ο κανόνας χρησιμοποιείται αντί για τον ενσωματωμένο FilterMergeRule κανόνα και βελτιώνει το ρυθμό παραγωγής αποτελεσμάτων κατά πολύ σε ορισμένες περιπτώσεις.

Στην υλοποίηση μας, ο κανόνας FilterPushThroughFilter μας βοήθησε να δοκιμάσουμε την τρίτη φάση της διαδικασίας βελτιστοποίησης που δημιουργήσαμε Ο παραπάνω κανόνας απαιτεί τη χρήση του VolcanoPlanner (δυναμικός προγραμματισμός) και το σωστό ορισμό του μοντέλου κόστους με βάση το ρυθμό εισροής δεδομένων, προκειμένου να επιβληθεί σωστά.

JoinTableScanSupportRule

Ο κανόνας JoinTableScanSupportRule προσθέτει έναν Project τελεστή πάνω από ένα TableScan που είναι είσοδος σε ένα Join. Αυτός ο κανόνας μας βοηθάει να εκτελέσουμε τελικά το πλάνο μας στο SABER.

0.7 Μοντέλο Κόστους

Σκοπός

Ένα καλά ορισμένο μοντέλο κόστους παίζει πολύ σημαντικό ρόλο σε ένα σύστημα διαχείρισης ροών δεδομένων γιατί:

- Χρησιμοποιούμε το μοντέλο κόστους για την εκτίμηση των επιπτώσεων των βελτιστοποιήσεων σε ένα σχέδιο ερωτήματος (π.χ. ρυθμός αποτελεσμάτων, χρήση πόρων). Ως εκ τούτου, είναι απαραίτητη προϋπόθεση για τη βελτιστοποίηση ερωτημάτων με βάση το κόστος (και τη προσαρμοστική διαχείριση των πόρων).
- Οι μετρήσεις χρόνου εκτέλεσης δίνουν πληροφορίες σχετικά με την παρόν και το παρελθόν, ενώ ένα μοντέλο κόστους καθιστά δυνατή την εκτίμηση των παραμέτρων του συστήματος στο μέλλον, αν είναι ακριβείς.
- Μπορούμε να μειώσουμε το υπολογιστικό κόστος χρησιμοποιώντας ένα ακριβές μοντέλο κόστους, επειδή η παρακολούθηση παραμέτρων του συστήματος κατά το χρόνο εκτέλεσης μπορεί να είναι ακριβή.

Παράμετροι του μοντέλου κόστους

Το μοντέλο του κόστους μας βασίζεται σε ένα μοντέλο με βάση το ρυθμό εισροής και εκροής των δεδομένων [37, 26], με συγκεκριμένες επεκτάσεις, ώστε να ταιριάζει με τη σημασιολογία μας. Αυτό το μοντέλο κόστους χρησιμοποιείται για την στατική βελτιστοποίηση με σταθερούς ρυθμούς εισόδου, αλλά μπορεί να επεκταθεί για να καλύψει τις ανάγκες ενός προσαρμοστικού συστήματος διαχείρισης πόρων. Υποθέτουμε ότι οι υπολογιστικοί πόροι του συστήματος μας είναι επαρκείς (εφικτά ερωτήματα) και δεν χρησιμοποιούμε την τεχνική αποβολής φορτίου του αρχικού μοντέλου.

Η σημασιολογία των συρόμενων παραθύρων στο μοντέλο μας

Στο μοντέλο μας κάνουμε κάποιες παραδοχές για την απλοποίηση του:

- Οι πλειάδες που φτάνουν στη ροή είναι σε μονοτονικά αύξουσα σειρά και δεν υπάρχει εκτός σειράς άφιξη (για αυτό έχει ληφθεί μέριμνα από το σύστημα).
- Δεν χρησιμοποιούμε σχεσιακούς πίνακες στα ερωτήματα.
- Οι πλειάδες έρχονται με σταθερό ρυθμό.
- Οι υπολογιστικοί πόροι του συστήματός μας είναι επαρκείς.

Έχουμε τους εξής τελεστές:

- Τελεστής Επιλογής: Αυτός ο τελεστής λαμβάνει ως είσοδο ένα ρεύμα S και δίνει ως αποτέλεσμα ένα άλλο ρεύμα O, του οποίου τα στοιχεία είναι ένα υποσύνολο του S που ικανοποιούν τα κατηγορήματα της επιλογής. Μπορεί να είναι είτε μια προβολή (τελεστής επιλογής με επιλεκτικότητα ίση με 1) ή ένα φίλτρο.
- Τελεστής Συνένωσης: Αυτός ο τελεστής αντιπροσωπεύει την συνένωση με συρόμενα παράθυρα πηγών συνεχούς ροής. Είναι ένας συμμετρικός τελεστής που λαμβάνει δυο ρεύματα εισόδου L και R, και για κάθε πλειάδα που έρχεται σε οποιαδήποτε από αυτές τις ροές ενώνει την πλειάδα αυτή με τα περιεχόμενα του τρέχοντος παραθύρου της άλλης ροής εισόδου.
- Τελεστές Συναθροιστικών Συναρτήσεων/Παραθύρων: Στην υλοποίηση μας, τόσο οι τελεστές Συναθροιστικών Συναρτήσεων όσο και Παραθύρων μας δίνουν ως έξοδο το ίδιο αποτέλεσμα (και τα δύο μεταφράζονται σε Group By ή MultiGroup By συναρτήσεις σε αντίθεση με τους ορισμούς που δώσαμε στην ενότητα που αφορούσε το Calcite Streaming). Παίρνουν ως είσοδο ένα ρεύμα S, υπολογίζουν τις συναθροιστικές συναρτήσεις που ορίζονται από τον τελεστή, και παράγουν ένα άλλο ρεύμα O με αυτούς τους υπολογισμούς.

Οι παράμετροι παραθύρων και ρυθμού παραγωγής αποτελεσμάτων

Οι βασικές μας παράμετροι είναι ο ρυθμός παραγωγής αποτελεσμάτων και το μέγεθος του παραθύρου. Με αυτές τις παραμέτρους, υπολογίζουμε τις απαιτήσεις των πόρων για έναν τελεστή. Κάθε τελεστής λαμβάνει ως είσοδο ρυθμούς άφιξης δεδομένων και το μέγεθος ενεργών παραθύρων και εξάγει ένα ρυθμό αποτελεσμάτων και ένα νέο μέγεθος ενεργού παραθύρου, ανάλογα με τον τύπο του τελεστή. Ενεργό παράθυρο είναι *"ο μέσος αριθμός*

των στοιχείων εξόδου που είναι επιλέξιμα για συμμετοχή ως είσοδος εάν η έξοδος του τελεστή τροφοδοτείται στην είσοδο ενός δεύτερου τελεστή" [37]. Υποθέτουμε σταθερό ρυθμό άφιξης και ότι υπάρχει αρκετή μνήμη για να εξυπηρετήσει τις ρυθμιστικές απαιτήσεις (buffering) των ερωτημάτων μας. Το μοντέλο κόστους χρησιμοποιείται και για τα παράθυρα που βασίζονται σε πλειάδες και για αυτά που βασίζονται στο χρόνο. Στην περίπτωση των παραθύρων βάσει χρόνου, ο αριθμός των ενεργών πλειάδων σε ένα παράθυρο i δίνονται από την ακόλουθη εξίσωση: Wi = λi * T, όπου T είναι το μέγεθος του παραθύρου με βάση το χρόνο και λi ο ρυθμός άφιξης των πλειάδων από την i πηγή. Οι μεταβλητές που χρησιμοποιούνται για τον υπολογισμό του κόστους μας παρουσιάζονται στο επόμενο σχήμα:

Table 1. Variables used in estimating resource requirements.

C_σ	Cost of performing a selection on a single tuple
C_P	Cost to probe an active window for a matching tuple just arriving
C_I	Cost to insert an arriving tuple into the sliding window
C_V	Cost to invalidate an expired tuple from the sliding window
σ	Selectivity factor of a selection predicate
f	Join selectivity factor
λ_i	Rate of arrival of tuples from source i
W	Size of a tuple-based window
T	Size of a time-base window

Εικ. 0.7.2, Οι μεταβλητές του μοντέλου κόστους μας [37].

Στη συνέχεια θα παρουσιάσουμε τις σχέσεις που περιγράφουν τις παραμέτρους και τους πόρους για όλους τους τελεστές. Οι παράμετροι μας είναι:

- λο: ο παραγόμενος ρυθμός δεδομένων για κάθε τελεστή
- Wo: το μέγεθος παραθύρου στην έξοδο ενός τελεστή
- C: το κόστος χρησιμοποίησης ενός τελεστή
- M: η μνήμη που χρειάζεται για κάθε τελεστή

Στον επόμενο πίνακα έχουμε όλες τις σχέσεις για τον υπολογισμό των παραπάνω παραμέτρων:

	Επιλογή	Συνένωση	Συναθροιστικές Συναρτήσεις/Παράθυρα
λο	λi * σ	f*(WR*λL + WL*λR)	multiplier*λi
Wo	Wi * σ	f*WR*WL	Wi
C	λi * Cσ	(λL+λR)*(Cp+Ci+Cv)	multiplier*Cσ*λi
M	0	WR+WL	Wi

Πίνακας. 0.7.3, Σχέσεις κόστους για τους τελεστές του συστήματος.

Στόχος Βελτιστοποίησης

Λαμβάνοντας υπόψη ένα συνδυαστικό ερώτημα για πηγές συνεχόμενων ροών, ορίζουμε δύο συναρτήσεις για οποιουδήποτε σχέδιο εκτέλεσης p για αυτό το ερώτημα:

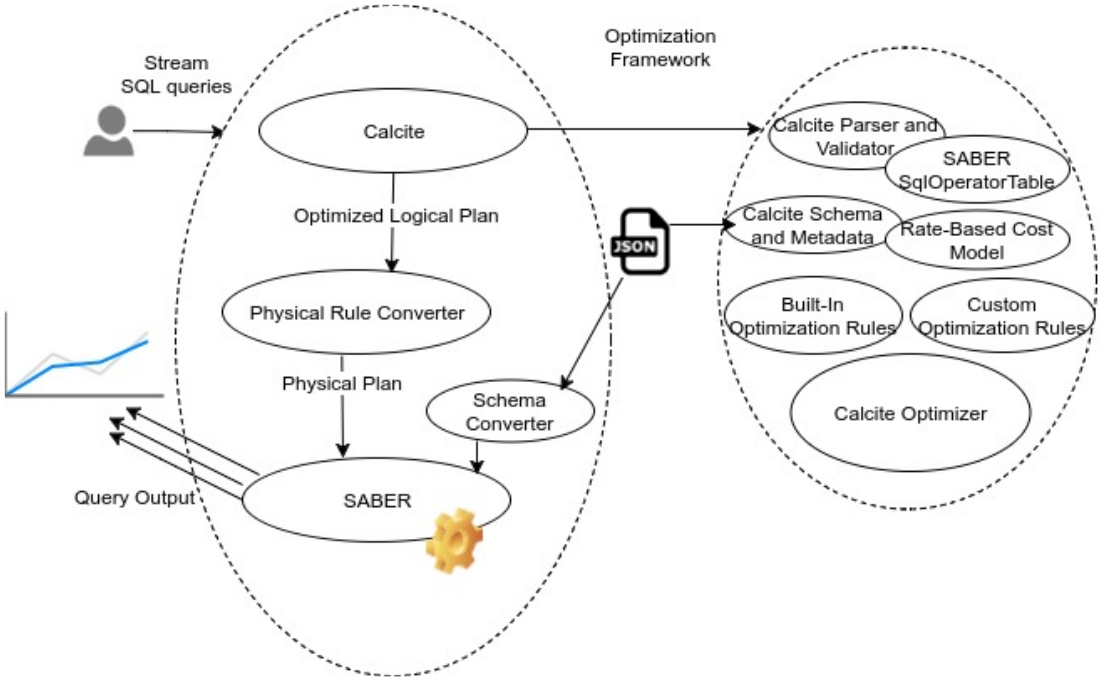
- λ(p): το throughput του σχεδίου είναι ίσο με $\lambda(p)=\prod_{i=1}^{n}\lambda i(p)$
- C(p): το κόστος αρχικοποίησης πόρων είναι ίσο με $C(p)=\sum_{i=1}^{n}Ci(p)$

Ο στόχος της βελτιστοποίησης μας είναι να βρούμε το σχέδιο με το μέγιστο R(p)= λ(p)/C(p), καθώς θέλουμε να μεγιστοποιηθεί η ταχύτητα παραγωγής δεδομένων και να ελαχιστοποιηθεί το κόστος αρχικοποίησης Επιπλέον, λαμβάνουμε υπόψη τη μνήμη που απαιτείται για κάθε πλάνο. Θα αξιολογήσουμε το μοντέλο μας στο κεφάλαιο της αξιολόγησης.

0.8 Περιγραφή της υλοποίησης

Η αρχιτεκτονική του RBStream

Το σύστημά μας βασίζεται στο Apache Calcite που παίζει κεντρικό ρόλο στην αρχιτεκτονική μας (Εικ. 0.8.1.α). Παρακάτω, θα συνοψίσουμε τα τέσσερα στάδια του συστήματός μας:



Εικ. 0.8.1.a, Η αρχιτεκτονική του RBStream.

Στάδιο πριν τη βελτιστοποίηση

Σε αυτό το στάδιο, ο χρήστης ορίζει τις πηγές ροής εισόδου για το Calcite, είτε προγραμματιστικά (χρησιμοποιώντας SchemaFactory, Schema, TableFactory και ScannableTable) ή δηλώνοντας σχήματα και τους πίνακες σε ένα αρχείο JSON. Αυτό το σχήμα μετατρέπεται στους αντίστοιχους πίνακες και τύπους δεδομένων των SABER. Πριν από την υποβολή ενός ερωτήματος στο σύστημα, υπάρχουν διάφορες παράμετροι της διαδικασίας βελτιστοποίησης για να επιλέξουμε, όπως:

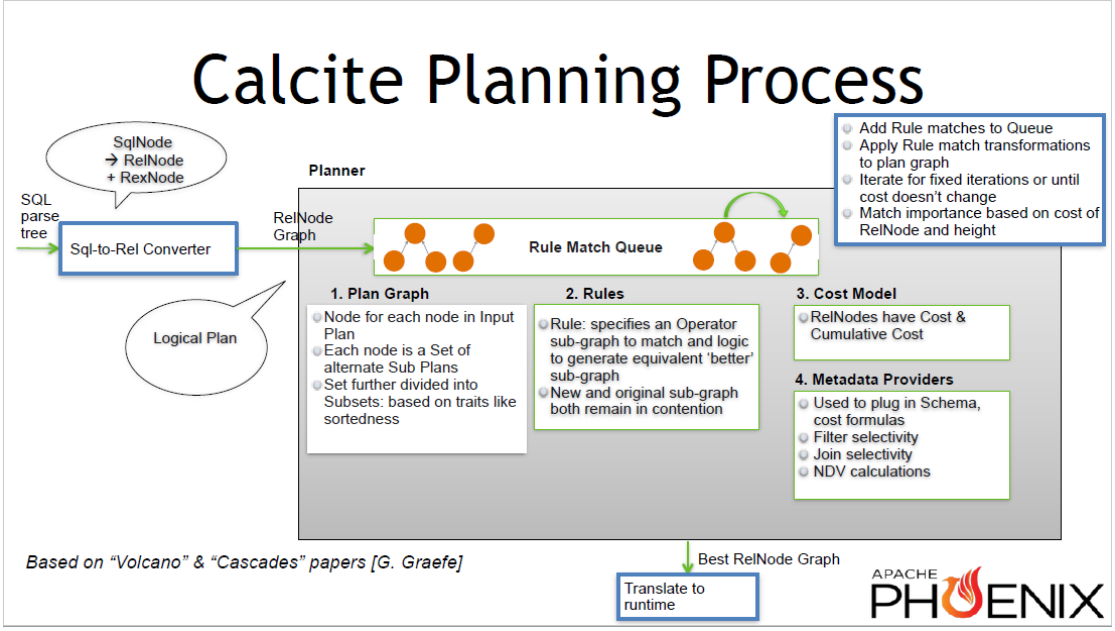
- Είτε να χρησιμοποιήσουμε το μοντέλο κόστους με βάση το ρυθμό εισροής των δεδομένων το ή το ενσωματωμένο μοντέλο κόστους του Calcite.
- Ο αλγόριθμος αλλαγής σειράς των συνενώσεων: μπορεί να επιλέξει είτε τον εξαντλητικό αλγόριθμο, ο οποίος προτιμάται για ένα μικρό αριθμό συνενώσεων, ή τον ευρετικό αλγόριθμο, ο οποίος προτιμάται στη γενική περίπτωση.

Επιπλέον, ο χρήστης μπορεί να προσαρμόσει τη διαμόρφωση του συστήματος του SABER και να επιλέξει τις παραμέτρους εκτέλεσης, όπως το μέγεθος των buffer που χρησιμοποιούνται, την πολιτική χρονοδρομολόγισης, τον αριθμό των νημάτων που θα αρχικοποιηθούν ή τον τρόπο εκτέλεσης του συστήματος (CPU, GPU ή και τα δύο).

Στάδιο Βελτιστοποίησης

Χρησιμοποιούμε ένα συνδυασμό των Volcano and Heuristic planners του Calcite. Η βελτιστοποίηση διαχωρίζεται σε έξι φάσεις, προκειμένου να είναι πιο αποτελεσματική και πιο εύκολη στη διαχείριση. Η αποτελεσματικότητα του διαχωρισμού αυτού παρουσιάστηκε από το γεγονός ότι σε κάθε φάση εφαρμόζονται μόνο σχετικές υποκατηγορίες κανόνων, μειώνοντας το χώρο αναζήτησης και την πολυπλοκότητα της διαδικασίας βελτιστοποίησης. Αν όλοι οι κανόνες εφαρμόζονταν σε μία μόνο φάση, είτε θα έπαιρνε πολύ χρόνο για να υπολογιστεί το βέλτιστο σχέδιο ενός σύνθετου ερωτήματος, με τη χρήση του Volcano planner, είτε δεν θα παίρναμε το βέλτιστο πλάνο με τη χρήση του Heuristic planner. Επίσης στον Heuristic planner, η σειρά με την οποία εφαρμόζονται οι κανόνες μπορεί να οδηγήσει σε διαφορετικές λύσεις. Λαμβάνοντας αυτές τις παρατηρήσεις υπόψιν, έχουμε κατηγοριοποιήσει τους κανόνες μας με βάση τη λειτουργικότητά τους και τους έχουμε διατάξει ανάλογα για να επιτευχθεί καλύτερο αποτέλεσμα χρησιμοποιώντας το σωστό planner σε κάθε φάση. Ως εκ τούτου, μπορούμε να προσθέσουμε, να καταργήσουμε και να αλλάξουμε τους κανόνες μόνο στη φάση που θέλουμε, χωρίς να επηρεάσουμε την υπόλοιπη αλυσίδα βελτιστοποίησης. Τα βήματα αυτής της διαδικασίας είναι:

- Το ερώτημα αναλύεται και επικυρώνεται τόσο έναντι ενός Κατάλογου των πηγών δεδομένων όσο και στους προσαρμοσμένους SqlValidator (επικυρώνει το συντακτικό δένδρο και παρέχει σημασιολογική πληροφορία για αυτό) και SqlOperatorTable.
- Στη συνέχεια, το ερώτημα μετατρέπεται σε αναπαράσταση λογικού πλάνου για το Calcite. Αυτή η αναπαράσταση είναι ένα δέντρο-γράφημα με τελεστές SQL ως κόμβους και τη ροή των δεδομένων μεταξύ τους ως ακμές.
- Η βελτιστοποίηση με βάση το κόστος (Cost Based Optimization - CBO) [82] του Calcite εφαρμόζει ενσωματωμένους και προσαρμοσμένους κανόνες ανάλογα με το επιλεγμένο μοντέλο κόστους. Η υλοποίηση του CBO μας συνδυάζει τόσο το VolcanoPlanner όσο και το HepPlanner και χωρίζει τη διαδικασία βελτιστοποίησης σε μικρότερα τμήματα. Αυτό μας βοηθά να επωφεληθούμε από την ανεξαρτησία των διάφορων τύπων κανόνων. Αν και η διαδικασία βελτιστοποίησης θα δημιουργήσει πολλά πιθανά πλάνα εκτέλεσης, τελικά μόνο αυτό με το χαμηλότερο δυνατό κόστος θα επιλεγεί. Οι φάσεις βελτιστοποίησης θα συζητηθούν αργότερα σε αυτό το κεφάλαιο πιο διεξοδικά.



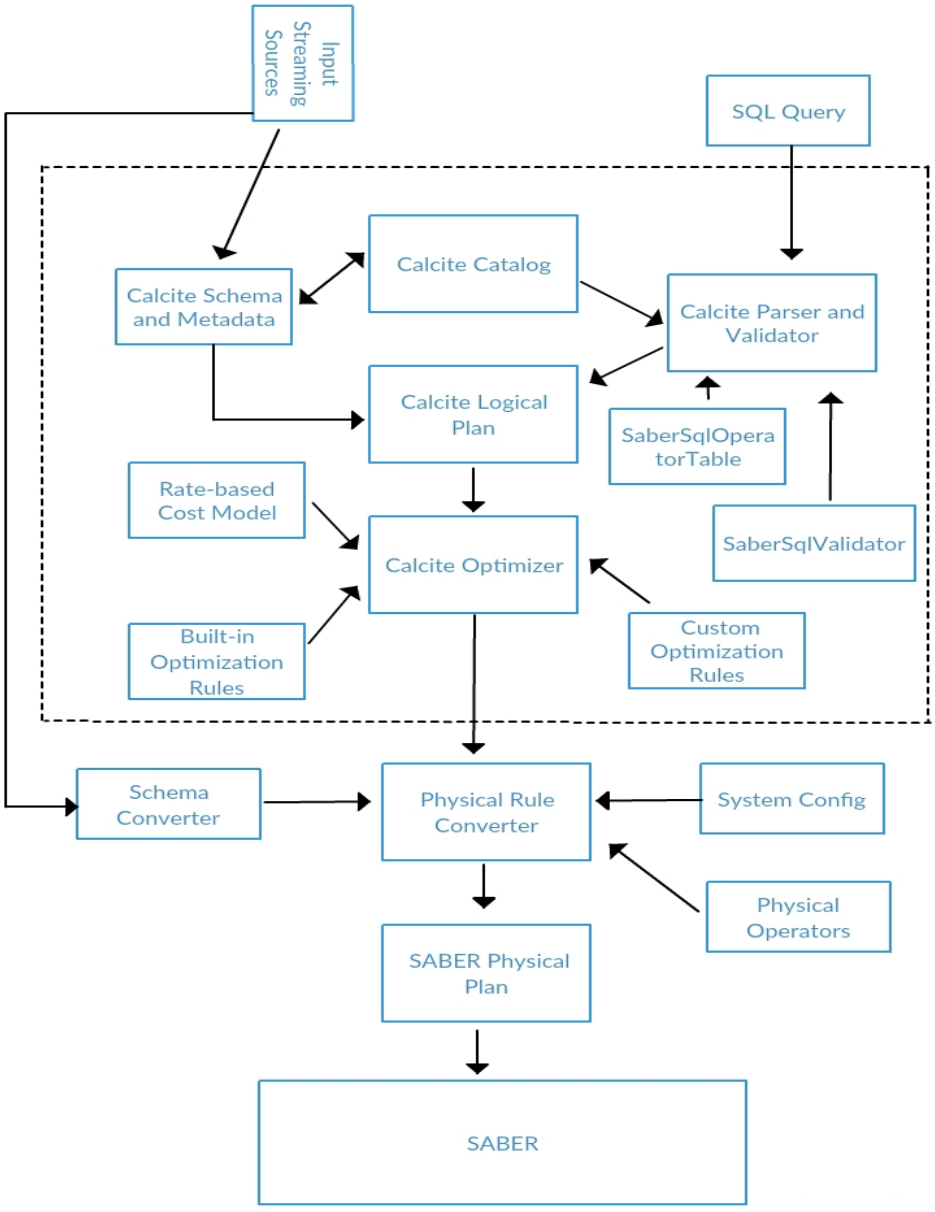
Εικ. 0.8.1.b, Η διαδικασία επιλογής πλάνου στο Calcite [53].

Στάδιο Μετατροπής

Σε αυτό το στάδιο το βελτιστοποιημένο λογικό πλάνο μετατρέπεται από τον Physical Rule Converter του συστήματος μας σε ένα φυσικό σχέδιο, έτοιμο να εκτελεστεί από το SABER. Σύνθετα ερωτήματα SQL μετατρέπονται σε δεκάδες γραμμές κώδικα που μπορούν να εκτελεστούν στο SABER, με μικρή επιβάρυνση χιλιοστών του δευτερολέπτου πριν την έναρξη της εκτέλεσης του ερωτήματος.

Στάδιο Εκτέλεσης

Τέλος, το φυσικό πλάνο εκτελείται στο SABER, και οι διάφορες μετρήσεις του συστήματος συλλέγονται και παρουσιάζονται στο χρήστη.



Εικ. 0.8.1.c, Πιο λεπτομερής επισκόπηση του RBStream.

Σημασιολογία δεδομένων ροής

Χρησιμοποιούμε τις ακόλουθες συναρτήσεις του Calcite [14] για να ορίσουμε την SQL για τα δεδομένα ροής στο σύστημα μας:

Scalar functions:

- FLOOR(dateTime TO intervalType): στρογγυλοποιεί προς τα κάτω μια ημερομηνία, το χρόνο ή μια σφραγίδα χρόνου σε ένα συγκεκριμένο χρονικό διάστημα.

- CEIL(dateTime TO intervalType): στρογγυλοποιεί προς τα πάνω μια ημερομηνία, το χρόνο ή μια σφραγίδα χρόνου σε ένα συγκεκριμένο χρονικό διάστημα.

Partitioning functions:

- HOP(t, emit, retain): επιστρέφει μια συλλογή από group keys για μια γραμμή για να είναι μέλος σε ένα hopping window.
- HOP(t, emit, retain, align): επιστρέφει μια συλλογή από group keys για μια γραμμή για να είναι μέλος σε ένα hopping window με μια συγκεκριμένη ευθυγράμμιση χρόνου.
- TUMBLE(t, emit): επιστρέφει μια συλλογή από group keys για μια γραμμή για να είναι μέλος σε ένα tumbling window.
- TUMBLE(t, emit, align): επιστρέφει μια συλλογή από group keys για μια γραμμή για να είναι μέλος σε ένα tumbling window με μια συγκεκριμένη ευθυγράμμιση χρόνου.

Η συνάρτηση TUMBLE(t, e) είναι ισοδύναμη με την TUMBLE(t, e, TIME '00:00:00').

Η συνάρτηση TUMBLE(t, e, a) είναι ισοδύναμη με την HOP(t, e, e, a).

Η συνάρτηση HOP(t, e, r) είναι ισοδύναμε με την HOP(t, e, r, TIME '00:00:00').

Ενώ το SABER προσφέρει συρόμενα παράθυρα τόσο με βάση το χρόνο και όσο και τις πλειάδες οποιουδήποτε μεγέθους και ολίσθησης, το Calcite περιορίζει τις επιλογές μας με την τρέχουσα υλοποίηση του. Ως εκ τούτου, έχουμε περιορισμένους τρόπους έκφρασης παραθύρων στο σύστημά μας αυτή τη στιγμή:

- Αν δεν ορίζεται παράθυρο από την SQL, χρησιμοποιούμε το προεπιλεγμένο παράθυρο. Στην εφαρμογή μας, το προεπιλεγμένο παράθυρο είναι το now-window, που είναι παράθυρο με βάση το χρόνο μεγέθους και ολίσθησης 1.
- Ο χρήστης μπορεί να ορίσει ένα σταθερό παράθυρο χρησιμοποιώντας κατάλληλα τις συναρτήσεις FLOOR και CEIL κατά τη δήλωση του GROUP BY. Προς το παρόν, μπορούν να οριστούν μόνο παράθυρα με βάση το χρόνο με τον τρόπο αυτό, τα οποία έχουν το μέγεθος των 1 sec, 1 λεπτό, 1 ώρα ή 1 ημέρα (που εκφράζονται σε νανοδευτερόλεπτα). Για παράδειγμα, το επόμενο ερώτημα:

```
select rowtime, productid, sum(units)
from s.orders
group by rowtime, productid, floor(rowtime to hour)
```

μετατρέπεται σε ένα τελεστή συναθροιστικής συνάρτησης του αθροίσματος υπολογιζόμενου σε σταθερό παράθυρο με μέγεθος μία ώρα σε νανοδευτερόλεπτα (τα σταθερά παράθυρα είναι συρόμενα παράθυρα με μέγεθος ίσο με την ολίσθηση) της πηγής δεδομένων ροής s.orders. Αυτό το παράθυρο μετατρέπεται σε WindowType.RANGE_BASED παράθυρο εύρους = ολίσθησης = 3600000 στο SABER.

- Ο χρήστης μπορεί επίσης να ορίσει ένα σταθερό παράθυρο που βασίζεται στο χρόνο χρησιμοποιώντας τις συναρτήσεις TUMBLE, TUMBLE_START και TUMBLE_END. Για παράδειγμα σε αυτό το ερώτημα:

```
select tumble_end(rowtime, interval '2' hour) as rowtime
from s.orders
group by tumble(rowtime, interval '2' hour), productid
```

έχουμε σταθερά παράθυρα μεγέθους ίσο με 7200000 (δύο ώρες σε νανοδευτερόλεπτα). Αυτές οι συναρτήσεις μας δίνουν την ευκαιρία να εκφράσουμε πιο περίπλοκα παράθυρα σε αντίθεση με την προηγούμενη περίπτωση, τα οποία μπορούν επίσης να ευθυγραμμιστούν δίνοντας μια τρίτη παράμετρο στη συνάρτηση TUMBLE. Αυτή η λειτουργία δεν υποστηρίζεται από το SABER ακόμα.

- Ο χρήστης μπορεί να ορίσει συρόμενα παράθυρα με βάση το χρόνο και τις πλειάδες με ολίσθηση 1 με τη χρήση των συναρτήσεων OVER και Windows. Στη δική μας υλοποίηση, δεν χρησιμοποιούμε τη συνάρτηση OVER σύμφωνα με τον ορισμό που δώσαμε στην ενότητα για το Calcite Streaming, αλλά για τον ορισμό απλών συναθροιστικών συναρτήσεων (το SABER δεν υποστηρίζει τον ορισμό της συνάρτησης OVER ακόμα). Ένα παράδειγμα ενός τέτοιου παραθύρου (WindowType.ROW_BASED σε SABER) είναι:

```
select rowtime, productid, SUM(units) over pr
from s.orders
window pr as (PARTITION BY productid ROWS BETWEEN 5
      PRECEDING AND 10 FOLLOWING)
```

Σε αυτό το ερώτημα έχουμε μια συνάρτηση Aggregate αθροίσματος με συρόμενο παράθυρο μεγέθους 15 και ολίσθησης 1, η οποία μετατρέπεται στο SABER σε WindowType.ROW_BASED παράθυρο.

Αντίστοιχα, ένα παράδειγμα παραθύρου με βάση του χρόνου με μέγεθος 1000 και ολίσθηση 1 είναι:

```
select rowtime, productid, SUM(units) over pr
from s.orders
window pr as (PARTITION BY rowtime, productid RANGE
INTERVAL '1' SECOND PRECEDING)
```

- Ο χρήστης μπορεί να καθορίσει συρόμενα παράθυρα μόνο με βάση το χρόνο όπως πριν, αλλά επιπλέον με ολίσθηση μεγέθους 1000 (1 δευτερόλεπτο), 60000 (1 λεπτό), 3600000 (1 ώρα) ή 86400000 (1 ημέρα) χρησιμοποιώντας είτε τη FLOOR είτε τη CEIL συνάρτηση στο partiotion statement του ορισμού του παραθύρου. Για παράδειγμα:

```
select rowtime, min(orderid) over pr
from  s.orders
window pr as (PARTITION BY floor(rowtime to second) RANGE
      INTERVAL '1' HOUR PRECEDING)
```

  Αυτό το ερώτημα υπολογίζει το ελάχιστο orderid πάνω σε ένα συρόμενο παράθυρο μεγέθους 3600000 και ολίσθησης 1000.

- Τέλος, ο χρήστης μπορεί να ορίσει παράθυρα hopping με βάση το χρόνο με τις συναρτήσεις HOP, HOP_START και HOP_END. Για παράδειγμα, στο επόμενο ερώτημα:

```
select hop_start(rowtime, interval '1' hour, interval '3'
hour) as rowtime, count(*) as c
from s.orders
group by hop(rowtime, interval '1' hour, interval '3'
hour);
```

  έχουμε ένα hopping παράθυρο μεγέθους=10800000 (3 ώρες) και ολίσθησης=3600000 (1 ώρα). Με αυτές τις συναρτήσεις μπορούμε να εκφράσουμε πιο πολύπλοκα συρόμενα παράθυρα με βάση το χρόνο από ό,τι θα μπορούσαμε στις παραπάνω περιπτώσεις, καθώς έχουμε μεγαλύτερο έλεγχο της ολίσθησης.

Φάσεις Βελτιστοποίησης

Στην υλοποίηση μας, το Apache Calcite χρησιμοποιείται για να εισάγουμε έναν Λογικό Βελτιστοποιητή με βάση το κόστος (Cost Based Logical Optimizer - CBO) στο SABER. Ο σκοπός της βελτιστοποίησης είναι να αυξηθεί ο ρυθμός παραγωγής αποτελεσμάτων του ερωτήματος, ελαχιστοποιώντας παράλληλα το μέγεθος των ενδιάμεσων αποτελεσμάτων

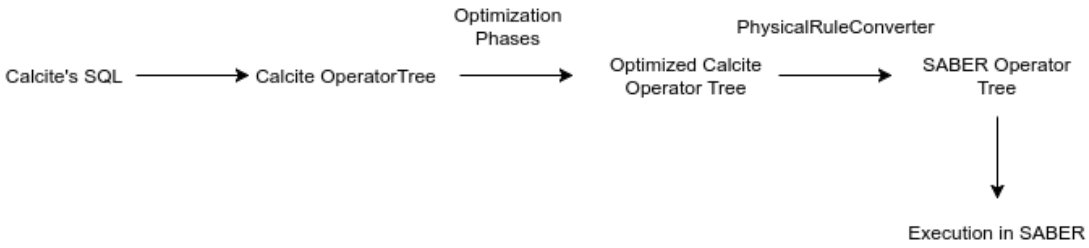
που δημιουργούνται. Αυτές οι λογικές βελτιστοποιήσεις μπορούν να βελτιώσουν το latency ενός ερωτήματος και διευκολύνουν σε μεγάλο βαθμό τη χρήση του SABER (για παράδειγμα, οι χρήστες δεν χρειάζεται να υποβάλουν ένα βελτιστοποιημένο ερώτημα με τη σωστή σειρά συνένωσης, για να μπορεί να εκτελεστεί αποτελεσματικά). Αντί για το selectivity ενός πίνακα, το μοντέλο του κόστους μας χρησιμοποιεί το ρυθμό εισόδου των δεδομένων και το μέγεθος του παραθύρου για να αποφασίσει ποια είναι η βέλτιστη σειρά συνένωσης των ροών και να μειώσει σημαντικά το latency του ερωτήματος.

Οι βελτιστοποιήσεις ερωτημάτων σε μια μηχανή επεξεργασίας ερωτημάτων μπορούν να ταξινομηθούν σε δύο κατηγορίες: τις λογικές βελτιστοποιήσεις ερωτήματος και τις φυσικές βελτιστοποιήσεις ερωτήματος. Θα ασχοληθούμε μόνο με την πρώτη κατηγορία, καθώς η δεύτερη αφορά βελτιώσεις που έχουν να κάνουν με την εκτέλεση του ερωτήματος στο SABER (π.χ. χρονοδρομολόγιση), οι οποίες εκτελούνται ήδη από το SABER εσωτερικά, και την επιλογή μεταξύ διαφορετικών υλοποιήσεων, που δεν υποστηρίζει το SABER ακόμα. Οι λογικές βελτιστοποιήσεις ερωτημάτων μπορούν να προσδιοριστούν με βάση τη σχεσιακή άλγεβρα, ανεξάρτητα το φυσικό στρώμα στο οποίο εκτελείται το ερώτημα.

Οι λογικές βελτιστοποιήσεις ερωτήματος που εφαρμόζονται στο SABER μπορούν να κατηγοριοποιηθούν σε:

- Βελτιστοποίηση για κλάδεμα προβολής (Projection Pruning).
- Βελτιστοποιήσεις για το κατέβασμα κατηγορημάτων (Predicate Push Down).
- Κανόνες που συγχωνεύουν δύο συνεχόμενους τελεστές προβολών ή φίλτρων σε έναν.
- Κανόνες Βελτιστοποίηση για την αναδιάταξη της σειράς συνένωσης ροών.
- Κανόνες σχετικά με το συμπερασμό μεταβατικών κατηγορημάτων (Transitive Predicates).

Ωστόσο, δεν μπορούν όλοι οι κανόνες, που χρησιμοποιούνται για την υλοποίηση μας, να επωφεληθούν από ένα CBO. Ως εκ τούτου, χρησιμοποιούμε ένα συνδυασμό των Volcano και Heuristic planners του Calcite σε στάδια (όπως στο Hive [38] ή στο Drill [39]).



Εικ. 0.8.3.a, Από την SQL μέχρι την εκτέλεση στο SABER.

Οι φάσεις βελτιστοποίησης που εφαρμόζονται σε ένα δεδομένο σχέδιο διαφέρουν ανάλογα με τον τύπο βελτιστοποίησης που επιλέγεται από το χρήστη. Αν ο χρήστης έχει επιλέξει να μην βελτιστοποιήσει το ερώτημα, εφαρμόζονται μόνο κανόνες που θα μας βοηθήσουν να δηλώσουμε τα παράθυρα για τη σημασιολογία των δεδομένων ροής μας.

Φάση 1: Υποστήριξη παραθύρων

Στην πρώτη φάση θα επιβάλουμε τους κανόνες που ξαναγράφουν το αρχικό σχέδιο ερωτήματος μας και δημιουργούν τους τελεστές παραθύρου χρησιμοποιώντας τον HepPlanner. Αυτό είναι ένα βασικό στάδιο, καθώς μας παρέχει την κατάλληλη υποστήριξη παραθύρων που εκφράζεται από την SQL του Calcite και βοηθά το σύστημα με τη σημασιολογία των δεδομένων ροής.

Φάση 2: Ευριστικοί Κανόνες

Οι κανόνες που ανήκουν σε αυτή τη φάση δεν επωφελούνται από το CBO και εφαρμόζονται από το HepPlanner. Αυτοί οι κανόνες πρέπει να εφαρμόζονται γενικά, ανεξάρτητα από το μοντέλο κόστους που χρησιμοποιείται στην εφαρμογή μας, καθώς μειώνουν το συνολικό κόστος. Μπορούν να κατηγοριοποιηθούν ως εξής:

- Distinct Aggregate Rewrite
- Συγχώνευση, αφαίρεση και μείωση προβολών.
- Προσθήκη constant propagation και folding, transitive inference, όχι null filters και υποστήριξη για Joins με WHERE.
- Κλάδεμα άδειων αποτελεσμάτων κανόνων.

Φάση 3: Pre-Join Κανόνες που επηρεάζονται από το CBO

Στη φάση 3, θέλουμε να αναζητήσουμε εξαντλητικά για το βέλτιστο σχέδιο στο χώρο αναζήτησης που ορίζεται από τους κανόνες μας. Ως αποτέλεσμα, χρησιμοποιούμε το VolcanoPlanner. Οι κανόνες αυτοί εφαρμόζονται σε συνδυασμό με τους κανόνες του μετατροπέα, και επιστρέφουν ένα σχέδιο ερωτήματος με τελεστές που έχουν μια συγκεκριμένη σύμβαση. Κατά τη χρήση του μοντέλου που βασίζεται στο ρυθμό εισροής δεδομένων, οι κανόνες μετατροπής αλλάζουν τους ενσωματωμένους τελεστές σε τελεστές που υποστηρίζουν το μοντέλο μας. Η φάση αυτή είναι απαραίτητη προκειμένου να εφαρμόσουμε αργότερα βελτιστοποιήσεις μας χρησιμοποιώντας το μοντέλο κόστους που δημιουργήθηκε.

Οι παραπάνω κανόνες χρησιμοποιούν το προσαρμοσμένο SaberRelFactory αντί το ενσωματωμένο RelFactory του Calcite. Με αυτόν τον τρόπο, έχουμε εισάγει το μοντέλο κόστους μας και βελτιστοποιούμε ένα συγκεκριμένο πλάνο με βάση τη λογική μας. Αν ο χρήστης έχει επιλέξει να μην χρησιμοποιήσει το μοντέλο μας, θα χρησιμοποιηθούν οι ίδιοι κανόνες, αλλά με τα ενσωματωμένα RelFactory και μοντέλο κόστους.

Φάση 4: Κανόνες Αναδιάταξης Συνενώσεων

Αυτή είναι η πιο σημαντική φάση στα σύνθετα ερωτήματα με πολλαπλές συνενώσεις. Ο χρήστης μπορεί να επιλέξει είτε να χρησιμοποιήσει το VolcanoPlanner, προκειμένου να βρεθεί η βέλτιστη λύση με εξαντλητική αναζήτηση (κατάλληλο μόνο για λιγότερες από 6 συνενώσεις), ή την ταχύτερη ευρετική υλοποίηση χρησιμοποιώντας το HepPlanner. Τα στατιστικά στοιχεία που χρησιμοποιούνται για την αναδιάταξη συνενώσεων εξαρτώνται από τις ρυθμίσεις που έχει επιλέξει ο χρήστης: αν είναι το μοντέλο που βασίζεται στο ρυθμό εισροής δεδομένων, χρησιμοποιεί το ρυθμό εισόδου και τα μεγέθη των παραθύρων, ενώ αν είναι το ενσωματωμένο μοντέλο κόστους, χρησιμοποιεί το cardinality των πινάκων.

Φάση 5: Εφαρμογή After-Join Κανόνων

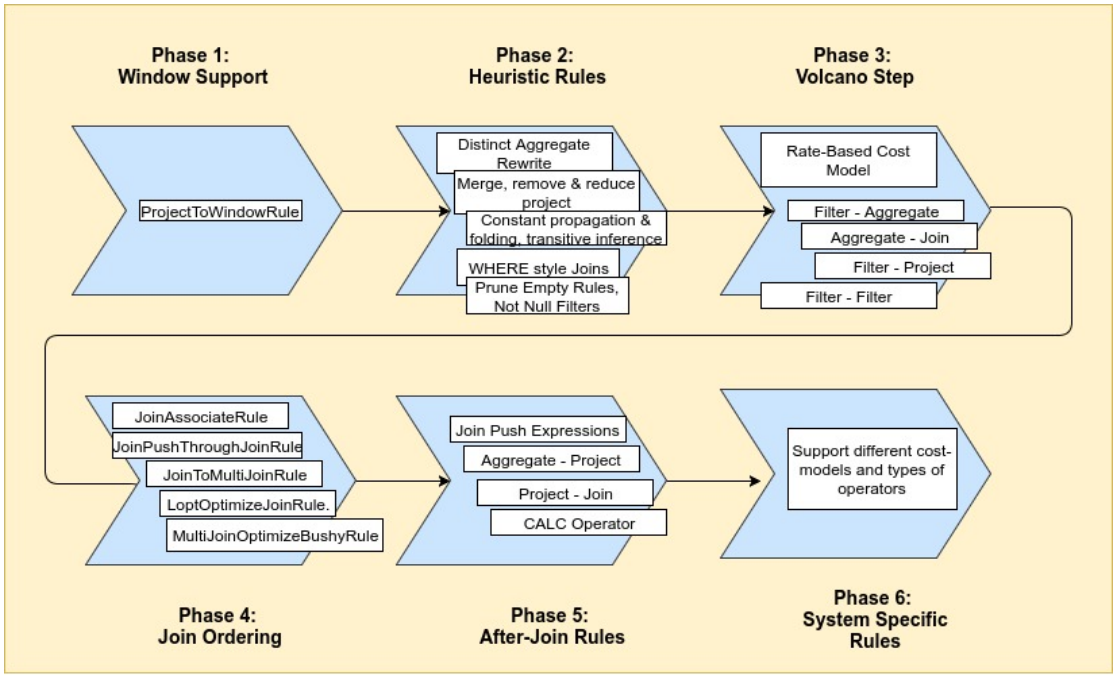
Αυτή η τελική φάση βελτιστοποίησης χρησιμοποιείται για να εφαρμόσουμε κάποιους τελευταίους κανόνες, ώστε να διασφαλιστεί ότι θα έχουμε ένα βελτιστοποιημένο σχέδιο μετά τη φάση 4. Οι βελτιστοποιήσεις αυτές δεν χρειάζονται στατιστικά στοιχεία.

Η φάση 5 χωρίζεται επίσης σε μικρότερες φάσεις, οι οποίες ομαδοποιούν ορισμένους διαδοχικούς τελεστές (για παράδειγμα ένα Φίλτρο και μια Προβολή) σε έναν σύνθετο τελεστή. Αυτός ο τελεστής μετατρέπεται σε μία ενιαία εργασία ερωτήματος με πολλαπλούς τελεστές στο SABER, προκειμένου να εκμεταλλευτεί την εκτέλεση συγκεκριμένων περιπτώσεων χρήσης. Εφαρμόζοντας αυτούς τους κανόνες, μειώνουμε την επικοινωνία των νημάτων και βελτιώνουμε την απόδοση.

Σε όλες τις παραπάνω περιπτώσεις, ο HepPlanner χρησιμοποιείται με από κάτω προς τα πάνω σειρά εφαρμογής των εγγεγραμμένων κανόνων.

Φάση 6: Μετατροπή είτε Enumerable είτε SaberRel τελεστών σε Λογικούς

Αυτή είναι μια φάση που δεν αλλάζει τη διάταξη των τελεστών του πλάνου. Ωστόσο, απαιτείται, προκειμένου να έχουμε μια γενική αναπαράσταση των πλάνων για τον physical rule converter που θα χρησιμοποιήσουμε αργότερα, ανεξάρτητα από τη σύμβαση των τελεστών. Με τη χρήση απλών κανόνων μετασχηματιστή, ώστε να διατηρηθεί η σύμβαση και τα χαρακτηριστικά των φορέων μας ανέπαφα, μπορούμε να υποστηρίξουμε διαφορετικά μοντέλα κόστους και τύπους τελεστών.



Εικ. 0.8.3.b, Οι φάσεις της βελτιστοποίησης.

Μοντέλο Κόστους με βάση το ρυθμό εισροής δεδομένων

Στο σύστημά μας, προτείνουμε μια νέα υλοποίηση της κλάσης RelOptCost που ονομάζεται SaberCostBase. Η κλάση SaberCostBase θα υπολογίσει τη χρήση της CPU, τη χρήση της μνήμης, την ρυθμό εκροής δεδομένων, το μέγεθος του παραθύρου και το στόχο βελτιστοποίησης R (χρησιμοποιούμε το αντίστροφο R του ορισμού που δίδεται στην ενότητα του μοντέλου κόστους) για κάθε τελεστή. Ο αλγόριθμος σύγκρισης του κόστους θα δώσει σημασία στο στόχο βελτιστοποίησης R πρώτα και στη συνέχεια στην αρχικοποίηση μνήμης, όταν προσπαθούμε να βρούμε το βέλτιστο σχέδιο Αυτός είναι ο ψευδοκώδικας της υλοποίησης μας:

```
public boolean isLt(RelOptCost other) {
    SaberCostBase that = (SaberCostBase) other;
    if (true)
        return ( (this.R <= that.R) // R is the main optimization
                                    // goal. The R used here is the
                                    // reversed R from 4.2 Section.
           && (this.memory <= that.memory) // this is memory
                                           // utilization
           );
        return ((this.cpu + this.memory + this.R)
          < (that.cpu + that.memory + that.R)); //alternative way of
                                    // checking for the smaller cost
}
```

Αυτή είναι η συνάρτηση που χρησιμοποιείται από τον planner για να βρει το σχέδιο με το μικρότερο κόστος, σύμφωνα με το στόχο βελτιστοποίησης που έχει οριστεί. Ως εκ τούτου, επιλέγουμε το σχέδιο με τη μικρότερη κατανάλωση μνήμης από τα σχέδια που έχουν τη μικρότερη παράμετρο R.

Εκτός από τις παραπάνω μεταβολές, έπρεπε να ξαναγράψουμε τη λογική της μεθόδου αθροίσματος του SaberCostBase, η οποία χρησιμοποιείται για να υπολογιστεί το συσσωρευτικό κόστος.

Ακολουθούν οι μεταβλητές που θα χρησιμοποιηθούν στον υπολογισμό του κόστους:

```
public static final int BASE_CPU_COST = 1;
public static final int PROJECT_CPU_COST = 4 * BASE_CPU_COST;
public static final int HASH_CPU_COST = 8 * BASE_CPU_COST;
public static final int Cs = PROJECT_CPU_COST;
public static final int Cj = HASH_CPU_COST;
```

Οι τρεις πρώτες μεταβλητές είναι εκτιμήσεις που πήραμε από το Drill [58]. Με αυτές τις μεταβλητές του κόστους προσπαθούμε να υπολογίσουμε τις μεταβλητές Cs και Cj του μοντέλου μας. Το κόστος της εκτέλεσης ενός join περιλαμβάνει το κόστος της ανίχνευσης, εισαγωγής και διαγραφής που απαιτείται: Cj = (Cp + Ci + Cv).

Για να χρησιμοποιήσουμε το καινούργιο μας μοντέλο, έπρεπε να αλλάξουμε την ComputeSelfCost() μέθοδο όλων των τελεστών που χρησιμοποιούνται στην υλοποίηση μας (custom SaberRelBase τελεστές)

Ο υπολογισμός του κόστους κάθε τελεστή βασίζεται στον πίνακα με τις παραμέτρους στην ενότητα του Μοντέλου Κόστους. Το κόστος αυτό υπολογίζεται κάθε φορά που υπολογίζουμε το συσσωρευτικό κόστος ενός συνόλου τελεστών που συνιστούν ένα ερώτημα.

Physical Rule Converter

Η μετατροπή του λογικού πλάνου του Calcite στο αντίστοιχο φυσικό πλάνο του SABER διαδραματίζει κρίσιμο ρόλο στην υλοποίηση του συστήματός μας. Η κλάση PhysicalRuleConverter είναι υπεύθυνη για την κατασκευή του δέντρου με τους φυσικούς τελεστές από ένα συγκεκριμένο λογικό πλάνο. Ο αναδρομικός αλγόριθμος που χρησιμοποιείται για την κατασκευή του φυσικού πλάνου, μαζί με τη μέθοδο εκτέλεσης του παρουσιάζονται με τον παρακάτω ψευδοκώδικα:

```
HashMap chainOfOperators
Map tablesMap
```

```
HashSet queries
List aggregates

void convert (logicalPlan) {
      // If the given logical plan has no inputs, it represents a
      // simple
      // LogicalTableScan (select * from stream). This case is treated
      // seperately.
      if logicalPlan.getInputs() == 0 then
            convertSingleRelNode(logicalPlan)
      else convertMultipleRelNodes(logicalPlan)
}

void convertSingleRelNode(logicalPlan) {
      inputStream = logicalPlan.getStreamFrom(tablesMap)
      create a Project Operator over the inputStream
      add the operator in the queries HashSet
}

Pair<Integer, String> convertMultipleRelNodes(logicalPlan) {
      List children = logicalPlan.getInputs();
      if children.size() == 0 then {
            inputStream = logicalPlan.getStreamFrom(tablesMap)
            create a Project Operator over the inputStream
            add parameters that will be used by later operators in
                  the chainOfOperators HashMap
            add the operator in the queries HashSet
            return (logicalPlan.getId(), logicalPlan.getRelTypeName)
      }
      else if children.size() == 1 then {
            // create its child
            chainTail = children.get(0)
            Pair <Integer, String> node =
```

```
            convertMultipleRelNodes(chainTail)
      get the parameters needed from the child operator using
            chainOfOperators
      create the given Operator with these parameters
      add parameters that will be used by later operators in
            the chainOfOperators HashMap
      add the operator in the queries HashSet
      if the child isn't a LogicalTableScan then connect it
            with its child
      if the operator is Aggregate or Window then add it to the
            aggregates list
      return (logicalPlan.getId(), logicalPlan.getRelTypeName)
}
else if children.size() == 2 then {
      leftChainTail = children.get(0)
      // create left child
      Pair <Integer, String> leftNode =
            convertMultipleRelNodes(leftChainTail)
      rightChainTail = children.get(1)
      // create right child
      Pair <Integer, String> rightNode =
            convertMultipleRelNodes(rightChainTail)
      get the parameters needed from the children operators
            using chainOfOperators
      create the given Operator with these parameters
      add parameters that will be used by later operators in
            the chainOfOperators HashMap
      add the operator in the queries HashSet
      if the  left child isn't a LogicalTableScan then connect
           with it
      if the  right child isn't a LogicalTableScan then connect
            with it
      return (logicalPlan.getId(), logicalPlan.getRelTypeName)
```

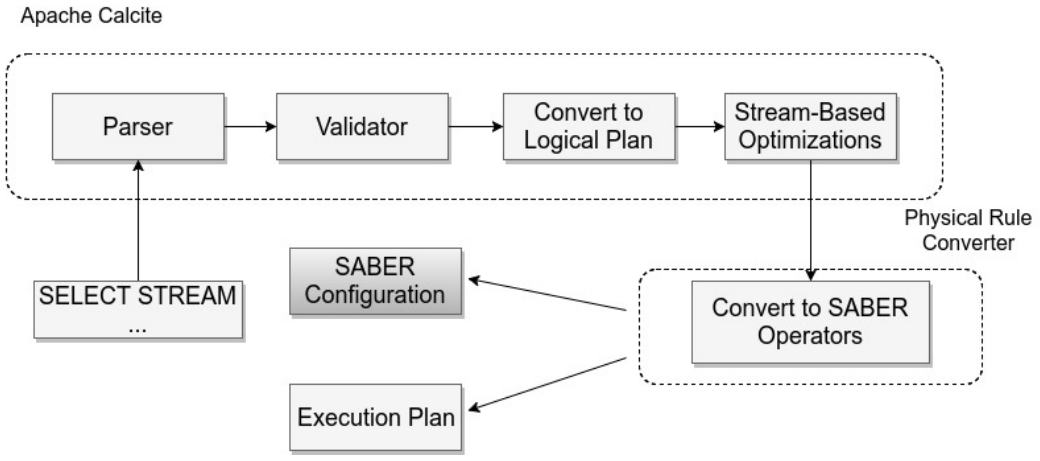
```
	}
}

void execute () {
	QueryApplication application = new QueryApplication(queries);
	application.setup();
	/* The path is query -> dispatcher -> handler -> aggregator */
	for ( SaberRule agg : aggregates){
		if (systemConf.CPU)
			agg.getQuery().setAggregateOperator((IAggregateOpera
		tor)
				agg.getCpuCode());
		else
			agg.getQuery().setAggregateOperator((IAggregateOpera
			tor)
				agg.getGpuCode());
	}
	/* Execute the query. */
	while (true) {
		for (Map.Entry<Integer,ChainOfRules> c :
		chainOfOperators.entrySet())
			if(c.getValue().getIsFirst()) {
				if(c.getValue().getFlag() == false) {
					application.processData
						(c.getValue().getData());
				} else {
					if(c.getValue().getHasMore() == true) {
						application.processSecondStream
							(c.getValue().getData());
					} else {
						application.processFirstStream
							(c.getValue().getData());
						application.processSecondStream
```

```
                                                    (c.getValue().getData2());
                                        }
                                    }
                                }
                            }
                        }
    }
```

Στην πρώτη περίπτωση, όταν το RelNode δεν έχει κόμβους παιδιά, μπορούμε να υποθέσουμε ότι πρόκειται για έναν τελεστή LogicalTableScan. Σε όλες τις άλλες περιπτώσεις, χρησιμοποιούμε την dumb υλοποίηση του τελεστή LogicalTableScan που μας βοηθάει να οικοδομήσουμε τους μετέπειτα τελεστές. Στην περίπτωση που το RelNode έχει ένα παιδί, είναι Προβολή, Φίλτρο, Συναθροιστική συνάρτηση ή Παράθυρο και πρέπει να φτιάξουμε όλα τα παιδιά του πριν από τη δημιουργία του. Τέλος, ένα RelNode με δύο παιδιά είναι Συνένωση, και πρέπει επίσης να κατασκευάσουμε τα δύο παιδιά του πρώτα.

Η μέθοδος execute() χρησιμοποιείται για να εκτελέσει το φυσικό πλάνο μας. Χρησιμοποιούμε το chainOfOperators map και μια aggregates list για όλες τις παραμέτρους που απαιτούνται. Τα δεδομένα υποβάλλονται σε επεξεργασία σε μια pipelined διαδικασία από SABER στο βρόχο while.



Εικ. 0.8.5, Query Planner.

Κατασκευή των φυσικών τελεστών

Προκειμένου να διευκολυνθεί η καλύτερη κατανόηση της διαδικασίας μετατροπής, στην παρούσα υποενότητα θα εξηγήσουμε πώς κατασκευάζονται οι φυσικοί τελεστές. Μερικές γραμμές SQL μετατρέπονται σε δεκάδες γραμμές κώδικα, έτοιμες για εκτέλεση στο SABER. Η κλάση PhysicalRuleConverter είναι η κύρια κλάση που ενορχηστρώνει τη μετατροπή ενός λογικού πλάνου σε φυσικό. Χρησιμοποιεί έξι κανόνες ώστε να επιτευχθεί το έργο της: SaberAggregateRule, SaberFilterRule, SaberJoinRule, SaberProjectRule, SaberScanRule και SaberWindowRule. Οι κανόνες αυτοί αφορούν την υλοποίηση των τελεστών του SABER και δεν έχουν σχέση με τους κανόνες του Calcite που χρησιμοποιήσαμε πριν.

Τα βασικά συστατικά που χρησιμοποιούνται από τις φυσικούς κανόνες μετατροπής είναι:

1) AggregationUtil

Αυτή η κλάση υλοποιεί την κατασκευή των Aggregates και Group By attributes για το SABER. Παίρνει ως είσοδο μια λίστα με τα aggregates (aggregate.getAggCallList ()) και ένα bit set με τα group by attributes (aggregate.getGroupSet()) είτε μιας συναθροιστικής συνάρτησης είτε ενός παραθύρου, και δημιουργεί τις αντίστοιχες δομές στο SABER. Μας παρέχει επίσης το σωστό σχήμα εξόδου, το οποίο είναι απαραίτητο για την pipelined διαδικασία στην οποία θα πρέπει να καθορίσουμε τους τελεστές μας. Η συνάρτηση count αναφέρεται πάντα την πρώτη στήλη, επειδή ο χρήστης μπορεί να γράψει COUNT (*) ή COUNT () στην SQL του Calcite. Αυτή τη στιγμή το SABER υποστηρίζει μόνο πέντε aggregates: min, max, count, sum and avg. Το επόμενο ερώτημα:

```
select rowtime, sum(units), count(orderid)
from  s.orders
group by rowtime,units,orderid, floor(rowtime to hour)
```

μετατρέπεται στο SABER σε ένα παράθυρο εύρους και ολίσθησης ίσα με 3600000. Ο αριθμός group by attributes είναι τέσσερα, όπως αναμενόταν, και υπολογίζουμε το sum και το count των αντίστοιχων στηλών. Ο τελεστής συνάθροισης που χρησιμοποιήθηκε περιέχει incremental aggregation types. Ωστόσο, κατά το χρόνο εκτέλεσης ο incremental υπολογισμός δεν ενεργοποιείται, αφού το μέγεθος και η ολίσθηση του παραθύρου είναι ίσα.

Ο αυξητικός υπολογισμός ενεργοποιείται μόνο όταν η ολίσθηση του παραθύρου είναι μικρότερη από το μέγεθος.

2) PredicateUtil

Αυτή η κλάση χρησιμοποιείται από το SaberJoinRule και το SaberFilterRule για να μετατρέψουμε τα κατηγορήματα από ένα δεδομένο RelNode Φίλτρου ή Συνένωσης στις αντίστοιχες δομές που χρησιμοποιούνται από το SABER. Ο αλγόριθμος που χρησιμοποιείται είναι αναδρομικός, προκειμένου να υποστηρίξει πολύπλοκες συνθήκες γραμμένες σε SQL. Για παράδειγμα, σε αυτό το ερώτημα:

```
select *
from s.orders
where (((productid = 5) or (units > 25 and units <100)) and
        customerid=100) and 136>42
```

η συνθήκη στο where μετατρέπεται στο SABER σε:

(("2" = Constant 5) OR (("3" > Constant 25) AND ("3" < Constant 100))) AND ("4" = Constant 100) AND (Constant 136 > Constant 42)

3) ExpressionBuilder

Στο SaberProjectRule χρησιμοποιούμε την κλάση ExpressionBuilder για να κατασκευάσουμε σύνθετες εκφράσεις προβολής για το SABER. Η κλάση αυτή χρησιμοποιεί έναν αναδρομικό αλγόριθμο, προκειμένου να κατασκευάσει την αντίστοιχη προβολή από το δέντρο ενός τελεστή. Προς το παρόν υποστηρίζει μόνο απλές εκφράσεις με μαθηματικές πράξεις και τα RexCalls ceil και floor. Μας βοηθά επίσης να καθορίσουμε τη σημασιολογία παραθύρου στο RBStream. Για παράδειγμα:

```
select ((units+10) * 25 ) /100
from s.orders
```

μετατρέπεται στο SABER σε : (/(*(+($3, 10), 25), 100): int).

Χαρακτηριστικά Υλοποίησης

Σε αυτήν την τελευταία ενότητα, θα παρουσιάσουμε κάποιες βοηθητικές κλάσεις, που αυτοματοποιούν κάποιες βασικές λειτουργίες, και το γραφικό περιβάλλον του RBStream.

DataGenerator

Ο DataGenerator, όπως υποδηλώνει το όνομά του, χρησιμοποιείται για την παραγωγή δεδομένων για το σύστημά μας. Μπορεί είτε να παράγει dummy δεδομένα ή να χρησιμοποιηθεί για τη σύνδεση του συστήματος μας με πηγές δεδομένων.

SchemaConverter

Η κλάση αυτή μετατρέπει ένα συγκεκριμένο σχήμα του Calcite στο αντίστοιχο σχήμα στο SABER. Αυτή τη στιγμή, το SABER υποστηρίζει μόνο integers, floats και longs. Επομένως, όταν ο μετατροπέας βρίσκει ένα RelDataType που δεν υποστηρίζεται από το SABER, το μετατρέπει σε ακέραιο στο σχήμα του συστήματός μας από προεπιλογή. Ο SchemaConverter δεν υποστηρίζει ακόμα εμφωλευμένα σχήματα.

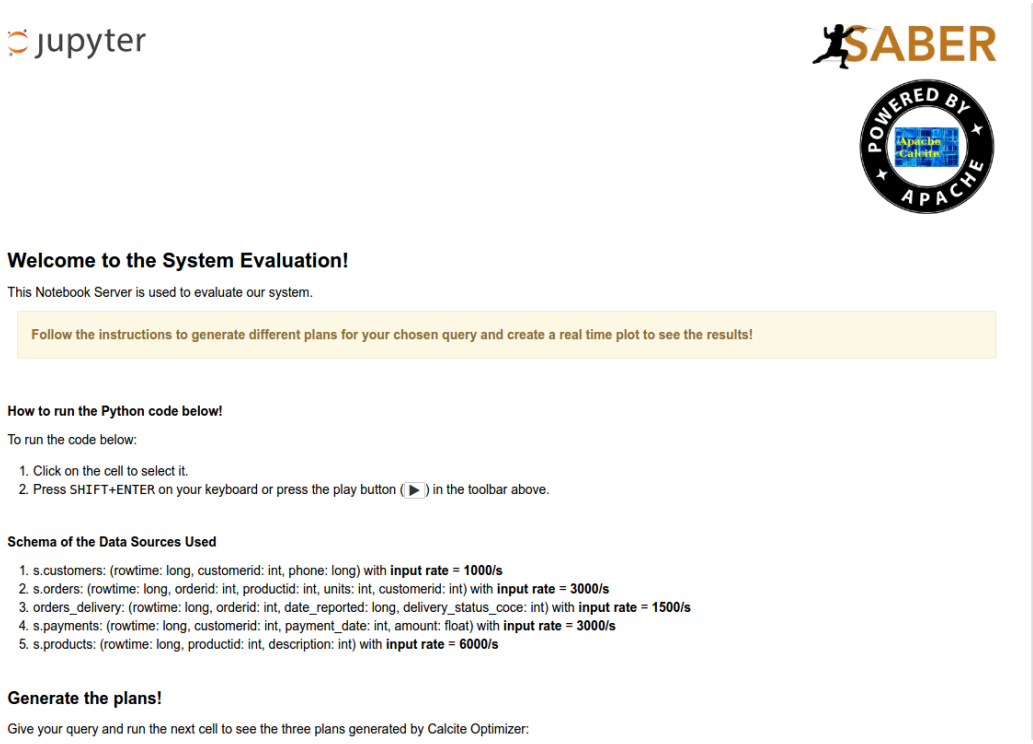
SystemConfig

Η κλάση αυτή χρησιμοποιείται για να διαμορφώσετε το SABER πριν από την εκτέλεση ενός ερωτήματος. Ανάλογα με τις παραμέτρους που επιλέγονται, το σύστημα μπορεί να αποτύχει να εκτελέσει ένα ερώτημα λόγω της έλλειψης αρκετής μνήμης για την αρχικοποίηση των Circular και Unbounded buffers ή το μέγεθος του Hash Table. Με αυτή την κλάση μπορούμε επίσης να ορίσουμε εάν χρησιμοποιείται η λειτουργία GPU και τον αριθμό των νημάτων του RBStream.

Το Γραφικό Περιβάλλον του RBStream

Για να παρουσιάσουμε το σύστημά μας, χρησιμοποιήσαμε ένα ολοκληρωμένο, πραγματικού χρόνου GUI που οι χρήστες θα χρησιμοποιούν για να αλληλεπιδράσουν με το σύστημα σε επίπεδο dataset, query και σύγκρισης. Χρησιμοποιήσαμε το Jupyter Notebook [62] για τη δημιουργία ενός διαδραστικού web interface για να παρουσιάσουμε το

σύστημα. Το UI μας ελέγχει ένα κεντρικό deployment του SABER σε μια εικονική μηχανή στον ~Okeanos IaaS [61].

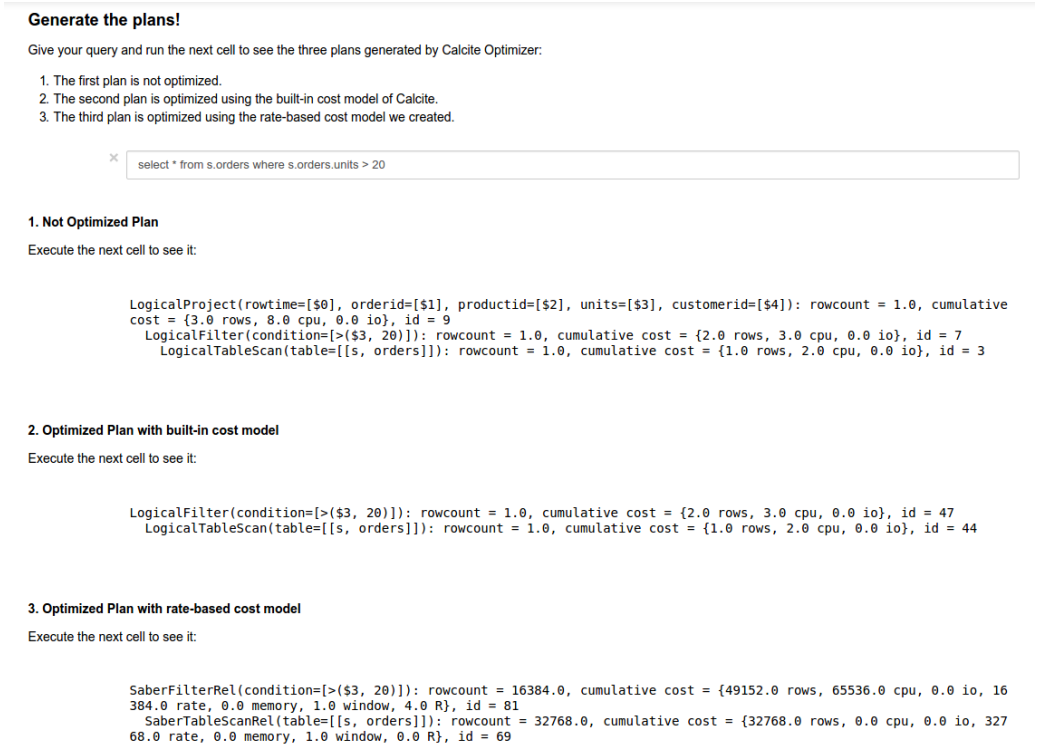


Εικ. 0.8.6.4.a, Jupyter Notebook.

Οι χρήστες έχουν την επιλογή να αλληλεπιδράσουν με δύο διαφορετικά σύνολα δεδομένων. Παρέχουμε:

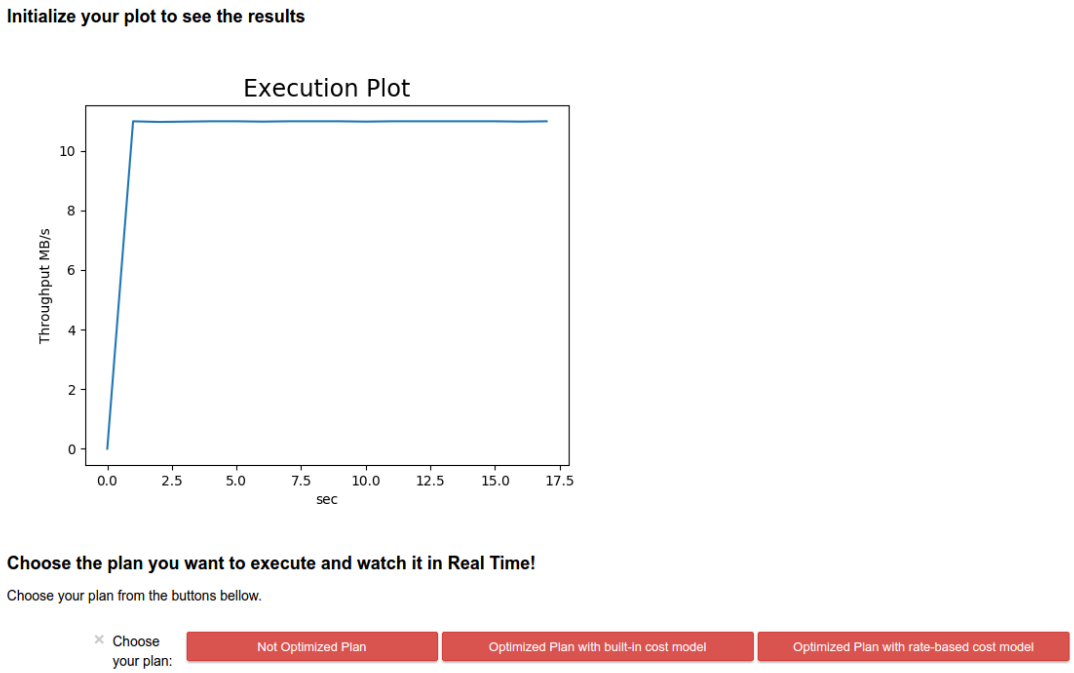
(i) Το Linear Road Benchmark (LRB) για την αξιολόγηση των επιδόσεων επεξεργασίας ροής δεδομένων. Το benchmark μοντελοποιεί ένα δίκτυο δρόμων με διόδια, στα οποία η επιβολή διοδίων εξαρτάται από το επίπεδο της κυκλοφοριακής συμφόρησης και το χρόνο της ημέρας. Οι πλειάδες στη ροή δεδομένων εισόδου υποδηλώνουν γεγονότα θέσης των οχημάτων σε μια λωρίδα αυτοκινητόδρομου, που οδηγούν με συγκεκριμένη ταχύτητα σε μία κατεύθυνση.

(ii) Συναλλαγές Αγορών-Παραγγελιών, με προσαρμοσμένα δεδομένων.



Εικ. 0.8.6.4.b, Δώσε το ερώτημα σου και πάρε τα τρία αντίστοιχα πλάνα.

Διαφορετικά ερωτήματα ροών μπορούν να εκτελεστούν στα φορτωμένα σύνολα δεδομένων. Ανεξάρτητα από τα σύνολο δεδομένων, οι χρήστες μπορούν να καθορίσουν το δικό τους ερώτημα χρησιμοποιώντας ένα πεδίο κειμένου, όπως στην εικόνα Εικ. 1.8.6.4.b. Για κάθε ένα από τα ερωτήματα, θα εμφανιστούν τα τρία σχέδια: ένα σχέδιο χωρίς βελτιστοποίηση, ένα σχέδιο με βελτιστοποίηση χρησιμοποιώντας το ενσωματωμένο μοντέλο κόστους και ένα σχέδιο με βελτιστοποίηση χρησιμοποιώντας το μοντέλο κόστους που βασίζεται στο ρυθμό εισροής δεδομένων. Στη συνέχεια, ο χρήστης μπορεί να επιλέξει ποιο πλάνο σκοπεύει να εκτελέσει. Τελικά, η πρόοδος θα είναι διαθέσιμη σε πραγματικό χρόνο μέσω ενός διαγράμματος, το οποίο παρουσιάζει τις μετρήσεις απόδοσης του υποβεβλημένου ερωτήματος.



Εικ. 0.8.6.4.c, Διάγραμμα πραγματικού χρόνου για το throughput.

0.9 Πειραματική Αξιολόγηση

Σε αυτήν την ενότητα θα τρέξουμε πειράματα με 10 αντιπροσωπευτικά ερωτήματα για την αξιολόγηση της αποτελεσματικότητας του RBStream.

0.9.1 Πειραματική Διάταξη

Η διαμόρφωση του συστήματος: Η πειραματική διάταξη αποτελείται από ένα OpenStack VM με 8x2GHz Intel Xeon E312xx πυρήνες CPU και 16GB μνήμη RAM.

Τα πλάνα που συγκρίναμε: Θα συγκρίνουμε την απόδοση εκτέλεσης των ερωτημάτων ροής με τρία διαφορετικά πλάνα: ένα πλάνο χωρίς βελτιστοποίηση, ένα βελτιστοποιημένο πλάνο με τη χρήση του ενσωματωμένου μοντέλου κόστους του Calcite και βελτιστοποιημένο πλάνο με τη χρήση του μοντέλου κόστους με βάση το ρυθμό εισροής των δεδομένων (rate-based).

Οι μετρήσεις που χρησιμοποιούνται για την αξιολόγηση μας είναι:

- Throughput: παρακολουθήσαμε το throughput με μια ενσωματωμένη λειτουργία του SABER.
- Latency: παρακολουθήσαμε το latency του συστήματος χρησιμοποιώντας την κλάση ResultCollector του SABER.
- Χρησιμοποίηση CPU: παρακολουθήσαμε την χρησιμοποίηση της cpu με το nmon [63].

Χρησιμοποιούμε ένα σύνολο δεδομένων για την αξιολόγηση μας: ένα Benchmark με συναλλαγές Αγορών-Παραγγελιών, με ελεγχόμενη κατανομή σε συνθετικά δεδομένα. Το σχήμα των πηγών δεδομένων ροής μας είναι:

```
s.customers Schema : Stream (rowtime: long, customerid: int, phone: long)
s.orders Schema : Stream (rowtime: long, orderid: int, productid: int, units: int, customerid: int)
s.orders_delivery Schema : Stream (rowtime: long, orderid: int, date_reported: long, delivery_status_code: int)
s.payments Schema : Stream (rowtime: long, customerid: int, payment_date: int, amount: float)
s.products Schema : Stream (rowtime: long, productid: int, description: int, price: float)
```

Προκειμένου να δοθεί μια άμεση σύγκριση μεταξύ των διαφόρων σχεδίων εκτέλεσης, θα δοκιμάσουμε την απόδοση των επιλεγμένων ερωτημάτων μας για όλα τα πιθανά σχέδια. Έχουμε ξεπεράσει το μη βελτιστοποιημένο πλάνο στα περισσότερα ερωτήματα, όταν τρέχουμε τα πειράματα μας σε small server configuration, λόγω της μικρότερης χρήσης της μνήμης και της δημιουργίας λιγότερων καθηκόντων ερωτημάτων για το SABER. Η διαφορά στην απόδοση οφείλεται κυρίως στη μείωση των ενδιάμεσων αποτελεσμάτων που δημιουργούνται, η οποία είναι κρίσιμη για τους υπολογισμούς συνεχόμενων ροών στη μνήμη (in memory) και επηρεάζει τόσο τη χρησιμοποίηση της μνήμης όσο και του κύκλους της CPU. Η βελτιστοποιημένη σειρά εκτέλεσης των τελεστών όχι μόνο αυξάνει το ρυθμό παραγωγής δεδομένων εξόδου, αλλά κάνει και τους υπολογισμούς εφικτούς σε πολλές περιπτώσεις, δεδομένου ότι μειώνει τις απαιτήσεις μνήμης. Επομένως, για ερωτήσεις μεγάλης έντασης με βάση τα δεδομένα, το σύστημά μας δίνει ένα εφικτό πλάνο που καθιστά δυνατή την εκτέλεση του ερωτήματος.

0.9.2 Πειράματα

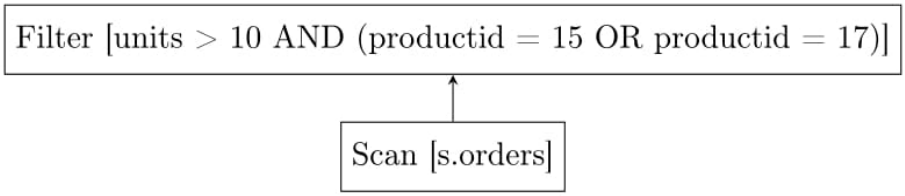
Θα δοκιμάσουμε την εκτέλεση ορισμένων ερωτημάτων, τα οποία είναι αντιπροσωπευτικά των περιπτώσεων χρήσης βελτιστοποίησης που μπορεί να χειριστεί το RBStream. Τα ερωτήματα που χρησιμοποιούνται, αν και απλά, καλύπτουν τις βασικές λειτουργίες SQL και μπορούν να χρησιμοποιηθούν για να σχηματίσουν πιο σύνθετες περιπτώσεις χρήσης. Με αυτά τα πειράματα, θα συγκρίνουμε την απλοϊκή εκτέλεση ερωτημάτων συγκριτικά με τη βελτιστοποιημένη εκδοχή τους. Για τα περισσότερα από αυτά τα ερωτήματα, τόσο το ενσωματωμένο όσο και το δικό μας μοντέλο κόστους δίνουν τα ίδια αποτελέσματα, λόγω της φύσης των κανόνων βελτιστοποίησης που εφαρμόζονται στο αρχικό σχέδιο. Κανόνες που ωθούν προς τα κάτω τελεστές (όπως φίλτρα, προβολές ή συναρτήσεις συνάθροισης), συγχωνεύουν ή καταργούν διαδοχικούς τελεστές και απλοποιούν SQL εκφράσεις, είναι ευριστικοί και μπορούν να εφαρμοστούν ανεξάρτητα από το μοντέλο κόστους που χρησιμοποιείται από το βελτιστοποιητή. Επομένως, σε αυτές τις περιπτώσεις θα παρουσιάσουμε μόνο τα αποτελέσματα του μοντέλου κόστους με βάση το ρυθμό εισροής δεδομένων. Ωστόσο, άλλοι κανόνες, όπως αυτοί που επηρεάζουν τη σειρά συνένωσης ροών, μπορούν να επωφεληθούν από το νέο μοντέλο κόστους.

0.9.2.1) Με το επόμενο ερώτημα δοκιμάζουμε τους κανόνες

FilterProjectTransposeRule, FilterMergeRule και ProjectMergeRule.

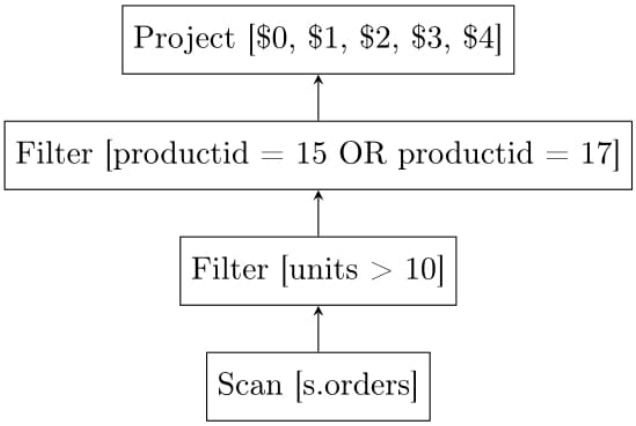
Q1:
```
select * from (
select *
from s.orders
where s.orders.units > 10) as s1
where s1.productid = 15 or s1.productid = 17;
```

a) Το πλάνο που προέκυψε από το rate-based μοντέλο κόστους είναι:



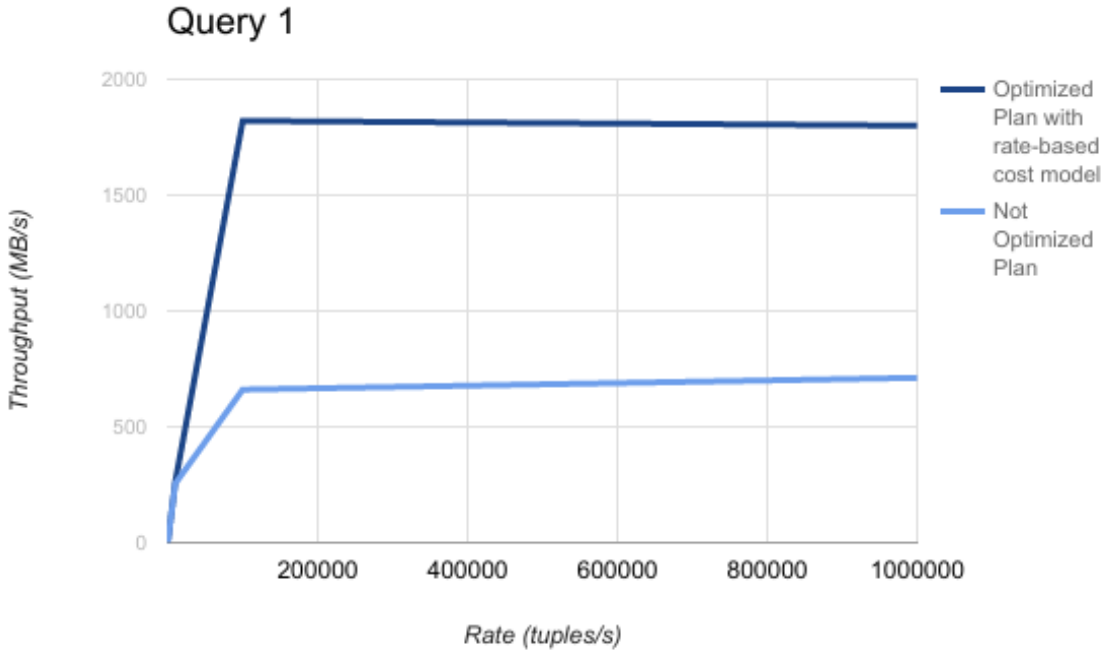
Εικ. 0.9.2.1.a, rate-based πλάνο για το Ερώτημα 1.

b) Το πλάνο που προέκυψε χωρίς βελτιστοποίηση είναι:



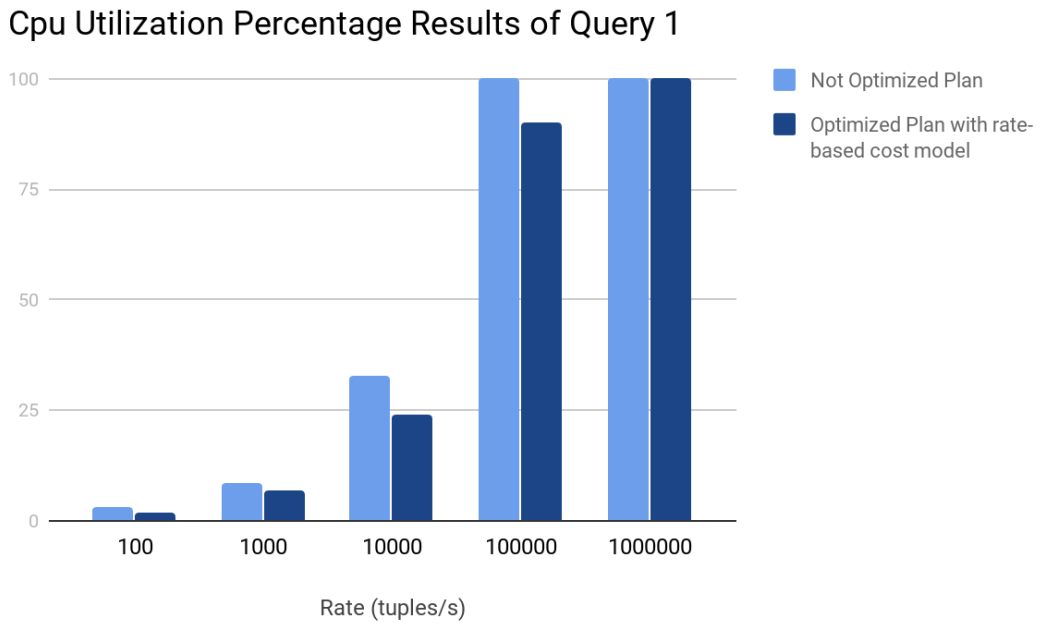
Εικ. 0.9.2.1.b, μη βελτιστοποιημένο πλάνο για το Ερώτημα 1.

Το configuration που χρησιμοποιήσαμε για τις επόμενες μετρήσεις που πήραμε ήταν: 2 Threads, 64 MB Circular Buffer Size και 128 MB Unbounded Buffer Size.



Εικ. 0.9.2.1.c, Τα αποτελέσματα του Throughput για το Ερώτημα 1.

Στο παραπάνω γράφημα, το πρώιμο φιλτράρισμα των πλειάδων που θα "πεταγόντουσαν" διαφορετικά από τους τελεστές που ακολουθούν, έχει ως αποτέλεσμα μεγαλύτερο throughput (περίπου 2,7 φορές περισσότερα MB/s). Αυτό συμβαίνει επειδή μεταφέρονται λιγότερα ενδιάμεσα αποτελέσματα και ακριβέστερα, στην περίπτωση αυτή, αρχικοποιούνται λιγότεροι χειριστές για τον υπολογισμό της τελικού αποτελέσματος. Μπορεί επίσης να παρατηρηθεί ότι με την επιλεγμένη διαμόρφωσή μας, καθώς αυξάνουμε τον ρυθμό εισόδου, η απόδοση παραμένει σταθερή και αρχίζει να μειώνεται σταθερά κατά έναν μικρό παράγοντα από ένα ορισμένο όριο και επάνω.



Εικ. 0.9.2.1.d, Τα αποτελέσματα της Χρήσης Cpu για το Ερώτημα 1.

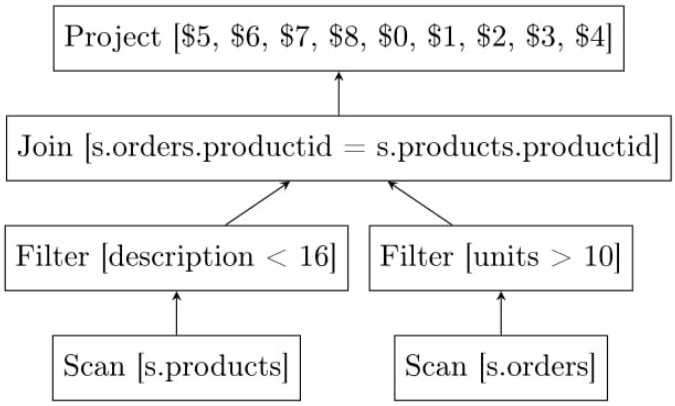
Αυτό το ερώτημα χρησιμοποιεί stateless τελεστές και ως αποτέλεσμα, η αξιοποίηση της CPU είναι ελαφρώς μικρότερη στο σχέδιο βελτιστοποίησης βάσει ρυθμού σε σύγκριση με τη μη βελτιστοποιημένη έκδοση.

0.9.2.2) Το επόμενο ερώτημα απεικονίζει τη χρήση του κανόνα FilterJoinRule, που ωθεί έναν τελεστή φίλτρου μέσω ενός τελεστή συνένωσης στα παιδιά του.

Q2:
```
select *
from  s.products join s.orders
```

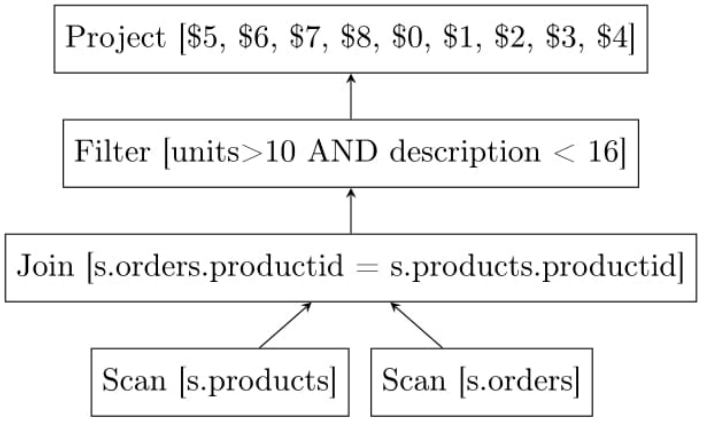
```
on s.orders.productid = s.products.productid
where units>10 and description < 16 ;
```

a) Το πλάνο που παίρνουμε με rate-based βελτιστοποίηση είναι:



Εικ. 0.9.2.2.a, Το rate-based πλάνο του Ερωτήματος 2.

b) Το πλάνο χωρίς βελτιστοποίηση:



Εικ. 0.9.2.2.b, το μη βελτιστοποιημένο πλάνο του Ερωτήματος 2.

Το πλάνο 1 είναι βελτιστοποιημένο με το rate-based μοντέλο κόστους. Το πλάνο 2 δεν είναι βελτιστοποιημένο. Στα κελιά με (-) το σύστημα ξεπέρασε τα όρια μνήμης και τερματίστηκε η λειτουργία του.

	Threads	Circular Buffer	Unbounded Buffer	Data Source 1 (tuples/s)	Data Source 2 (tuples/s)	Selectivity	Throughput Plan1	Latency Plan 1 (s)	CPU utilization	Throughput Plan 2	Latency Plan 2 (s)	CPU utilization Plan 2

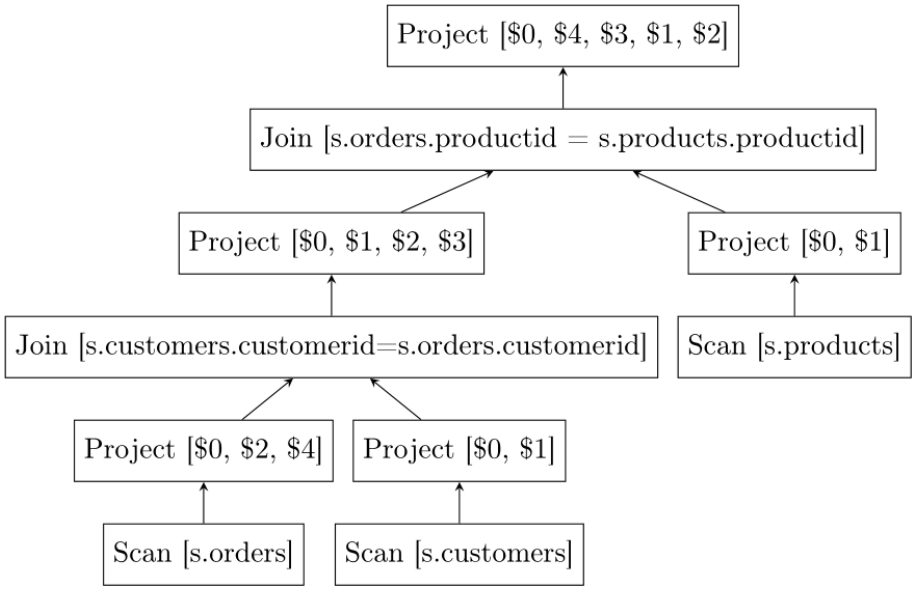
		(MB)	(MB)				(MB/s)		Plan 1	(MB/s)		
Q2,1	2	64	128	300	200	1/24	13,75	0,12	15%	13,16	0,032	45%
Q2,2	2	64	128	400	400	1/24	22,17	0,019	35%	20,73	0,019	70%
Q2,3	2	64	128	500	400	1/24	24,10			-	-	-
Q2,4	2	64	128	800	600	1/24	38,51			-	-	-
Q2,5	2	64	128	800	800	1/24	38,22			-	-	-
Q2,6	1	64	128	400	400	1/24	21,7	0,02	70%	-	-	-

Πίνακας 0.9.2.2, Μετρικές του Ερωτήματος 2.

0.9.2.3) Με αυτό το ερώτημα, δείχνουμε το αποτέλεσμα της αναδιάταξης των συνενώσεων, ακόμα και με 2 συνενώσεις, κατά την διάρκεια εκτέλεσης.

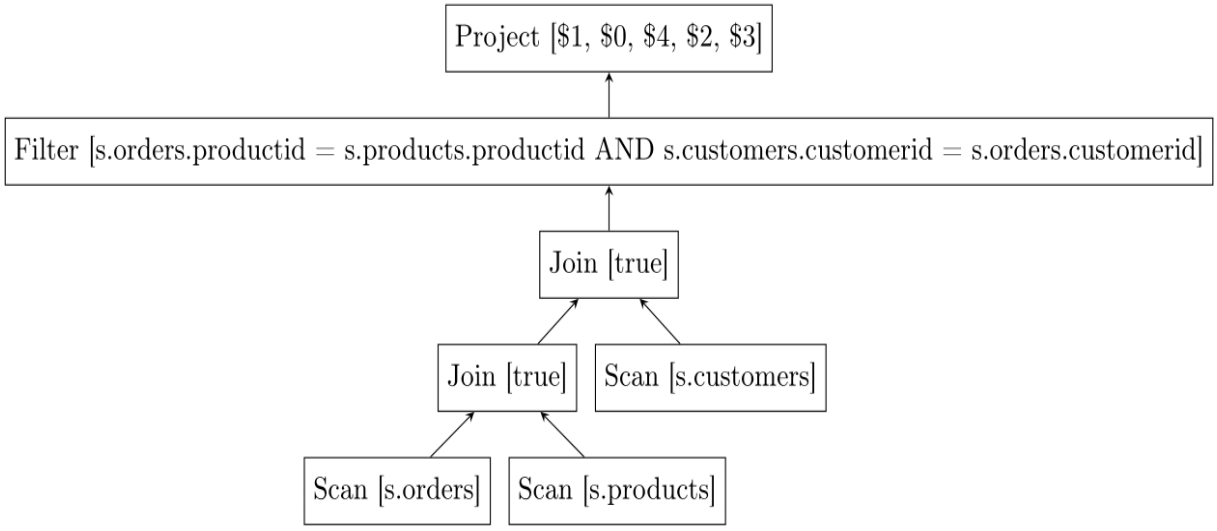
Q3:
```
select s.orders.rowtime, s.products.rowtime,
s.customers.rowtime,
s.orders.productid,s.orders.customerid
from  s.products, s.orders, s.customers
where s.orders.productid = s.products.productid and
s.customers.customerid=s.orders.customerid;
```

a) Το πλάνο με την rate-based βελτιστοποίηση είναι:



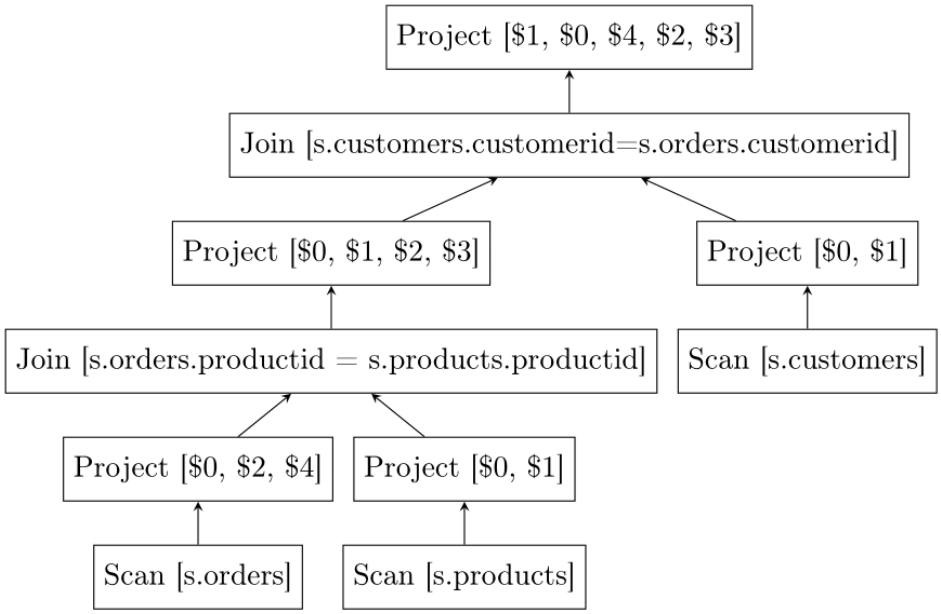
Εικ. 0.9.2.3.a, rate-based βελτιστοποιημένο πλάνο του Ερωτήματος 3.

b) Το πλάνο χωρίς βελτιστοποίηση είναι (δεν μπορούσαμε να εκτελέσουμε αυτό το πλάνο εξαιτίας των απαιτήσεων μνήμης που είχε):



Εικ. 0.9.2.3.b, μη βελτιστοποιημένο πλάνο του Ερωτήματος 3.

c) Το πλάνο με τη built-in βελτιστοποίηση:



Εικ. 0.9.2.3.c, built-in βελτιστοποιημένο πλάνο του ερωτήματος 3.

Το πλάνο 1 είναι βελτιστοποιημένο με το rate-based μοντέλο κόστους. Το πλάνο 2 δεν είναι βελτιστοποιημένο. Στα κελιά με (-) το σύστημα ξεπέρασε τα όρια μνήμης και τερματίστηκε η λειτουργία του.

	Threads	Circular Buffer (MB)	Unbounded Buffer (MB)	Data Source 1 (tuples/s)	Data Source 2 (tuples/s)	Data Source 3 (tuples/s)	Selectivity	Throughput Plan 1 (MB/s)	Latency Plan 1 (s)	CPU utilization Plan1	Throughput Plan 2 (MB/s)	Latency Plan 2 (s)	CPU utilization Plan 2
Q3,1	2	128	128	100	400	400	1/24	27,16	4,76	9%	23,50	5,67	75%
Q3,2	2	128	128	200	400	400	1/24	26,84	1,6	20%	27,22	2,57	85%
Q3,3	2	64	128	200	600	600	1/24	38,9	3,5	20%	-	-	-
Q3,4	2	64	128	200	800	600	1/24	43	3,7	20%	-	-	-
Q3,5	2	64	256	200	800	800	1/24	49	3,2	>80%	-	-	-
Q3,6	2	64	200	350	700	600	1/24	43	1,6	>80%	-	-	-
Q3,7	2	64	128	200	600	300	1/24	30	2,03	40%	30	3,33	60%
Q3,8	2	64	128	200	600	300	1/14	31	0,68	10%	31	0,90	35%

Q3,9	2	64	128	200	600	300	1/10	29,92	0,44	50%	30	0,62	70%
Q3,10	1	128	128	100	400	400	1/24	24,9	5,15	15%	-	-	-
Q3,11	1	64	128	200	600	300	1/24	29,98	3,88	25%	-	-	-

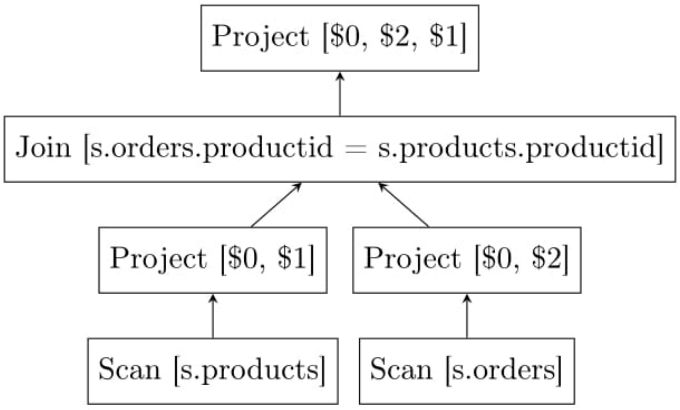
Πίνακας 0.9.2.3, Μετρικές του Ερωτήματος 3.

0.9.2.4) Αυτό το ερώτημα δείχνει το αποτέλεσμα του κανόνα

ProjectJoinTransposeRule.

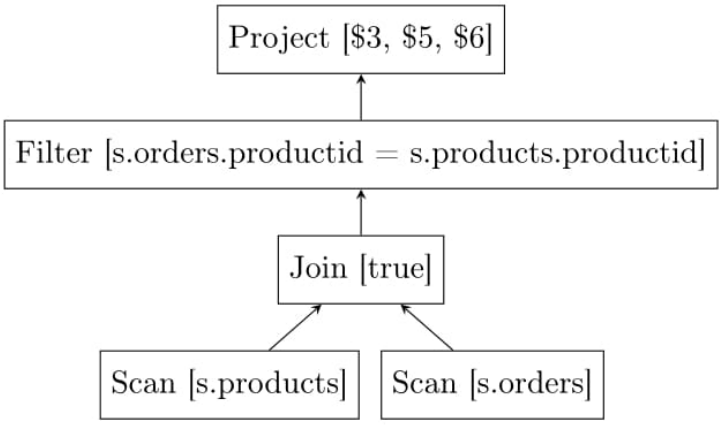
Q4:
```
select s.orders.rowtime, s.products.rowtime, s.orders.productid
from  s.products, s.orders
where s.orders.productid = s.products.productid;
```

a) Το πλάνο αυτό προέκυψε από το rate-based μοντέλο κόστους:



Εικ. 0.9.2.4.a, rate-based πλάνο για το ερώτημα 4.

b) Το πλάνο χωρίς βελτιστοποίηση είναι:



Εικ. 0.9.2.4.b, μη βελτιστοποιημένο πλάνο του ερωτήματος 4 (μη εφικτό).

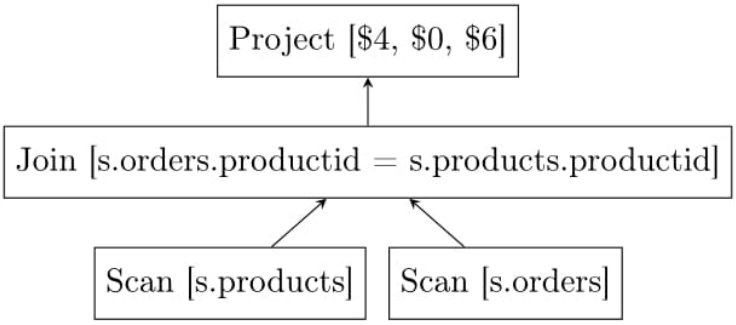
Η εκτέλεση αυτού του πλάνου δεν είναι εφικτή (το μέγεθος των buffer δεν είναι αρκετό), γιατί προσπαθούμε να συνενώσουμε δύο ροές δεδομένων με τη συνθήκη [true] και δημιουργείται μεγάλη ποσότητα ενδιάμεσων αποτελεσμάτων που δεν θα χρησιμοποιηθούν:

```
java.nio.BufferOverflowException
      at java.nio.HeapByteBuffer.put(HeapByteBuffer.java:183)
      at
uk.ac.imperial.lsds.saber.buffers.UnboundedQueryBuffer.put(UnboundedQue
ryBuffer.java:156)
      at
uk.ac.imperial.lsds.saber.buffers.CircularQueryBuffer.appendBytesTo(Cir
cularQueryBuffer.java:296)
      at
uk.ac.imperial.lsds.saber.cql.operators.cpu.ThetaJoin.processData(Theta
Join.java:154)
...
```

Για να μπορεί να εκτελεστεί πρέπει να το ξαναγράψουμε ως εξής:

```
select s.orders.rowtime, s.products.rowtime, s.orders.productid
from  s.products join s.orders
on s.orders.productid = s.products.productid;
```

και παίρνουμε το ακόλουθο πλάνο:



Εικ. 0.9.2.4.c, μη βελτιστοποιημένο πλάνο του ερωτήματος 4.

Το πλάνο 1 είναι βελτιστοποιημένο με το rate-based μοντέλο κόστους. Το πλάνο 2 δεν είναι βελτιστοποιημένο. Στα κελιά με (-) το σύστημα ξεπέρασε τα όρια μνήμης και τερματίστηκε η λειτουργία του.

	Threads	Circular Buffer (MB)	Unbounded Buffer (MB)	Data Source 1 (tuples/s)	Data Source 2 (tuples/s)	Selectivity	Throughput Plan 1 (MB/s)	Latency Plan 1 (s)	CPU utilization Plan 1	Throughput Plan 2 (MB/s)	Latency Plan 2 (s)	CPU utilization Plan 2
Q4,1	1	64	128	100	100	1/24	5,52	0,13	13%	5,49	0,05	20%
Q4,2	1	64	128	400	250	1/24	17,01	0,03	50%	17,31	0,006	65%
Q4,3	1	64	128	400	280	1/24	17,9	0,02	60%	-	-	-
Q4,4	1	64	128	500	400	1/24	-	-	-	-	-	-
Q4,5	1	64	128	800	800	1/24	-	-	-	-	-	-
Q4,6	2	64	128	400	280	1/24				18,11	0,005	65%

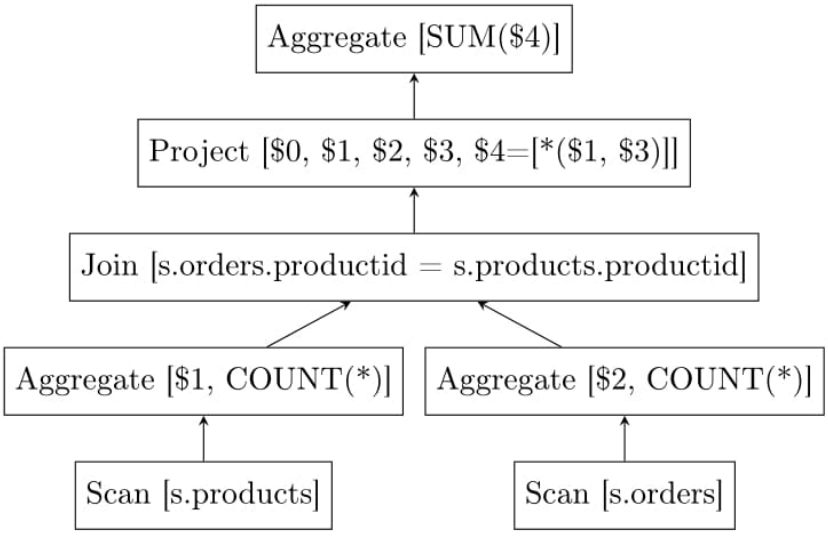
Πίνακας 0.9.2.4, Μετρικές του Ερωτήματος 4.

0.9.2.5) Με αυτό το ερώτημα παρουσιάζουμε τη χρήση του κανόνα AggregateJoinTransposeRule.

Q5:
```
select count(*)
from s.orders
```

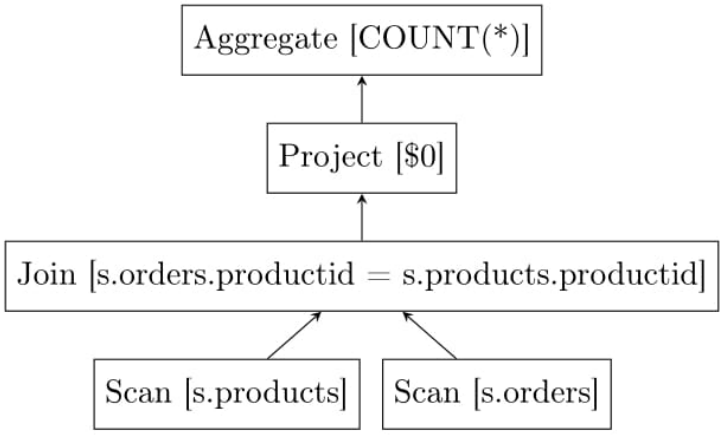
```
join s.products
on s.orders.productid = s.products.productid;
```

a) Όταν χρησιμοποιούμε το rate-based μοντέλο κόστους παίρνουμε το ακόλουθο πλάνο:



Εικ. 0.9.2.5.a, rate-based πλάνο του ερωτήματος 5.

b) Χωρίς βελτιστοποίηση παίρνουμε:



Εικ. 0.9.2.5.b, μη βελτιστοποιημένο πλάνο του ερωτήματος 5.

Το πλάνο 1 είναι βελτιστοποιημένο με το rate-based μοντέλο κόστους. Το πλάνο 2 δεν είναι βελτιστοποιημένο. Στα κελιά με (-) το σύστημα ξεπέρασε τα όρια μνήμης και τερματίστηκε η λειτουργία του.

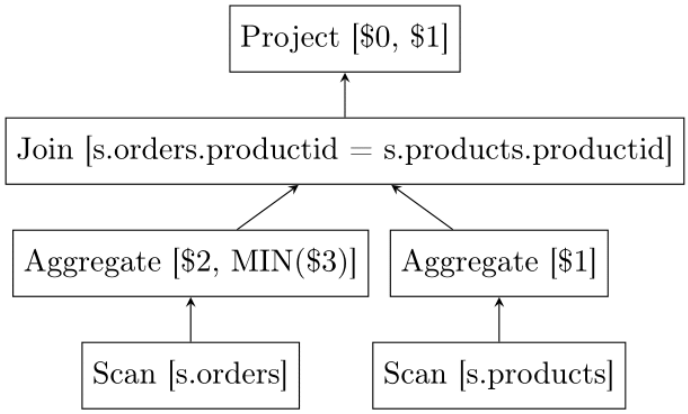
	Threads	Circular Buffer (MB)	Unbounded Buffer (MB)	Data Source 1 (tuples/s)	Data Source 2 (tuples/s)	Selectivity	Throughput Plan1 (MB/s)	Latency Plan 1 (s)	CPU utilization Plan 1	Throughput Plan2 (MB/s)	Latency Plan 2 (s)	CPU utilization Plan 2
Q5,1	1	32	128	100	100	1/16	5,28	0,1	10%	5,25	0,26	30%
Q5,2	1	32	256	200	200	1/16	9,96		30%	10,52		80%
Q5,3	1	32	380	400	400	1/16	21,93			-	-	-
Q5,4	1	32	380	500	400	1/16	24,67			-	-	-
Q5,5	2	32	380	400	400	1/16	21,93		16%	-	-	-

Πίνακας 0.9.2.5, Μετρικές του Ερωτήματος 5.

0.9.2.6) Αυτό το ερώτημα χρησιμοποιεί τους κανόνες AggregateJoinTransposeRule και AggregateProjectMerge.

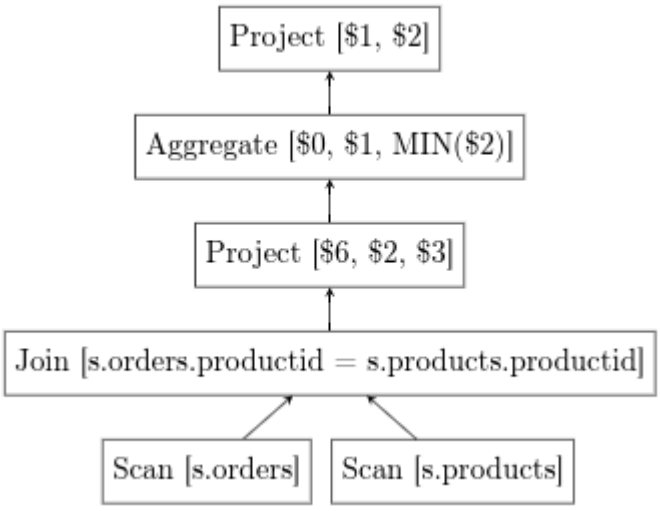
Q6:
```
select o.productid, min(units)
from s.orders as o
join s.products as p
on p.productid=o.productid
group by p.productid,o.productid;
```

a) Το πλάνο που παράγεται από τη rate-based βελτιστοποίηση είναι:



Εικ. 0.9.2.6.a, rate-based βελτιστοποίηση του ερωτήματος 6.

b) Στην περίπτωση που δεν χρησιμοποιήσουμε βελτιστοποίηση έχουμε:



Εικ. 0.9.2.6.b, μη βελτιστοποιημένο πλάνο 6.

Το πλάνο 1 είναι βελτιστοποιημένο με το rate-based μοντέλο κόστους. Το πλάνο 2 δεν είναι βελτιστοποιημένο. Στα κελιά με (-) το σύστημα ξεπέρασε τα όρια μνήμης και τερματίστηκε η λειτουργία του.

	Threads	Circular Buffer (MB)	Unbounded Buffer (MB)	Data Source 1 (tuples/s)	Data Source 2 (tuples/s)	Selectivity	Throughput Plan1 (MB/s)	Latency Plan 1 (s)	CPU utilization Plan 1	Throughput Plan2 (MB/s)	Latency Plan 2 (s)	CPU utilization Plan 2
Q6,1	1	32	128	100	100	1/16	5,38	0,1	13%	5	0,26	35%
Q6,2	1	32	256	200	200	1/16	10,97		30%	10,29		85%
Q6,3	1	32	360	400	400	1/16	21,59		35%	-	-	-
Q6,4	1	32	360	500	400	1/16	22,55			-	-	-
Q6,5	2	32	360	400	400	1/16	21,59		16%	-	-	-

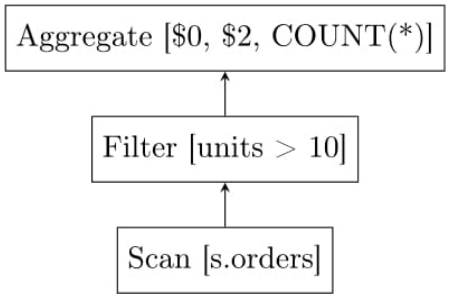
Πίνακας 0.9.2.6, Μετρικές του Ερωτήματος 6.

0.9.2.7) Με αυτό το ερώτημα αξιολογούμε τον κανόνα

AggregateProjectPullUpConstantsRule.

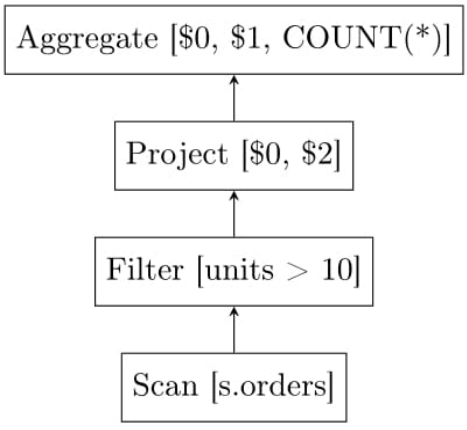
Q7:
```
select rowtime,productid, count(*)
from s.orders
where units > 10
group by rowtime,productid;
```

a) Το πλάνο που παίρνουμε από τη rate-based βελτιστοποίηση:



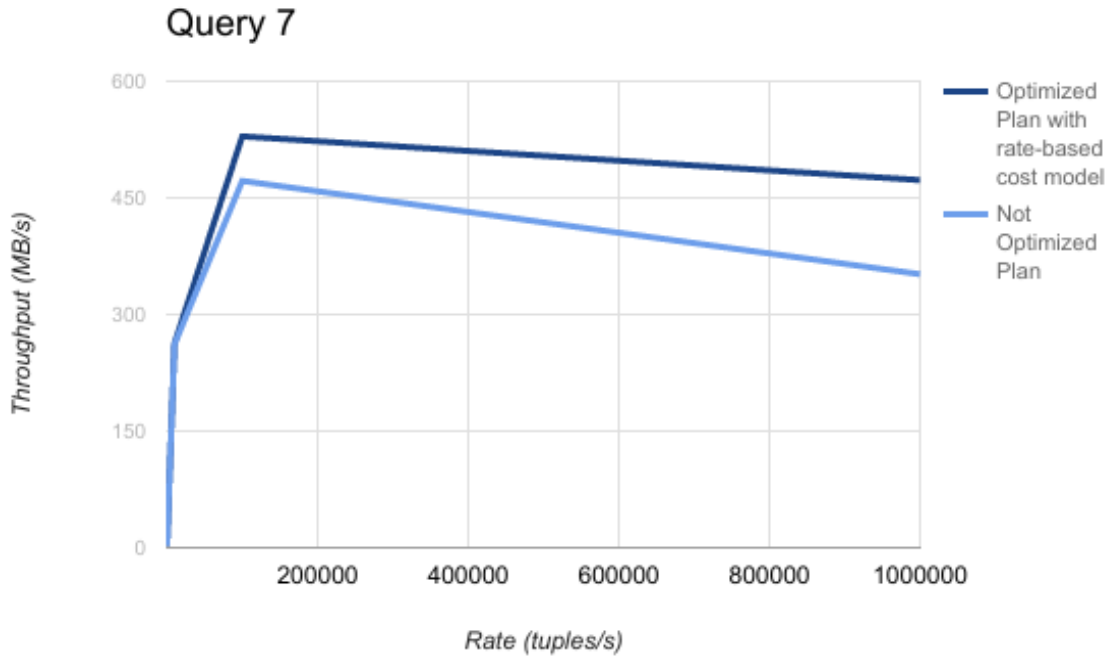
Εικ. 0.9.2.7.a, rate-based βελτιστοποίηση του ερωτήματος 7.

b) Το πλάνο που παίρνουμε χωρίς βελτιστοποίηση:



Εικ. 0.9.2.7.b, μη βελτιστοποιημένο πλάνο του ερωτήματος 7.

Το configuration που χρησιμοποιήσαμε για τις επόμενες μετρήσεις που πήραμε ήταν: 2 Threads, 64 MB Circular Buffer Size και 128 MB Unbounded Buffer Size.



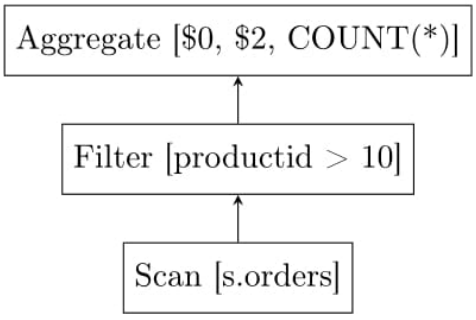
Εικ. 0.9.2.7.c, Throughput Αποτελέσματα για το ερώτημα 7.

0.9.2.8) Με αυτό το ερώτημα παρουσιάζουμε τον κανόνα

FilterAggregateTransposeRule, που ωθεί ένα filter κάτω από ένα aggregate.

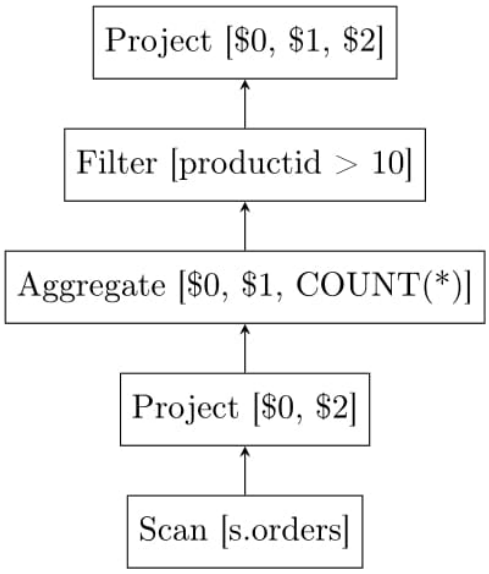
Q8:
```
select * from (
select rowtime,productid, count(*)
from s.orders
group by rowtime,productid
) as o
where o.productid > 10;
```

a) Όταν χρησιμοποιήσαμε το rate-based μοντέλο κόστους:



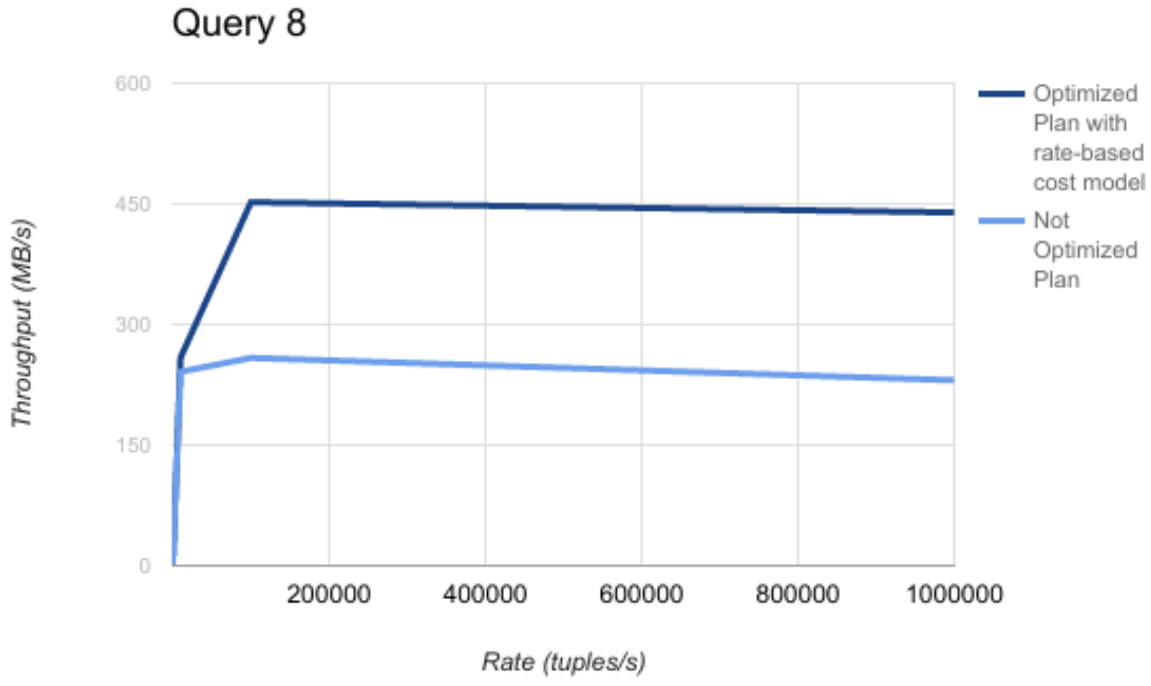
Εικ. 0.9.2.8.a, rate-based πλάνο του ερωτήματος 8.

b) Όταν δεν χρησιμοποιήσαμε βελτιστοποίηση:



Εικ. 0.9.2.8.b, μη βελτιστοποιημένο πλάνο του ερωτήματος 8.

Το configuration που χρησιμοποιήσαμε για τις επόμενες μετρήσεις που πήραμε ήταν: 2 Threads, 64 MB Circular Buffer Size και 128 MB Unbounded Buffer Size.

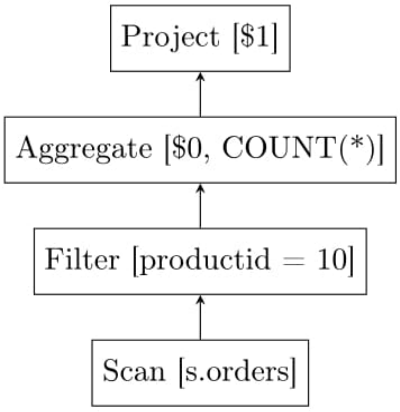


Εικ. 0.9.2.8.c, Throughput αποτελέσματα του ερωτήματος 8.

0.9.2.9) Αυτό το ερώτημα δείχνει το αποτέλεσμα του κανόνα AggregateConstantKeyRule.

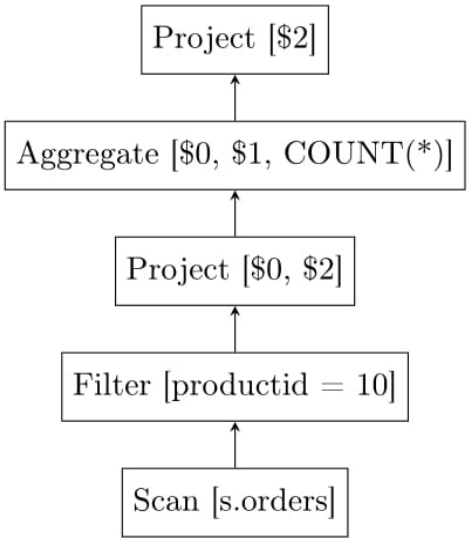
Q9:
```
select count(*)
from s.orders
where productid=10
group by rowtime,productid;
```

a) Το πλάνο που προκύπτει από το rate-based μοντέλο κόστους είναι:



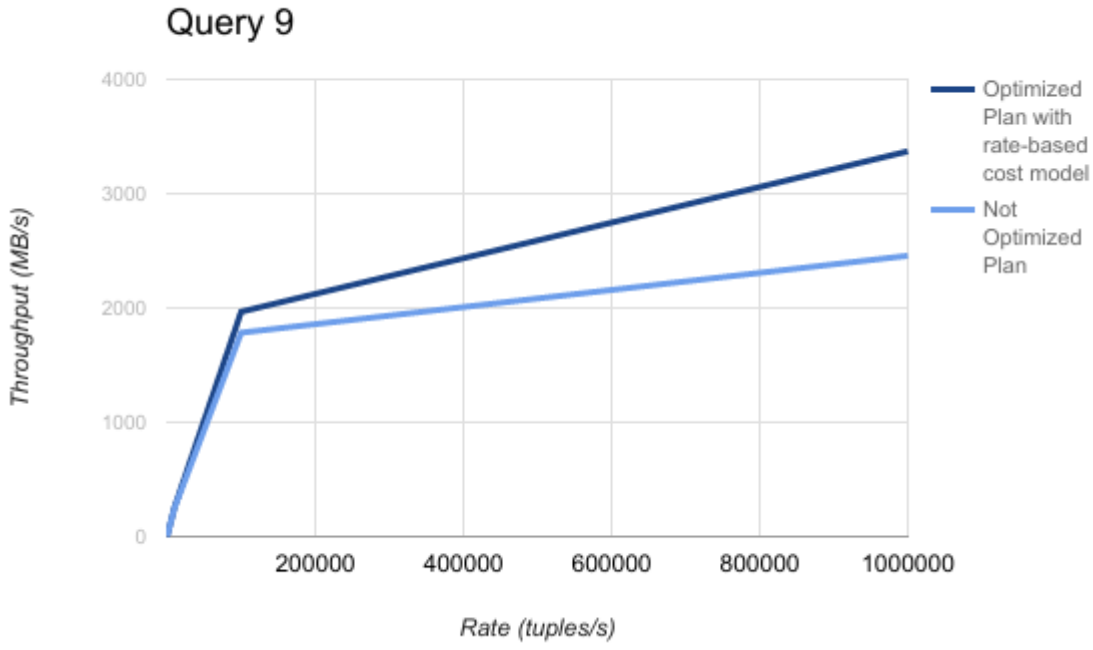
Εικ. 0.9.2.9.a, rate-based πλάνο του ερωτήματος 9.

b) Το πλάνο που προέκυψε χωρίς βελτιστοποίηση:



Εικ. 0.9.2.9.b, μη βελτιστοποιημένο πλάνο του ερωτήματος 9.

Το configuration που χρησιμοποιήσαμε για τις επόμενες μετρήσεις που πήραμε ήταν: 2 Threads, 64 MB Circular Buffer Size και 128 MB Unbounded Buffer Size.

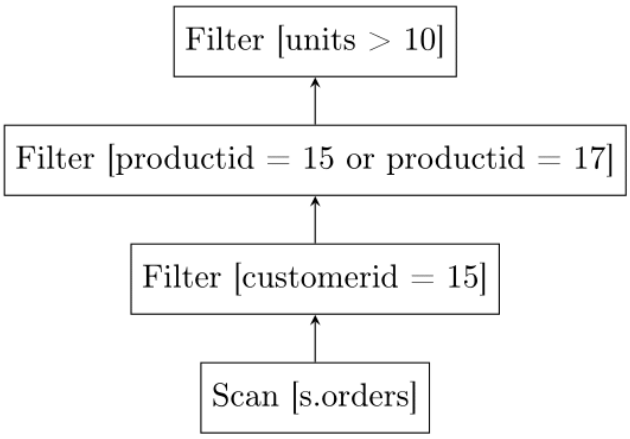


Εικ. 0.9.2.9.c, Throughput αποτελέσματα του ερωτήματος 9.

0.9.2.10) Με αυτό το ερώτημα θα εξετάσουμε το αποτέλεσμα του κανόνα FilterPushThroughFilterRule, που δημιουργήσαμε, συγκριτικά με τον κανόνα FilterMergeRule.

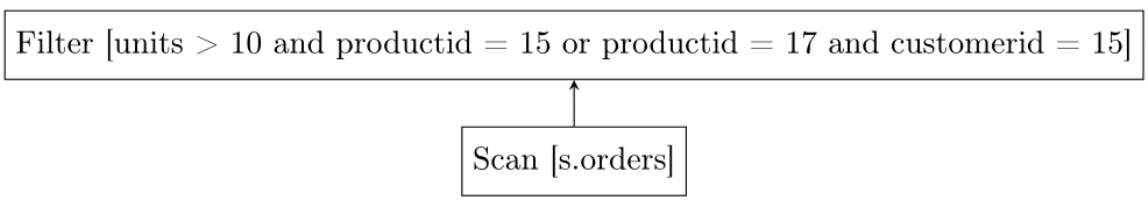
Q10:
```
select * from
(select * from (
select *
from s.orders
where productid = 15 or productid = 17) as s1
where s1.units > 10) as s2
where s2.customerid  = 15;
```

a) Το παραγόμενο πλάνο με τον κανόνα FilterPushThroughFilterRule είναι:



Εικ. 0.9.2.10.a, βελτιστοποιημένο πλάνο με τον κανόνα FilterPushThroughFilterRule για το ερώτημα 10.

b) Το πλάνο που δημιουργήθηκε με τη χρήση του κανόνα FilterMergeRule είναι:



Εικ. 0.9.2.10.b, βελτιστοποιημένο πλάνο με τον κανόνα FilterMergeRule για το ερώτημα 10.

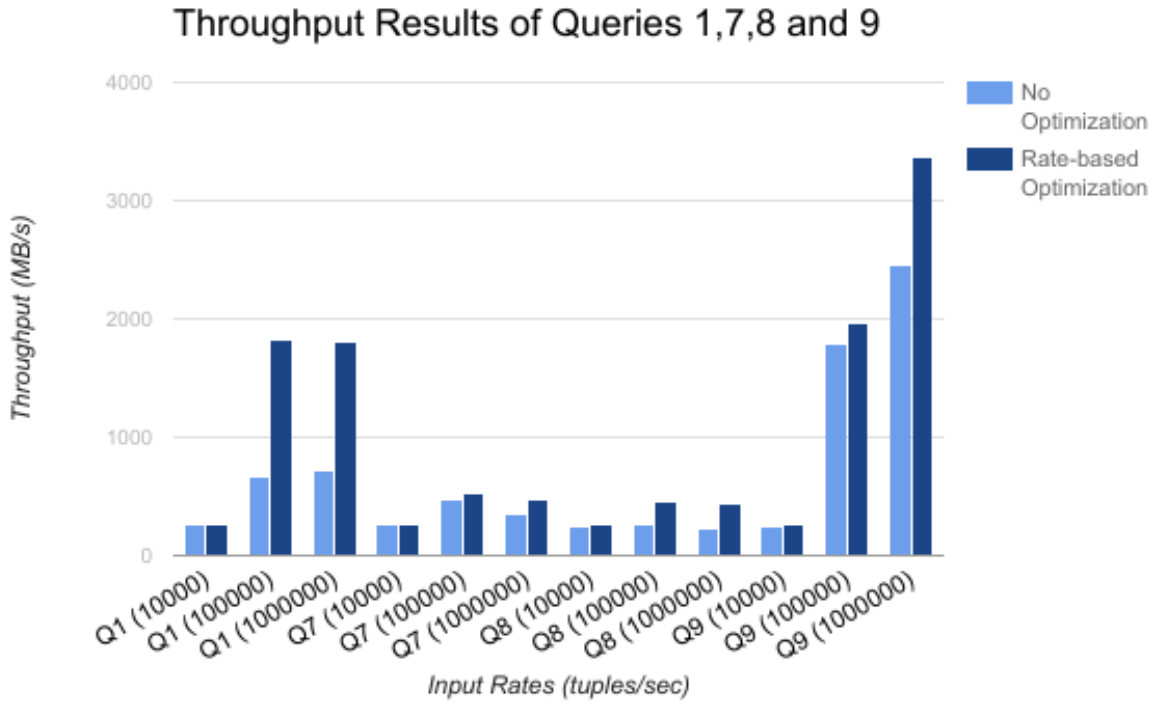
Το configuration που χρησιμοποιήσαμε για τις επόμενες μετρήσεις που πήραμε ήταν: 2 Threads, 64 MB Circular Buffer Size και 128 MB Unbounded Buffer Size.

Εικ. 0.9.2.10.c, Throughput αποτελέσματα του ερωτήματος 10.

0.9.3 Ανάλυση αποτελεσμάτων

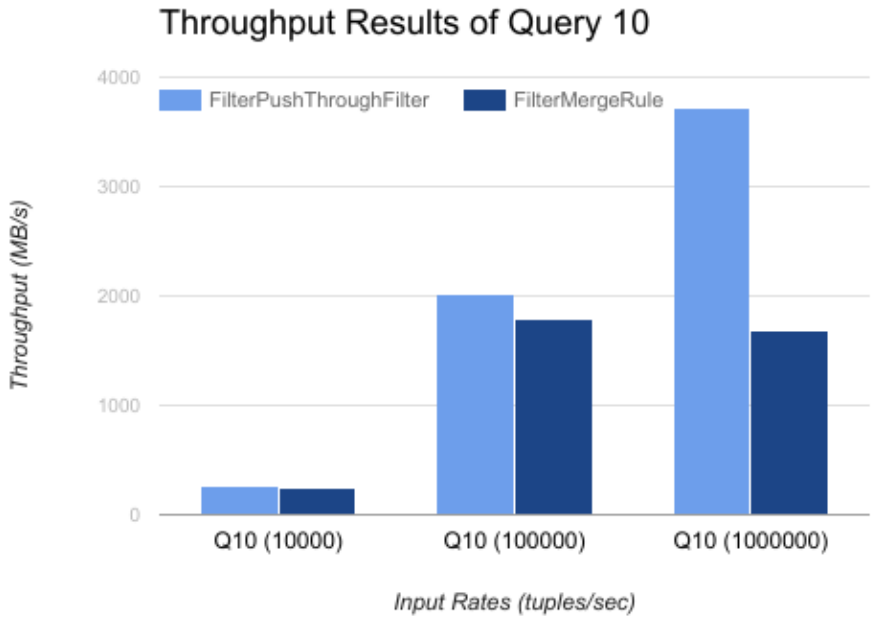
Από τα πειραματικά αποτελέσματα μας, παρατηρούμε ότι έχουμε δύο κατηγορίες ερωτημάτων με βάση τις μετρικές που έχουμε συλλέξει: τα ερωτήματα χωρίς και με τελεστή συνένωσης. Στην πρώτη κατηγορία, τα βελτιστοποιημένα σχέδια ξεπερνούν τα μη βελτιστοποιημένα, όσον αφορά το throughput των πηγών δεδομένων εισόδου. Είναι προφανές, ότι όταν η τιμή εισόδου είναι αρκετά μικρή για να αντιμετωπιστεί από το σύστημά μας, το throughput και για τις δύο περιπτώσεις είναι σχεδόν το ίδιο. Ωστόσο, καθώς αυξάνεται ο ρυθμός εισροής δεδομένων, το σύστημα επιτυγχάνει υψηλότερη απόδοση με το βελτιστοποιημένο σχέδιο, λόγω των λιγότερων ενδιάμεσων αποτελεσμάτων που δημιουργούνται και της μικρότερης επιβάρυνσης σε cpu cycles (απαιτούνται λιγότερα

tasks και υπολογισμοί). Ένα μεγάλο ποσοστό των ενδιάμεσων αποτελεσμάτων που παράγονται από τα μη βελτιστοποιημένα σχέδια συνήθως απορρίπτονται (τα "πετάμε") επειδή δεν χρειάζονται για τον υπολογισμό του τελικού αποτελέσματος. Για παράδειγμα, όταν ωθήσουμε προς τα κάτω ένα φίλτρο μέσα από έναν άλλο τελεστή, εξασφαλίζουμε ότι τα δεδομένα που περνούν στα επόμενα στάδια επεξεργασίας θα συμμετέχουν στους υπολογισμούς που ακολουθούν και αποφεύγουμε τη χρήση περιττών πόρων. Αυτή η διαδικασία δεν επηρεάζει μόνο τις απαιτήσεις χρήσης μνήμης, αλλά επίσης προκαλεί μία δραματική μείωση στη δημιουργία tasks για το SABER, καταλήγοντας σε μικρότερη επιβάρυνση του συστήματος όσο αφορά τη χρονοδρομολόγιση και τον υπολογισμό αποτελεσμάτων. Στις μη βελτιστοποιημένες περιπτώσεις χρήσης, το σύστημα λαμβάνει περισσότερα δεδομένα από ό,τι μπορεί να καταναλώσει, καταλήγοντας σε μικρότερες τιμές throughput και υψηλότερη χρήση CPU και μνήμης, καθώς δυσκολεύεται να αντιμετωπίσει τους ρυθμούς εισροής δεδομένων. Επιπλέον, υπάρχουν ορισμένες περιπτώσεις με μη βελτιστοποιημένα σχέδια, όπου τα ποσοστά απόδοσης δεν είναι σταθερά λόγω της αδυναμίας του συστήματος να χειριστεί τον υπολογισμό τους, με αποτέλεσμα προκαλείται τερματισμός του συστήματος. Το επόμενο σχήμα παρουσιάζει τα αποτελέσματα των ερωτημάτων Q1, Q7, Q8 και Q9 (2,75, 1,34, 1,9, 1,37 φορές παραπάνω MB/s αντίστοιχα):



Εικ. 0.9.3.a, Throughput αποτελέσματα των ερωτημάτων 1, 7, 8 και 9.

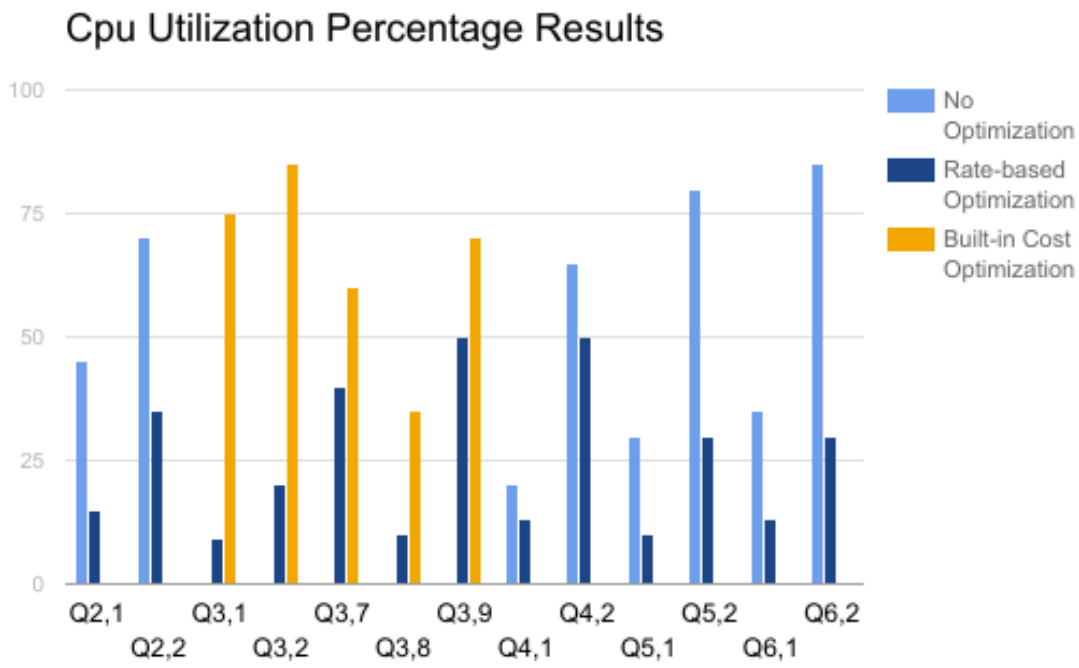
Όσον αφορά το δέκατο ερώτημα, μας βοήθησε να δοκιμάσουμε τη διαδικασία βελτιστοποίησης μας πάνω σε κανόνες που απαιτούν δυναμική βελτιστοποίηση, χρησιμοποιώντας το μοντέλο κόστους βάσει ρυθμού εισροής δεδομένων. Παρατηρήσαμε ότι η σωστή αναδιάταξη των διαδοχικών τελεστών Φίλτρου μπορεί να οδηγήσει σε μεγαλύτερο throughput σε κάποιες περιπτώσεις., σε σύγκριση με τη συγχώνευση τους. Η παρακάτω εικόνα παρουσιάζει τα αποτελέσματά μας (ακόμη και 2,22 φορές παραπάνω MB/s):



Εικ. 0.9.3.b, Throughput αποτελέσματα των ερωτήματος 10.

Στη δεύτερη κατηγορία των ερωτημάτων, που περιείχαν τελεστές συνένωσης, δεν παρατηρήσαμε αύξηση στο throughput των διαφόρων πλάνων εκτέλεσης. Ο τελεστής συνένωσης λειτουργεί ως εμπόδιο (bottleneck) στην εκτέλεση του ερωτήματος, λόγω της υλοποίησης του, και ως εκ τούτου δεν μπορούσαμε να βρούμε μια περίπτωση χρήσης κατά την οποία τα πλάνα μας να έχουν σημαντική διαφορά σε επίπεδο throughput σε σύγκριση με την μη βελτιστοποιημένη υλοποίηση. Πειραματιστήκαμε με το selectivity, τον αριθμό των νημάτων και το μέγεθος των buffers που χρησιμοποιούνται για τον υπολογισμό των ερωτημάτων μας, χωρίς να παρατηρήσουμε κάποια εντυπωσιακή διαφορά μεταξύ των εκτελούμενων πλάνων. Επομένως, υποθέτουμε ότι αυτό συμβαίνει λόγω της άνισης κατανομής του φορτίου επεξεργασίας μεταξύ των νημάτων, το οποίο είναι πιθανό να

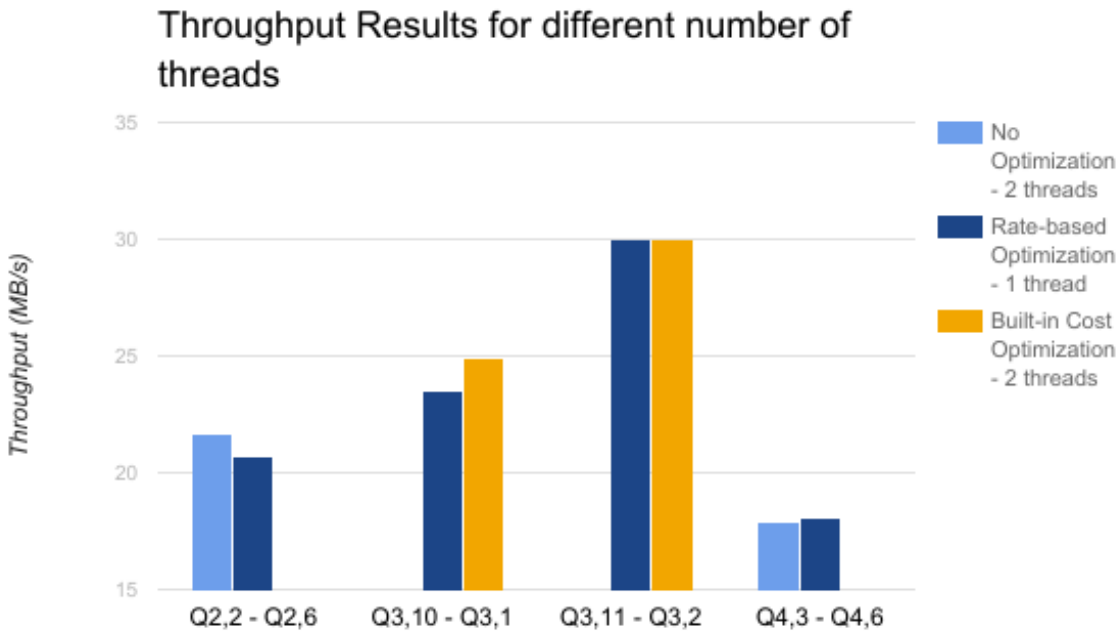
συμβεί σε μεγάλα συστήματα με δυναμικά μεταβαλλόμενο φόρτο εργασίας. Για να ξεπεραστεί αυτό το εμπόδιο και να πραγματοποιήσουμε συγκρίσεις για την αξιολόγηση του συστήματος μας, έπρεπε να παρακολουθήσουμε άλλες μετρικές, όπως το latency και τη χρησιμοποίηση cpu. Τα παρακάτω στοιχεία και η ανάλυση αφορούν τα ερωτήματα Q2, Q3, Q4, Q5 και Q6 (τα νούμερα κάτω από τις μπάρες στα επόμενα γραφήματα αντιστοιχούν στα νούμερα που δώσαμε στις μετρήσεις μας στους πίνακες της προηγούμενης ενότητας).



Εικ. 0.9.3.c, αποτελέσματα ποσοστού χρησιμοποίησης της CPU για τα ερωτήματα 2, 3, 4, 5 και 6.

Σε όλες τις περιπτώσεις χρήσης των ερωτημάτων 2, 3, 4, 5 και 6, η χρήση της CPU του rate-based βελτιστοποιημένου πλάνου είναι σημαντικά μικρότερη από το αντίστοιχο είτε του μη βελτιστοποιημένου πλάνου είτε του βελτιστοποιημένου πλάνου με το ενσωματωμένο μοντέλο κόστους. Έχουμε πειραματιστεί με διαφορετικό αριθμό νημάτων και μεγέθη buffer και παρατηρήσαμε ότι στο rate-based πλάνο, η cpu υπολειτουργεί σε σχέση με τα άλλα σχέδια, όταν έχουμε χαμηλό ρυθμό εισροής δεδομένων. Έτσι, αυξήσαμε το ρυθμό εισροής σε όλες τις περιπτώσεις χρήσης μας και παρατηρήσαμε ότι το rate-based πλάνο μπορεί να χειριστεί ακόμα και το διπλό ρυθμό χωρίς πρόβλημα. Αντίθετα, τόσο το μη βελτιστοποιημένο πλάνο όσο και το ενσωματωμένο μοντέλο κόστους αδυνατούν να

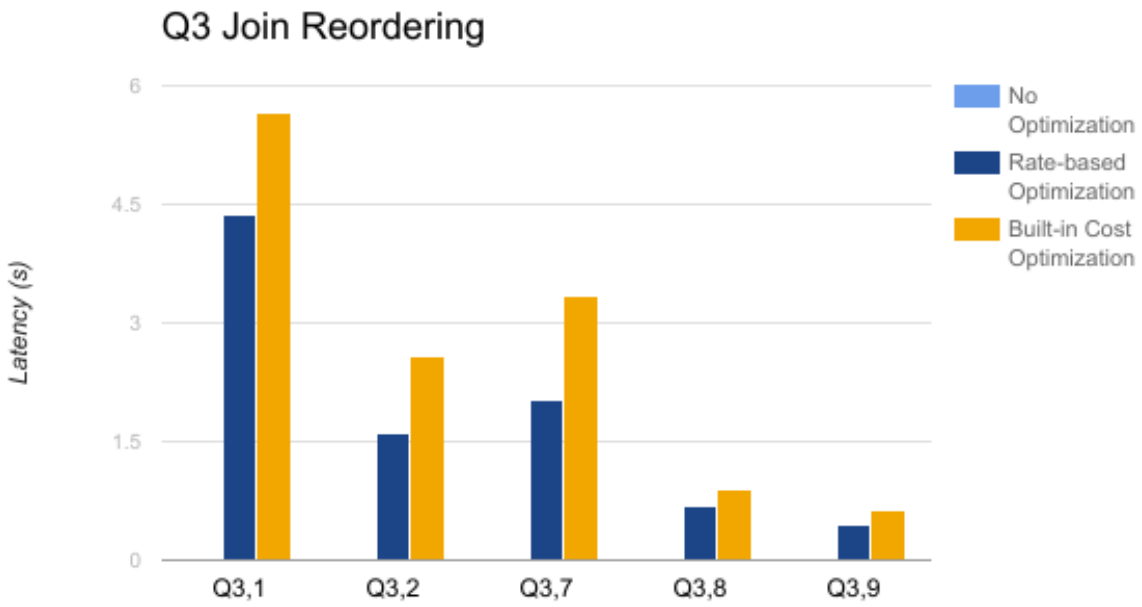
καταναλώσουν τα εισερχόμενα δεδομένα, προκαλώντας τερματισμό του συστήματος. Αυτό το αποτέλεσμα μπορεί εύκολα να εξηγηθεί από τον στόχο που επιτυγχάνουμε με την rate-based βελτιστοποίηση, κατά την οποία μειώνουμε τα ενδιάμεσα δεδομένα και τα task ερωτημάτων που παράγονται από κάθε στάδιο της διαδικασίας υπολογισμού. Για να κάνουμε τα αποτελέσματά μας πιο εύκολα αντιληπτά, πραγματοποιήσαμε κάποια πειράματα για τα ερωτήματα 2, 3 και 4 που δείχνουν ότι έχουμε επιτύχει τα ίδια αποτελέσματα throughput με τη χρήση ενός μόνο νήματος στο rate-based πλάνο σε αντίθεση με τις άλλες δύο επιλογές, στις οποίες χρειάζονται τουλάχιστον δύο νήματα για να χειριστούμε την χρονοδρομολόγιση και τις υπολογιστικές εργασίες:



Εικ. 0.9.3.d, Throughput αποτελέσματα για διαφορετικό αριθμό νημάτων για τα ερωτήματα 2, 3, και 4.

Παρά το γεγονός ότι σε ορισμένα από τα παραπάνω πειράματα το σύστημα θα μπορούσε να χρησιμοποιήσει λιγότερη μνήμη για τη δημιουργία των buffers που απαιτούνται για τους υπολογισμούς κατά την εκτέλεση των rate-based πλάνων, η στατική δέσμευση του μεγέθους των buffer δεν μας επιτρέπει να γενικεύσουμε την παρατήρηση μας και να υποθέσουμε ότι το μοντέλο κόστους εκτός από τη χρήση λιγότερων νημάτων, μπορεί να τρέξει με μικρότερα μεγέθη buffer για τους Unbounded και Circular buffers.

Σε όλα τα προηγούμενα παραδείγματα, τόσο το built-in και όσο και το rate-based μοντέλο κόστους παράγουν τα ίδια σχέδια. Στη συνέχεια, θα επικεντρωθούμε στο ερώτημα Q3, το οποίο δείχνει τη διαφορά μεταξύ των δύο μοντέλων κόστους. Η αναδιάταξη των συνενώσεων είναι ένα σημαντικό χαρακτηριστικό της διαδικασίας της rate-based βελτιστοποίησης, που επηρεάζει την εκτέλεση των πολύπλοκων ερωτημάτων με πολλές συνενώσεις. Όσον αφορά τη χρήση της CPU, στο rate-based πλάνο υποχρησιμοποιείται με τις ίδιες παραμέτρους του συστήματος. Η πρώτη βελτίωση που παρατηρούμε είναι ότι καθώς αυξάνουμε το ρυθμό εισροής, τα πλάνα με το ενσωματωμένο μοντέλο κόστους γίνονται ανέφικτα και οδηγούμαστε σε κατάρρευση του συστήματος (αποτέλεσμα που προκύπτει επίσης από την παραπάνω εικόνα). Η δεύτερη βελτίωση αφορά στο latency που παρατηρείται και στις δύο περιπτώσεις. Στο rate-based πλάνο, επιλέγουμε να υπολογίσουμε νωρίτερα τις συνενώσεις που παράγουν μικρότερα αποτελέσματα, σε σχέση με το ρυθμό εισόδου, με τα παράθυρα που χρησιμοποιούνται και με τη συνθήκη συνένωσης. Ως εκ τούτου, παρατηρούμε ακόμη και λιγότερο από το 70% του latency του built-in μοντέλο κόστους στις περισσότερες περιπτώσεις (Σχήμα παρακάτω), το οποίο επιβεβαιώνει τις προσδοκίες μας.



Εικ. 0.9.3.e, Αποτελέσματα αναδιάταξης των συνενώσεων για το ερώτημα 3.

Συνοψίζοντας, καταλήγουμε στο συμπέρασμα ότι η βελτιστοποίηση που επιβάλλουμε έχει ως αποτέλεσμα μικρότερη αξιοποίηση επεξεργαστή και μνήμης και εξασφαλίζει τη δημιουργία εφικτών προς εκτέλεση πλάνων. Στις περιπτώσεις που δεν έχουμε τελεστές συνένωσης, παρατηρούμε επίσης σημαντικές βελτιώσεις απόδοσης για τα πλάνα των ερωτημάτων μας.

Τέλος, θα θέλαμε να παρατηρήσουμε ότι το rate-based μοντέλο θα είχε μεγαλύτερο αντίκτυπο στην περίπτωση που είχαμε περισσότερες από μία φυσικές υλοποιήσεις (αλγόριθμους) για να επιλέξουμε για οποιοδήποτε από τους τελεστές μας. Για παράδειγμα, αν το σύστημα είχε τουλάχιστον δύο διαφορετικούς τύπους συνενώσεων, η σημασιολογία συνεχούς ροής, η οποία είναι ενσωματωμένη στο rate-based μοντέλο κόστους, θα δημιουργούσε το βέλτιστο πλάνο με βάση τα χαρακτηριστικά των πηγών δεδομένων ροής μας.

0.10 Συμπεράσματα και Μελλοντικές Επεκτάσεις

Σύνοψη και συμπεράσματα

Κατά τη διάρκεια αυτής της εργασίας, είχαμε την ευκαιρία να διερευνήσουμε τη σημασιολογία των ροών δεδομένων και να χρησιμοποιήσουμε τις γνώσεις μας για να εισάγουμε μια δηλωτική γλώσσα SQL στο SABER. Επιπλέον, εξετάσαμε τις τεχνικές βελτιστοποίησης των ερωτημάτων σε σχεσιακές βάσεις δεδομένων, επεκτείναμε την ενσωματωμένη λογική βελτιστοποίησης του Calcite, προκειμένου να αντιμετωπίσουμε τις απαιτήσεις ενός περιβάλλοντος ροής δεδομένων, και δοκιμάσαμε την απόδοση τους στο σύστημα μας. Εδώ θα συνοψίσουμε τα συμπεράσματα και τις συνεισφορές που προκύπτουν από αυτήν την εργασία:

1. Τα σύγχρονα συστήματα ροής απαιτούν τη χρήση SQL για πολλούς λόγους. Η γλώσσα SQL είναι δηλωτική, πράγμα που σημαίνει ότι πρέπει να καθορίσουμε μόνο τι θέλουμε και όχι πώς να το υπολογίσουμε. Επιπλέον, μπορεί να βελτιστοποιηθεί και να αξιολογηθεί αποτελεσματικά τόσο σε κατανεμημένα όσο και σε κεντρικά (centralized) περιβάλλοντα και είναι η καλύτερη λύση για την έκφραση σύνθετων αναλυτικών ερωτημάτων σε δεδομένα, τα οποία εξάγουν πολύτιμες πληροφορίες για τις περισσότερες βιομηχανικές και ακαδημαϊκές

περιπτώσεις χρήσης. Επομένως, στο RBStream εισάγαμε υποστήριξη SQL στην μηχανή ροής που επιλέξαμε, με βάση ένα υποσύνολο στοιχείων παραθύρων που μπορούν να οριστούν στην SQL-99 για συναρτήσεις OLAP. Η ευκολία χρήσης του συστήματός μας έναντι της απλής υλοποίησης του SABER ήταν προφανής και μας δόθηκε η ευκαιρία να δοκιμάσουμε ακόμη περισσότερο το SABER πάνω σε πιο σύνθετα και απαιτητικά ερωτήματα. Ένα απλό ερώτημα SQL με περισσότερους από τρεις τελεστές μεταφράζεται σε δεκάδες γραμμές κώδικα εκτέλεσης στο SABER, με μικρή επιβάρυνση χιλιοστών του δευτερολέπτου μόνο πριν από την εκτέλεση του ερωτήματος για τον υπολογισμό του βέλτιστου πλάνου.

2. Με βάση τους τελεστές της SQL, θα μπορούσαμε να εφαρμόσουμε βελτιστοποίηση τόσο βάσει ευρετικών όσο και κανόνων που βασίζονται στο κόστος, για να αντιμετωπίσουμε τις προκλήσεις που παρουσιάζονται σε περιβάλλοντα ροής. Η βασική μας ιδέα ήταν η εισαγωγή ενός Cost Based Logical Optimizer στο SABER. Αποφασίσαμε να χρησιμοποιήσουμε ένα συνδυασμό του ευρετικού και του δυναμικού βελτιστοποιητή του Calcite, με χρήση τους σε διαφορετικές φάσεις. Διαχωρίσαμε τη διαδικασία βελτιστοποίησης σε διάφορες φάσεις για να μειώσουμε τον χώρο αναζήτησης και τον χρόνο που απαιτείται για τη βελτιστοποίηση, εκμεταλλευόμενοι την εφαρμογή σχετικών υποσυνόλων κανόνων. Επομένως, δημιουργήσαμε μια διαδικασία πιο εύκολα διαχειρίσιμη και επιτύχαμε τα καλύτερα αποτελέσματα με τους κανόνες που είχαμε ήδη. Είναι πολύ εύκολο να προσθέσουμε και να καταργήσουμε κανόνες χωρίς να επηρεαστεί η υπόλοιπη αλυσίδα βελτιστοποίησης. Τέλος, εξετάσαμε και αξιολογήσαμε πολλές ενσωματωμένες και προσαρμοσμένες τεχνικές βελτιστοποίησης ερωτημάτων στο σύστημά μας, όπως το κατέβασμα κατηγορημάτων, το κλαδέμα προβολών, τη συγχώνευση τελεστών και την αναδιάταξη τελεστών (π.χ. συνενώσεων ή φίλτρων).
3. Τα περισσότερα συστήματα ροής που διατίθενται για εμπορική χρήση, δεν έχουν υιοθετήσει ακόμα ένα κατάλληλο μοντέλο κόστους ικανό να καταγράψει τις μετρικές μιας πηγής δεδομένων συνεχούς ροής και να κάνει βελτιστοποιήσεις με βάση αυτές τις παραμέτρους. Διαπιστώσαμε αυτήν την ανάγκη και προσθέσαμε ένα μοντέλο κόστους βάσει του ρυθμού εισροής δεδομένων στο σύστημά μας, για να αντιμετωπίσουμε τα χαρακτηριστικά των δεδομένων ροής. Προηγούμενες προσεγγίσεις για το μοντέλο κόστους με βάση το ρυθμό εισροής δεδομένων δεν έχουν υποστήριξη για συναθροιστικές συναρτήσεις. Προσπαθήσαμε να το επεκτείνουμε για να υποστηρίξουμε την αποτελεσματική εκτίμηση της κατανάλωσης πόρων, να ταιριάξουμε το μοντέλο με τη σημασιολογία μας και να

επιτύχουμε την καλύτερη βελτιστοποίηση για τις πηγές ροής μας. Ο κύριος στόχος του μοντέλου κόστους ήταν η αύξηση του ρυθμού παραγωγής δεδομένων, ελαχιστοποιώντας παράλληλα τα ενδιάμεσα αποτελέσματα και τα query tasks που δημιουργούνται από το SABER. Το μοντέλο κόστους μας βοηθά επίσης να εκτιμήσουμε τον αντίκτυπο των βελτιστοποιήσεων που εφαρμόζονται σε ένα πλάνο ερωτήματος.

4. Η βελτιστοποίηση μας έχει ως αποτέλεσμα τη μικρότερη αξιοποίηση της cpu και της μνήμης, και διασφαλίζει ότι η εκτέλεση των πλάνων είναι εφικτή. Ένα μεγάλο ποσοστό των ενδιάμεσων αποτελεσμάτων που παράγονται από τα μη βελτιστοποιημένα σχέδια συνήθως απορρίπτονται, επειδή δεν απαιτούνται για τον υπολογισμό του τελικού αποτελέσματος. Με τη διαδικασία βελτιστοποίησης που εφαρμόζουμε, μειώνουμε τα ενδιάμεσα αποτελέσματα που παράγονται και μειώνουμε την αναπαραγωγή εργασιών στο SABER, με αποτέλεσμα τη μείωση της επιβάρυνσης λόγω του υπολογιστικού κόστους και της χρονοδρομολόγισης για σύνθετα ερωτήματα. Στις περισσότερες περιπτώσεις δοκιμής, η βελτιστοποίηση βάσει ρυθμού δημιούργησε ένα εφικτό πλάνο, ενώ τόσο τα μη βελτιστοποιημένα πλάνα όσο και τα βελτιστοποιημένα πλάνα με ενσωματωμένο μοντέλο κόστους παρήγαγαν αποτελέσματα που εξαντλούν τη μνήμη και προκαλούν τερματισμό του συστήματος. Επιπλέον, στις περιπτώσεις που δεν είχαμε τελεστή συνένωσης, παρατηρήσαμε μεγάλες βελτιώσεις στο throughput (από 1,3 έως 2,7 φορές περισσότερα MB/s), καθώς ο ρυθμός παραγωγής του βελτιστοποιημένου μας πλάνου επωφελήθηκε πολύ από το μοντέλο κόστους. Τέλος, παρατηρήσαμε από τα πειράματά μας ότι ο τελεστής Join ενεργεί ως εμπόδιο στην εκτέλεση των ερωτημάτων για το SABER. Υποθέτουμε ότι αυτό συμβαίνει λόγω μη σωστής κατανομής του φόρτου επεξεργασίας μεταξύ των νημάτων. Επομένως, έπρεπε να μετρήσουμε διαφορετικές μετρικές για αυτές τις περιπτώσεις, όπως η χρησιμοποίηση του cpu και το latency. Συγκεκριμένα, στο ερώτημα αναδιάταξης των τελεστών συνένωσης, παρατηρήσαμε λιγότερο από το 70% της καθυστέρησης του ενσωματωμένου μοντέλου κόστους. Τέλος, διερευνήσαμε την δυνατότητα συγχώνευσης τελεστών στην ίδιο query task ή τη διάσπαση τους, προκειμένου να προσαρμόσουμε το granularity της εργασία μας και να επιτύχουμε μεγαλύτερη απόδοση.
5. Παρουσιάζουμε τα αποτελέσματα throughput και latency στον χρήστη με μια διαδραστική διεπαφή που χρησιμοποιεί το Jupyter Notebook και βιβλιοθήκες της python για γραφικές παραστάσεις πραγματικού χρόνου. Ο χρήστης μπορεί να

υποβάλει ένα ερώτημα και να επιλέξει ένα από τα τρία διαφορετικά πλάνα που μπορούν να εκτελεστούν. Αυτά τα πλάνα είναι: μη βελτιστοποιημένο πλάνο, βελτιστοποιημένο πλάνο με ενσωματωμένο μοντέλο κόστους και βελτιστοποιημένο πλάνο με βελτιστοποίηση βάσει ρυθμού εισροής δεδομένων.

Μελλοντικές επεκτάσεις

Δημιουργία νέων φυσικών υλοποιήσεων τελεστών στο SABER.

Στην τρέχουσα έκδοση του SABER, έχουμε μια μοναδική φυσική υλοποίηση (αλγόριθμο) για κάθε τελεστή. Παρόλο που για ορισμένους τελεστές, όπως η προβολή, αυτή η προσέγγιση φαίνεται αποδεκτή, για άλλους, όπως το Join, πρέπει να έχουμε περισσότερες από μία υλοποιήσεις για να αντιμετωπίσουμε διαφορετικές περιπτώσεις χρήσης (Hash Join vs B+tree Index Nested Loops Join [24]). Σε ένα απρόβλεπτο περιβάλλον, όπως αυτό που αντιμετωπίζουμε σε εφαρμογές συνεχούς ροής, είναι σημαντικό να έχουμε τη δυνατότητα να επιλέξουμε την καλύτερη επιλογή για κάθε περίπτωση χρήσης. Η εισαγωγή νέων αλγορίθμων και η υλοποίηση τους στο SABER (για την εκτέλεση τόσο σε cpu όσο και σε gpu) θα ωφελήσει περισσότερο τη χρήση του μοντέλου κόστους και θα οδηγήσει σε καλύτερη βελτιστοποίηση.

Υλοποίηση νέων κανόνων μετασχηματισμού για το Calcite.

Οι ενσωματωμένοι κανόνες μετασχηματισμού του Calcite στοχεύουν σε σχεσιακούς μετασχηματισμούς και δεν εκμεταλλεύονται τη σημασιολογία των δεδομένων ροής. Ως αποτέλεσμα, μια πιθανή λύση σε αυτήν την έλλειψη θα ήταν η υλοποίηση κανόνων μετασχηματισμού προσανατολισμένων προς τα δεδομένα ροής, που θα τοποθετηθούν στην τρίτη φάση της διαδικασίας βελτιστοποίησης του συστήματός μας. Σε αυτή τη φάση, αναζητούμε εξαντλητικά το βέλτιστο σχέδιο και την χρησιμοποιούμε για κανόνες που εφαρμόζονται πριν από την εφαρμογή της αναδιάταξης των Join.

Κατασκευή μοντέλου κόστους με μεγαλύτερη ακρίβεια.

Απαιτούνται ορισμένες τροποποιήσεις προκειμένου να αναμορφωθεί το μοντέλο κόστους μας και να δημιουργήσουμε μια πιο ακριβή έκδοση του, που θα καθορίζεται περισσότερο από το SABER. Αυτές οι αλλαγές θα μας βοηθούσαν να εκτιμήσουμε καλύτερα τον αντίκτυπο των βελτιστοποιήσεων και να δώσουμε πιο βέλτιστα σχέδια σε περίπλοκες

περιπτώσεις χρήσης, βασιζόμενοι σε μετρήσεις που συλλέχθηκαν από την εκτέλεση του SABER.

Προσθήκη προσαρμοστικότητας στο σύστημα.

Η επεξεργασία ροών δεδομένων σε πραγματικές εφαρμογές συνδέεται στενά με την προσαρμοστικότητα, διότι η διαδικασία της επεξεργασίας πρέπει να εξελίσσεται κατά την εκτέλεση. Παρόλο που η δυναμική αναδιάρθρωση είναι ένα επιθυμητό χαρακτηριστικό, τα συστήματα ροής ενδέχεται να υποφέρουν από τη διατάραξη και την επιβάρυνση που προκαλείται από την αναδιάρθρωση αυτή. Επομένως, η ασφαλής και μη αποδιοργανωτική αναδιάταξη των τελεστών εξακολουθεί να αποτελεί ανοιχτό πρόβλημα. Είναι πολύ δύσκολο να κάνουμε τις σωστές αλλαγές στο SABER, προκειμένου να υποστηρίξουμε προσαρμοστικούς τελεστές (dataflow routing του Eddies [27]) με την ελάχιστη δυνατή επιβάρυνση. Ωστόσο, η τρέχουσα διαδικασία βελτιστοποίησης θα μπορούσε να χρησιμοποιηθεί ως ένα καλό σημείο εκκίνησης για την προσαρμοστική βελτιστοποίηση.

Επέκταση του συστήματος μας για την υποστήριξη ερωτημάτων σε ιστορικά δεδομένα.

Προς το παρόν, το SABER έχει σχεδιαστεί μόνο για ερωτήματα σε ροές δεδομένων. Ωστόσο, σε πραγματικές εφαρμογές, απαιτούνται πιο σύνθετα ερωτήματα και ανάλυση δεδομένων προκειμένου να εξαχθούν πολύτιμες πληροφορίες. Επομένως, πρέπει να μπορούμε να εκτελέσουμε ερωτήματα τόσο σε δεδομένα ροής όσο και σε ιστορικά δεδομένα (πίνακες). Είναι δύσκολο να σχεδιάσουμε ένα ενοποιημένο μοντέλο που να υποστηρίζει την ανάλυση και για τις δύο αυτές πηγές και τον συνδυασμό τους, με ακριβώς μία φορά σημασιολογία.

Υποστήριξη Flow Control: Backpressure μηχανισμός.

Είναι δύσκολο να εγγυηθούμε την αντοχή σε σφάλματα και την υψηλή απόδοση της επεξεργασίας ροών, σε αντίθεση με την επεξεργασία σε δέσμες (batches). *"To Backpressure αναφέρεται στην κατάσταση όπου ένα σύστημα λαμβάνει δεδομένα με υψηλότερο ρυθμό από ό,τι μπορεί να επεξεργαστεί κατά τη διάρκεια μιας προσωρινής έκρηξης φόρτου"* [70]. Εάν το backpressure δεν αντιμετωπιστεί σωστά, μπορεί να οδηγήσει σε εξάντληση των πόρων ή ακόμη και, στη χειρότερη περίπτωση, απώλεια δεδομένων. Η τρέχουσα έκδοση του SABER δεν υποστηρίζει έναν κατάλληλο μηχανισμό backpressure, κάτι που έχει ως

αποτέλεσμα τον τερματισμό του συστήματος και την απώλεια δεδομένων στις περιπτώσεις που δεν μπορεί να χειριστεί τα εισερχόμενα δεδομένα.

1

Introduction

1.1 Data Stream Management Systems

In the last decades, relational database management systems (RDBMSs) fulfilled the needs of traditional business applications. Data was already loaded into the database when the users submitted complex queries, which were evaluated efficiently by DBMSs. These data records can be considered relatively static and remain valid until explicitly modified or removed. As a result, we assume that the number of queries exceeds the number of data modifications. The goal of a DBMS is to provide persistent, consistent, and recoverable storage, along with efficient query execution.

However, nowadays we have the emergence of a new type of data-intensive applications that requires processing data which is continuously arriving at the system, called stream data sources. Web applications, sensor networks, location-based services, network traffic monitoring, electronic trading and transactions logging are a few examples that belong to this new class of stream-oriented applications and produce data at colossal rates. The increasing use of wireless computing devices, the increasing number of web applications and the advances in sensor-technologies contribute to the growth of such streaming applications. Even, as throughput is increasing, businesses and users crave for even more timely data. While the latest advances in hardware offer some help, it is not enough to keep up with the throughput and latency issues. It is extremely expensive to extract valuable information in acceptable time from massive data volumes accumulated over months or even years. Therefore, the archived data has to be discarded without analyzing it, to free space for new data.

In order to process this rapidly flowing data and produce fast results, the technology of stream processing is being used. The basic concepts behind this technology derive from the database research community. Although relational databases and streaming engines share many common characteristics, they are not the same. In a database, the users apply queries on data that has arrived, has been pre-processed, stored and indexed in advance. Conventional DBMSs are not designed for the continual evaluation of queries over

streaming data. On the contrary, in a streaming query engine, the queries are applied before the data.

Data Stream Management Systems (DMSMs) provide the required support for queries to be evaluated over data that continuously arrives as potentially unbounded, rapid and time-varying data streams. We pass data flows through a number of continuously executing queries and operators, and we get their transformations as a result. One might say that a relational database processes data at rest with transient queries, whereas a streaming query engine processes data in flight [17] with persistent queries.

1.2 Problem Motivation

Modern data stream processing platforms like Apache Storm [40], Spark Streaming [41] and Apache Samza [43] lack support for SQL like declarative query languages and require sound knowledge on imperative style programming and distributed systems to effectively utilize them [4]. These systems provide low-level programming API and substantial effort is required in order to produce decent code, because of the complexity involved the maintenance overhead [20]. In addition to that, several well-known solutions based on Hadoop utilize SQL support, like Hive [56], Impala [74] or Presto [75]. Thus, the question asked is whether we can extend these Big Data streaming engines to support SQL with clear semantics or not.

We already know from traditional databases, that an SQL like language is a mathematically sound way of expressing a question to a DBMS. In order to make streaming systems more easily accessible to users not familiar with programming and distributed systems, we have to introduce the support for SQL like continuous query languages or SQL with streaming extensions. The reasons for using a SQL like query language are [52, 76, 77, 80]:

- SQL is declarative. As a result, the users specify what information they want to extract, but not how to compute them.
- They enable sophisticated real-time analytics to be expressed using simple queries, because of their expressiveness. Most developers understand SQL and it is easier for them to learn it. Only if it doesn't cover a specific use case, they can dive in more complex solutions, like imperative style programming. SQL is capable of encapsulating the underlying complexity of the data management operations, and

thus simplifies real-time analytics. Also, real-time analytics can better be optimized with an SQL like model. By using SQL support to our system, we enable optimizations based on a structured procedure which derives from traditional relational databases. The optimization techniques and algorithms used in DBMS can be adapted and expanded accordingly, in order to deal with issues that emerge from stream-oriented applications.

- *"SQL can be applied for dynamic query planning and optimization, as it is mathematically tractable and designed for this purpose."* [52]
- SQL queries can be optimized automatically over both centralized and distributed systems. Therefore we can achieve significantly better performance with less effort, eliminating the need for manual and often poor optimization of queries. "*SQL offers powerful and elegant constructs that are general, can be optimized dynamically for optimum performance given the current underlying hardware configuration, and can be combined with other SQL operations"* [52]. Streaming SQL applicaitions outperform the equivalents Storm or Spark applications, as developing a streaming analytics application on this platforms requires writing code in Java or Scala, which are less compact than SQL. These programming languages "*require significant garbage collection which is particularly inefficient and troublesome for in-memory processing"* [79].
- We can decouple the implementation of SQL applications, as they can be built in less time compared to for low-level open source platforms and proprietary SQL-like platforms.
- In contrary to Complex event processing (CEP [55]) and many current open source platforms, SQL platforms can be updated on the fly without having to recompile them, which is essential in many applications. For example, when we use stored procedures, the changes apply immediately without having to recompile the system.
- Also, we can define user defined operations written in a high level language and deployed in a SQL query, which partially solves the problem of SQL not supporting all real life use cases.
- Finally, we can use platforms for GUI-driven analysis of data streaming and visualization of streaming analytics, as SQL is easy to auto-generate.

However, traditional SQL, relational data model and algebra [73] were not designed for streaming processing [80]. Relational database systems deal with finite sequences of tuples, which are completely available and produce fixed sized results. Some of these systems

feature eager maintenance of materialized views, which is defined as a SQL query like a regular view. This type of processing is similar to evaluating SQL queries on streams of data, with the difference that in streaming systems we have to update the view whenever its base relations are modified.

Therefore, it is evident that there is a necessity for streaming processing engines to possess a SQL declarative language and as a result query optimization support. Query optimization is a key component of database technology because it acts as a bridge between SQL queries and efficient execution. Moreover, in most cases, optimizers make declarative SQL queries a far better choice than imperative code. However, "*although most Big Data platforms revisit many of the concepts pioneered in the database community in the 2000s, functionalities, such as the typical query optimizers in these frameworks, are rudimentary compared to many mature commercial databases but is quickly evolving"* [78]. Some frameworks, like Spark [41] or Storm [40], offer some heuristic query optimizations, but these are not based on streaming characteristics. In order to optimize their performance, we have either to write custom code (how to cache/partition data) or to tune our applications with configuration changes. On the other hand, although Apache Flink [42] is adopting streaming SQL semantics along with algebraic transformations in its latest release (1.3.0), using Apache Calcite, it doesn't use variables from the streaming context to find the optimal execution plan. As a result, a more concrete solution should be provided with a focus on streaming semantics, including cost-based optimization techniques, and not just in topology transformations.

These two aspects of a DSMS, SQL support and query optimization, do not only offer ease of use, but also improve dramatically a system's performance. When submitting a complex query, the user does not have to h time and thought in writing the query in an optimized way. For example, it is not required to define the join ordering of the streams, as it is the responsibility of the optimizer to handle such cases. Simple optimization techniques, like projection pruning (pushing down projection operators to prune columns early), can affect both the overall throughput and the resources used, as the data that is transferred in the pipelined execution procedure of a DSMS is reduced.

1.3 Thesis Statement and Contributions

The thesis introduces support for a declarative query language, by utilizing a state of the art open-source framework for query parsing, validation and optimization. We present rate-based optimization techniques over data streams and apply our findings to our streaming processing engine, SABER [2]. These techniques are classified into static optimization and have a huge impact on the performance of RBStream, the system we created.

Our contributions are summarized as follows:

1. Query optimization: We employ and configure a set of query optimization techniques, such as predicate push-down, projection pruning, operator merging and join order optimization, based on rules and cost models, that are proposed in the literature, to optimize complex SQL processing in streaming queries, by taking into account incoming rates, the size of windows, system utilization and memory consumption.
2. Cost Model: We modified and extended the utilized models, in order to optimize query execution throughput while minimizing resource consumption, such as memory and cpu usage. Until now, stream processing systems, such as Spark Streaming [41] and Flink [42], only used automatic query optimizations with heuristic rules, not based on streaming semantics. In addition, Calcite's built-in cost model [1] utilizes statistics that can't be used for streaming data, as they are unknown in advance. Thus, we introduce a rate-based cost model on stream characteristics. This model relies on previous work [37, 26] and was modified appropriately to fit our streaming semantics. Its purpose is to allow the estimation of resource consumption and help the optimizer to select the best execution plan.
3. SQL support: We add SQL support to SABER, a hybrid relational stream processing engine. In this thesis, we introduce our declarative query language to express continuous queries over streaming data sources, based on a subset of window constructs definable in SQL-99 for OLAP functions [64]. A SQL query is translated in dozens of lines of code, ready to be executed in SABER.
4. System implementation: We build the system atop Apache Calcite [1], an optimization framework, we create a physical converter, to convert the logical operators to the corresponding physical of our execution engine. In our

implementation, we extend Apache Calcite to utilize the aforementioned techniques, while allowing the user to perform expressive SQL queries following the SQL-99 standard and offer an end-to-end system where we have integrated the modified streaming-aware Apache Calcite with SABER, a hybrid stream processing engine.

5. Evaluation: We analyze and evaluate the effectiveness of RBStream under various system configurations over a Purchase-Orders Transaction Benchmark, with synthetic data, where we show that our implementation can outperform the naive execution approach in an interactive web UI, as far as throughput and cpu utilization are concerned. In certain cases without join operators, the optimized plan resulted in even 2.7 times greater throughput, utilizing nearly half of the resources compared to the non optimized plan. When one or more join operators were used in the query, less than 70% of latency was observed against the conventional plan.

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

- In Chapter 2, we focus on the theoretical background required for understanding the core concepts of Streaming, Query Optimization and Data Stream Management Systems, alongside with related work on these fields. We also present the optimization framework, Apache Calcite, and the streaming processing engine, SABER, which are used in our integration.
- In Chapter 3, we describe the optimization procedure of Calcite. We present some of Calcite's built-in rules along with our custom rules, which are used in the implementation of our system.
- In Chapter 4, we define the cost model used for query optimization in our system. We specify our model's parameters and optimization goal, in order to fit in streaming semantics.
- In Chapter 5, we present the implementation of RBStream along with an interactive web interface to present it. The design choices and the characteristics of our integration are analytically explained.
- In Chapter 6, we present the results and diagrams from the experiments run for our system, for different cost models.

- Finally, in Chapter 7, we summarize our conclusions from the results and refer to issues we did not have the chance to work on and which could be regarded as future work.

2

Background and Related Work

In the context of this thesis, we are going to use Apache Calcite [1], a dynamic data framework, and SABER [2], a centralized hybrid high-performance relational stream processing engine for CPUs and GPGPUs. In order to justify our choice and familiarize the reader with the main concepts concerning streaming systems and query optimization, in this chapter we are going to provide the necessary background, describing the basic mechanisms and programming models used nowadays, as well as a brief reference to related work.

In Section 2.1, we introduce the reader to some key ideas of streaming. In Section 2.2, we discuss CQL, a continuous query language that provides the appropriate semantics for streaming. In Section 2.3, we examine the query optimization procedure in general. In Section 2.4, we describe the main concepts of Calcite. Finally, in Section 2.5, we discuss streaming processing engines and more specifically focus on SABER.

2.1 Basic Streaming Concepts

First of all, we have to cover some important background information about streaming in general [5]. These concepts are partitioned into three specific sections:

- Terminology: in order to dive into complex issues, we have to give precise definitions.
- Capabilities: this section is about oft-perceived shortcomings of streaming systems.
- Time domains: there are two primary domains of time that are relevant to data processing.

A streaming engine is an execution engine designed for unbounded (ever-growing, essentially infinite) data sets, as Tyler Akidau states in his post [5]. The term “streaming” does not imply approximate or speculative results. A well-designed streaming system can produce correct, consistent, repeatable results as any existing batch engine (Lambda

Architecture [72]). In Lambda Architecture we run a streaming system alongside a batch system. Although they both perform the same calculation, the streaming system returns low-latency inaccurate results while the batch system provides the correct results some time later.

However, maintaining such a system is inconvenient, as we recompute the result twice using two complex pipelines [35]. So streaming engines should assure correctness (strong consistency is required for exactly-once processing) and tools for reasoning about time (dealing with unbounded, unordered data of varying event-time skew). Designing and implementing robust streaming processing systems remains a challenging task, not only because of the massive scale of typical deployments, the complexity and unpredictability of system executions but also due to the Volume and the Velocity challenge of large-scale data at the same time. The transition from the Lambda to Kappa Architecture systems [34, 35] requires a powerful stream processor in order to process data at high incoming rates and secure data retention.

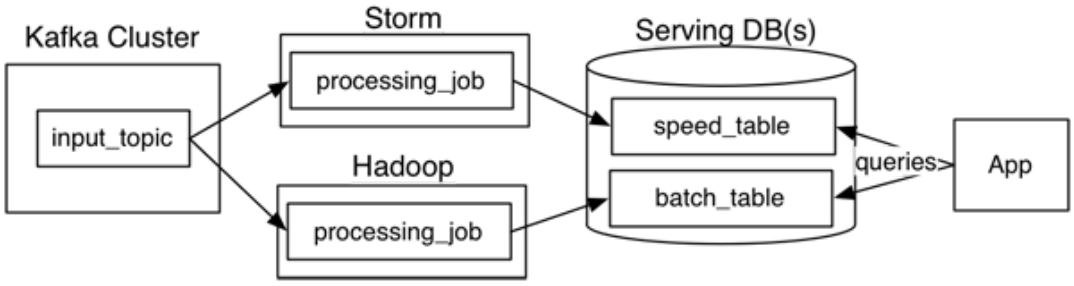


Fig. 2.1.a, Lambda Architecture [35].

Next, we have to understand the two primary time domains involved in the stream processing:

- Event time, which is the time at which the event occurred.
- Processing time, which is the time at which events are observed in the system.

In an ideal world, both would always be equal. However, the skew that exists between them can't be passed by. There are use cases where we care for either of them and create windows according to this particular domain, and cases where both are needed in which several problems arise.

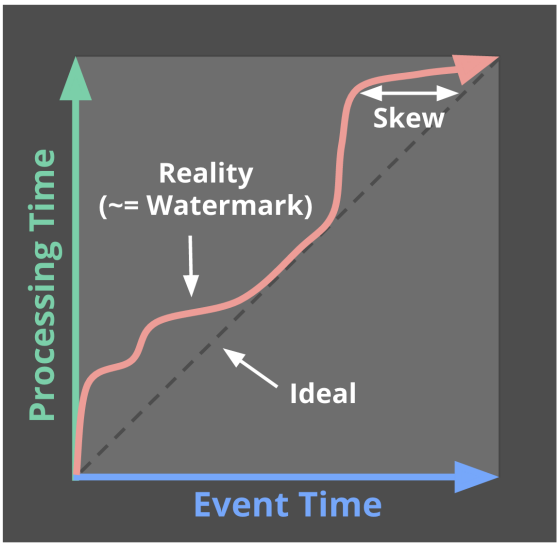


Fig. 2.1.b, Event Time vs Processing Time [5].

In traditional databases, we know how to deal with bounded data processing. The problem arises when we have to face unbounded data sets. In streaming engines we have five different approaches to deal with unbounded data:

- Time-agnostic: This type of processing is used when time is essentially irrelevant (e.g. all relevant logic is data driven). Streaming systems should support these use cases, which are the simplest streaming scenarios where our computations don't require the use of timestamps. Such time-agnostic operations are Filtering and Projection. These operations would return the same result regardless the time dimension, and thus are considered stateless and we don't need memory to compute them. For example, in the next query, we don't care about the time and the result is computed as if we had a conventional table:

  SELECT *

  FROM StreamSource

  WHERE column1 > 100;

- Approximation: The second type of processing is approximation algorithms such as approximate Top-N [65], streaming K-means [66]. These algorithms are designed for unbounded data and are low overhead. However, they are complicated and their approximate nature limits their use.
- Windowing by processing time.
- Windowing by event time.

- Windowing by tuples.

In order to proceed with the last three types of processing, the definition of windowing has to be given. "*Windowing is simply the notion of taking a data source (either unbounded or bounded), and chopping it up along temporal boundaries into finite chunks for processing* [5]."

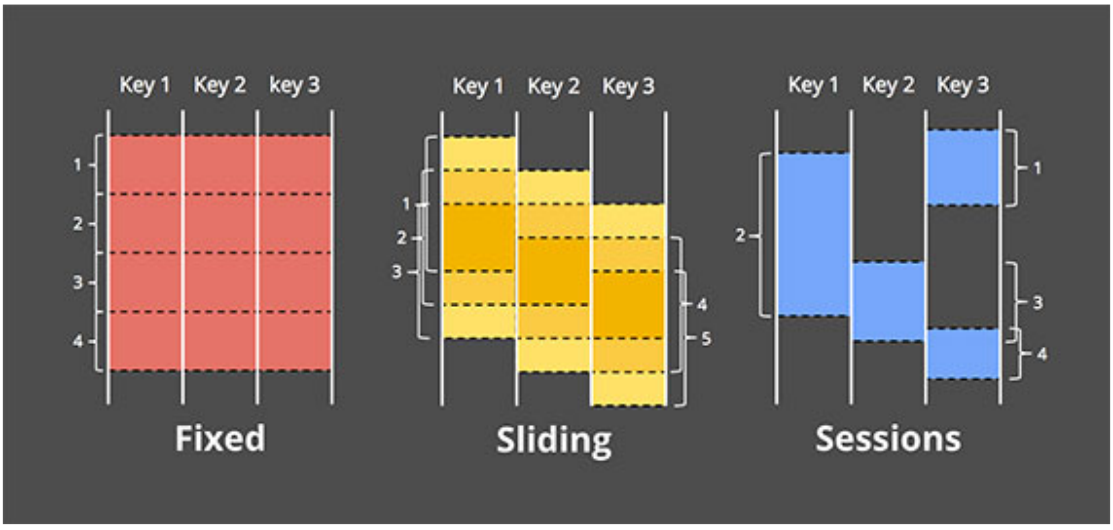


Fig. 2.1.c, Window Types [5].

As shown in the figure above, there are three different windowing patterns (the horizontal lines show the different keys produced by the Group By clause):

- Fixed Windows: These windows slice up time into segments with a fixed size temporal length, which are applied either uniformly across the entire data set (aligned windows as in Fig. 2, above) or to different subsets of data (unaligned windows). As is observed by the figure, the windows are formed with data arriving from the top, until we reach the fixed size of the window. Next, all this data is computed to produce the result and then is dropped, while new data arrives and forms the following window. They are also called tumbling windows.
- Sliding Windows: They are a generalization of fixed windows and are defined by two parameters, a fixed length and a fixed period of slide. In this type of windows, we gather data for a fixed length of time before we compute the result and we drop it according to the fixed period of slide. For instance, if we have length and sliding period equal to 60 seconds and 1 second relatively, we compute our result when we have collected tuples for at least 60 seconds and we drop the tuples that were inserted in the first second of the previous period when we add tuples for every new

second. If the period is less than the length, the windows overlap. If they are equal, we have fixed windows. Finally, when period is greater than length, we have something like a sampling window, that only looks at subsets of the data over time.

- Sessions: Sessions are dynamic windows, composed of sequences of events terminated by a gap of inactivity greater than some timeout. The length of the sessions can't be defined in advance and depends on the actual data involved. Although this type of window is not used in our current implementation, it will be discussed more thoroughly in Calcite Streaming section.

In windowing by processing time, we buffer up incoming data for a certain amount of processing time. This type of processing is simple, eases the check of window completeness and fits in cases where we want to infer information about the source as it is observed.

On the contrary, in cases where data is related to event time, it is difficult to process it, as it may arrive out of order. In such cases, we separate the data sources in finite chunks that reflect the time that these events occurred. However, it is difficult to preserve event time correctness and more buffering is required, but we can create sessions.

In windowing by tuples, the size of the windows is counted in a number of elements and they are called tuple-based. Tuple-based windowing is a form of (processing) time windowing, where elements have monotonically increasing timestamps.

Taking into consideration that while old data is being processed new data arrive, we have to use different algorithms for streaming operators. These algorithms should be able to keep pace with the data stream by having low computation time per data element. As a result, the computation is prefered to be confined in main memory without accessing disk. Also, the input of blocking operators (such as Join, Aggregate and Sort) needs to be materialized before start processing. For example, we need the whole data to be present before applying a sorting operation. These operators are usually called stateful, as we need to use memory to compute their result. Extensions of already known algorithms are used, in order to reshape the blocking behaviour of the known algorithms of such operators to non-blocking (e.g. non-blocking join [36]).

In order to speed up query execution and keep the computation time low, there are several techniques that can be used [54]. Windows are used to set bounds over unbounded data and help the streaming system to compute the result of blocking operators. Furthermore, there are certain applications in which high-quality approximate answers are acceptable. In these cases, we use approximation algorithms to compute our result.

2.2 CQL: Continuous Query Language

CQL [3] is not SQL, but a SQL based declarative language for querying streaming and stored relations. Abstract semantics of CQL relies on three types of operations: stream-to-relations, relation-to-relation and relation-to-stream on two types of data, streams and relations.

Streams and relations

The main difference between the relations in CQL and the standard relations is the additional notion of time in the semantics of the relations R of CQL.

- Stream - A stream S is a (possibly infinite) bag of elements (s,t), where s is a tuple of the schema that describes the stream S and τ is the timestamp attached to the element. The timestamp is not considered to be a part of the schema of the stream, and there could be zero, one, or multiple elements which share same timestamp in our stream. The only requirement that we have is that there is a finite (but unbounded) number of elements with the same timestamp. There are two "types" of streams: the source data streams that arrive to the streaming engine, which are called base streams, and the streams that are produced by operators, which are called derived streams.
- Relation - A relation R is a mapping from a discrete, ordered time domain T to a finite but unbounded bag of tuples of schema of this relation. For a specific time instant $\tau \in T$, we have $R(\tau)$, which is an unordered bag of tuples for this time instance.

Operators

Some regular SQL operators from relational databases, such as join and aggregation operators, are blocking by default and can not be used for evaluation in a streaming context. To overcome this obstacle, we use window operators, to divide the stream into possibly overlapping subsets, after stream scan, in order to reduce the scope of the query to a window extent.

The concept of window is embedded into CQL semantics, by using the concept of instantaneous relation, as discussed above. This allows streaming query execution engines to implement blocking operators and the integration of stored relations to streaming queries. When a stream is converted to instantaneous relation we can work like we did on relations.

- **Stream-to-relation** - The input of this operator is a stream S and the output is a relation R with the same schema as S. Every relation R(τ) should be computable from S up to τ, at any instant τ. In CQL we have only one stream-to-relation operator, the Window operator. There are three classes of sliding window operators in CQL, which are defined below: time-based, tuple-based, and partitioned. The sliding window operator is based on the window specification language derived from SQL-99.
    - **Time-based sliding windows**: This type of window is specified by a time-interval T. It is assumed that it specifies a computable period of application time. By definition, the output of the sliding window is computed by all the latest tuples of an ordered stream that belong to an interval of size T time units. The output relation R of "S [Range T]" is defined as:

$$R(\tau) = \{s \mid \langle s, \tau' \rangle \in S \ \wedge \ (\tau' \leq \tau) \ \wedge \ (\tau' \geq \max\{\tau - T, 0\})\}$$

      There are two special cases: T = 0 and T = ∞. In the first case, the syntax "S [Now]" is introduced, and R(τ) consists of tuples obtained from elements of S with timestamp τ. In the second case, the syntax "S [Range Unbounded]" is used, and R(τ) consists of tuples obtained from all elements of S up to τ.

These two cases will be referred as Now window and Unbounded window in the rest of this text.

- **Tuple-based windows**: A tuple-based sliding window on a stream S is specified by a number N of rows. The output relation computed over time by sliding tuple-based window include the last N tuples of an ordered stream ("S [Rows N]"). A relation R(τ) consists of the N tuples of S with the largest timestamps $\leq \tau$. If there are several tuples with the Nth most recent timestamp (while for clarity let us assume the other $N - 1$ more recent timestamps are unique), we must "break the tie" in some way to generate exactly N tuples in the window. For this reason, tuple-based windows may be not appropriate for cases when timestamps are not unique. There is a special case when $N = \infty$ and it is specified by [Rows Unbounded], which is equivalent to [Range Unbounded].
- **Partitioned Windows**: This type of sliding window takes a positive integer N and a subset {A1, . . . , Ak} of S's attributes as parameters. It is could be defined by [Partition By A1,...,Ak Rows N]. This window logically partitions S into different substreams based on equality of attributes A1, . . . , Ak, computes a tuple-based sliding window of size N independently and computes the result from the union of these windows.

- **Relation-to-relation** - The relation-to-relation operators are derived from traditional relational operators of SQL. The main concept behind these operators is that once we have used a Stream-to-relation operator over a stream and converted it to a window, we can treat it as a relation. These operators have straightforward semantic mapping to time-varying relations. All the traditional relations of a SQL query can be described with CQL semantics.

- **Relation-to-stream** - A relation-to-stream operator takes as input a relation R and gives as output a stream S with the same schema as R. Every stream S should be computable from R up to τ, at any instant τ. We use the difference between previous and current instantaneous relation to convert a relation to a stream. With these operators we can convert the result of a stream-to-relation operation back into a stream for further processing, which is required for the pipelined procedure of query execution in a streaming processing engine. In CQL, there are three

relation-to-stream operators: Istream, Dstream, and Rstream. We assume that the operators ∪, ×, and − are the multiset versions in the definitions that follow.

- 1. Istream ("insert stream") applied to relation R contains a stream element (s,τ), whenever tuple s is in R(τ) − R(τ−1). This operator is used to create a data stream from a relation where each time a tuple is inserted into the relation, a copy is sent to the data stream, but only if the tuples that are inserted are not duplicates. Formally, we have:

$$\texttt{Istream}(R) = \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \{\tau\})$$

- 2. Dstream ("delete stream") applied to relation R contains a stream element (s, τ), whenever tuple s is in R(τ − 1) − R(τ). Using the DSTREAM operator, a data stream is created from a relation where each tuple is removed from a relation, this tuple is sent to the data stream. Formally, we have:

$$\texttt{Dstream}(R) = \bigcup_{\tau > 0} ((R(\tau - 1) - R(\tau)) \times \{\tau\})$$

- 3. Rstream ("relation stream") applied to relation R contains a stream element (s,τ), whenever tuple s is in R at time τ. The operator RSTREAM converts an entire relation into a data stream. That is, all the tuples present at the present time in the relation are sent to the data stream. Formally, we have:

$$\texttt{Rstream}(R) = \bigcup_{\tau \geq 0} (R(\tau) \times \{\tau\})$$

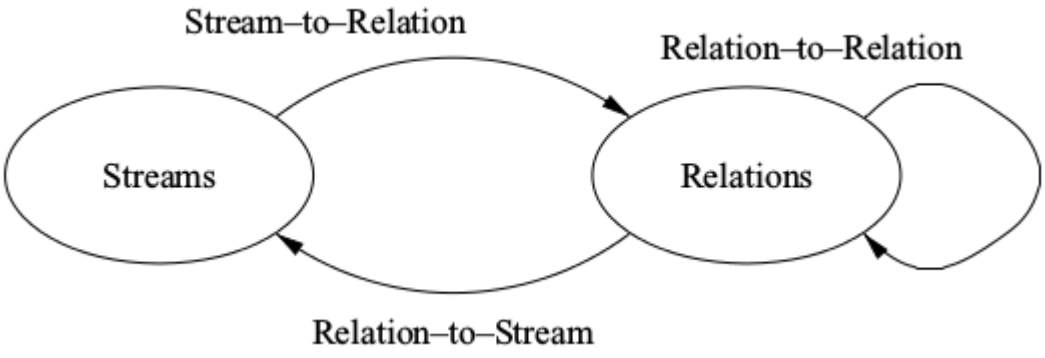


Fig. 2.2.a , CQL Operators [3].

Let's use some examples from CQL [3] paper in order to clarify the use of the above operators:

- ```
  Select Distinct vehicleId
  From PosSpeedStr [Range 30 Seconds]
  ```

  This query is constructed from a stream-to-relation operator (a sliding window of size 30 seconds and slide 1) and a relation-to-relation operator (distinct), which performs projection and duplicate-elimination and it is familiar to us from traditional databases. The query is taken from Linear Road Benchmark, which is used in CQL paper for evaluation, and outputs a relation with all the active vehicles that have transmitted a position speed measurement over the last 30 seconds.

- ```
  Select Istream(*)
  From PosSpeedStr [Range Unbounded]
  Where speed > 65
  ```

  In this query we can distinguish three operators: a stream-to-relation unbounded window operator, which contains at a time τ all the speed position measurements until τ, a relation-to-relation filter operator, which reduces the output according to its filter condition, and a relation-to-stream Istream operator, which streams new values in the filtered relation as the result of the query. We use Istream operator with Unbounded windows to express filter conditions or to stream the results of sliding-window join queries. On the contrary, Rstream operator is used with Now windows to express filter conditions or to stream the results of joins between streams and relations. Finally, Dstream is used less commonly than Istream or Rstream.

STREAM Query Plans

Apart from the above semantic features of CQL, it is interesting to explain the query execution strategy of STREAM engine, which uses CQL, to better understand the basic components of a streaming engine. A runtime stream is represented as a sequence of

timestamped insert tuples. This sequence can be infinite or can be limited with two different approaches, either by using compression techniques, that try to summarize the data, or window techniques, that bound the data into finite parts. A relation is represented as timestamped insert and delete tuples, which show the changing state of this relation. By taking into consideration the above definitions, it is easier to implement incremental processing and query optimization of streams. Incremental processing emerges from the observation that several computations on the incoming streams can be estimated only once and be reused for upcoming computations (e.g. incremental sliding window aggregation). Also, the basic concepts for optimizing continuous queries are analogous to those from traditional databases. If our system supports relational data streams and our logical plan is based on relational operators, algebraic equivalences can be produced by a query optimizer.

Here's an example of how two query plans are executed in STREAM:

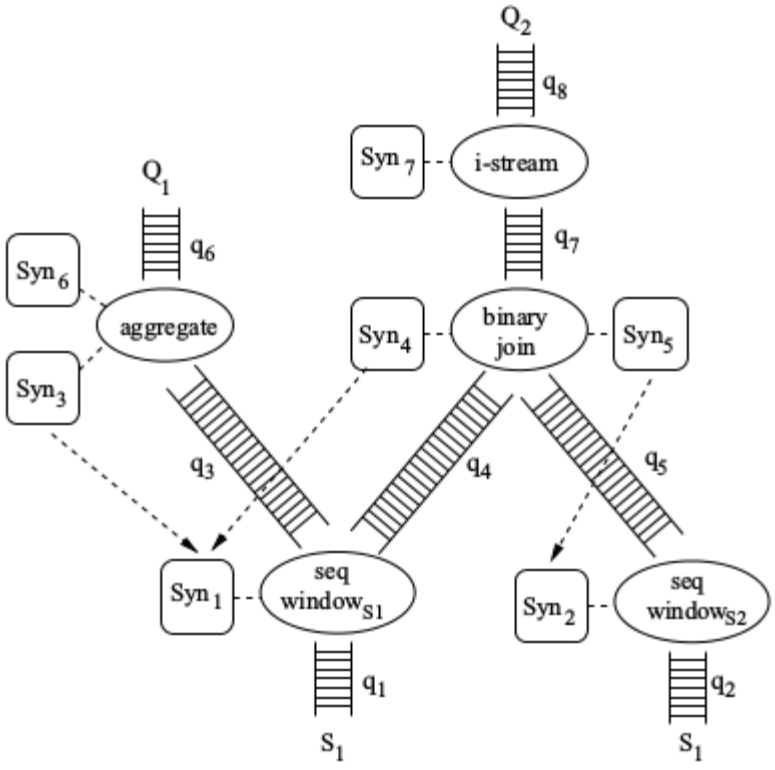


Fig. 2.2.b, STREAM continuous query plan for two queries [3].

Q1:
```
Select B, max(A)
From S1 [Rows 50,000]
Group By B
```

Q2:
```
Select Istream(*)
From
S1 [Rows 40,000],
S2 [Range 600 Seconds]
Where S1.A = S2.A
```

In the previous example, we notice that the output is calculated in a pipelined way. We bound the input data with windows and we keep the state of the operators, in order to advance in our computations. The queues q1 to q8 represent the data movement of the system and synopses s1 to s7 are used to retain the operator state. We can also see the window operators (seq-windows) used on the data stream inputs, to bound the data into finite chunks. More precisely, in the first query, we have a tuple-based sliding window of range 50,000 rows and slide one, while in the query Q2 we have both a tuple-based and a time-based window with slide one and range 40,000 rows and 600 seconds respectively.

We have already discussed stateful and stateless operators. Operators that require memory for their computation, such as Aggregate or Join, play a critical role in the output rate and the resources utilization of a streaming engine. On the contrary, stateless operators aren't affected by window boundaries and thus can give the same result regardless the window chosen. However, placing a filter or a projection early can significantly reduce the amount of data that is produced and processed in later execution stages. From the above, it is obvious that the correct placement of operators can maximize the output rate and reduce the needed computational resources. This is what we try to achieve with query optimization.

2.3 Query Optimization

This section is about the main concepts of query optimization, which derive from traditional databases. Query optimization is one of the key components of data management systems' core, the bridge that connects SQL queries to efficient execution. In traditional databases, SQL queries can have more than one corresponding physical plans. Although all these plans give the same result when executed, they may differ in the execution time or the required

resources. In this case, the optimizer has to find the optimal or a close to the optimal plan that will be executed.

Each logical or physical plan is represented by an operator tree, in which the nodes represent the operators and the edges represent the data flow. When we have complex queries over multiple input sources, the different ways of implementing them may be millions or more. Decisions about the join order, the order of consecutive operators (e.g. if a filter should be pushed through a projection) or which algorithm to use for implementing an operator (e.g. natural join or hash join) produce a large search space. The detection of the optimal plan even in simple queries has a huge impact in time spent to compute the result and the utilization of resources.

Query optimization can be viewed as a search problem and it needs:

- A search space of possible plans.
- A cost model that will be used to compute the cost of every candidate plan.
- An enumeration algorithm to search through the possible plans and find the one with the optimal cost. This algorithm can be combined with a pruning mechanism to reduce the size of the search space.

An optimizer should have the following characteristics:

- The plans with the lowest cost are included in the search space.
- The cost model used provides accurate cost estimation.
- The enumeration algorithm is computationally efficient.

There are two different query optimization architectures used over the last decades: System R-style bottom-up dynamic programming optimizers [50] and Volcano-style top-down memoization with branch-and-bound pruning optimizers [11, 8]. We could always apply heuristics for a certain subset of the search space, but in general these transformations do not ensure that the cost will be reduced in every case. Therefore, optimization rules must be applied in a cost-based manner. Both optimizers achieve the goal of finding the optimal plan based on a cost model. The quality of the plan depends on the transformation rules used for generating it and the correctness of the cost model and not on the type of optimizer. However, top-down optimizers have the advantage that they can prune the search space early with branch-and-bounding. The pruning will be effective depending on the search

order and how fast it reaches the best plan. We will discuss later the Volcano architecture in detail.

The same principles of query optimization can be applied in streaming data management systems, by introducing the stream-to-relation and relation-to-stream operators. If we add the appropriate rules about these operators and combine them with the relation-to-relation rules, we can optimize our streaming queries like we do with traditional database queries. Two streaming query plans are equivalent if they generate snapshot-equivalent results. "*The snapshot of a temporal relation at time t is the conventional relation containing those tuples (without the time periods) from the temporal relation that have time periods containing t* [23]". This means that for any time instant t their snapshots are equal.

2.3.1 Cost Metrics and Statistics

Query optimization in traditional DBMSs uses selectivity information and available resources to choose efficient query plans (for example plans with the fewer cpu cycles or the fewest disk accesses). However, these metrics and statistics can't be used for query optimization of streaming queries, where processing cost per-unit-time is more appropriate [24, 25, 26]. Changing the optimization goal to the highest output rate [24], could give us a reliable solution. Providing that we have the stream arrival rates and output rates of query operators we could also search for a plan that takes the least time to output a given number of tuples (rate-based model [25]).

Sometimes, operators may need to be re-ordered on-the-fly according to changes in system conditions in execution time, because of the nature of the input data sources (adaptive optimization). There are three reasons that may change the initial cost of a query plan: change in the processing time of an operator, change in the selectivity of a predicate, and change in the arrival rate of a stream. There has been some work in query optimization, either static [24, 25, 26] or adaptive [27, 28], on streaming queries.

Some possible cost metrics for streaming queries along with necessary statistics that are required to be maintained for both static and adaptive optimization are [51]:

- Accuracy and reporting delay vs. memory and cpu usage: In some cases, we have to decide about accuracy and reporting delay against memory and cpu usage trade off,

according to the nature of our application. We can use sampling and load shedding to decrease memory usage and increase the result error.

- Output rate: If the stream arrival rates and output rates of query operators are known, it is possible to optimize for the highest output rate or to find a plan that takes the least time to output a given number of tuples [25].
- Power usage: Energy consumption plays a critical role in certain type of applications (e.g. in a wireless network of battery operated sensors).

2.4 Calcite

Apache Calcite [1] is a dynamic data management framework written in Java, previously known as Optic, inspired by the Volcano Optimizer [7, 8]. There are four stages in query optimization: a) parse the query with a JavaCC generated parser, b) validate the queries with known database metadata, c) optimize logical plan and convert it to physical expressions and d) convert the physical plan to application specific execution. Calcite is "*a collection of software libraries and tools that can be used"* [20] to implement and add custom logic to each of the former four stages. Users can add their custom optimizer rules and extend the built-in logical and physical algebras, according to the query engine used for executing the queries, in order to achieve custom optimizations. For example, they can plug rules and operators based on a specific cost-model, that corresponds to the physical implementation of query operators. Many modern big data solutions, such as Apache Hive [56], Apache Kylin [57], Apache Drill [58] and Apache Phoenix [59], use Calcite for their query planning and optimization modules. Another interesting characteristic of Calcite's utility libraries is that they help the code generation phase of the query planner. For all these reasons, it is a preferred solution for parsing, validating, optimizing and converting SQL queries to an execution plan in our streaming engine.

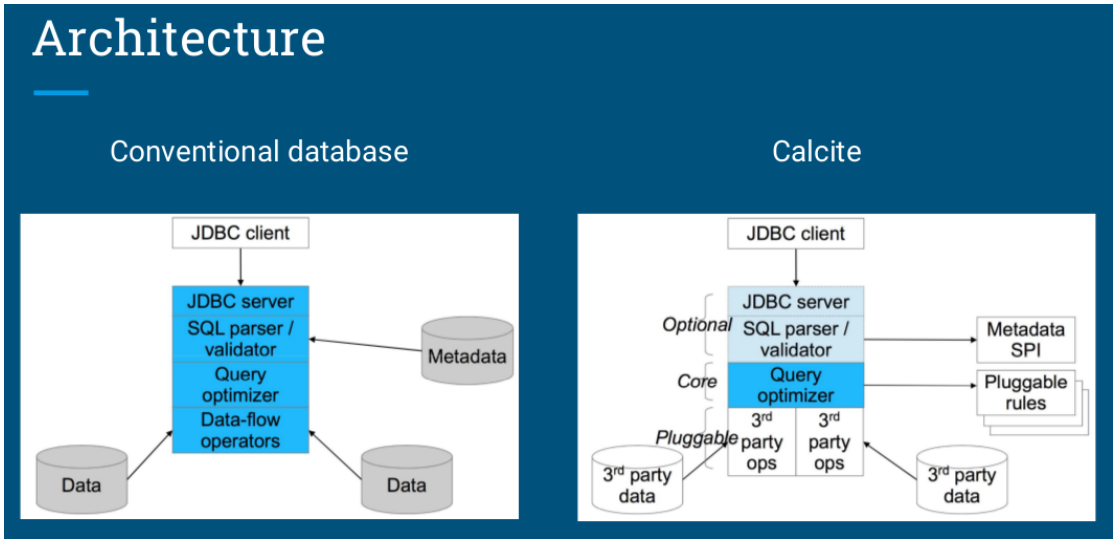


Fig. 2.4, Architecture with Calcite [13].

"Calcite contains many of the pieces that comprise a typical database management system, but omits some key functions: storage of data, algorithms to process data, and a repository for storing metadata [1]*."* This choice is intentional, as it makes Calcite a very good option for mediating between applications and one or more data storage locations and data processing engines. It can also be used as a foundation for building a database by just adding data and showing to Calcite where to find this data format and how to handle it. In order, to add a data source, we need to write a proper adapter to describe to Calcite what collections in the data source it should consider "tables".

Furthermore, we can add custom optimizer rules and algebra. The optimizer rules are used to optimize the logical plan, to convert it to physical operators, to access data in new format and to register new custom operators. The user can combine his custom logic with the built-in rules and operators, apply cost-based optimization (based on custom or built-in cost model), and generate an efficient plan. These rules, also called planner rules, operate by searching for patterns in the query parse tree. For example, when the rule FilterProjectTransposeRule finds a Filter operator over a Project operator, it transposes these operators. Another example of planner rules' use is when we want to define a custom table or schema in order to make the access efficient.

"Calcite doesn't fire rules in a prescribed order. The query optimization process follows many branches of a branching tree, just like a chess playing program examines many

possible sequences of moves. If rules A and B both match a given section of the query operator tree, then Calcite can fire both. Also, Calcite uses cost in choosing between plans, but the cost model doesn't prevent rules from firing which may seem to be more expensive in the short term."[1]

There are optimizers that use a linear optimization scheme. This means that, when the optimizer has to make a choice between two rules at a certain time, it must choose immediately which rule to enforce, depending on a policy. Nevertheless, Calcite does not face these restrictions when trying to combine rules.

As already mentioned, the Calcite chooses the cheaper plan according to a cost model. This cost model defines the cost formulas used to describe the operators (statistics) and the optimization goal of the optimizer. By using a well described cost model we can prune the search tree to prevent the search space from exploding. However, the cost model never forces the optimizer to choose between two rules and avoids falling into local minima in the search space that are not actually optimal.

2.4.1 Volcano Optimizer

Volcano Optimizer [11, 8] is a top-down style optimization framework. This family of optimizers feature top-down goal-directed search with branch-and-bound pruning. This section is about some high-level design characteristics of Volcano that relate to the previous discussion on Query Optimization.

1. Search space: Top-down optimizers use two types of rules: the transformation rules and the implementation rules [11, 8]. The transformation rules map an algebraic expression into another. The implementation maps an algebraic expression into an operator tree. Volcano uses only one optimization phase because all transformation are algebraic and cost-based, the mapping from algebraic to physical operators occurs in a single step and it performs goal-driven application of rules. In Volcano rules are translated independently from one another and are combined only by the search engine when optimizing a query. These rules represent the knowledge of search space in top-down optimizers.

2. Cost estimation: The cost estimation has to be performed in bottom-up way. This occurs because the cumulative cost of each intermediate level of the plan depends on costs and statistics of the previous sub-plans.

3. Enumeration/Pruning algorithm: During the optimization phase, when the optimizer encounters a query expression, it checks whether it has already been computed by looking in a memoization table of plans that have been optimized in the past. If its not found, it applies a logical transformation rule, an implementation rule, or uses an enforcer to modify properties of the data stream. Plans are captured in a hash table of expressions and equivalence classes, in order to detect redundant derivations of the same expressions and plans during optimization and reduce the optimization effort. An equivalence class represents two collections, one of equivalent logical and one of physical expressions. As mentioned above, top-down optimizers use branch-and-bounding to prune the search space (directed dynamic programming). If a sub-plan exceeds a certain cost threshold it can be safely pruned early in the optimization process and save time by not enumerating its subtrees. Pruning is embedded in enumeration and hence has to be in a specific top-down order as well.

2.4.2 Components of Calcite [9]

2.4.2.1 Catalog

Catalog defines namespaces and metadata that can be accessed in SQL queries, such as:

- Schema:
    - A collection of schemas and tables
    - Can be arbitarily nested
- Table:
    - Represents a single data set
    - Fields defined by RelDataType
- RelDataType
    - Represents fields in a data set
    - Supports all SQL data types, including structs and arrays [10]

It also provides table statistics used in optimization. These statistics refer to

- approximate number of rows in the table
- collections of columns on which this table is sorted (List<RelCollation>)
- the distribution of the data in this table (RelDistribution)
- if a given set of columns is a unique key, or a superset of a unique key, of the table (ImmutableBitSet columns)

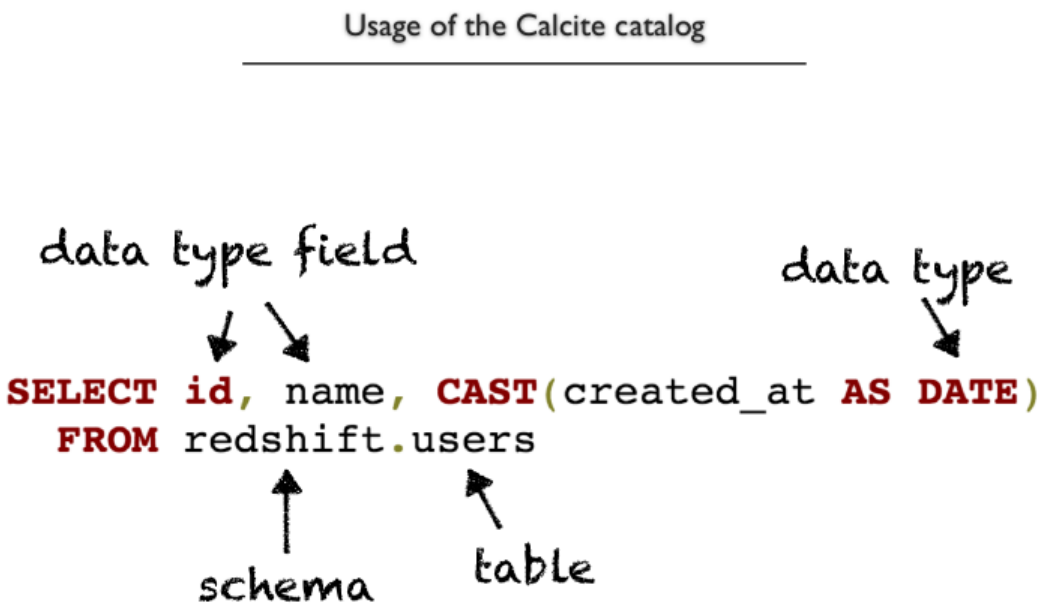


Fig. 2.4.2.1, Calcite Catalog [9].

2.4.2.2 SQL parser

SQL parser is an LL(k) parser (a top-down parser [67] for a subset of context-free languages [68]) written in JavaCC. The parser takes queries as input and parses them into an abstract syntax tree (AST). This tree consists of SqlNodes. An SqlNode could be a parsed tree by itself, as it may be an SqlOperator operator, SqlLiteral literal, SqlIdentifier identifier and so forth. These nodes can also be converted back to a SQL string via unparse method.

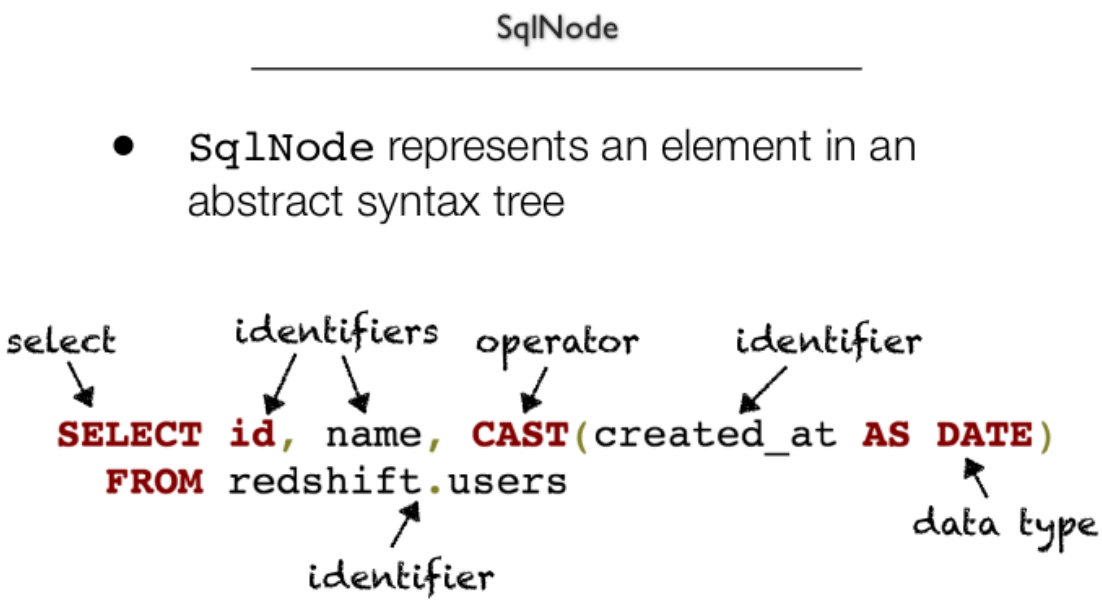


Fig. 2.4.2.2, SqlNode [9].

2.4.2.3 SQL validator

Sql validator validates the parse tree of a SQL statement, and provides semantic information about the parse tree. In order to resolve names to objects, the validator builds a map of the structure of the query. This map consists of two types of objects. A SqlValidatorScope describes the tables and columns accessible at a particular point in the query and a SqlValidatorNamespace, which is a description of a data source used in a query. The validator builds the map by making a quick scan over the query when the root SqlNode is first provided. Thereafter, it supplies the correct scope or namespace object when it calls validation methods.

2.4.2.4 Query Optimizer

After the validation of the AST, comes the stage of query optimization. Query plans represent the steps necessary to execute a query. First, we optimize the logical plan according to certain optimization goal, such as reducing the amount of data that must be processed early in the plan. Then, we convert the logical plan into a physical plan, that is engine specific and represents the physical execution stages.

Some possible optimizations could be:

- Prune unused fields
- Merge Projections
- Convert subqueries to joins
- Reorder Joins
- Push down Projections
- Push down Filters

The key concepts of query optimization are:

1. **Relational Algebra** :

Every query is represented as a tree of relational operators. These operators are called RelNodes. Relational expressions process data, so their names are typically verbs : Sort, Join, Project, Filter, Scan, Sample. Planner rules transform expression trees using mathematical identities that preserve semantics. For example, it is valid to push a filter into an input of an inner join if the filter does not reference columns from the other input. Calcite optimizes queries by repeatedly applying planner rules to a relational expression. A cost model guides the process, and the planner engine generates an alternative expression that has the same semantics as the original but a lower cost.

2. **Row Expressions** :

Every row-expression has a type. Some common row-expressions are: RexLiteral (constant value), RexVariable (variable), RexCall (call to operator with operands). Expressions are generally created using a RexBuilder factory. Row expression could be projection fields, filter conditions, join conditions, window expressions and sort fields.

Row Expressions :

- Input column ref (RexInputRef)
- Literal (RexLiteral)
- Struct field access (RexFieldAccess)
- Function call (RexCall)
- Window expression (RexOver)

3. **Traits** :

"RelTrait represents the manifestation of a relational expression trait within a trait definition [21]*."* These traits don't alter execution. For example, a CallingConvention.JAVA is a trait of the ConventionTraitDef trait definition. Traits are used to validate plan output. There are three primary trait types:

- Convention
- RelCollation
- RelDistribution

4. **Conventions** :

A set of conversion information about the traits of an expression. A convention is a type of RelTrait and is associated with a RelNode interface. Conventions are used to represent a single data source. Inputs to a relational expression must be in the same convention. For example Convention.NONE is a convention for a relational expression that does not support any convention. It is not implementable, and has to be transformed to something else in order to be implemented. Relational expressions generally start off in this form. Such expressions always have infinite cost.

5. **Rules** :

Rules are used to modify query plans. There are two types of rules: converters and transformers :

- Converter rules implement Converter and convert from one convention to another. Converter rules applied via convert.
- Tranformer rules are matched to elements of a query plan using pattern matching onMatch is called for matched rules and are defined by the RelOptRule interface.

RelOptRules register the rules in their constructor method and transform an expression into another. Creating a rule involves creating an operand with children. It has a list of RelOptRuleOperands, which determine whether the rule can be applied to a particular section of the tree. This will be matched with incoming query and when it matches, the onMatch() method in the Rule class will be called.

6. **Planners:**

Planners implement the RelOptPlanner interface. There are two types of planners:

- HepPlanner : is a heuristic optimizer similar to Spark's optimizer. It applies all matching rules until none can be applied. Heuristic optimization is faster than cost-based optimization, but has the risk of infinite recursion if rules make opposing changes to the plan.
- VolcanoPlanner: is a cost-based optimizer, that applies matching rules iteratively, selecting the plan with the cheapest cost on each iteration. Costs are provided by relational expressions. However, not all possible plans will be computed, because the optimization procedure terminates when the cost does not significantly improve through a determinable number of iterations.

Cost is provided by each RelNode and is represented by RelOptCost. It typically includes row count, I/O and CPU cost. The estimates are relative and statistics are used to improve the accuracy of cost estimations. Calcite provides utilities for computing various resource-related statistics, which can be used for cost estimations.

7. **Programs** :

A Program is a RelOptPlanner and a set of rules to apply. We can add several Programs to our Planner by using the various static Programs methods, e.g. Programs.ofRules(rules), if we want to use the VolcanoPlanner, or Programs.hep, if we want to use the HepPlanner. When we have set the proper configurations, we use the Planner's transform method to apply one of those Programs to the converted RelNode and define a set of required output traits.

2.4.2.5 SQL Generator

SQL Generator converts physical plans to SQL. It is an extension to StringBuilder used for creating SQL queries and expressions. It also helps to prevent SQL injection attacks, incorrectly quoted identifiers and strings.

2.4.3 Calcite Streaming

Calcite has extended SQL and relational algebra in order to support streaming queries [14]. There are cases where business or users want to query streams like they do with relational tables. This need comes with the requirement to have some of the advantages of the

standard databases: use a high-level language based (if possible) on relational SQL, validated according to a schema and optimized to take advantage of the resources of our query engine and the algorithms used. Calcite's SQL is an extension to standard SQL and has the following characteristics:

- It is easy to learn, because it is an extension of standard SQL.
- The semantics are clear, because they take after the corresponding ones from standard SQL. The results on stream should be the same as if we had the same data in a table.
- The user has the capability to combine streams and tables (or the history of a stream, which is basically an in-memory table) and create more complex queries.
- There are already many existing tools that can be used to generate standard SQL.

If the user doesn't use the STREAM keyword, he is back in regular standard SQL. Streaming SQL follows the approach set out in CQL [3]. This means that every relational query is converted to a stream using the STREAM keyword in the top-most SELECT. During the preparation time of the query, Calcite distinguishes whether we refer to a stream or a historical relation. In some cases, there is available some of the history of a stream (for example the last 24 hours of data in an Apache Kafka topic [60]). At run time, Calcite figures out whether there is sufficient history to run a query that requires the use of a historical relation.

For Calcite streaming semantics, there is a Stream - Database duality [13]:

- *"Your database is just a cache of my stream"*
- *"Your stream is just change-capture of my database"*

A table can be used as a stream and a stream as a table, by having the system to determine where to find the records. In addition, streams can be considered the derivatives of tables, while tables can be considered respectively the integrals of streams. With the STREAM keyword we refer to records from now to $+\infty$. Without it, we refer to records from $-\infty$ to now, as we would if we had a standard table. Streams complement tables and represent what is happening in the present and future of the records. Tables represent the past (history) of the records and it is very common for a stream to be archived into a table. As a

result, the user can combine past and future in complex queries and achieve high expressivity.

The replay principle applies to these semantics: "*A streaming query produces the same result as the corresponding non-streaming query would if given the same data in a table"*[13].

A simple query example that projects all the columns of a stream named Orders :

```
SELECT STREAM *
FROM Orders;
```

2.4.3.1 Punctuation

Another basic concept of Calcite streaming is making progress. In real-time applications it is not enough only to get the right results. We have to compute them in the right time period, in order to be worth. These are the ways to make progress without compromising safety:

- Monotonic columns (e.g. rowtime) and expressions (e.g. floor(rowtime to hour))
- Punctuations (aka watermarks)
- Or a combination of both

We can use Punctuation [15, 16] to ensure that our query will make progress even if there not enough values in monotonic key to push the results out. When a stream has punctuation enabled, it can be sorted even though is out of order initially. This helps us use our semantics as if the stream was sorted. Also, an out of order stream is also sortable if it is t-sorted or k-sorted and queries can be planned in a similar way as if we had punctuation.

Sometimes we want to aggregate over attributes that are not time-based but are nevertheless monotonic. "*The number of times a team has shifted between winning-state and losing-state*" could be a monotonic attribute. In this case, the system needs to figure out for itself that it is safe to aggregate over such an attribute and punctuation does not add any extra information.

Certain cost metrics (metadata) should be implemented for the planner in order to support punctuation, as it only supports monotonic columns with BuiltInMetadata.Collation.

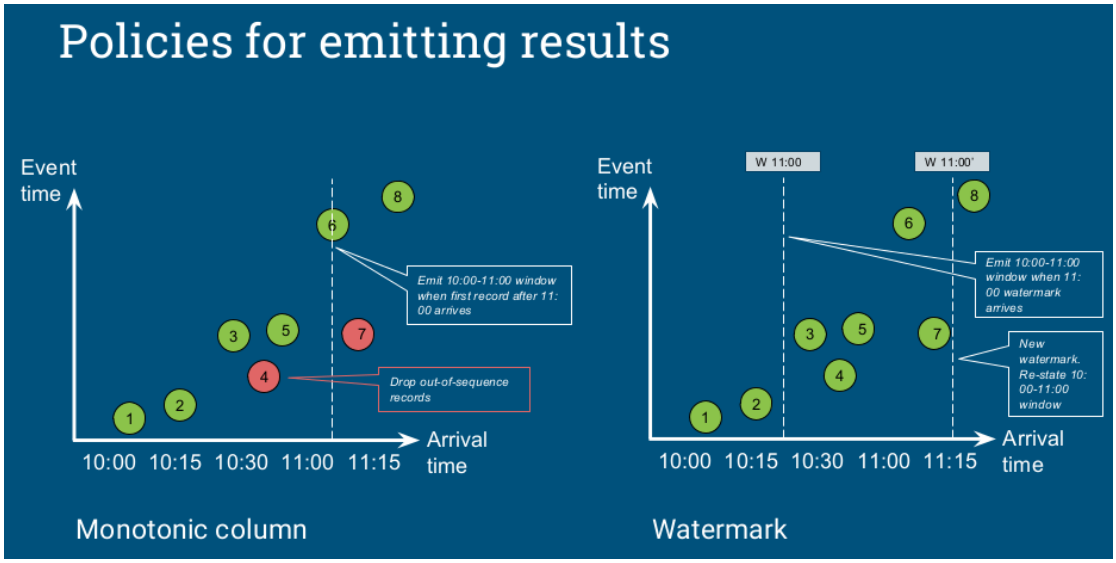


Fig. 2.4.3.1, Punctuation - Watermarks example [13]

2.4.3.2 Window Definitions

In Calcite, we can compute aggregate functions on streams with several different ways. There are various window types, that vary depending on [14]:

- *"How many rows come out for each row in?"*
- *"Does each incoming value appear in one total, or more?"*
- *"What defines the "window", the set of rows that contribute to a given output row?"*
- *"Is the result a stream or a relation?"*

Standard **Group By** aggregates multiple rows into subtotals, as each row contributes to exactly one sub-total. On the contrary, when we have a **Multi-Group By** (e.g. HOP or Grouping Sets) a row can contribute to more than one sub-total. We also have **Window functions**, by using OVER, which leave the number of rows unchanged and compute extra expressions for each row.

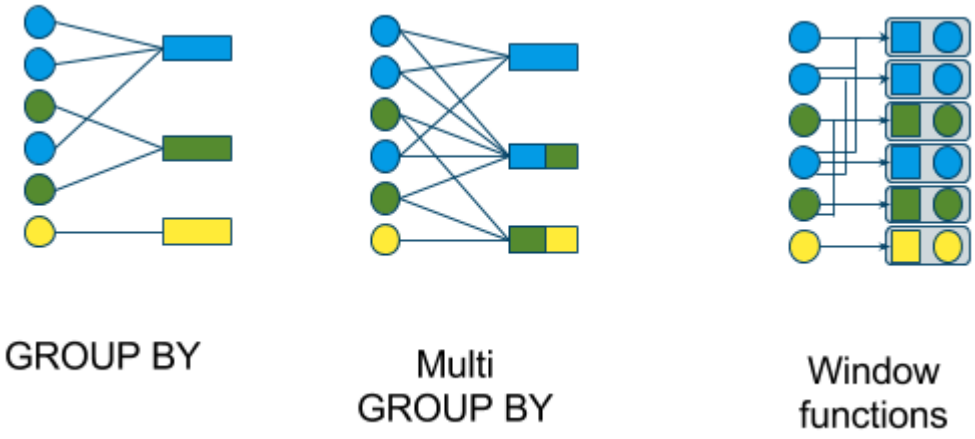


Fig. 2.4.3.2, Different Aggregate Functions [14]

The above windows can be categorized in the following window types:

- tumbling window (GROUP BY)
- hopping window (multi GROUP BY)
- sliding window (window functions)
- cascading window (window functions)

2.4.3.2.1 Tumbling Windows

"Tumbling windows are a series of fixed-sized, non-overlapping and contiguous time intervals, expressed with group by" [22]. In order to allow the query to be executed in Calcite, we have to use monotonic or quasi-monotonic expression in the GROUP BY clause (see Punctuation above). Apart from using group by, Calcite has introduced TUMBLE, TUMBLE_START and TUMBLE_END functions to give the user the opportunity to define more complex expressions.

2.4.3.2.2 Hopping Windows

"Hopping windows are a generalization of tumbling windows that allow data to be kept in a window for a longer than the emit interval" [14]. We can define hopping windows, by using the functions HOP, HOP_START and HOP_END. Unlike tumbling windows, they model overlapping windows, in which each input record contributes in the computation of at least one output record.

2.4.3.2.3 Grouping Sets

We use Grouping Sets to generate an output equivalent to that generated by a UNION ALL of multiple simple GROUP BY clauses. For example the following queries contain a single grouping set and give the same result [48]:

q1:
```
SELECT a, b, c, SUM(x)
FROM v53
GROUP BY GROUPING SETS ((a, b, c));
```

q2:
```
SELECT a, b, c, SUM(x)
FROM v53
GROUP BY a, b, c;
```

Although Grouping Sets can be expressed by ROLLUP and CUBE operations [47], these operations are not valid for streaming queries, as they will generate a grouping set that aggregates everything (GROUP BY ()) [14]. On the other hand, Grouping Sets that contain monotonic or quasi-monotonic expressions are valid in streaming computation.

2.4.3.2.4 Sliding Windows

Sliding windows derive from standard SQL features "analytic functions" that can be used in the SELECT clause. For each record that goes in, one record comes out, unlike group by which collapses the input records. Some other features of the windowed aggregation syntax are:

- Row-based windows can be defined, apart from time-based windows.
- Rows that haven't arrived yet, can be referenced by the windows.
- Order-dependent functions such as RANK and median can be computed.

2.4.3.2.5 Cascading Windows

"If we want a query that returns a result for every record, like a sliding window, but resets totals on a fixed time period, like a tumbling window" [14], we can use a cascading window. For example:

```
SELECT STREAM rowtime,
productId,
units,
SUM(units) OVER (PARTITION BY FLOOR(rowtime TO HOUR)) AS
```

```
        unitsSinceTopOfHour
FROM Orders;
```

The difference with the previous sliding window definition is that the monotonic expression occurs within the PARTITION BY clause of the window. Every time an hour passes in our example, the value of FLOOR(rowtime TO HOUR) changes, starting a new partition. Taking into consideration that old partitions will not be used in later computations, all sub-totals for them are removed from Calcite's internal storage. Therefore, we can assume that cascading windows are sliding windows with slide greater than 1.

2.4.3.2.6 Session Windows

Emit groups of records that are separated by gaps of no more than T seconds. When we use session windows, the elements will be splitted among them based on their timestamp. If two elements have time distance smaller than the T seconds gap, they will be in the same session. Elements with larger time distance will be into different sessions, if there is no element to "close" the session gap between them [49]:

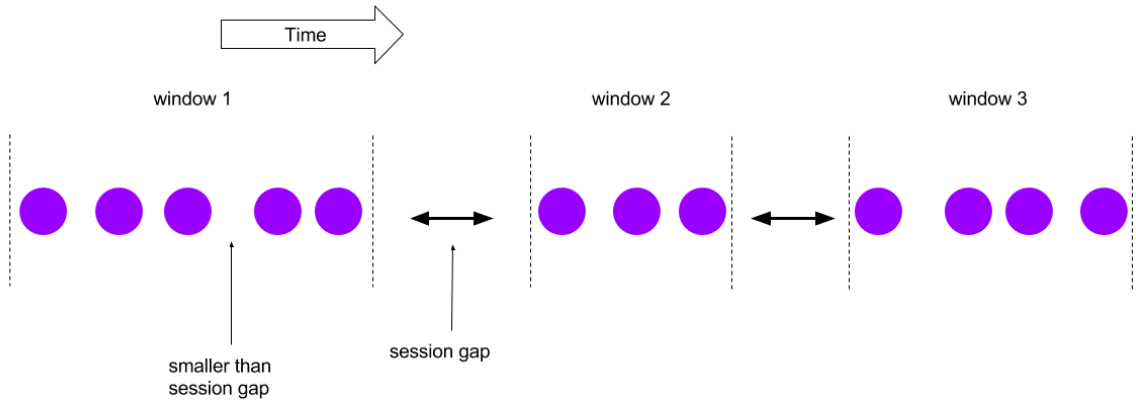


Fig 2.4.3.2.6.a, Session Windows [49].

Some of the above definitions can be clarified by the next figures:

Window types

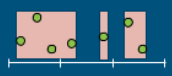
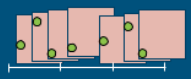
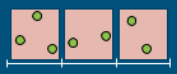
Tumbling window	"Every T seconds, emit the total for T seconds"	
Hopping window	"Every T seconds, emit the total for T2 seconds"	
Session window	"Emit groups of records that are separated by gaps of no more than T seconds"	
Sliding window	"Every record, emit the total for the surrounding T seconds" "Every record, emit the total for the surrounding R records"	

Fig 2.4.3.2.6.b, Window Types [13].

Tumbling, hopping & session windows in SQL

Tumbling window



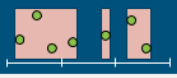
```
select stream … from Orders
group by floor(rowtime to hour)
```

```
select stream … from Orders
group by tumble(rowtime, interval '1' hour)
```

Hopping window

```
select stream … from Orders
group by hop(rowtime, interval '1' hour,
    interval '2' hour)
```

Session window



```
select stream … from Orders
group by session(rowtime, interval '1' hour)
```

Fig 2.4.3.2.6.c, Tumbling, Hopping and Session Windows [13].

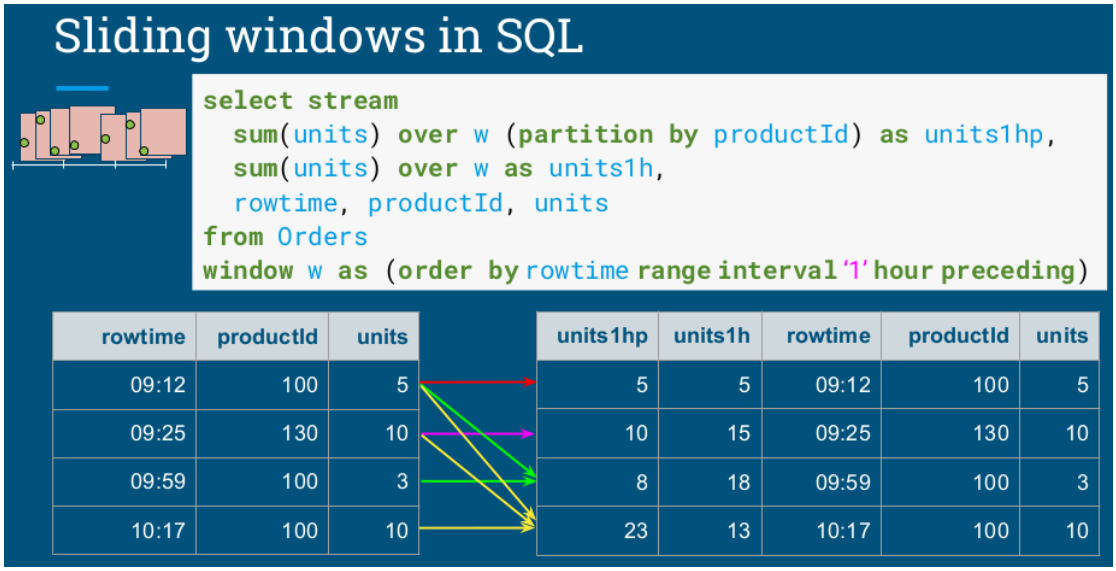


Fig 2.4.3.2.6.d, Sliding Windows [13].

2.5 Data Stream Management Systems

Stream Processing Engines (SPEs) are designed to perform SQL-style processing on the incoming records as they arrive, without necessarily storing them. In cases when it is necessary to store state, conventional SQL databases, embedded in the system, can be used for efficiency. These engines support high-volume and low-latency processing of stream inputs. Many SPEs have emerged in the last decades, serving different types of applications and introducing new characteristics in stream processing.

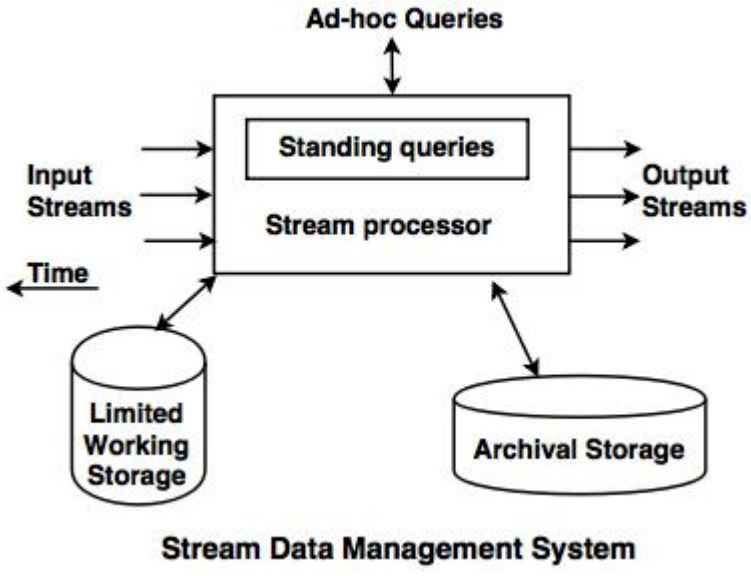


Fig. 2.5.a, Stream Data Management System.

Systems like Niagara [33] (an XML query processing engine), Borealis [29] (a distributed stream-processing system that inherits core stream-processing functionality from Aurora and distribution functionality from Medusa), STREAM [30] (a general-purpose prototype Data Stream Management System) and TelegraphCQ [31] (a system for implementing continuously adaptive query processing based on Eddy) set the foundations of modern data stream processing platforms. Some other well known, non-commercial and widely adopted options used in real-life applications are:

- Apache Storm [40]: Storm is an open source framework that provides massively scalable event collection, created by Twitter.
- Apache Spark [41]: A general framework for large-scale data processing that supports lots of different programming languages and concepts such as stream processing.
- Apache Flink [42]: An open source framework for distributed big data analytics.
- Apache Samza [43]: A distributed stream processing framework processor, recently open-sourced by LinkedIn.
- Esper [44]: Esper is an open-source Java-based software product for complex event processing (CEP [55]) and Event stream processing (ESP), that analyzes series of events for deriving conclusions from them.

However, most of these systems face difficulties in meeting the requirements that a streaming processing engine has to fulfill [32]. Systems based on batch processing, such as MapReduce frameworks, FlumeJava [45] and Spark [41], suffer from latency problems. Many systems cannot remain fault-tolerant at large scale inputs (e.g. Aurora [29], TelegraphCQ [31], Niagara [33], Esper [44]). Some other systems, provide fault tolerance and scalability on the cost of expressiveness or correctness. There are also cases, where systems lack the ability to provide exactly-once semantics, which results in the output correctness (Storm [40], Samza [43]). Others provide limited window semantics (only tuple or processing-time-based windows), such as Spark Streaming [41] or Trident [40]. Most of the existing streaming engines do not support the combination of event and process time-based windowing in their semantics. "*MillWheel and Spark Streaming are both sufficiently scalable, fault-tolerant, and low-latency to act as reasonable substrates, but lack high-level programming models that make calculating event-time sessions straightforward*" [32]. Also, Lambda Architecture systems may achieve many of the desired

requirements, but face several restrictions, and maybe should be replaced by Kappa Architecture systems [34].

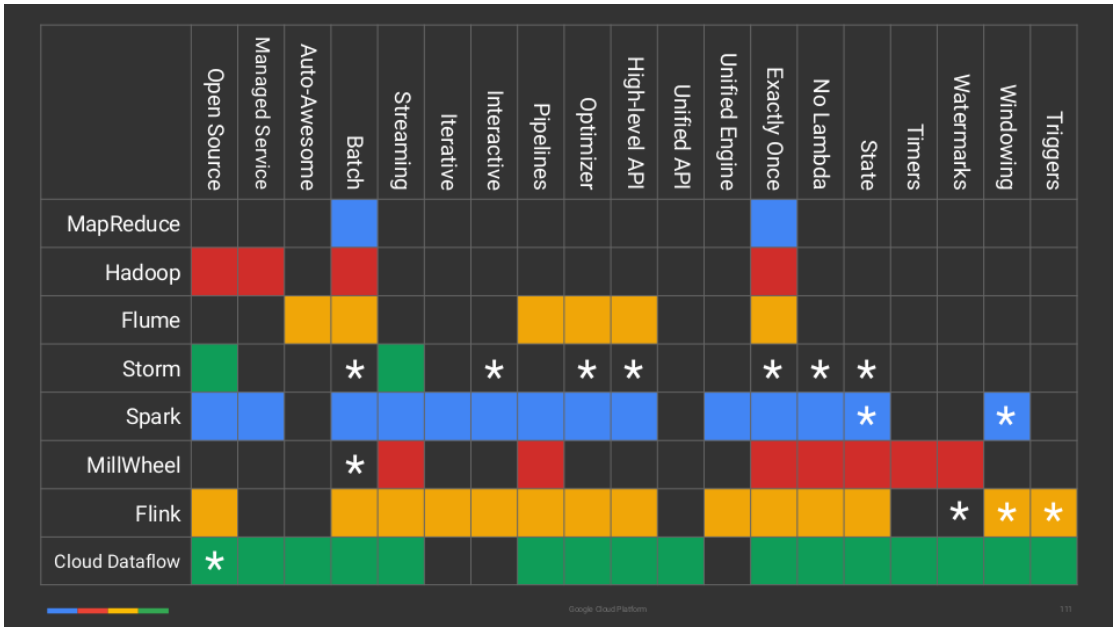


Fig. 2.5.b, Comparison of different Streaming frameworks [69].

Another very important shortcoming, that most of the above streaming systems face, is the lack of support for an SQL like declarative query language (e.g. Storm, Spark , Samza). As a result, good knowledge of imperative style programming and distributed systems is required to use these systems efficiently. By using an SQL-like language, that expands traditional SQL to serve the streaming semantics, we provide the system a structured way of expressing queries and we enable their optimization with well known database techniques used in traditional database systems. The optimization is not only restricted in known rules and algorithms of relational databases, but can be expanded to cover the streaming issues. This is the functionality that we try to introduce with our system to our chosen streaming processing engine, SABER [2] .

2.5.1 SABER

In this section, we will discuss some high-level concepts of SABER [2]. Previous research efforts on streaming processing engines, were trying to exploit either task parallelism or data parallelism on distributed or centralized environments, without considering the option of heterogeneous architectures. However, nowadays servers with heterogeneous

architectures are becoming available in data centers and cloud services, introducing a new source of parallelism for streaming processing. GPGPUs offer the opportunity of performing certain types of complex computation in high degrees of parallelism, and as a result maximize total system's throughput. SABER is a hybrid relational stream processing engine, that executes streaming SQL queries on such heterogeneous systems, in a data-parallel fashion. It uses all available CPU and GPGPU cores to achieve high processing throughput.

SABER' s key technical contributions are:

- SABER uses a hybrid stream processing model, in which tasks run in parallel across the CPU and GPGPU and are further parallelised across the GPGPU' s core. The scheduler uses a heterogeneous lookahead scheduling (HLS) algorithm, in order to assign each query task to the appropriate processor type. This algorithm matches a given query task to a heterogeneous processor based on past behaviour (highest throughput) and resolves possible conflicts, in order to optimize the utilization of system' s resources and increase the throughput.
- It supports sliding window semantics and maintains high throughput for small window sizes and slides. In contrast with other engines, SABER decouples window semantics from system performance. This has a performance impact in throughput, especially when window slides are small.
- It supports incremental computation when processing a query task with a sliding window, using results that were computed for preceding window fragments in the same batch.
- In order to avoid delays caused by data movement, SABER introduces a five stage pipelining mechanism to "hide" these overheads on the GPGPU.
- SABER' s memory management implementation minimizes memory allocation by:
    - Lazy deserialization: Tuples are deserialized only when they are needed and are stored in their byte representation. Also, deserialisation only generates primitive types.
    - Object pooling: SABER uses statically allocated pools of objects (one per thread) for all query tasks, and of byte arrays, for storing intermediate window fragment results, in order to avoid dynamic memory allocation.
- SABER's architecture consists of four processing stages:
    - Dispatching stage: creates fixed sized query tasks, which can be executed both in CPUs and GPUs, and place them in a system-wide queue.

- Scheduling stage: decides which task will be executed next by each processor using HLS.
- Execution stage: runs a query task, either on CPU cores or GPGPU, and evaluates the result by applying the batch operator function on the input window fragments.
- Result stage: reorders the query task results and assembles the window results by applying the assembly operator function on the window fragment results from the previous stage.

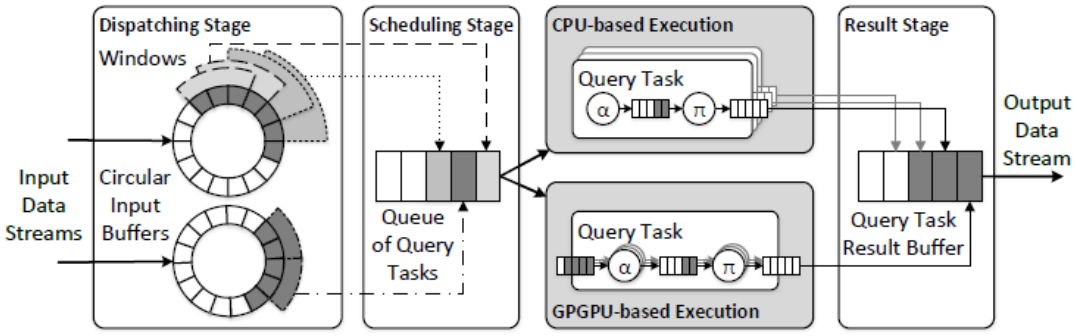


Fig. 2.5.1.a, Overview of the SABER architecture [2].

We used SABER as our streaming processing engine, because it is centralized (which helps in conducting experiments on a single machine) and supports sliding window semantics, but lacks SQL-like support and query optimization.

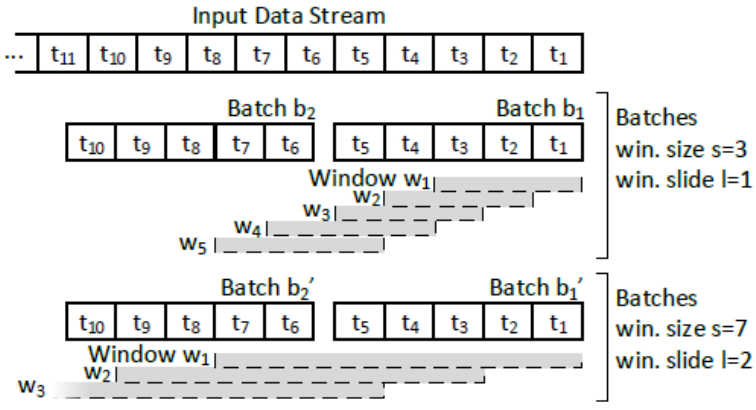


Fig. 2.5.1.b , Stream batches under two different window definitions [2].

3

Calcite Optimization Rules

Recall that we use rules to optimize our query plans. In this chapter we will focus mainly on built-in transformer rules of Calcite, which modify our logical plan by pattern matching, and some custom rules that we use in our implementation. After registering the rules that we want to use in our optimizer, they are applied when matching operands are found, according to their definition.

In Section 3.1, we describe the general idea of how to use the optimizer and we show an example of applying rules to a simple query plan. In Section 3.2, we describe some of the built-in planner rules that we used in our implementation. Finally, in Section 3.3, we present some custom transformer and converter rules that we created.

3.1 How to use the RelOptPlanner

These are the steps we have to follow in order to optimize a relational expression R :

- Choose the desired implementation of RelOptPlanner. We can choose between HepPlanner (heuristic optimizer similar to Spark Catalyst [12]) and VolcanoPlanner (dynamic programming optimizer) or a combination of them.
    - Register the relational expression R to the optimizer.
    - Create rule-calls for all the rules we want to apply.
    - Rank the rule calls by importance. The importance of a rule is determined by the probability to produce better implementation of a relational expression on the plan's critical path. As a result, it produces a member of an important relation and its children are cheap.
        - Call the most important rule.
        - Repeat.

A rule can be triggered by any of its matching operands in our relational expression. For example (taken from [18]) we have the rules:

1. PushFilterThroughProjectRule, which applies in operands (Filter (Project(...)))
2. CombineProjectsRule, which applies in operands (Project (Project(...)))

Consider the next query plan:

```
Project (deptno)                          [exp 1, subset A]
  Filter (gender='F')                     [exp 2, subset B]
    Project (deptno, gender, empno)       [exp 3, subset C]
      Project (deptno, gender, empno, salary)  [exp 4, subset D]
        TableScan (emp)                   [exp 0, subset X]
```

When we apply PushFilterThroughProjectRule to [exp 2, exp 3], we get the following plan:

```
Project (deptno)                          [exp 1, subset A]
  Project (deptno, gender, empno)         [exp 5, subset B]
    Filter (gender='F')                   [exp 6, subset E]
      Project (deptno, gender, empno, salary)  [exp 4, subset D]
        TableScan (emp)                   [exp 0, subset X]
```

We see that two new expressions are created by applying this rule. Expression 5 is in subset B (because it is equivalent to expression 2), and expression 6 is in a new equivalence class, subset E. In this new generated plan, we notice that new rules are enabled to be applied (CombineProjectsRule could not be triggered earlier), because of the transpose of the initial rules. As a result, two more rules are triggered:

The exp 5 triggers CombineProjectsRule to [exp 1, exp 5], which creates the following plan:

```
Project (deptno)                          [exp 7, subset A]
  Filter (gender='F')                     [exp 6, subset E]
    Project (deptno, gender, empno, salary)  [exp 4, subset D]
      TableScan (emp)                     [exp 0, subset X]
```

The newly registered exp 6 triggers PushFilterThroughProjectRule(exp 6, exp 4), which creates the following plan:

Project (deptno)	[exp 1, subset A]
Project (deptno, gender, empno)	[exp 5, subset B]
Project (deptno, gender, empno, salary)	[exp 8, subset E]
Filter (gender='F')	[exp 9, subset F]
TableScan (emp)	[exp 0, subset X]

Finally, CombineProjectsRule to [exp 7, exp 8] further reduces the depth of the tree to our final plan:

Project (deptno)	[exp 10, subset A]
Filter (gender='F')	[exp 9, subset F]
TableScan (emp)	[exp 0, subset X]

Rule matches can be triggered by a subset child for any of its parents. In the process of applying rules, new relexps could be registered, which can trigger several rules, including the rule that created it. Also, when enforcing rules, it is possible to merge subsets.

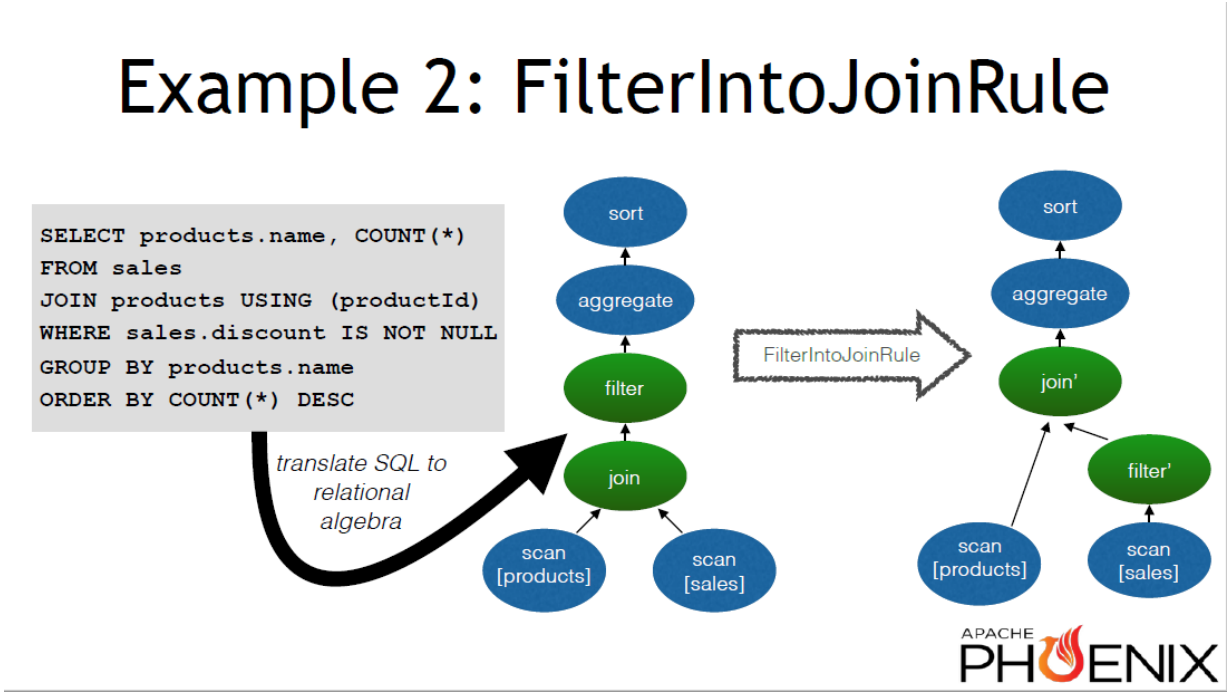


Fig. 3.1, FilterJoinRule application [53].

3.2 Built-in Planner Rules

In this section we describe the built-in rules used in our implementation. The following definitions are taken from [19].

3.2.1 AggregateExpandDistinctAggregatesRule

This is a planner rule that expands distinct aggregates from a LogicalAggregate. It has two instances:

- The default instance of the rule:

  For example,

```
SELECT productid, COUNT(DISTINCT orderid)
FROM orders
GROUP BY productid
```

  becomes:

```
SELECT productid, COUNT(distinct_orderid)
FROM (
        SELECT DISTINCT productid, orderid AS
              distinct_orderid
        FROM orders
        GROUP BY productid)
GROUP BY productid
```

- An instance of the rule that generates a Join

  For example,

```
SELECT productid, COUNT(DISTINCT orderid), MAX(units)
FROM orders
GROUP BY productid
```

  becomes

```
SELECT a.productid, c.count_orderid, a.max_units
FROM (
  SELECT deptno, MAX(units) as max_units
  FROM orders GROUP BY productid) AS a
```

```
    JOIN (
     SELECT productid, COUNT(orderid) AS count_orderid FROM (
          SELECT DISTINCT productid, sal FROM orders)
                AS b
          GROUP BY productid)) AS c
    ON a.productid = c.productid)
GROUP BY a.productid
```

3.2.2 AggregateFilterTransposeRule

This planner rule pushes an Aggregate operator through a Filter operator.

3.2.3 AggregateJoinTransposeRule

This rule pushes an Aggregate operator through a Join operator.

3.2.4 AggregateProjectMergeRule

This rule merges an aggregate with a projection, if the grouping expressions and arguments to the aggregate functions are field references. In some cases, this rule has the effect of trimming and as a result the aggregate will use fewer columns than the project did.

3.2.5 AggregateProjectPullUpConstantsRule

This rule is used to remove constant keys from an Aggregate. We deduce these constant fields using RelMetadataQuery.getPulledUpPredicates(RelNode). The constants are placed in a projection above the reduced Aggregate, in order to have a result that matches the original query.

3.2.6 AggregateReduceFunctionsRule

This rule reduces Aggregate functions in Aggregates to simpler forms. These reductions are:

- AVG(x) → SUM(x) / COUNT(x)

- STDDEV_POP(x) → SQRT((SUM(x * x) - SUM(x) * SUM(x) / COUNT(x)) / COUNT(x))
- STDDEV_SAMP(x) → SQRT((SUM(x * x) - SUM(x) * SUM(x) / COUNT(x)) / CASE COUNT(x) WHEN 1 THEN NULL ELSE COUNT(x) - 1 END)
- VAR_POP(x) → (SUM(x * x) - SUM(x) * SUM(x) / COUNT(x)) / COUNT(x)
- VAR_SAMP(x) → (SUM(x * x) - SUM(x) * SUM(x) / COUNT(x)) / CASE COUNT(x) WHEN 1 THEN NULL ELSE COUNT(x) - 1 END

The rule gathers common sub-expressions , because most of the above reductions require the use of COUNT(x).

3.2.7 AggregateRemoveRule

AggregateRemoveRule removes an Aggregate if it computes no aggregate functions. For example it removes GROUP BY c1, where c1 is a column with unique values and no aggregate functions have to be computed.

3.2.8 FilterAggregateTransposeRule

It is a planner rule that transposes a Filter operator with an Aggregate operator.

3.2.9 FilterJoinRule

With this rule we can push filters from above a join node into this node and/or its children or filters from within a join node to its children. It has several instances to cover all these cases:

- FILTER_ON_JOIN: This instance is used to push predicates from a Filter into the Join below them.
- JOIN: It is used to push predicates in a Join into the inputs to the join.
- DUMB_FILTER_ON_JOIN: This is a dumb version of FilterJoinRule used for testing.

3.2.10 FilterMergeRule

This rule merges to consecutive Filter operators into one.

3.2.11 FilterMultiJoinMergeRule

FilterMultiJoinMergeRule merges a LogicalFilter into a MultiJoin, creating a richer MultiJoin.

3.2.12 FilterProjectTransposeRule

Rule that transposes a LogicalFilter with a LogicalProject.

3.2.13 FilterRemoveIsNotDistinctFromRule

Planner rule that replaces IS NOT DISTINCT FROM in a LogicalFilter with logically equivalent operations. For example, it converts an expression like 'x IS NOT DISTINCT FROM y' to 'CASE WHEN x IS NULL THEN y IS NULL WHEN y IS NULL THEN x IS NULL ELSE x = y END'.

3.2.14 JoinAssociateRule

JoinAssociateRule changes a join based on the associativity rule. For example, we have a transformation like this ((a JOIN b) JOIN c) → (a JOIN (b JOIN c)) We can use JoinCommuteRule to convert (a JOIN (b JOIN c)) → ((a JOIN b) JOIN c).

3.2.15 JoinCommuteRule

Planner rule that permutes two consecutive Join operators. If needed, in order to preserve the order of columns in the output row, the rule adds a Project. It has two instances:

- The default INSTANCE : This instance swaps inner joins.
- SWAP_OUTER: This instance swaps outer joins and inner joins.

3.2.16 JoinPushExpressionsRule

JoinPushExpressionsRule pushes down expressions in “equal” join condition. For example, when we have a condition like "table1 JOIN table2 ON table1.column1 + 1 =

table2.column1", adds a project above "table1" in order to compute the expression "table1.column1 + 1".

3.2.17 JoinPushThroughJoinRule

This planner rule pushes the right input of a join through the left input of the join, if the left input is also a join. As a result, (A join B) join C becomes (A join C) join B. The advantage of applying this rule is that it may be possible to apply conditions earlier. For example:

```
(tableA join tableB on true)
join tableC
on tableA.col1 = tableC.col1 and tableC.col2 = tableB.col2
```

becomes

```
(tableA join tableC on tableA.col1 = tableC.col1)
join tableB
on  tableC.col2 = tableB.col2
```

Before the rule, one join has two conditions and the other has none (ON TRUE), while after enforcing the rule, each join has one condition. This rule has two instances:

LEFT: Instance of the rule for logical joins that pushes to the left.

RIGHT: Instance of the rule for logical joins that pushes to the right.

However, if they are both used in the heuristic planner, they cause exhaustive search.

3.2.18 JoinPushTransitivePredicatesRule

JoinPushTransitivePredicatesRule takes predicates from a Join and creates Filters if those predicates can be pushed to its inputs.

3.2.19 JoinToMultiJoinRule

This rule is used to flatten a tree of LogicalJoins into a single MultiJoin with N inputs. Join conditions are also pulled up from the inputs into the topmost MultiJoin. Here are examples

[19] of the MultiJoins constructed, after we have enforced this rule on the following join trees.

- A JOIN B → MJ(A, B)
- A JOIN B JOIN C → MJ(A, B, C)
- A LEFT JOIN B → MJ(A, B), left outer join on input#1
- A RIGHT JOIN B → MJ(A, B), right outer join on input#0
- A FULL JOIN B → MJ[full](A, B)
- A LEFT JOIN (B JOIN C) → MJ(A, MJ(B, C))), left outer join on input#1 in the outermost MultiJoin
- (A JOIN B) LEFT JOIN C → MJ(A, B, C), left outer join on input#2
- (A LEFT JOIN B) JOIN C → MJ(MJ(A, B), C), left outer join on input#1 of the inner MultiJoin
- A LEFT JOIN (B FULL JOIN C) → MJ(A, MJ[full](B, C)), left outer join on input#1 in the outermost MultiJoin
- (A LEFT JOIN B) FULL JOIN (C RIGHT JOIN D) → MJ[full](MJ(A, B), MJ(C, D)), left outer join on input #1 in the first inner MultiJoin and right outer join on input#0 in the second inner MultiJoin

3.2.20 LoptOptimizeJoinRule

This rule implements the heuristic planner for determining optimal join orderings. It is triggered by the pattern LogicalProject (MultiJoin). When this rule has access to accurate metadata, it can give almost optimal join ordering for many joins very fast. However, it is only capable of producing left-deep joins,

3.2.21 MultiJoinOptimizeBushyRule

This rule implements also a heuristic algorithm for optimal join orderings and it is triggered by the pattern LogicalProject (MultiJoin). It is similar to LoptOptimizeJoinRule. While LoptOptimizeJoinRule is only capable of producing left-deep joins, this rule is capable of producing bushy joins.

3.2.22 ProjectFilterTransposeRule

This rule transposes a Project operator with a Filter.

3.2.23 ProjectJoinTransposeRule

ProjectJoinTransposeRule pushes a Project operator through a Join operator, by splitting the projection into a projection on top of each child of the join.

3.2.24 ProjectMergeRule

This rule is used to merge two consecutive project operators to one, only if these operators aren't projecting identical sets of input references.

3.2.25 ProjectMultiJoinMergeRule

This planner rule is used to merge Project with a MultiJoin, in order to create a richer MultiJoin.

3.2.26 ProjectRemoveRule

This rule converts a Project node that merely returns its input into its child. For example, Project(ArrayReader(a), {$input0}) becomes ArrayReader(a).

3.2.27 ProjectToWindowRule

This rule splits a Project into two sections: the section that contains windowed aggregate functions and the section that does not. The first section with the windowed agg functions becomes instances of LogicalWindow. This rule is very useful for declaring windows for our streaming semantics. It has two instances:

- the default INSTANCE: This instance applies to a Calc that contains windowed aggregates and converts it into a mixture of LogicalWindow and Calc.
- PROJECT: This instance applies to a Project and produces a mixture of LogicalProject and LogicalWindow.

3.2.28 ProjectWindowTransposeRule

ProjectWindowTransposeRule transposes a LogicalProject with a LogicalWindow.

3.2.29 PruneEmptyRules

This is a set of rules that removes sections of a query plan known never to produce any rows. It has many different instances for most of the operators that Calcite supports:

- AGGREGATE_INSTANCE: it converts an Aggregate to empty if its child is empty.
- FILTER_INSTANCE: it converts a LogicalFilter to empty if its child is empty.
- JOIN_LEFT_INSTANCE: it converts a Join to empty if its left child is empty.
- JOIN_RIGHT_INSTANCE: it converts a Join to empty if its right child is empty.
- PROJECT_INSTANCE: it converts a LogicalProject to empty if its child is empty.
- SORT_FETCH_ZERO_INSTANCE: it converts a Sort to empty if it has LIMIT 0.
- SORT_INSTANCE: it converts a Sort to empty if its child is empty.
- UNION_INSTANCE: It removes empty children of a LogicalUnion.

3.2.30 ReduceDecimalsRule

ReduceDecimalsRule reduces decimal operations (such as casts or arithmetic) into operations involving more primitive types (such as longs and doubles). This rule helps us to deal with decimals in a consistent manner in Calcite. It can be is optionally applied, in order to support cases that we want to push down decimal operations to an external database.

3.2.31 ReduceExpressionsRule

This rules contain a set of rules that simplify various transformations on RexNode trees. There are two transformations:

- Constant reduction: it evaluates constant subtrees and replaces them with a corresponding RexLiteral.
- Removal of redundant cast: it removes casts that result in the same type as the expression used.

It has five instances:

- CALC_INSTANCE: reduces constants inside a LogicalCalc.
- EXCLUSION_PATTERN: Regular expression that matches the description of all instances of this rule and ValuesReduceRule also.
- FILTER_INSTANCE: reduces constants inside a LogicalFilter.
- JOIN_INSTANCE: reduces constants inside a Join.
- PROJECT_INSTANCE: reduces constants inside a LogicalProject.

3.2.32 TableScanRule

TableScanRule converts a LogicalTableScan to the result of calling RelOptTable.toRel(org.apache.calcite.plan.RelOptTable.ToRelContext).

3.2.33 SubQueryRemoveRule

SubQueryRemoveRule converts IN, EXISTS and scalar sub-queries into joins.

3.2.34 Other Rules

The above rules are the core rules used in our implementation. They are called transformer rules. Although some rules like ProjectFilterTransposeRule or JoinProjectTransposeRule don't optimize the general cases, they can be used in order to change the order of given operators and create the right conditions for other pattern matching rules to be applied. Finally, there are many other rules that are used to optimize RelNodes with Union, Minus, Intersect, Sort, Semi-Join, Subqueries or specific types of expressions, which are omitted as they don't match to the operators supported by our execution engine.

3.3 Custom Rules

In RBStream, we have divided the optimization procedure into separate phases, to make it more effective and more easily sustainable (it will be discussed in greater detail in the implementation chapter). These phases use either VolcanoPlanner or HepPlanner. The VolcanoPlanner is used in an intermediate optimization phase, with a subset of rules, to search dynamically in a certain search space and find the optimal solution. When using the VolcanoPlanner we have to use converter rules, that convert our operators to operators with

a certain convention and traits (e.g. Enumerable) in the output. However, most of the transformer rules we have described above, apply only to Logical operators and can't be used on Enumerable operators. As a result, we have to apply custom transformer rules, which don't effect the convention and the traits of the operators but change them to Logical. With this simple transformation, we are able to reuse our built-in rules in later phases, without having to rewrite them from the start. We describe these rules in Section 3.3.1.

We also created converter rules in order to introduce our custom operators, that follow the rate-based cost model used in RBStream. These converter rules convert the Logical operators to our custom convention operators and are described in Section 3.3.2.

Finally, in Section 3.3.3 we will show some custom transformer rules used either for optimization in certain cases or for preparing our final logical plan for its transformation to the corresponding physical plan in SABER.

3.3.1 Transformer Rules used after VolcanoPlanner

We have two implementations of this type of transformer rules for each operator. The operators used are: Aggregate, Filter, Join, Project, TableScan and Window. We have one transformer rule that transforms Enumerable operators (built-in convention) to Logical operators (e.g. EnumerableJoin becomes LogicalJoin) and one that transforms SaberRel operators (our custom convention) to Logical operators. In both cases, the convention and the traits are sustained. This is important, because each operator must have as input(s) only operators of the same convention. For example, a Join can't have as children an operator of Convention.NONE and an operator of Enumerable convention.

3.3.2 SaberRel Converter Rules

There are six converter rules, one for each of the operators. These rules convert the built-in Logical operators of Calcite to our custom SaberRel operators. SaberRel operators extend the base operator classes of Calcite and add our custom logic in the computation of plan's cost.

3.3.3 Custom Transformer Rules

3.3.3.1 FilterPushThroughFilterRule

This planner rule is used to transpose two consecutive filter operators, depending on their selectivity. The filter with lower selectivity is pushed down, in order to reduce the amount of data transferred to later stages of the query execution. This rule can be used instead of FilterMergeRule and increases greatly the output rate in certain cases.

In our implementation, FilterPushThroughFilter rule helped us test the third phase of our optimization procedure. It required the use of VolcanoPlanner (dynamic programming) and the proper definition of our rate-based cost model, in order to be enforced correctly.

3.3.3.2 JoinTableScanSupportRule

JoinTableScanSupportRule adds a Project operator over a TableScan that is an input of a Join. This transformer rule helps us convert our final logical plan to the corresponding physical plan that will be executed in SABER.

4

Cost Model

In this chapter we will focus on the cost model we used for estimating the optimal query execution plan. First, we present the purpose of our cost model in Section 4.1. In Section 4.2 we define our model parameters and how we estimate them.

4.1 Purpose

A well-defined cost model plays a crucial role in a Data Stream Management System (DSMS):

- We use the cost model to estimate the impact of optimizations on a query plan (e.g. stream rates, resource usage). As a result, it is a prerequisite for cost-based query optimization (and adaptive resource management).
- Runtime measurements give information about the presence and the past, while a cost model makes it possible to estimate system parameters in the future, if it is accurate.
- We can save computational cost by using an accurate cost model, because monitoring system parameters in runtime may be expensive.

4.2 Model Parameters

Our cost model is based on a rate-based model [37, 26], with specific extensions in order to match with our semantics. This cost model is used for static optimization with steady input rates but can be expanded to fit in an adaptive resource management system. We assume that the computational resources of our system are sufficient (feasible queries) and we don't use the load shedding technique of the initial model.

4.2.1 The Semantics of Sliding Window Continuous Queries

In our cost model we make some simplifying assumptions:

- Tuples arrive in the stream in a monotonically increasing order and there is no out of order arrival (it is taken care of by the DSMS).
- We don't use relational tables in the queries.
- Tuples arrive at stable rate.
- The computational resources of our system are sufficient.

We have the following operators:

- Selection operator: This operator takes as a input a stream S and gives as a result another stream O, whose elements are a subset of S that satisfy the predicates of the selection. It can be either a projection (selection operator with selectivity equal to 1) or a filter.
- Join operator: This operator represents sliding window joins of streaming sources. It is a symmetric operator that takes two input streams L and R, and for every arriving tuple on any of these streams it joins the tuple with the current window contents of the other input stream.
- Aggregate/Window operator: In our implementation, both aggregate and window operators output the same result. They take as input a stream S, compute the aggregate functions defined by the operator, and produce another stream O with these computations.

4.2.2 Rate and Window Parameters

Our basic parameters are rate and window size. With these parameters, we compute the resource requirements for an operator. Every operator takes as input rate(s) and active window size(s) and outputs a rate and an active window size, depending on the type of the operator. Active window is "*the average number of output elements that are eligible for participation as input if the output of the operator is fed into the input of a second one*"[37]. We assume stable arrival rate and that there is enough memory to hold the buffering requirements of our query plans. The cost-model is used for both tuple-based and time-based windows. In the case of time-based windows, the number of active tuples in a window i are given by the following equation: $W_i=\lambda_i*T$, where T is the size of the time-based window and λ_i the rate of the arrival tuples from source i. The variables used for estimating our costs are presented in the next figure:

Table 1. Variables used in estimating resource requirements.

C_σ	Cost of performing a selection on a single tuple
C_P	Cost to probe an active window for a matching tuple just arriving
C_I	Cost to insert an arriving tuple into the sliding window
C_V	Cost to invalidate an expired tuple from the sliding window
σ	Selectivity factor of a selection predicate
f	Join selectivity factor
λ_i	Rate of arrival of tuples from source i
W	Size of a tuple-based window
T	Size of a time-base window

Fig. 4.2.2, Variables of our cost model [37].

Next, we will present all the formulas of our model's parameters and resources for each operator. These are:

- λo: the output rate produced from each operator
- Wo: the active window size of the output
- C: the cost for using each operator
- M: the memory required for using each operator

4.2.2.1 Selections and Projections

The output rate depends on the selectivity σ of the selection (the tuples that qualify for the selection) and is equal to λo=σ*λi. The same applies to the output active window size, which is equal to Wo=σ*Wi. The cost of this operator is C(selection)=λi * Cσ and the memory resources needed for it are M=0, as it is stateless. Projections are a special case of selections, as they have σ=1.

4.2.2.2 Join

We can think of Join as a cartesian product, which passes through a filter defined by the join condition and only the tuples that qualify this condition are kept in the final output (selectivity f). The output tuples of a join are the tuples produced as a result of tuples arriving from the left side (f*WR*λL) added to the tuples produced as a result of tuples

arriving from the right side (f*WL*λR). Therefore, λo = f*WR*λL + f*WL*λR = f*(WR*λL + WL*λR). The active window size is estimated by the average number of valid tuples coming out of the join. There are WL and WR active tuples from the windows of both sides and because of the selectivity f, the output window size is equal to Wo=f*WR*WL. Kang et al. made an observation in [24], that the join cost can be divided into the cost of performing the left and the right parts of the join, as they are independent. The cost for each side is:

CL = λR * Cp(L) +λL*(Ci(R)+Cv(R)) and CR = λL * Cp(R) +λR*(Ci(L)+Cv(L)), so the total cost is C = (λL +λR)*(Cp +Ci+Cv). Finally, Join is a stateful operator, so it consumes memory equal to the sum of the active windows it uses, M=WR+WL.

4.2.2.3 Aggregate/Window

Window is assumed to have the same result of a corresponding aggregation in our implementation, and they are used to provide richer semantics for windows defined by Calcite's SQL. As a result the same formulas are used for both operators. For our estimations we utilize a built-in variable used from Calcite for aggregation cost (called multiplier) that is computed depending on the aggregation functions used. The aggregate operator is considered more expensive compared to Filter and Projection in the general case, and we secure this with the way that we estimate our variables. However, these estimations create the appropriate search space for possible shift between aggregate and join operators. The output rate is λo=multiplier*λi and the active window size is Wo=Wi. The cost of this operator is C=multiplier*Cσ*λi and the memory utilized is equal to the active window Wi used (M=Wi).

All the formulas of our operators are presented in the next table:

	Selection	Join	Aggregation/Window
λo	λi * σ	f*(WR*λL + WL*λR)	multiplier*λi
Wo	Wi * σ	f*WR*WL	Wi
C	λi * Cσ	(λL+λR)*(Cp+Ci+Cv)	multiplier*Cσ*λi
M	0	WR+WL	Wi

Fig. 4.2.2.3, Table of operators' cost functions.

4.2.3 Optimization Goal

Given a conjunctive query on streaming sources, we define two functions of any execution plan p for this query:

- λ(p): the throughput of the plan which is equal to $\lambda(p)=\prod_{i=1}^{n}\lambda i(p)$
- C(p): the utilization cost of the plan, $C(p)=\sum_{i=1}^{n}Ci(p)$

The objective of our optimization is to find the plan with the maximum R(p)= λ(p)/C(p), as we want to maximize the output rate and minimize the utilization cost. We also take into consideration the memory needed for each plan. We will evaluate our model in the evaluation chapter.

5

Design & Implementation

In the previous chapters, we provided the reader with the required background to understand the design choices made for the implementation of RBStream. The objective of this diploma thesis is the development of a system that takes continuous queries specified in SQL, generates optimized query evaluation plans and evaluates them against streaming data. This system extends our chosen streaming processing engine, SABER, by adding SQL support and an optimizer framework. More precisely, we created an integration of Apache Calcite with SABER, designed for streaming data sources only.

In Section 5.1, we describe the architecture of our system. In Section 5.2, we examine our system's streaming semantics. In Section 5.3, we discuss the optimization phases used in RBStream. In Section 5.4, we describe the implementation of the rate-based cost model. In Section 5.5, we present our Physical Rule Converter, which converts Calcite's logical plans into SABER execution plans. Finally, in Section 5.6, we describe some additional features of RBStream along with an interactive web interface.

5.1 RBStream Architecture

Our system is built on top of Apache Calcite, a popular SQL parser and optimizer framework which plays a central role in our architecture (Fig. 5.1.a). Below, we summarise the four stages of our system:

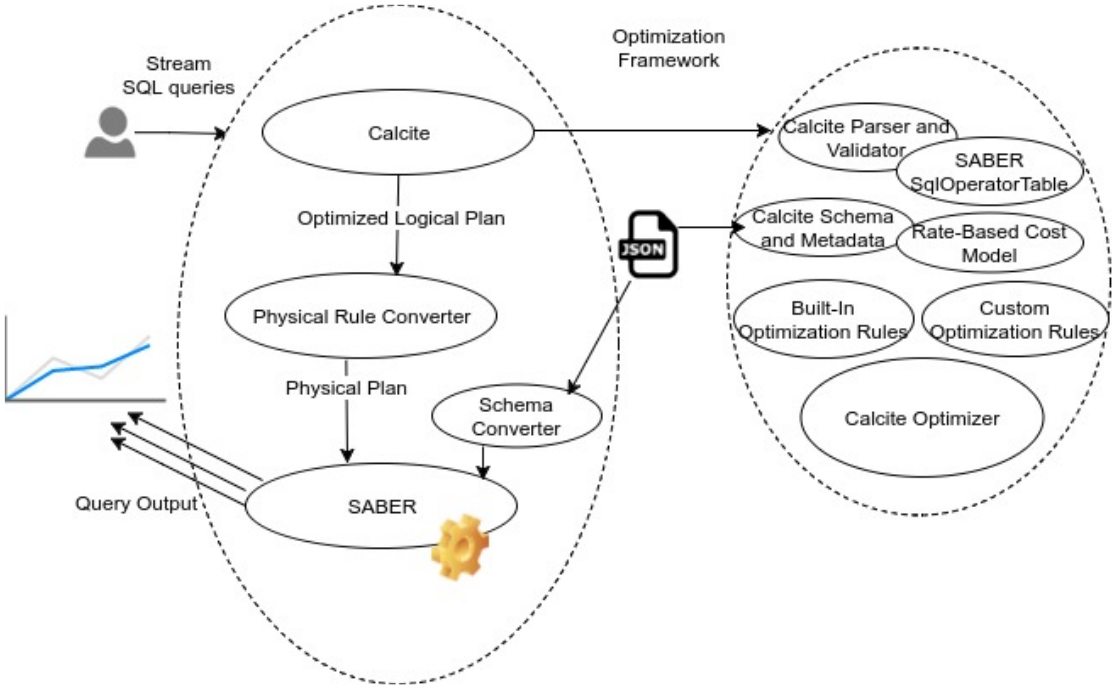


Fig. 5.1.a, RBStream Architecture.

PreOptimization Stage

At this stage, the user defines the input streaming sources to Calcite, either programmatically (using SchemaFactory, Schema, TableFactory and ScannableTable interfaces) or by declaring schemas and tables in a model JSON file. This schema is converted to the corresponding schema, tables and data types of SABER. Before submitting a query to the system, there are several parameters of the optimization procedure to choose from such as:

- Whether to use the rate-based cost model or the built-in cost model of Calcite.
- The algorithm of Join ordering: either the exhaustive algorithm, which is prefered for a small number of joins, or the heuristic algorithm, which is prefered in the general case. If the heuristic option is chosen, there are two algorithms for different cases, one that produces deep-left join orderings and one that produces bushy join orderings.

In addition, the user can customize the system configuration of SABER and choose the execution parameters, such as the size of the buffers used, the scheduling policy, the number of the threads that will be initialized or the execution mode of the system (CPU, GPU or both).

Optimization Stage

We use a combination of Calcite's Volcano and Heuristic planners in a phased manner. The optimization is separated into six phases, in order to be more effective and more easily sustainable. The effectiveness is occurred by the fact that in every phase only relevant subsets of rules are applied, reducing the search space and the time complexity of the whole optimization procedure. If all the rules were applied in a single phase, it would either take much time to compute the optimal plan of a complex query, when using the VolcanoPlanner, or it wouldn't give the optimal plan when using the HepPlanner as the order by which the rules are applied can lead to different solutions. Taking these observations into consideration, we have categorized our rules based on their functionality, ordered them accordingly to achieve the greatest result and used the proper planner in each phase. Therefore, we can add, remove and change rules depending only on the phase we want to affect in the optimization chain. The steps of this procedure are:

- The query is parsed and validated both against a Catalog of registered streaming data sources and custom implementations of SqlValidator (validates the parse tree and provides semantic information about it) and SqlOperatorTable (defines a directory for looking up SQL operators and functions) interfaces.
- Next, the query is converted into Calcite's representation of logical plans. This representation is a tree-like graph with sql operators as nodes and the flow of data between them as edges.
- Calcite's cost-based optimizer (CBO) [82] applies built-in and custom rules, depending on the chosen cost model. Our CBO implementation combines both VolcanoPlanner and HepPlanner and divides the optimization procedure into smaller parts. This helps us to take advantage of the independence of different rule types. Although the optimization procedure will generate many possible execution plans, finally only the one with the lowest cost will be chosen. In all cases below, the HepPlanner is used with Bottom Up matching order of applying the registered rules. The optimization phases are discussed more thoroughly later in this chapter.

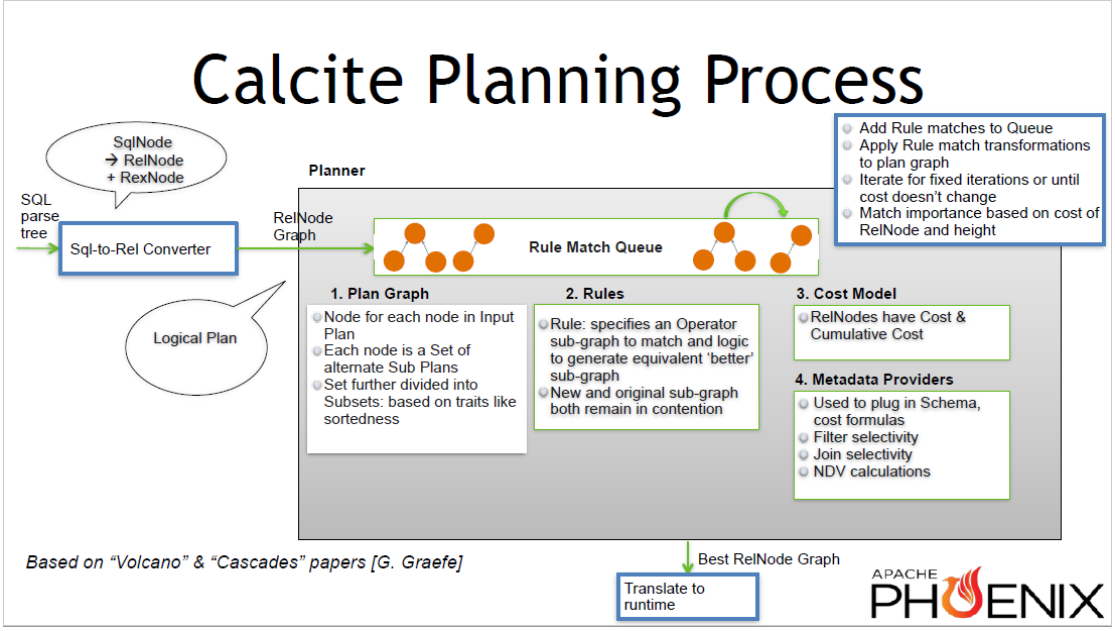


Fig. 5.1.b, Calcite Planning Process in detail [53].

Conversion Stage

In this stage the optimized logical plan is converted by our system's Physical Rule Converter to a physical plan, ready to be executed from SABER. Complex SQL queries are transformed to dozens of lines of code that can be executed in SABER, with a small overhead of milliseconds before the execution of the query.

Execution Stage. Finally, the physical plan is executed on SABER, and several metrics of the system are collected and presented to the user.

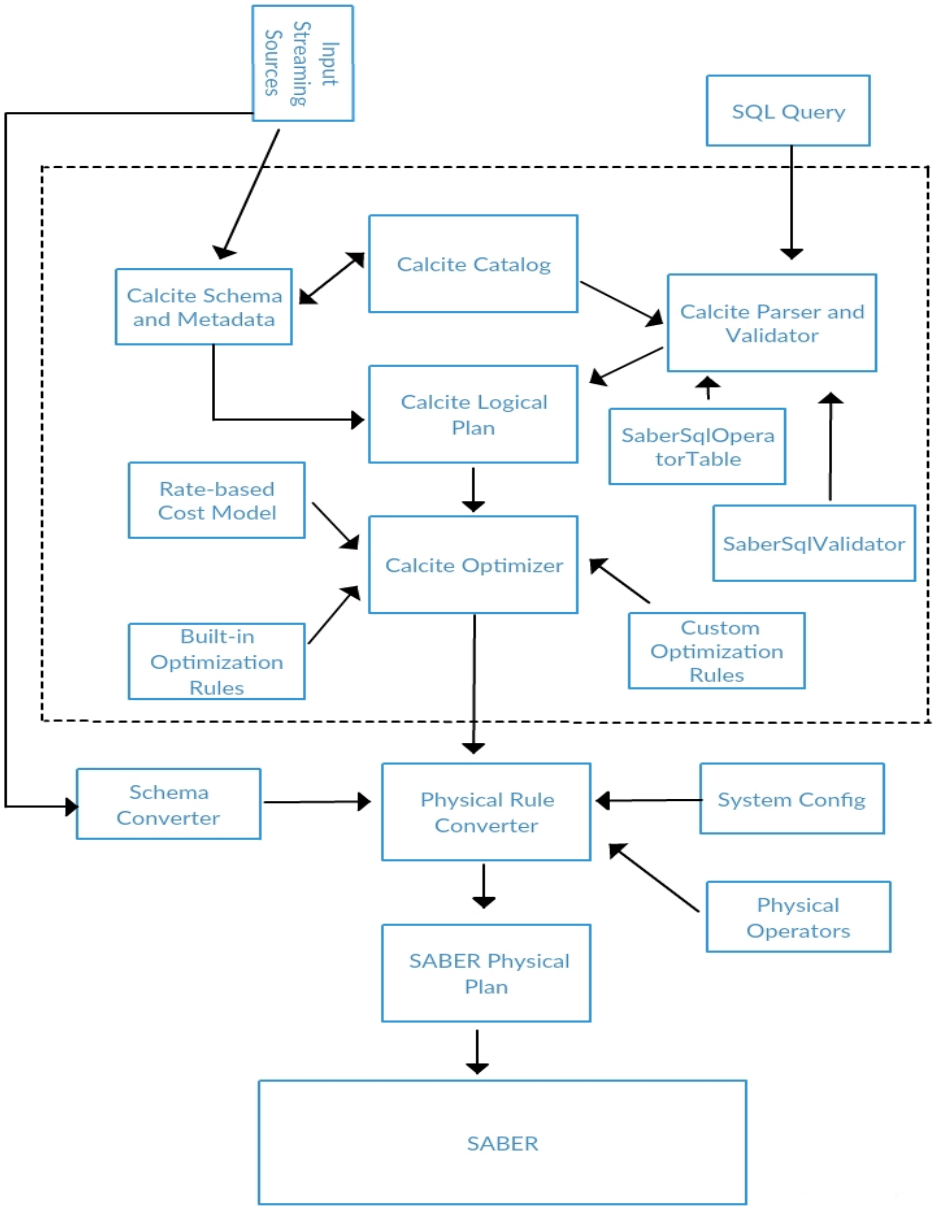


Fig. 5.1.c, More detailed RBStream Overview.

5.2 Streaming Semantics

We use the following functions of Calcite [14] to define our streaming SQL:

Scalar functions:

- FLOOR(dateTime TO intervalType) rounds a date, time or timestamp value down to a given interval type.
- CEIL(dateTime TO intervalType) rounds a date, time or timestamp value up to a given interval type.

Partitioning functions:

- HOP(t, emit, retain) returns a collection of group keys for a row to be part of a hopping window.
- HOP(t, emit, retain, align) returns a collection of group keys for a row to be part of a hopping window with a given alignment.
- TUMBLE(t, emit) returns a group key for a row to be part of a tumbling window.
- TUMBLE(t, emit, align) returns a group key for a row to be part of a tumbling window with a given alignment.

TUMBLE(t, e) is equivalent to TUMBLE(t, e, TIME '00:00:00').

TUMBLE(t, e, a) is equivalent to HOP(t, e, e, a).

HOP(t, e, r) is equivalent to HOP(t, e, r, TIME '00:00:00').

While SABER offers both time-based and row-based sliding windows of any size and slide, Calcite limits our options with its current implementation. Therefore, we have limited ways of expressing windows in our system at the moment:

- If no window is defined by SQL, we use the default window. In our implementation, the default window is a now-window, that is a time-based window of size 1.
- The user can define a tumbling window by using appropriately the FLOOR and CEIL functions in GROUP BY statement. At the moment, only time-based tumble windows can be defined by this way, which have the size of 1 sec, 1 minute, 1 hour or 1 day (expressed in nanoseconds). For example the next query:

  ```
  select rowtime, productid, sum(units)
  from s.orders
  group by rowtime, productid, floor(rowtime to hour)
  ```

  is converted to an Aggregation function of sum computed over a tumbling window with size one hour in nanoseconds (tumbling windows are sliding windows with size equal to slide) of the streaming data source s.orders. This window is converted to WindowType.RANGE_BASED window of range=slide=3600000 in SABER.
- The user can also define a time-based tumbling window by using TUMBLE, TUMBLE_START and TUMBLE_END functions. For example in this query:

  ```
  select tumble_end(rowtime, interval '2' hour) as rowtime
  ```

```
from s.orders
group by tumble(rowtime, interval '2' hour), productid
```

we have tumbling windows of size equal to 7200000 (two hours in nanoseconds). These functions give us the opportunity to express more complex tumble windows in contrary to the previous case, which can also be aligned by giving a third parameter to the TUMBLE function. This operation is not supported by SABER yet.

- The user can define both time and row based sliding windows of slide 1 with the use of OVER function and Windows. In our implementation, we don't use the OVER function with the definition we gave in Calcite Streaming Section, but as a tool for defining regular aggregate functions computed over sliding windows (SABER doesn't support OVER function yet). An example of a row-based window (WindowType.ROW_BASED in SABER) is:

```
select rowtime, productid, SUM(units) over pr
from s.orders
window pr as (PARTITION BY productid ROWS BETWEEN 5
      PRECEDING AND 10 FOLLOWING)
```

In this query we have an Aggregation function of sum computed over a sliding window of size 15 and slide 1, which is converted in SABER to WindowType.ROW_BASED window.

Respectively, an example of a time-based window with size 1000 and slide 1 is:

```
select rowtime, productid, SUM(units) over pr
from s.orders
window pr as (PARTITION BY rowtime, productid RANGE
INTERVAL '1' SECOND PRECEDING)
```

- The user can define only time-based sliding windows like before, but additionally with slide of size 1000 (1 sec), 60000 (1 minute), 3600000 (1 hour) or 86400000 (1 day) by using either FLOOR or CEIL functions in the partition statement of the window definition. For example:

```
select rowtime, min(orderid) over pr
from  s.orders
window pr as (PARTITION BY floor(rowtime to second) RANGE
```

```
INTERVAL '1' HOUR PRECEDING)
```

This query computes the minimum orderid over a sliding window of size 3600000 and slide 1000.

- Finally, the user can define time-based hopping windows with HOP, HOP_START and HOP_END functions. For example in the next query:

```
select hop_start(rowtime, interval '1' hour, interval '3'
hour) as rowtime, count(*) as c
from s.orders
group by hop(rowtime, interval '1' hour, interval '3'
hour);
```

we have a hopping window of size=10800000 (3 hours) and slide=3600000 (1 hour). With these functions, we can express more complex time-based sliding windows than we could in the cases above, as we have greater control over the slide parameter.

Window and aggregate functions are treated by SABER with the same way. As a result, we use Window functions only for defining windows in RBStream based on their expressiveness and not with their regular analytical functionality. In addition, session windows which are currently supported by Calcite's SQL, are not supported by SABER at the moment.

It is recommended that the user includes the rowtime in SELECT clause as seen in the above examples. Having a sorted timestamp in each stream and streaming query makes it possible to do advanced calculations later, such as GROUP BY and JOIN. We also have to note that we don't use the STREAM keyword of Calcite, as SABER is designed only for streaming data sources and we don't have to distinguish whether we have a static data source or not. By omitting the STREAM keyword, we simplify our system, because we don't use the extra streaming rules of Calcite.

5.3 Optimization Phases

In our implementation, Apache Calcite is used to introduce a Cost Based Logical Optimizer (CBO) to SABER. The purpose of the optimization is to increase the rate of the query's

result, while minimizing the size of the intermediate results created. These logical optimizations can improve SABER's query latency and ease of use greatly (for example, users don't have to submit an optimized query with the right join order, for it to be executed efficiently). Instead of table cardinality, our cost model uses input rate and active window size to decide which is the optimal Join ordering and reduce significantly query latency.

Query optimizations in a query processing engine can be classified in two categories: the logical query optimizations and the physical query optimizations. We will get involved only with the first category, as the second implies optimizations that have to do with the query execution in SABER (e.g. scheduling), which are performed by SABER internally, and choosing from different physical implementations (Hash Join vs B+tree Index Nested Loops Join [24]), which doesn't exist in SABER yet. Logical query optimizations can be determined based on relational algebra, regardless the physical layer in which query is executed.

Logical query optimizations applied in SABER can be categorized in:

- Optimization regarding Projection Pruning.
- Optimizations about Predicate Push Down.
- Rules that merge two consecutive Projections or Filters into one.
- Optimization rules for Join ordering.
- Rules about deducing Transitive Predicates.

However, not all of the rules used in our implementation (Chapter 3) can benefit from a CBO. In our case, only Join ordering and some certain rules, that transpose consecutive operators, belong in this category. Therefore, we use a combination of Calcite's Volcano and Heuristic planners in a phased manner (like Hive [38] or Drill [39]).

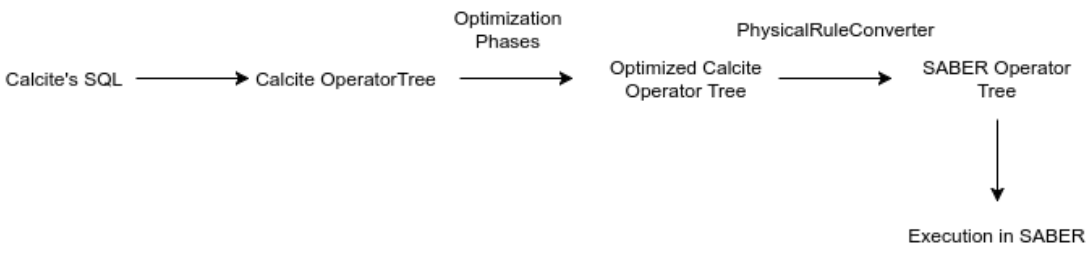


Fig. 5.3.a, From SQL to execution in SABER.

The optimization phases applied to a given plan, differ depending on the optimization type chosen from the user. If the user has chosen not to optimize the query, only rules that help us to declare windows for our streaming semantics (windowed aggregate functions) are used. The difference between the built-in and rate-based cost model optimization is in phases 3 and 4, in which we can have specific rules for each cost model, and in phases 5 and 6, which are reversed for the built-in cost model.

5.3.1 Phase 1: Window Support

In the first phase we enforce rules that rewrite our initial query plan and create Window operators using HepPlanner. We only use ProjectToWindowRule.PROJECT (3.2.27) and ReduceExpressionsRule.PROJECT_INSTANCE (3.2.31) in our implementation, as SABER doesn't support CALC operator [81]. This is a basic phase, as it provides us with the appropriate windowing support expressed by Calcite's SQL and helps the system with its streaming semantics.

5.3.2 Phase 2: Heuristic Rules that don't benefit from CBO

The rules that belong in this phase don't benefit from a CBO and are applied by the HepPlanner. These rules have to be enforced in general, regardless the cost model used in our implementation, as they reduce the overall cost. They can be categorized as follows:

- Distinct Aggregate Rewrite:

  the AggregateExpandDistinctAggregatesRule.INSTANCE (3.2.1). We have to run this optimization early, since it is expanding the operator pipeline.

- Merge, remove and reduce project if possible:
  - ProjectRemoveRule.INSTANCE (3.2.26)
  - ProjectWindowTransposeRule.INSTANCE (3.2.28)
  - ProjectMergeRule.INSTANCE (3.2.24)
  - ProjectTableScanRule.INSTANCE (3.2.7)
- Add constant propagation and folding, transitive inference, not null filters and support for WHERE style Joins:
  - FilterMergeRule.INSTANCE (3.2.10)
  - FilterAggregateTransposeRule.INSTANCE (3.2.8)

    - FilterProjectTransposeRule.INSTANCE (3.2.12)
    - FilterJoinRule.FILTER_ON_JOIN (3.2.9)
    - FilterJoinRule.JOIN (3.2.9)
    - JoinPushExpressionsRule.INSTANCE (3.2.16)
    - ReduceExpressionsRule.FILTER_INSTANCE (3.2.31)
    - ReduceExpressionsRule.PROJECT_INSTANCE (3.2.31)
    - ReduceExpressionsRule.JOIN_INSTANCE (3.2.31)
    - JoinPushTransitivePredicatesRule.INSTANCE (3.2.18)
    - AggregateProjectPullUpConstantsRule.INSTANCE (3.2.5)
    - AggregateReduceFunctionsRule.INSTANCE (3.2.6)
    - AggregateRemoveRule.INSTANCE (3.2.7)
- Prune empty result rules:
    - PruneEmptyRules.FILTER_INSTANCE (3.2.29)
    - PruneEmptyRules.PROJECT_INSTANCE (3.2.29)
    - PruneEmptyRules.AGGREGATE_INSTANCE (3.2.29)
    - PruneEmptyRules.JOIN_LEFT_INSTANCE (3.2.29)
    - PruneEmptyRules.JOIN_RIGHT_INSTANCE (3.2.29)

5.3.3 Phase 3: Pre-Join Rules that benefit from CBO

In phase 3, we want to search exhaustively for the optimal plan in the search space defined by our rules. As a result, we use the VolcanoPlanner. These rules are applied combined with converter rules, and return a query plan with operators that possess a certain Convention. When using the rate-based model, the converter rules change the build-in operators to our custom operators that support this model. This phase is essential in order to apply our later optimizations using the created cost model. The transformation rules used are:

- SABER_AGGREGATE_JOIN_TRANSPOSE_RULE
- SABER_FILTER_AGGREGATE_TRANSPOSE_RULE
- SABER_FILTER_PROJECT_TRANSPOSE_RULE
- SABER_PROJECT_FILTER_TRANSPOSE_RULE
- SABER_PROJECT_MERGE_RULE
- FilterPushThroughFilter

The rules above are 3.2.3, 3.2.8, 3.2.12, 3.2.22 and 3.2.24 rules respectively, which use our custom SaberRelFactory instead of Calcite's built-in RelFactory. In this way, we introduce our rate-based cost model and we optimize a given query plan based on this specific logic. If the user has chosen not to use the rate-based model, the same rules will be used, but with the built-in RelFactory and cost model.

5.3.4 Phase 4: Join Ordering Rules

This is the most important phase of complex queries with multiple joins. The user can choose either to use the VolcanoPlanner, in order to find the optimal join ordering with exhaustive search (appropriate only for less than 6 joins), or the faster heuristic implementation applied in HepPlanner (more useful in the general case). The statistics used for the Join ordering depend on the mode that the user has chosen: if it is the rate-based model, it uses the input rate and the active window sizes and if it is the built-in cost model, it uses the tables cardinality. In both cases, column boundaries can also help in the ordering estimation (e.g. number of distinct values).

With the VolcanoPlanner we use:

- JoinPushThroughJoinRule.LEFT (3.2.17)
- JoinPushThroughJoinRule.RIGHT (3.2.17)
- JoinAssociateRule.INSTANCE (3.2.14)
- JoinCommuteRule.INSTANCE (3.2.15)

With the HepPlanner we use:

- JoinToMultiJoinRule.INSTANCE (3.2.19)
- LoptOptimizeJoinRule.INSTANCE (3.2.20)

  or MultiJoinOptimizeBushyRule.INSTANCE (3.2.21)

5.3.5 Phase 5: Applying After-Join Rules

This is the final optimization phase that is used to enforce some last rules, in order to ensure that we get an optimized plan after we have found an optimal Join ordering. These are optimizations that don't need statistics such as:

- SABER_JOIN_PUSH_EXPRESSIONS_RULE,

- SABER_AGGREGATE_PROJECT_MERGE_RULE,
- SABER_AGGREGATE_JOIN_TRANSPOSE_RULE,
- ProjectRemoveRule.INSTANCE,
- SABER_PROJECT_MERGE_RULE,
- SABER_PROJECT_JOIN_TRANSPOSE_RULE,
- ProjectRemoveRule.INSTANCE,
- SABER_PROJECT_MERGE_RULE

The rules above use our custom SaberRelFactory instead of Calcite's built-in RelFactory.

Phase 5 is also divided into smaller phases, which group certain consecutive operators (for example Filter and Projection) into one composite operator. This operator is translated in a single Query Task with multiple operators in SABER, in order to exploit the execution granularity of specific use cases. In parallel computer, granularity can be defined as "*the ratio of computation time to communication time, wherein, computation time is the time required to perform the computation of a task and communication time is the time required to exchange data between processors"* [71]. By applying these rules and adjusting our task granularity, we explore a new dimension in our optimization process with an opportunity for performance increase. The reduction of thread communication improves throughput in some cases that we create larger tasks with more stateless operators.

5.3.6 Phase 6: Converting either Enumerable or SaberRel Operators to Logical Operators

This is a phase that doesn't change the arrangement of plan's operators. However, it is required in order to have a generic representation of the plans for the physical rule converter used later, regardless the convention enforced by the VolcanoPlanner in a previous stage. With the use of simple transformer rules, in order to keep the convention and the traits of our operators intact, we can support two different cost-models and types of operators.

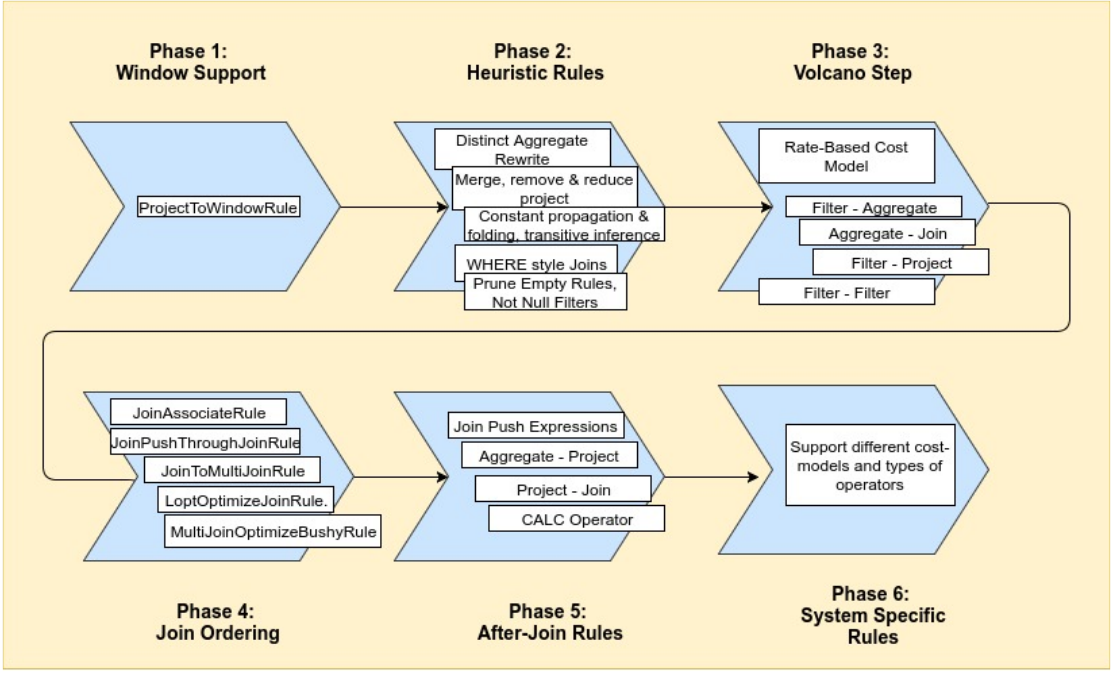


Fig. 5.3.b, The optimization phases.

5.4 Rate-based Cost Model

The rate-based cost model, discussed in Chapter 4, keeps track of cost in terms of rate and active window sizes of the input streaming sources. In our system, we propose a new RelOptCost implementation called SaberCostBase. SaberCostBase will compute the CPU usage, the memory utilization, the output rate, the active window size and the optimization goal R parameter (note that we use the reverse R of the definition given in 4.3) for each operator. The cost comparison algorithm will give importance to the optimization goal R first and then to the memory initialization when trying to find the optimal plan. This is the pseudocode of our implementation:

```
public boolean isLt(RelOptCost other) {
    SaberCostBase that = (SaberCostBase) other;
    if (true)
        return ( (this.R <= that.R) // R is the main optimization
                                    // goal. The R used here is the
                                    // reversed R from 4.2 Section.
            && (this.memory <= that.memory) // this is memory
```

```
                                                // utilization
            );
        return ((this.cpu + this.memory + this.R)
          < (that.cpu + that.memory + that.R)); //alternative way of
         // checking for the smaller cost
}
```

This is the function used from the planner in order to find the plan with the smaller cost, according to the optimization goal that is set. Therefore, we choose the plan with the smallest memory consumption from the plans that have the smallest parameter R.

In addition to the above changes, we had to rewrite the logic of the plus method of SaberCostBase, which is used to compute the accumulative cost, as we wanted both the rate and the active window size to be estimated by the product of the input rate and window size with a certain factor and not by addition:

```
public RelOptCost plus(RelOptCost other) {
    SaberCostBase that = (SaberCostBase) other;
    if ((this == INFINITY) || (that == INFINITY)) {
      return INFINITY;
    }
    return new SaberCostBase(
        this.rowCount + that.rowCount,
        this.cpu + that.cpu,
        this.io + that.io,
        this.rate,
        this.memory + that.memory,
        this.window,
        this.R);
}
```

Following are the cost variables that will be used in cost computation:

```
public static final int BASE_CPU_COST = 1;
```

```
public static final int PROJECT_CPU_COST = 4 * BASE_CPU_COST;
public static final int HASH_CPU_COST = 8 * BASE_CPU_COST;
public static final int Cs = PROJECT_CPU_COST;
public static final int Cj = HASH_CPU_COST;
```

The first three variables are taken from Drill [58], because they helped us to make estimations as much accurate as possible. These numbers give a balanced analogy of the costs that we wanted to describe. With these cost variables we try to compute the cost variables Cs and Cj of our model, which are defined in Section 4.2. The cost of performing a join includes the cost of probing, insertion and deletion needed: Cj=(Cp+Ci+Cv).

In order to use the rate-based cost model, we had to override the ComputeSelfCost() method of all the operators used in our implementation (custom SaberRelBase operators):

- SaberAggregateRelBase

```
@Override public RelOptCost computeSelfCost(RelOptPlanner
      planner, RelMetadataQuery mq) {
    RelOptCost previousCost = planner.getCost(this.input, mq);
    double rowCount = mq.getRowCount(this);
    float multiplier = 1f + (float) aggCalls.size() * 0.125f;
    for (AggregateCall aggCall : aggCalls) {
      if (aggCall.getAggregation().getName().equals("SUM")) {
        // Pretend that SUM costs a little bit more than $SUM0,
        // to make things deterministic.
        multiplier += 0.0125f;
      }
    }
    double inputRate = ((SaberCostBase) previousCost).getRate();
    double outputRate = multiplier * inputRate;
    double cpuCost = multiplier * SaberCostBase.Cs * inputRate
    double window = ((SaberCostBase)previousCost).getWindow();
    double memory = window;
    double R = (((SaberCostBase) previousCost).getCpu() +
      cpuCost) / outputRate;
```

```
    SaberCostFactory costFactory =
      (SaberCostFactory)planner.getCostFactory();
    return costFactory.makeCost(multiplier * rowCount, cpuCost,
      0, outputRate, memory, window, R);
}
```

- SaberFilterRelBase

```
@Override public RelOptCost computeSelfCost(RelOptPlanner
      planner, RelMetadataQuery mq) {
      RelOptCost previousCost = planner.getCost(this.input, mq);
      double rowCount = mq.getRowCount(this);
      double selectivity = mq.getSelectivity(this.getInput(),
            this.getCondition());
      double rate = selectivity * ((SaberCostBase)
            previousCost).getRate();
      double cpuCost = SaberCostBase.Cs * rate;
      //System.out.println("selectivity:" + selectivity );
      double window =  selectivity * ((SaberCostBase)
            previousCost).getWindow();
      window = (window < 1) ? 1 : window; // fix window size in
                                          // order to be >= 1
      double R = (((SaberCostBase) previousCost).getCpu() +
            cpuCost)/rate;
      if (Double.isNaN(R))
            R = Double.MAX_VALUE;
      if (Double.isInfinite(rate))
            rate = Double.MAX_VALUE;
      if (Double.isInfinite(cpuCost))
            cpuCost = Double.MAX_VALUE;
      SaberCostFactory costFactory =
            (SaberCostFactory)planner.getCostFactory();
      return costFactory.makeCost(rowCount, cpuCost, 0, rate, 0,
            window, R);
```

```
}
```

- SaberJoinRelBase

```
@Override public RelOptCost computeSelfCost(RelOptPlanner
      planner, RelMetadataQuery mq) {
      RelOptCost previousLeftCost = planner.getCost(this.left,
            mq);
      RelOptCost previousRightCost = planner.getCost(this.right,
            mq);
      double rowCount = mq.getRowCount(this);
      double selectivity = mq.getSelectivity(this.left,
            this.getCondition()); //fix it
      //System.out.println("selectivity:" + selectivity );
      double leftRate = ((SaberCostBase)
      previousLeftCost/*mq.getCumulativeCost(this.left)*/).getRa
            te();
      double rightRate = ((SaberCostBase)
            previousRightCost).getRate();
      double leftWindow = ((SaberCostBase)
            previousLeftCost).getWindow();
      double rightWindow = ((SaberCostBase)
            previousRightCost).getWindow();

      double rate = selectivity * (leftRate*rightWindow +
            rightRate*leftWindow);
      double cpuCost = SaberCostBase.Cj * (leftRate +
            rightRate);
      double memory =  leftWindow + rightWindow;
      double window =  selectivity * leftWindow * rightWindow;
      window = (window < 1) ? 1 : window; // fix window size in
                                          // order
      double R = (((SaberCostBase) previousLeftCost).getCpu() +
```

```
            ((SaberCostBase) previousRightCost).getCpu() +
        cpuCost) / rate;

        SaberCostFactory costFactory =
            (SaberCostFactory)planner.getCostFactory();
        return costFactory.makeCost(rowCount, cpuCost, 0, rate,
            memory, window, R);
    }
```

- SaberProjectRelBase

```
    @Override public RelOptCost computeSelfCost(RelOptPlanner
        planner, RelMetadataQuery mq) {
        RelOptCost previousCost = planner.getCost(this.input, mq);
        double rowCount = mq.getRowCount(this);
        double rate = ((SaberCostBase) previousCost).getRate();
        double cpuCost = SaberCostBase.Cs * rate;
        double window = ((SaberCostBase)
            previousCost).getWindow();
        List<RexNode> projectedAttrs = this.getChildExps();
        double windowRange = 0; // find it in a better way
        for (RexNode attr : projectedAttrs){
         if (!(attr.getKind().toString().equals("INPUT_REF"))) {
            Pair<Expression, Integer> pair = new
                ExpressionBuilder(attr).build();
            if (pair.right > 0) {
                // W = T * λi
                windowRange = pair.right * rate;
            }
         }
        }
        window = (windowRange > 0) ? windowRange : window;
        double R = (((SaberCostBase) previousCost).getCpu() +
            cpuCost) / rate;
```

```
            if (Double.isInfinite(R))
                  R = Double.MAX_VALUE;
            if (Double.isInfinite(rate))
                  rate = Double.MAX_VALUE;
            if (Double.isInfinite(cpuCost))
                  cpuCost = Double.MAX_VALUE;
            SaberCostFactory costFactory =
                  (SaberCostFactory)planner.getCostFactory();
            return costFactory.makeCost(rowCount, cpuCost, 0, rate, 0,
                  window, R);
   }
```

- SaberTableScanRelBase

```
@Override public RelOptCost computeSelfCost(RelOptPlanner planner,
     RelMetadataQuery mq) {
     double rowCount = mq.getRowCount(this); // this is the rate
     double window = 1;
     SaberCostFactory costFactory =
           (SaberCostFactory)planner.getCostFactory();
     return costFactory.makeCost(rowCount, 0, 0, rowCount, 0, window,
           0);
 }
```

- SaberWindowRelBase: it has the same ComputeSelfCost() method as Aggregation, because of the semantics used in our system.

The computation of each operator's cost is based on the table with the used parameters under the 4.2.2.3 Section. This cost is estimated every time we compute the cumulative cost of a set of operators that constitute a continuous query.

Finally, we have created our custom MetaDataProvider implementation, SaberDefaultRelMetaDataProvider, in order to control how to gather metadata from our streaming sources (e.g. how to get an estimate of distinct rows over a stream data source).

5.5 Physical Rule Converter

The conversion of Calcite's logical plan to the corresponding physical plan in SABER plays a crucial role in our system's implementation. Some lines of SQL are transformed to dozens of lines of code, ready to be executed in SABER. PhysicalRuleConverter class is responsible for constructing the physical operators tree from a given logical plan. The recursive algorithm used for the construction of the physical plan along with its execution method are presented with the following pseudocode:

```
HashMap chainOfOperators
Map tablesMap
HashSet queries
List aggregates

void convert (logicalPlan) {
      // If the given logical plan has no inputs, it represents a
      // simple
      // LogicalTableScan (select * from stream). This case is treated
      // seperately.
      if logicalPlan.getInputs() == 0 then
            convertSingleRelNode(logicalPlan)
      else convertMultipleRelNodes(logicalPlan)
}

void convertSingleRelNode(logicalPlan) {
      inputStream = logicalPlan.getStreamFrom(tablesMap)
      create a Project Operator over the inputStream
      add the operator in the queries HashSet
}
```

```
Pair<Integer, String> convertMultipleRelNodes(logicalPlan) {
      List children = logicalPlan.getInputs();
      if children.size() == 0 then {
            inputStream = logicalPlan.getStreamFrom(tablesMap)
            create a Project Operator over the inputStream
            add parameters that will be used by later operators in the
            chainOfOperators HashMap
            add the operator in the queries HashSet
            return (logicalPlan.getId(), logicalPlan.getRelTypeName)
      }
      else if children.size() == 1 then {
            // create its child
            chainTail = children.get(0)
            Pair <Integer, String> node =
                  convertMultipleRelNodes(chainTail)
            get the parameters needed from the child operator using
            chainOfOperators
            create the given Operator with these parameters
            add parameters that will be used by later operators in the
            chainOfOperators HashMap
            add the operator in the queries HashSet
            if the child isn't a LogicalTableScan then connect it with
            its child
            if the operator is Aggregate or Window then add it to the
            aggregates list
            return (logicalPlan.getId(), logicalPlan.getRelTypeName)
      }
      else if children.size() == 2 then {
            leftChainTail = children.get(0)
            // create left child
            Pair <Integer, String> leftNode =
                  convertMultipleRelNodes(leftChainTail)
```

```
            rightChainTail = children.get(1)
            // create right child
            Pair <Integer, String> rightNode =
                  convertMultipleRelNodes(rightChainTail)
            get the parameters needed from the children operators
            using chainOfOperators
            create the given Operator with these parameters
            add parameters that will be used by later operators in the
            chainOfOperators HashMap
            add the operator in the queries HashSet
            if the  left child isn't a LogicalTableScan then connect
            with it
            if the  right child isn't a LogicalTableScan then connect
            with it
            return (logicalPlan.getId(), logicalPlan.getRelTypeName)
      }
}

void execute () {
      QueryApplication application = new QueryApplication(queries);
      application.setup();
      /* The path is query -> dispatcher -> handler -> aggregator */
      for ( SaberRule agg : aggregates){
            if (systemConf.CPU)
                  agg.getQuery().setAggregateOperator((IAggregateOpera
                  tor) agg.getCpuCode());
            else
                  agg.getQuery().setAggregateOperator((IAggregateOpera
                  tor) agg.getGpuCode());
      }
      /* Execute the query. */
      while (true) {
            for (Map.Entry<Integer,ChainOfRules> c :
```

```
                chainOfOperators.entrySet())
                        if(c.getValue().getIsFirst()) {
                                if(c.getValue().getFlag() == false) {
                                        application.processData
                                                (c.getValue().getData());
                                } else {
                                        if(c.getValue().getHasMore() == true) {
                                                application.processSecondStream
                                                        (c.getValue().getData());
                                        } else {
                                                application.processFirstStream
                                                        (c.getValue().getData());
                                                application.processSecondStream
                                                        (c.getValue().getData2());
                                        }
                                }
                        }
                }
        }
}
```

In the first case, when the RelNode has zero children nodes, we can assume that it is a LogicalTableScan. These type of operators are created only if the whole given logical plan consists only of one LogicalTableScan. In all other cases, we use dumb implementations of the LogicalTableScan operator that help us build the later operators. This is the reason why we treat separately these two cases. In the case that the RelNode has one child, it is Project, Filter, Aggregate or Window and we must build all of its children before creating it. Finally, a RelNode with two children is a Join, and we also have to construct its both descendants before creating it.

The execute() method is used to execute our created physical plan. We use the chainOfOperators map and the aggregates list for all the parameters needed. Data are processed in a pipelined procedure by SABER in the while loop.

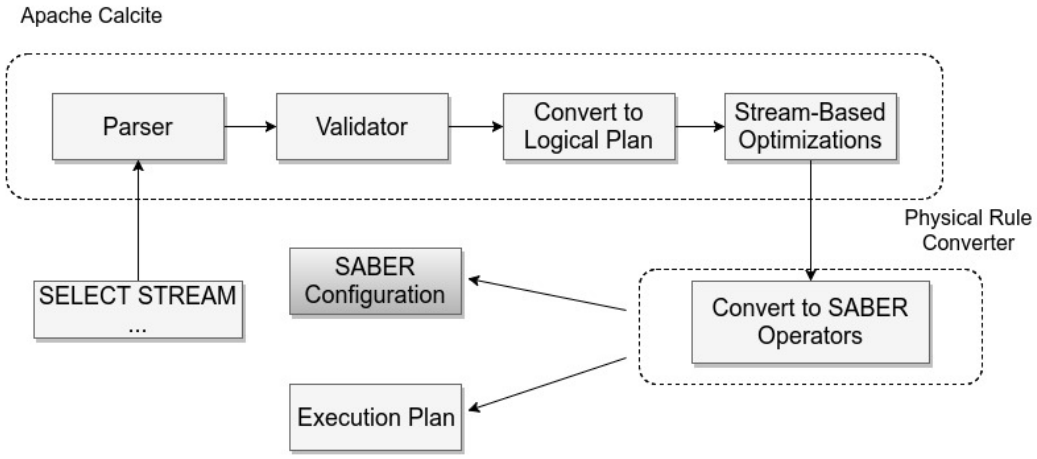


Fig 5.5.a, Query Planner.

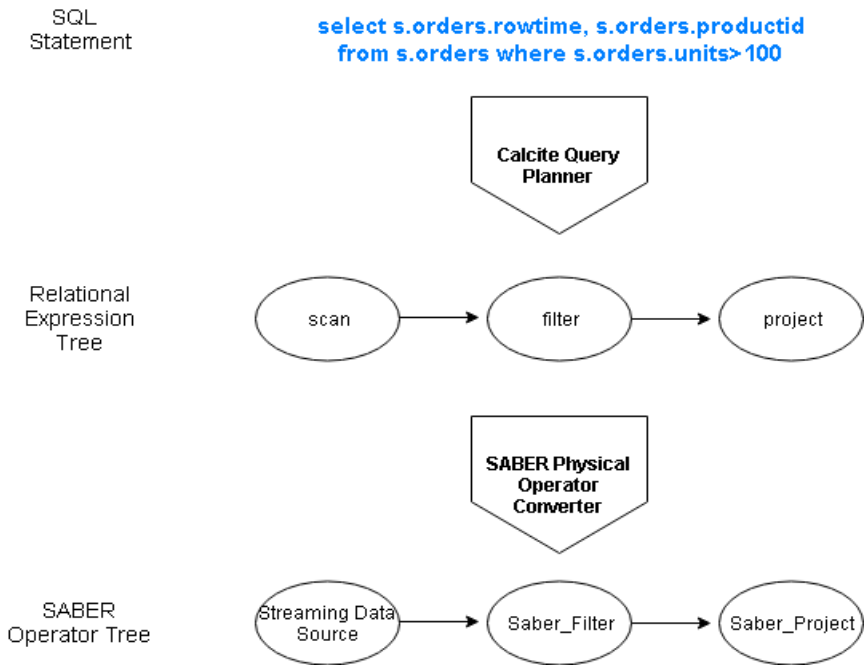


Fig 5.5.b, Simple SQL query example.

5.5.1 Physical Operator Construction

In order to facilitate better the understanding of the conversion procedure, we will explain how the physical operators are constructed in this subSection. The PhysicalRuleConverter class is the main class that orchestrates the conversion of a logical plan to physical. It uses

six rules as to achieve its task: SaberAggregateRule, SaberFilterRule, SaberJoinRule, SaberProjectRule, SaberScanRule and SaberWindowRule. These rules are specific to the implementation of operators in SABER and are not related to the Calcite rules we used before.

The basic components used by the physical conversion rules are:

5.5.1.1 AggregationUtil

This class implements the construction of Aggregates and Group By attributes for SABER. It takes as an input a list with the aggregates (aggregate.getAggCallList()) and a bit set with the group by attributes (aggregate.getGroupSet()) of either an Aggregate or a Window operator, and creates the corresponding structures in SABER. It also provides us with the correct output schema, which is essential in the pipelined procedure in which we have to define our operators. The count function always references the first column, because the user can write COUNT(*) or COUNT() in Calcite's SQL. At the moment SABER supports only five aggregates: min, max, count, sum and avg. The next query:

```
select rowtime, sum(units), count(orderid)
from  s.orders
group by rowtime,units,orderid, floor(rowtime to hour)
```

is converted in SABER as a window of range and slide equal to 3600000. The number of group by attributes is four, as expected, and we compute the sum and count of the corresponding columns. The aggregate operator used contains incremental aggregation types. However, in run time the incremental computation is not activated, as the sizes of the range and slide are equal. The incremental computation is enabled only when the slide of the window is smaller than its range.

5.5.1.2 PredicateUtil

This class is used by SaberJoinRule and SaberFilterRule to convert the predicates from a given Filter or Join RelNode to corresponding structures used by SABER. The algorithm used is recursive in order to support complex conditions written in SQL. For example in this query:

```
select *
from s.orders
```

```
where (((productid = 5) or (units > 25 and units <100)) and
        customerid=100) and 136>42
```

the where condition from SQL is converted in SABER as:

(("2" = Constant 5) OR (("3" > Constant 25) AND ("3" < Constant 100))) AND ("4" = Constant 100) AND (Constant 136 > Constant 42)

5.5.1.3 ExpressionBuilder

In SaberProjectRule we use the ExpressionBuilder class to construct complex projection expressions for SABER. This class uses a recursive algorithm in order to built the corresponding projection from a given operator's tree. At the moment it supports only simple expressions with mathematical operations and the RexCalls CEIL and FLOOR. It also helps us to define the window semantics of RBStream. For example:

```
select ((units+10) * 25 ) /100
from s.orders
```

is converted in SABER as : (/(*(+($3, 10), 25), 100): int).

5.6 Implementation Features

In this final Section, we will present the graphical interface and some ancillary classes, that automate some basic functionalities of RBStream.

5.6.1 DataGenerator

DataGenerator, as its name suggests, is used to generate data for our system. It can either generate dummy data or be used to connect our system with certain data sources.

5.6.2 SchemaConverter

This class converts a given Calcite schema to the corresponding schema in SABER. At the moment, SABER supports only integers, floats and longs. As a result, when the converter finds a RelDataType that is not supported by SABER, it converts it to integer in our system's schema by default. SchemaConverter doesn't support nested schemas yet.

5.6.3 SystemConfig

This class is used to configure the SABER before executing a query. Depending on the parameters chosen, the system may fail to execute a query due to lack of enough memory for initializing Circular and Unbounded buffers or the Hash Table size. With this class we also define whether the GPU mode is used and the number of threads of RBStream.

5.6.4 Graphical Interface of RBStream

For demonstrating our system, we used a comprehensive, real-time GUI that users will utilize to interact over dataset, query and comparison levels. We used Jupyter Notebook [62] to create an interactive web interface to present our system. The UI controls a centralized SABER deployment over a virtual machine from the ~Okeanos IaaS [61].

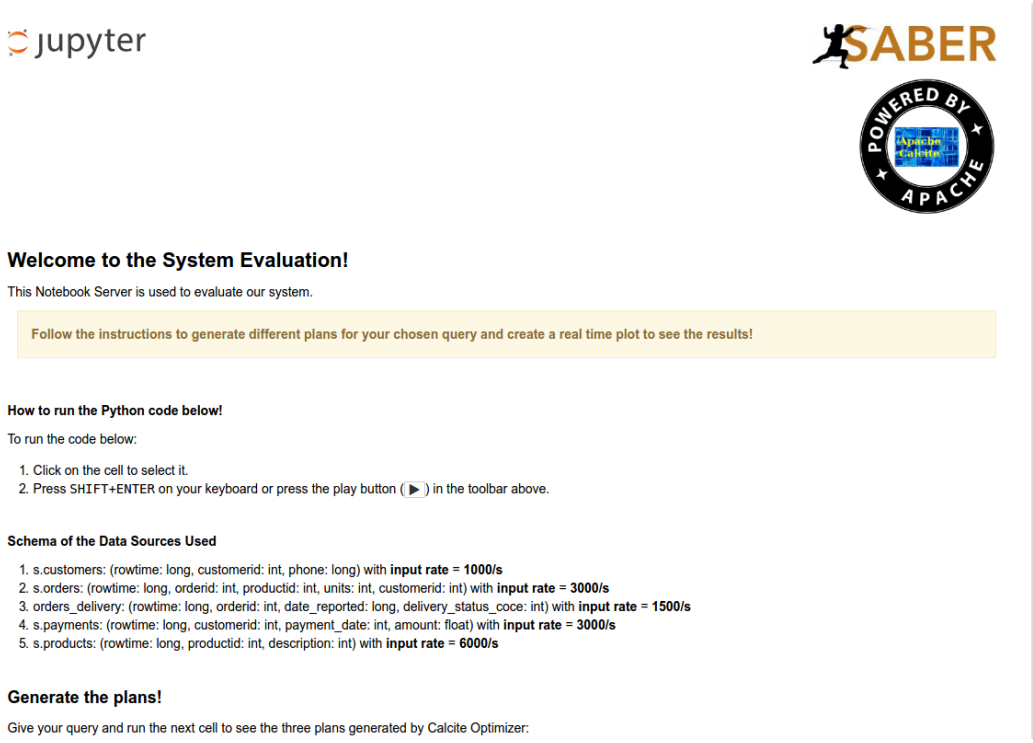


Fig. 5.6.4.a, Jupyter Notebook.

Users are given the choice to interact with two different datasets. We provide:

(i) Linear Road Benchmark (LRB) for evaluating stream processing performance.

The benchmark models a network of toll roads, in which incurred tolls depend on the level of congestion and the time-of-day. Tuples in the input data stream denote position events of vehicles on a highway lane, driving with a specific speed in a particular direction.

(ii) A Purchase-Orders Transaction Benchmark, with custom data.

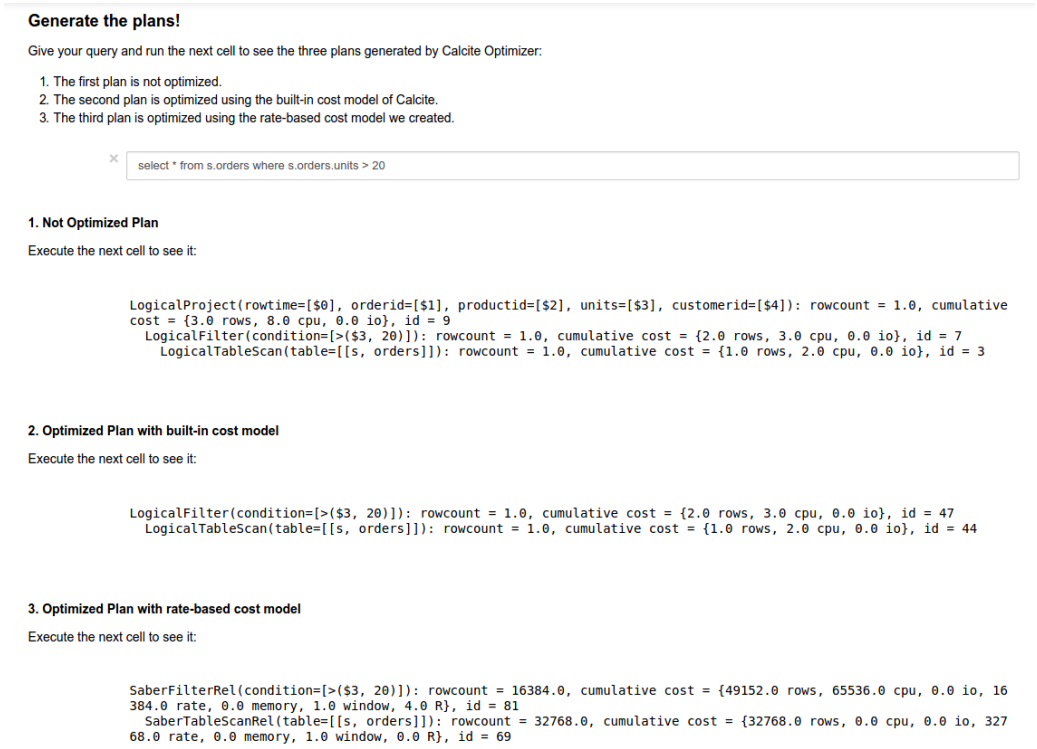


Fig. 5.6.4.b, Give your query and get the corresponding plans.

Different Steaming queries can be executed on the loaded datasets. Regardless the dataset, the users can specify their own query using a text-area field, as in Fig.5.6.4.b. For each of the queries, the three plans will be shown: a plan without optimization, a plan with optimization using the built-in cost model and a plan with optimization using the rate-based cost model. Upon proceeding with the execution, the user can choose which plan to execute; real-time progress will be available through a plot, which presents the throughput metrics of the submitted query.

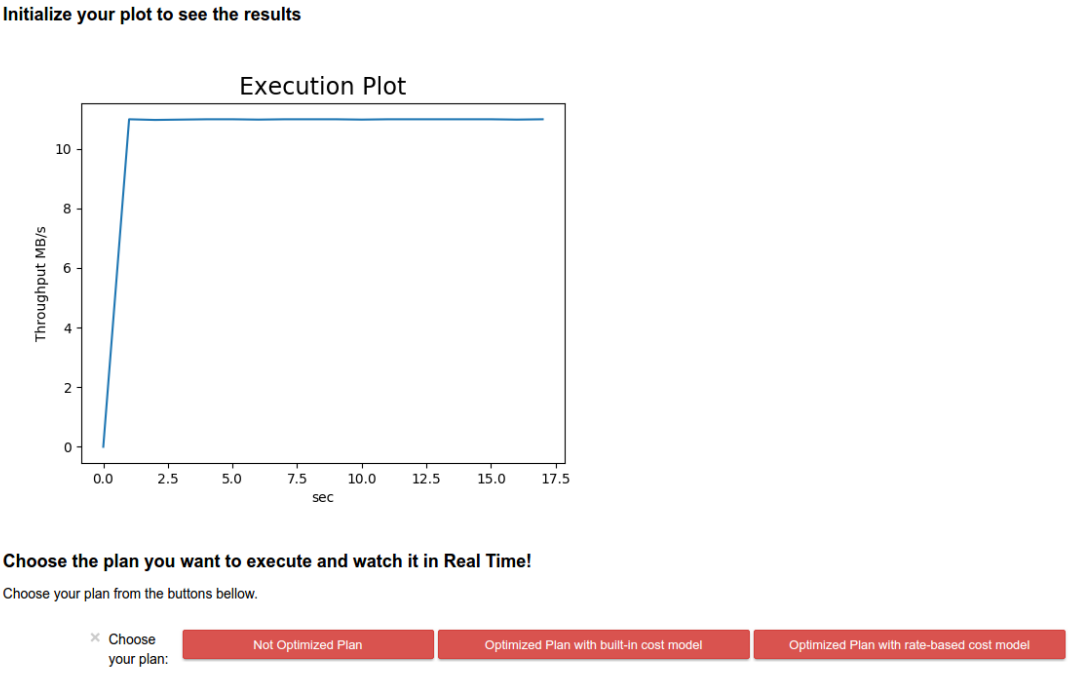


Fig. 5.6.4.c, Real Time Plot of Throughput.

6

Evaluation

In this chapter we run experiments with 10 representative queries to evaluate the efficiency of RBStream. We discuss our experimental setup in Section 6.1. We present our queries and their results in Section 6.2. We conclude this chapter with a discussion of our results in Section 6.3.

6.1 Experimental Setup

System configuration: The experimental setup consists of an OpenStack VM with 8x2GHz Intel Xeon E312xx CPU cores and 16GB of RAM.

Compared Plans: We compare the execution performance of streaming queries against three different plans: a plan without optimization, a plan optimized using Calcite's built in cost model and a plan optimized using our rate-based cost model.

Evaluation Metrics: The metrics we used for our evaluation are:

- Throughput: we monitored the throughput with a built-in functionality of SABER.
- Latency: we monitored the latency of the system by using the ResultCollector class of SABER.
- CPU Utilization: we monitored the cpu utilization with nmon [63].

Data Sets Used: We utilize one dataset in our evaluation: a Purchase-Orders Transaction Benchmark, with controlled distribution over synthetic data. The schema of our streaming data sources is:

```
s.customers Schema : Stream (rowtime: long, customerid: int, phone: long)
s.orders Schema : Stream (rowtime: long, orderid: int, productid: int, units: int, customerid: int)
s.orders_delivery Schema : Stream (rowtime: long, orderid: int, date_reported: long, delivery_status_code: int)
```

```
s.payments Schema : Stream (rowtime: long, customerid: int,
payment_date: int, amount: float)
s.products Schema : Stream (rowtime: long, productid: int, description:
int, price: float)
```

In order to provide a direct comparison among the different execution plans, we test the performance of our chosen queries for all the possible plans. We outperform the non-optimized plan in most of the queries, when running on the small server configuration, due to smaller memory usage and the creation of less query tasks for SABER. The difference in performance is mainly attributed to the decrease of the intermediate results created, which is crucial for the streaming in-memory computations, affecting both memory utilization and cpu cycles. The optimized order of execution of certain operators not only increases the output rate, but also makes the computations feasible in many cases, as it lowers the memory requirements. As a result, for more data-intensive queries, our system gives a feasible plan that makes the query execution possible.

6.2 Experiments

We will test the execution of certain queries, which are representative of the optimization use cases that RBStream can handle. The queries used, although simple, cover the basic SQL functionalities and can be used to form more complex use cases. With these experiments, we will compare the "naive" execution of queries against their optimized version. For most of these queries, both built-in and rate-based cost models give the same results, because of the nature of the optimization rules applied to the initial plan. Rules that push down operators (such as filters, projections or aggregates), merge or remove consecutive operators and simplify sql expressions are heuristic and can be applied regardless the cost model used by the optimizer. Thus, in such cases we will present only the rate-based model results. However, other rules, such as the rules that arrange the join or the filter ordering, can be greatly benefited by the rate-based cost model.

6.2.1) With the next query we will test FilterProjectTransposeRule, FilterMergeRule and ProjectMergeRule.

Q1:
```
select * from (
select *
from s.orders
where s.orders.units > 10) as s1
where s1.productid = 15 or s1.productid = 17;
```

a) The generated plan using the rate-based cost model is:

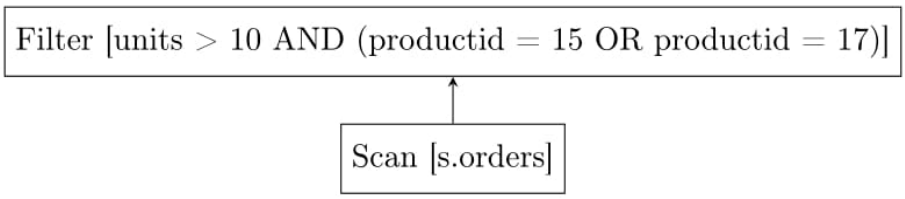


Fig. 6.2.1.a, rate-based plan of Query 1.

b) The generated plan with no optimization is:

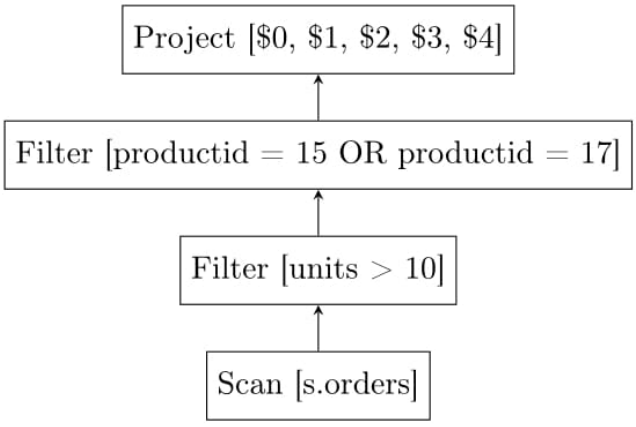


Fig. 6.2.1.b, not optimized plan of Query 1.

The configuration used for the following measurements of this test case was: 2 Threads, 64 MB Circular Buffer Size και 128 MB Unbounded Buffer Size.

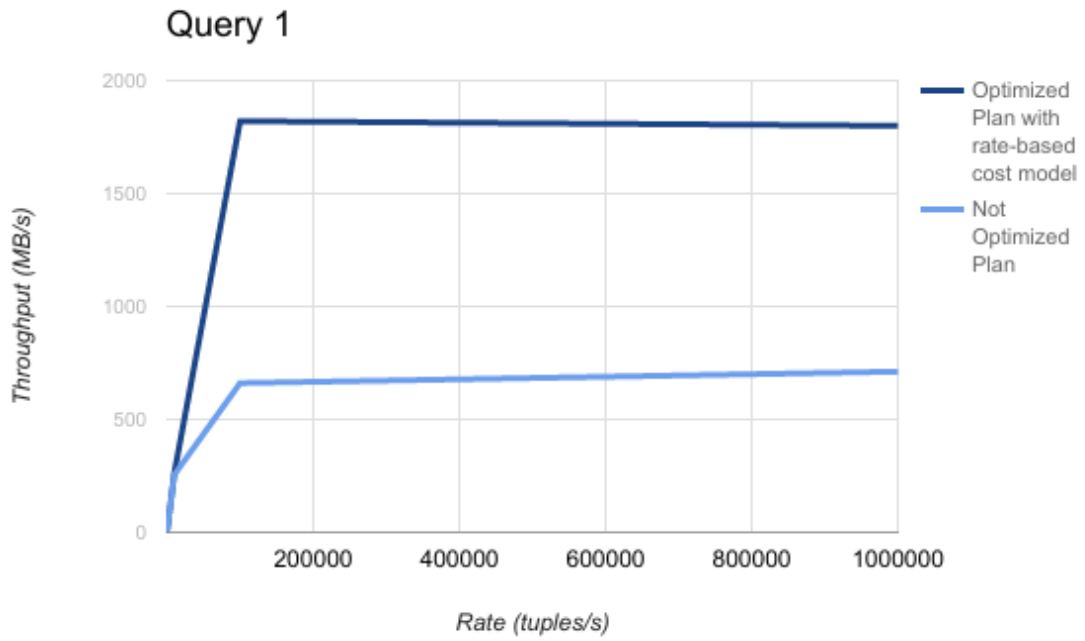


Fig. 6.2.1.c, Throughput Results of Query 1.

In the graph above, the early filtering of tuples that would be otherwise dropped by the operators that follow results in greater throughput results (nearly 2,7 times more MB/s). This occurs because less intermediate results are transferred and more precisely, in this case, fewer operators are initialized for the computation of the final output. It can also be observed that with our chosen configuration, as we increase the input rate, the throughput remains still and starts to decrease steadily by a minor factor from a certain threshold and on.

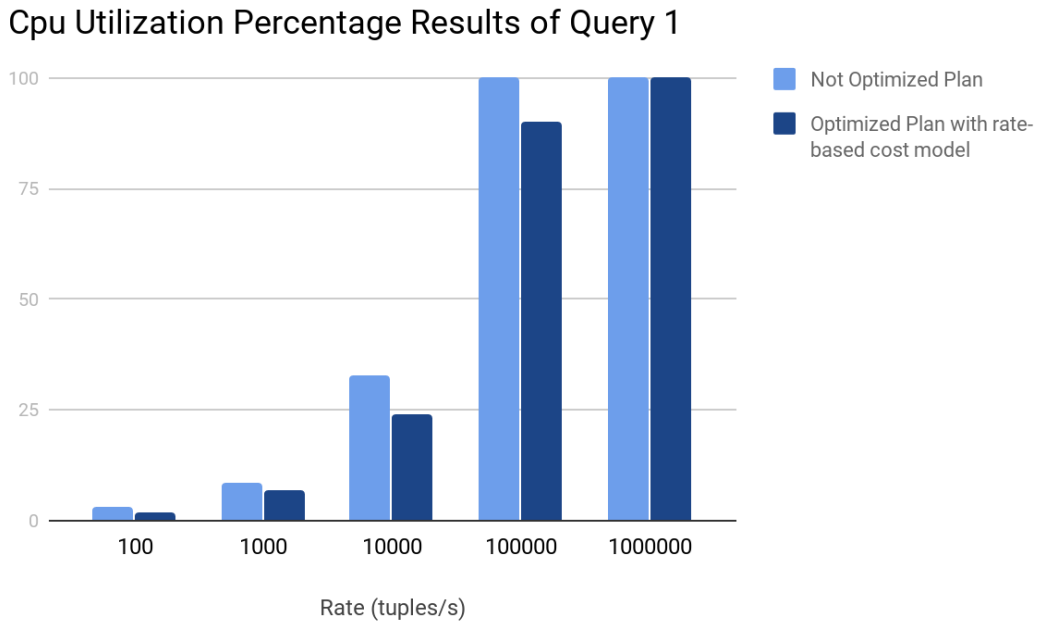


Fig. 6.2.1.d, Cpu Utilization Results of Query 1.

This query initializes stateless operators and as a result, the Cpu utilization is slightly smaller in the rate-based optimization plan in comparison to the non-optimized version.

6.2.2) This query illustrates the use of FilterJoinRule, which pushes filter through a join to its children.

Q2:
```
select *
from  s.products join s.orders
on s.orders.productid = s.products.productid
where units>10 and description < 16 ;
```

a) The plan we get with the rate-based optimization is:

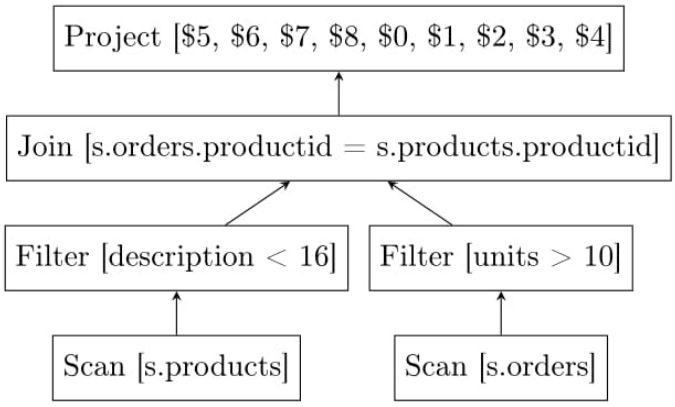


Fig. 6.2.2.a, rate-based plan of Query 2.

b) The plan we get with no optimization is:

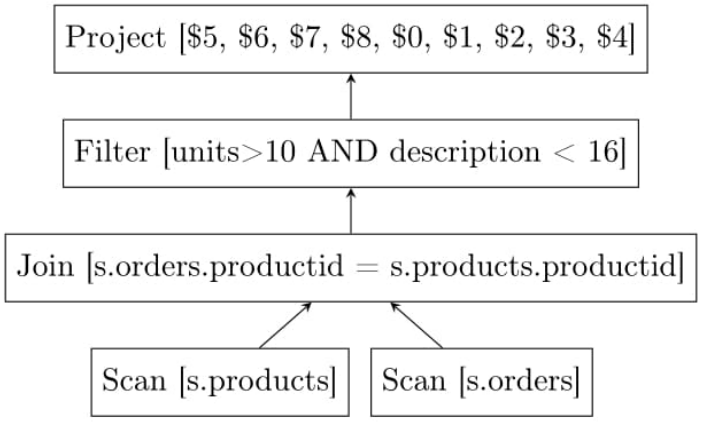


Fig. 6.2.2.b, not optimized plan of Query 2.

Plan 1 is optimized with rate-based cost model. Plan 2 is not optimized. In the cells with (-) the system run out of memory bounds.

	Threads	Circular Buffer (MB)	Unbounded Buffer (MB)	Data Source 1 (tuples/s)	Data Source 2 (tuples/s)	Selectivity	Throughput Plan1 (MB/s)	Latency Plan 1 (s)	CPU utilization Plan 1	Throughput Plan 2 (MB/s)	Latency Plan 2 (s)	CPU utilization Plan 2
Q2,1	2	64	128	300	200	1/24	13,75	0,12	15%	13,16	0,032	45%

Q2,2	2	64	128	400	400	1/24	22,17	0,019	35%	20,73	0,019	70%
Q2,3	2	64	128	500	400	1/24	24,10			-	-	-
Q2,4	2	64	128	800	600	1/24	38,51			-	-	-
Q2,5	2	64	128	800	800	1/24	38,22			-	-	-
Q2,6	1	64	128	400	400	1/24	21,7	0,02	70%	-	-	-

Table 6.2.2, Execution Metrics of Query 2.

6.2.3) With this query, we show the impact of join ordering, even with two joins, in execution. The same query will be computed with optimization, using built-in and rate-based cost models, and without optimization.

Q3:
```
select s.orders.rowtime, s.products.rowtime,
s.customers.rowtime,
s.orders.productid,s.orders.customerid
from  s.products, s.orders, s.customers
where s.orders.productid = s.products.productid and
s.customers.customerid=s.orders.customerid;
```

a) The plan using the rate-based cost model optimization is:

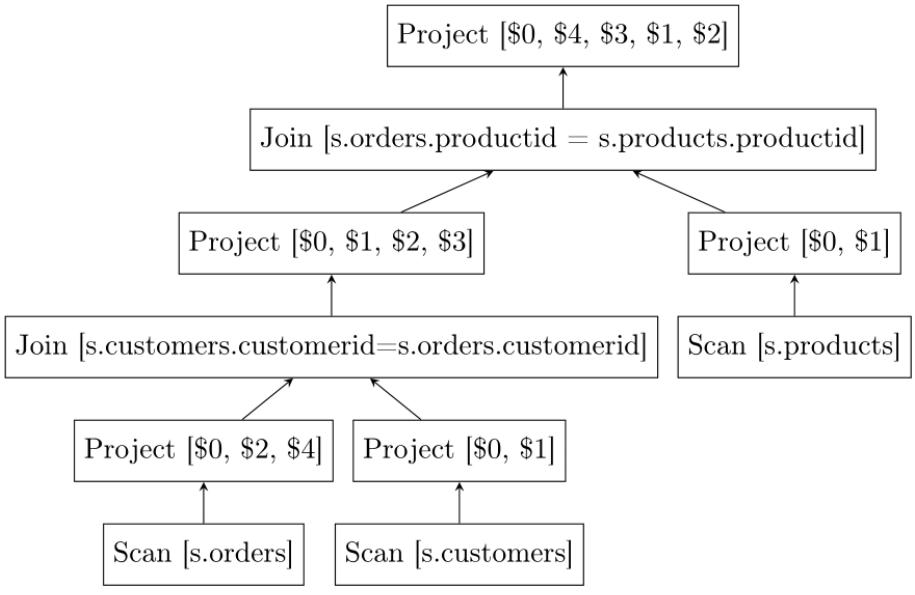


Fig. 6.2.3.a, rate-based optimized plan of Query 3.

b) Then plan without optimization is (we couldn't execute this plan because of the memory it required):

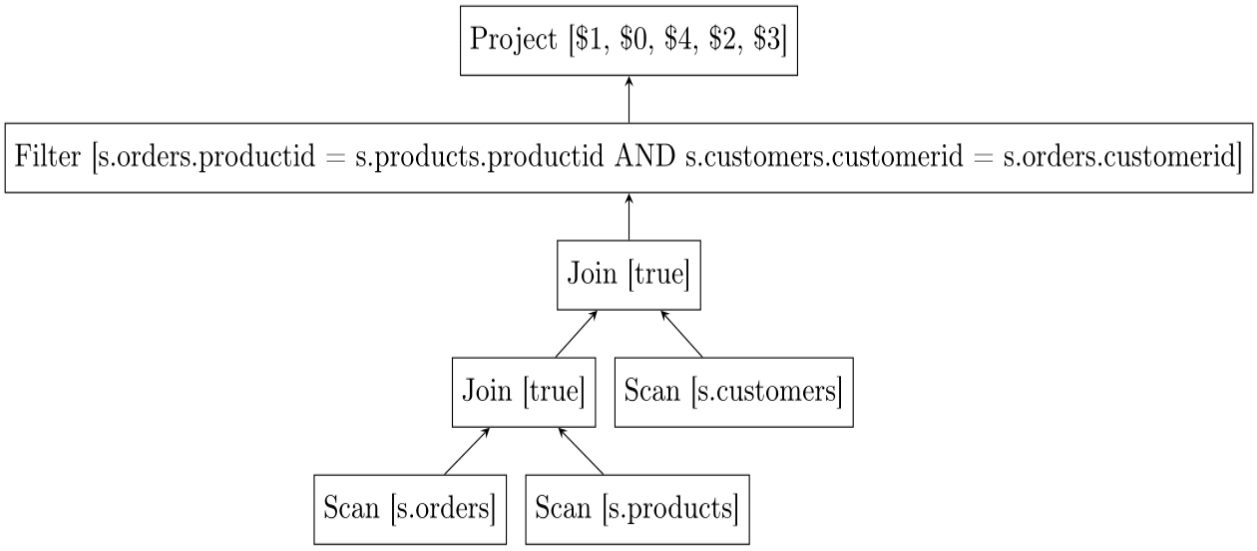


Fig. 6.2.3.b, not optimized plan of Query 3.

c) The plan with the built-in cost model optimization is:

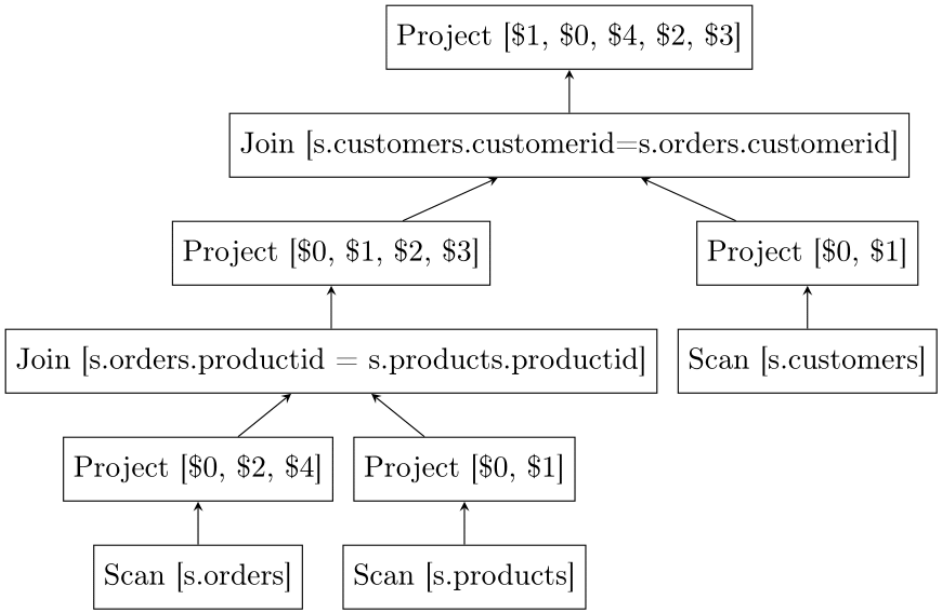


Fig. 6.2.3.c, built-in optimized plan of Query 3.

Plan 1 is optimized with rate-based cost model. Plan 2 is optimized with built-in cost model. In the cells with (-) the system run out of memory bounds.

	Threads	Circular Buffer (MB)	Unbounded Buffer (MB)	Data Source 1 (tuples/s)	Data Source 2 (tuples/s)	Data Source 3 (tuples/s)	Selectivity	Throughput Plan 1 (MB/s)	Latency Plan 1 (s)	CPU utilization Plan 1	Throughput Plan 2 (MB/s)	Latency Plan 2 (s)	CPU utilization Plan 2
Q3,1	2	128	128	100	400	400	1/24	27,16	4,76	9%	23,50	5,67	75%
Q3,2	2	128	128	200	400	400	1/24	26,84	1,6	20%	27,22	2,57	85%
Q3,3	2	64	128	200	600	600	1/24	38,9	3,5	20%	-	-	-
Q3,4	2	64	128	200	800	600	1/24	43	3,7	20%	-	-	-
Q3,5	2	64	256	200	800	800	1/24	49	3,2	>80 %	-	-	-
Q3,6	2	64	200	350	700	600	1/24	43	1,6	>80 %	-	-	-
Q3,7	2	64	128	200	600	300	1/24	30	2,03	40%	30	3,33	60%
Q3,8	2	64	128	200	600	300	1/14	31	0,68	10%	31	0,90	35%
Q3,9	2	64	128	200	600	300	1/10	29,92	0,44	50%	30	0,62	70%
Q3,10	1	128	128	100	400	400	1/24	24,9	5,15	15%	-	-	-
Q3,11	1	64	128	200	600	300	1/24	29,98	3,88	25%	-	-	-

Table 6.2.3, Execution metrics of Query 3.

6.2.4) This query shows the effect of ProjectJoinTransposeRule.

Q4:
```
select s.orders.rowtime, s.products.rowtime, s.orders.productid
from  s.products, s.orders
where s.orders.productid = s.products.productid;
```

a) The plan generated by the rate-based cost model optimization is:

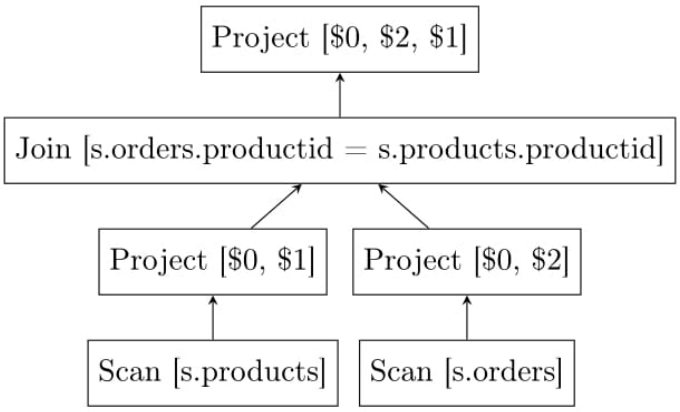


Fig. 6.2.4.a, rate-based plan of Query 4.

b) The plan generated without optimization is:

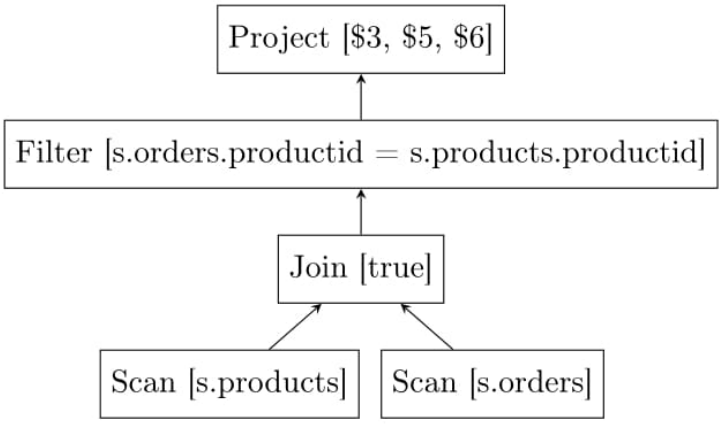


Fig. 6.2.4.b, not optimized plan of Query 4 (not feasible).

The execution of this plan crashes (the size of the buffers is not enough), because it tries to join the two sources with condition [true] and as a result it creates a huge amount of intermediate data that will be dropped:

```
java.nio.BufferOverflowException
      at java.nio.HeapByteBuffer.put(HeapByteBuffer.java:183)
      at
uk.ac.imperial.lsds.saber.buffers.UnboundedQueryBuffer.put(UnboundedQue
ryBuffer.java:156)
      at
uk.ac.imperial.lsds.saber.buffers.CircularQueryBuffer.appendBytesTo(Cir
cularQueryBuffer.java:296)
      at
uk.ac.imperial.lsds.saber.cql.operators.cpu.ThetaJoin.processData(Theta
Join.java:154)
```

```
        at
uk.ac.imperial.lsds.saber.QueryOperator.process(QueryOperator.java:75)
        at uk.ac.imperial.lsds.saber.tasks.Task.run(Task.java:63)
...
```

In order to execute this query we have to rewrite it like this:

```
select s.orders.rowtime, s.products.rowtime, s.orders.productid
from  s.products join s.orders
on s.orders.productid = s.products.productid;
```

and we get the following plan:

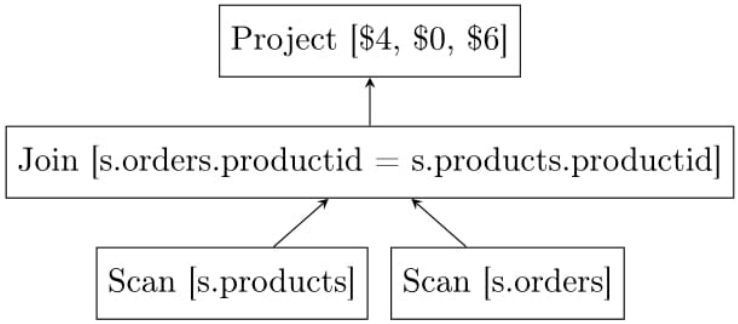


Fig. 6.2.4.c, not optimized plan of Query 4.

Plan 1 is optimized with rate-based cost model. Plan 2 is not optimized. In the cells with (-) the system run out of memory bounds.

	Threads	Circular Buffer (MB)	Unbounded Buffer (MB)	Data Source 1 (tuples/s)	Data Source 2 (tuples/s)	Selectivity	Throughput Plan 1 (MB/s)	Latency Plan 1 (s)	CPU utilization Plan 1	Throughput Plan 2 (MB/s)	Latency Plan 2 (s)	CPU utilization Plan 2
Q4,1	1	64	128	100	100	1/24	5,52	0,13	13%	5,49	0,05	20%
Q4,2	1	64	128	400	250	1/24	17,01	0,03	50%	17,31	0,006	65%
Q4,3	1	64	128	400	280	1/24	17,9	0,02	60%	-	-	-
Q4,4	1	64	128	500	400	1/24	-	-	-	-	-	-
Q4,5	1	64	128	800	800	1/24	-	-	-	-	-	-
Q4,6	2	64	128	400	280	1/24				18,11	0,005	65%

Table 6.2.4, Execution metrics of Query 4.

6.2.5) With this query we present the use of AggregateJoinTransposeRule.

Q5:
```
select count(*)
from s.orders
join s.products
on s.orders.productid = s.products.productid;
```

a) When we use the rate-based cost model with optimization, we get the following plan:

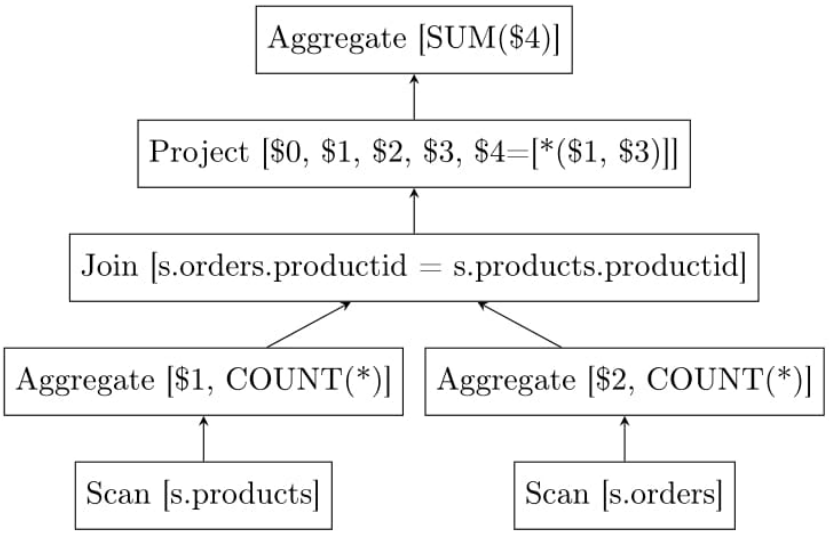


Fig. 6.2.5.a, rate-based optimized plan of Query 5.

b) With no optimization we get:

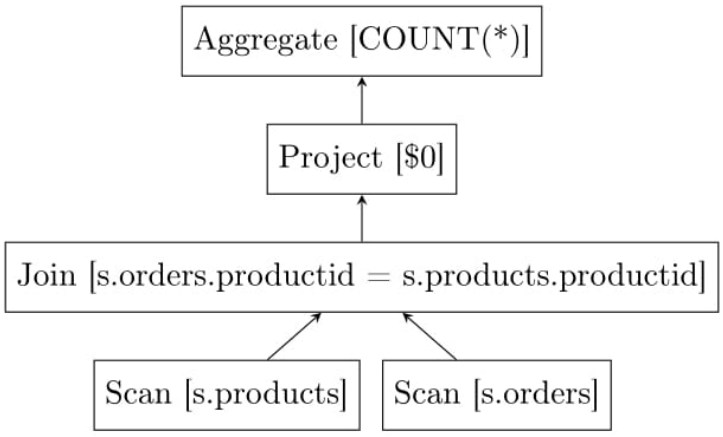


Fig. 6.2.5.b, not optimized plan of Query 5.

Plan 1 is optimized with rate-based cost model. Plan 2 is not optimized. In the cells with (-) the system run out of memory bounds.

	Threads	Circular Buffer (MB)	Unbounded Buffer (MB)	Data Source 1 (tuples/s)	Data Source 2 (tuples/s)	Selectivity	Throughput Plan1 (MB/s)	Latency Plan 1 (s)	CPU utilization Plan 1	Throughput Plan2 (MB/s)	Latency Plan 2 (s)	CPU utilization Plan 2
Q5,1	1	32	128	100	100	1/16	5,28	0,1	10%	5,25	0,26	30%
Q5,2	1	32	256	200	200	1/16	9,96		30%	10,52		80%
Q5,3	1	32	380	400	400	1/16	21,93			-	-	-
Q5,4	1	32	380	500	400	1/16	24,67			-	-	-
Q5,5	2	32	380	400	400	1/16	21,93		16%	-	-	-

Table 6.2.5, Execution metrics of Query 5.

6.2.6) This query uses AggregateJoinTransposeRule and AggregateProjectMerge.

Q6:
```
select o.productid, min(units)
from s.orders as o
join s.products as p
on p.productid=o.productid
group by p.productid,o.productid;
```

a) The plan generated from the rate-based optimization is:

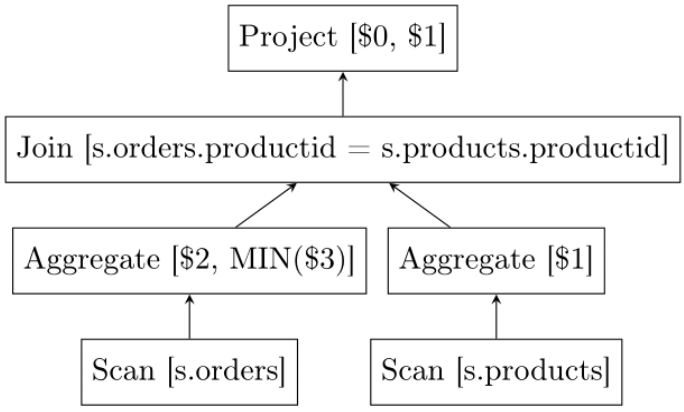


Fig. 6.2.6.a, rate-based optimized plan of Query 6.

b) If we don't use optimization we get:

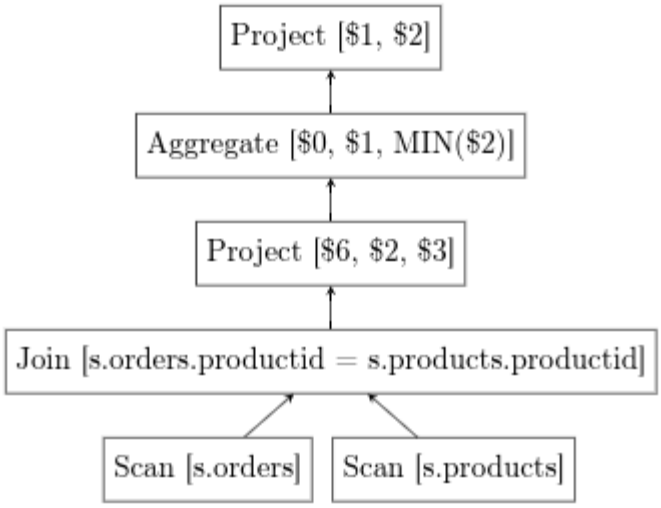


Fig. 6.2.6.b, not optimized plan of Query 6.

Plan 1 is optimized with rate-based cost model. Plan 2 is not optimized. In the cells with (-) the system run out of memory bounds.

	Threads	Circular Buffer (MB)	Unbounded Buffer	Data Source 1 (tuples/s)	Data Source 2 (tuples/s)	Selectivity	Throughput Plan1 (MB/s)	Latency Plan 1 (s)	CPU utilization Plan 1	Throughput Plan2 (MB/s)	Latency Plan 2 (s)	CPU utilization Plan 2

			(MB)									
Q6,1	1	32	128	100	100	1/16	5,38	0,1	13%	5	0,26	35%
Q6,2	1	32	256	200	200	1/16	10,97		30%	10,29		85%
Q6,3	1	32	360	400	400	1/16	21,59		35%	-	-	-
Q6,4	1	32	360	500	400	1/16	22,55			-	-	-
Q6,5	2	32	360	400	400	1/16	21,59		16%	-	-	-

Table 6.2.6, Execution metrics of Query 6.

6.2.7) In this query we evaluate the application of

AggregateProjectPullUpConstantsRule.

Q7:
```
select rowtime,productid, count(*)
from s.orders
where units > 10
group by rowtime,productid;
```

a) The plan we get with the rate-based optimization:

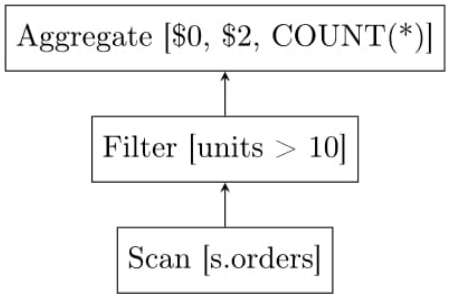


Fig. 6.2.7.a, rate-based optimized plan of Query 7.

b) The plan we get without optimization:

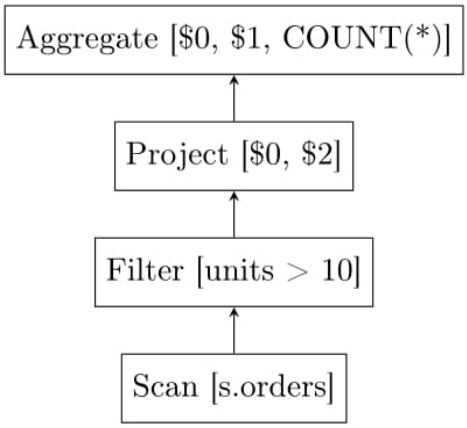


Fig. 6.2.7.b, not optimized plan of Query 7.

The configuration used for the following measurements of this test case was: 2 Threads, 64 MB Circular Buffer Size και 128 MB Unbounded Buffer Size.

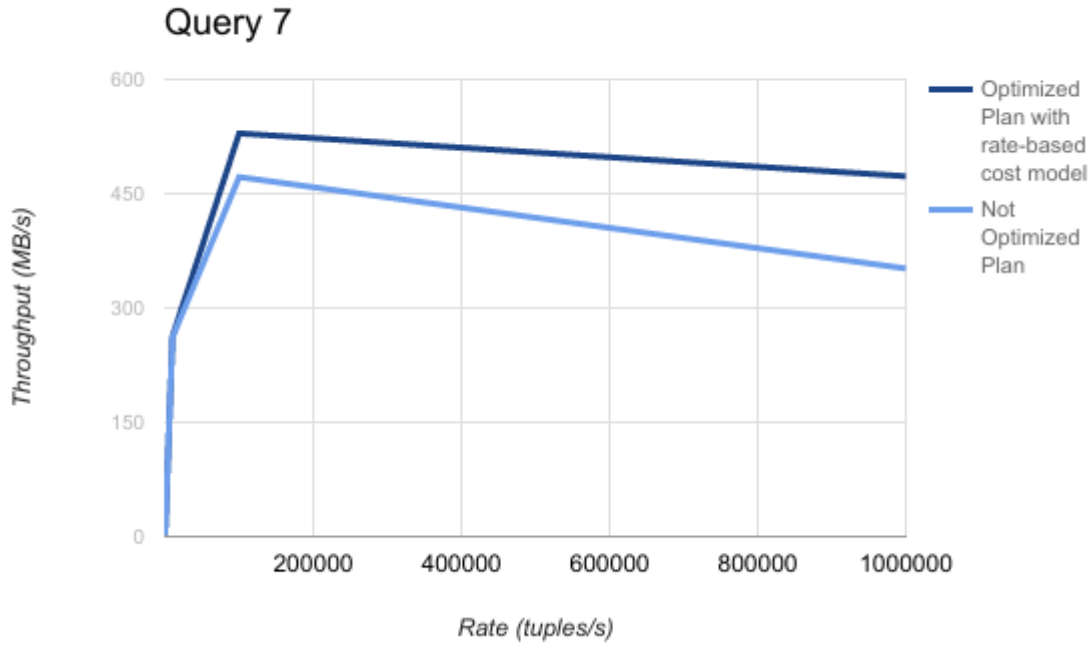


Fig. 6.2.7.c, Throughput Results of Query 7.

As observed in the line graph above, by pulling up the constants in the aggregate operator, the rate-based plan reveals higher throughput results, because we limit the operators required for computing the output. However, as aggregate operator demands bigger buffers

and hash table in order to escalate accordingly when the input rate increases, we notice a decrease in the throughput in both plans. This occurs because the system's configuration fails to manage with these rates.

6.2.8) With this query we present FilterAggregateTransposeRule, that pushes filter through aggregate.

Q8:
```
select * from (
select rowtime,productid, count(*)
from s.orders
group by rowtime,productid
) as o
where o.productid > 10;
```

a) We use the rate-based cost model with optimization:

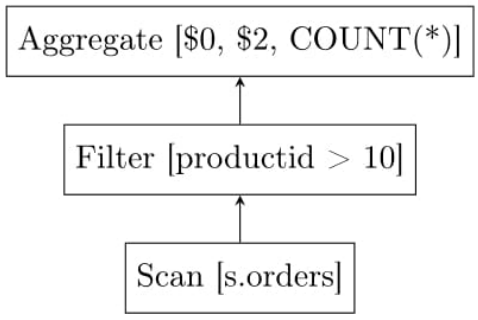


Fig. 6.2.8.a, rate-based plan of Query 8.

b) We didn't use optimization:

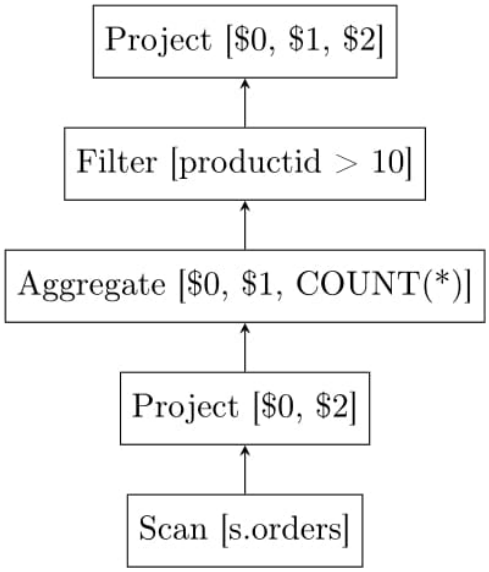


Fig. 6.2.8.b, not optimized plan of Query 8.

The configuration used for the following measurements of this test case was: 2 Threads, 64 MB Circular Buffer Size και 128 MB Unbounded Buffer Size.

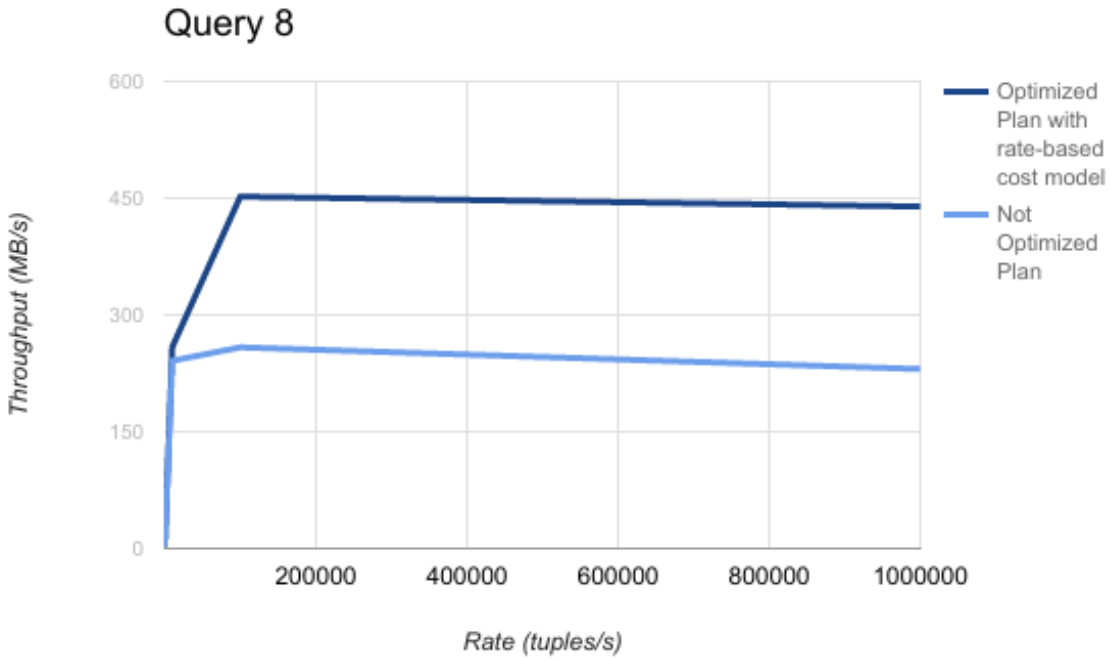


Fig. 6.2.8.c, Throughput Results of Query 8.

This case is similar to query 1. The rate-based plan results in nearly 1,9 more MB/s throughput in comparison to the naive version.

6.2.9) This query shows the effect of AggregateConstantKeyRule.

Q9:
```
select count(*)
from s.orders
where productid=10
group by rowtime,productid;
```

a) The plan generated with the rate-based cost model optimization is:

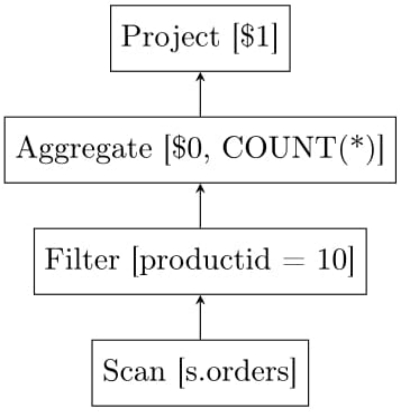


Fig. 6.2.9.a, rate-based plan of Query 9.

b) The plan generated wihtout optimization is:

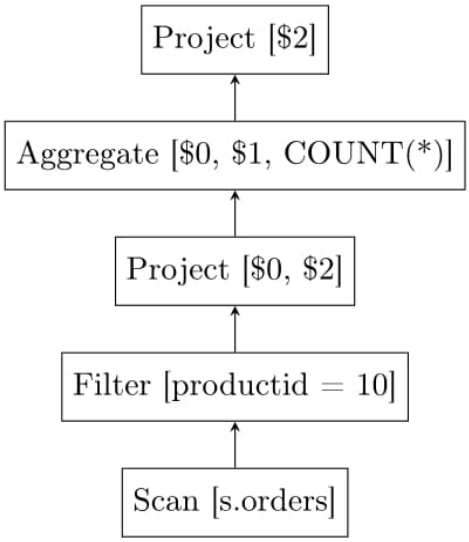


Fig. 6.2.9.b, not optimized plan of Query 9.

The configuration used for the following measurements of this test case was: 2 Threads, 64 MB Circular Buffer Size και 128 MB Unbounded Buffer Size.

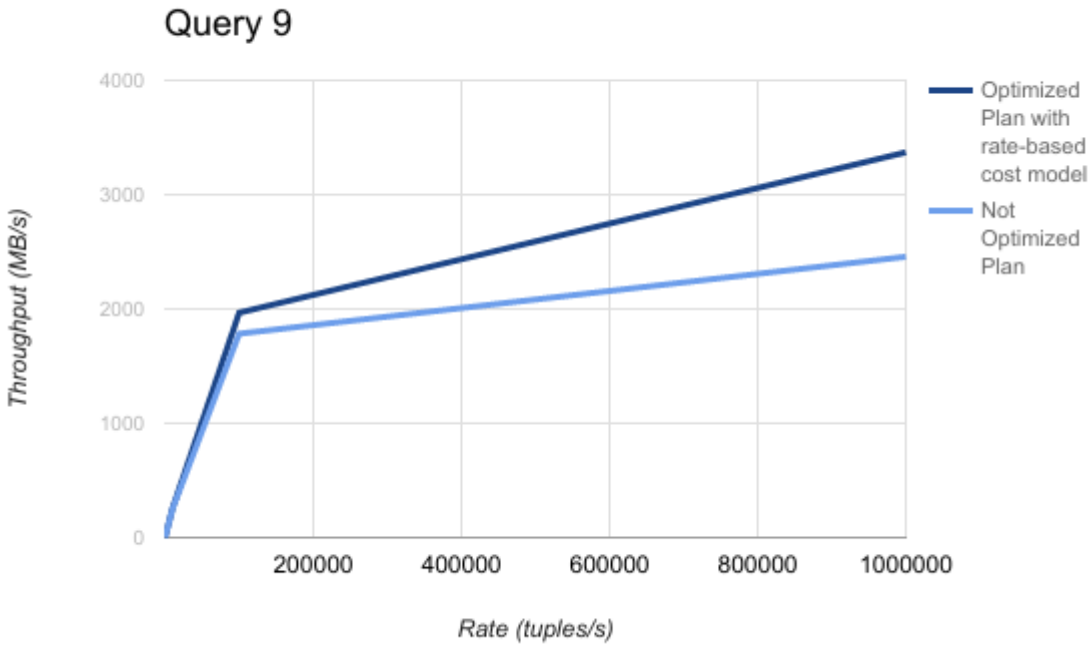


Fig. 6.2.9.c, Throughput Results of Query 9.

This rule finds the constants that can be omitted from the grouping procedure of the aggregate operator, resulting in greater throughput results, as it is less expensive to compute them.

6.2.10) With this query we will test the effect of FilterPushThroughFilterRule, we created, against FilterMergeRule.

Q10:
```
select * from
(select * from (
select *
from s.orders
where productid = 15 or productid = 17) as s1
where s1.units > 10) as s2
where s2.customerid  = 15;
```

a) The generated plan using the rate-based cost model and FilterPushThroughFilterRule is:

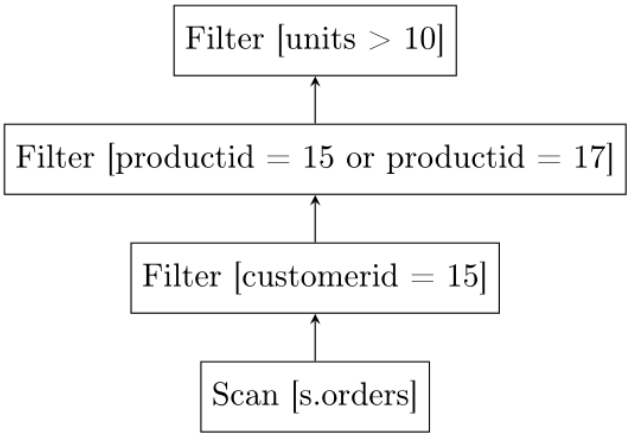


Fig. 6.2.10.a, optimized plan using FilterPushThroughFilterRule of Query 10.

b) The generated plan using the rate-based cost model and FilterMergeRule is:

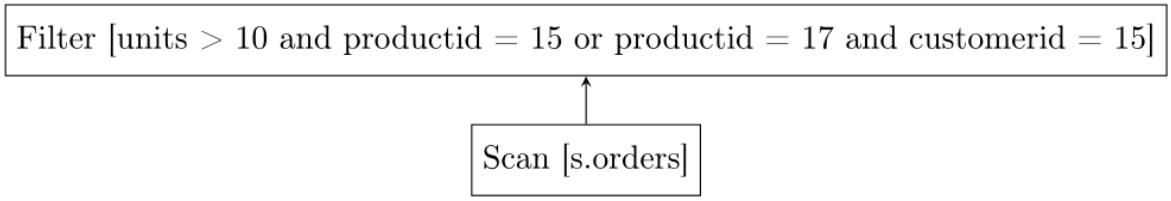


Fig. 6.2.10.b, optimized plan with FilterMergeRule of Query 10.

The configuration used for the following measurements of this test case was: 2 Threads, 64 MB Circular Buffer Size και 128 MB Unbounded Buffer Size.

Fig. 6.2.10.c, Throughput Results of Query 10.

From the line graph above, it can be noticed that regulating the granularity of filter operator can result in great throughput improvement in certain cases. The two examined rules follow opposite results, with FilterPushThroughFilterRule increasing significantly while FilterMergeRule decreases steadily.

6.3 Analysis of the Experimental Results

From our experimental results, we observe that we have two categories of queries based on the results we have collected: the queries with and without a Join operator. In the first category (queries 1, 7, 8, 9 and 10), the optimized plans outperform the non optimized ones, regarding the throughput of the input data sources. It can be observed, that when the input rate is small enough to be handled by our system, the throughput for both cases is almost the same. However, as the rate increases, the system achieves higher throughput with the optimized plan, because of the less intermediate results created and the smaller cpu overhead (less tasks and computations required). A great percentage of the intermediate results generated by the non optimized plans are usually dropped because they are not required for the computation of the final output. For example, when we push down a filter through another operator, we ensure that the data that pass through the next processing stages will participate in the following computations and we avoid the use of unnecessary resources. This process not only affects the memory utilization requirements but also causes a dramatic decrease in query task spawning for SABER, resulting in smaller scheduling and computational overhead of the system. In the not optimized use cases, the system receives more data than it can consume, resulting in smaller throughput rates, higher cpu and higher memory utilization, as it struggles to deal with incoming rates. In addition, there are some cases with the non optimized plans, where the throughput rates are not stable because of the inability of the system to handle their computation, and cause system failure. The next figure presents the result of queries Q1, Q7, Q8 and Q9 (2,75, 1,34, 1,9 and 1,37 times more MB/s respectively):

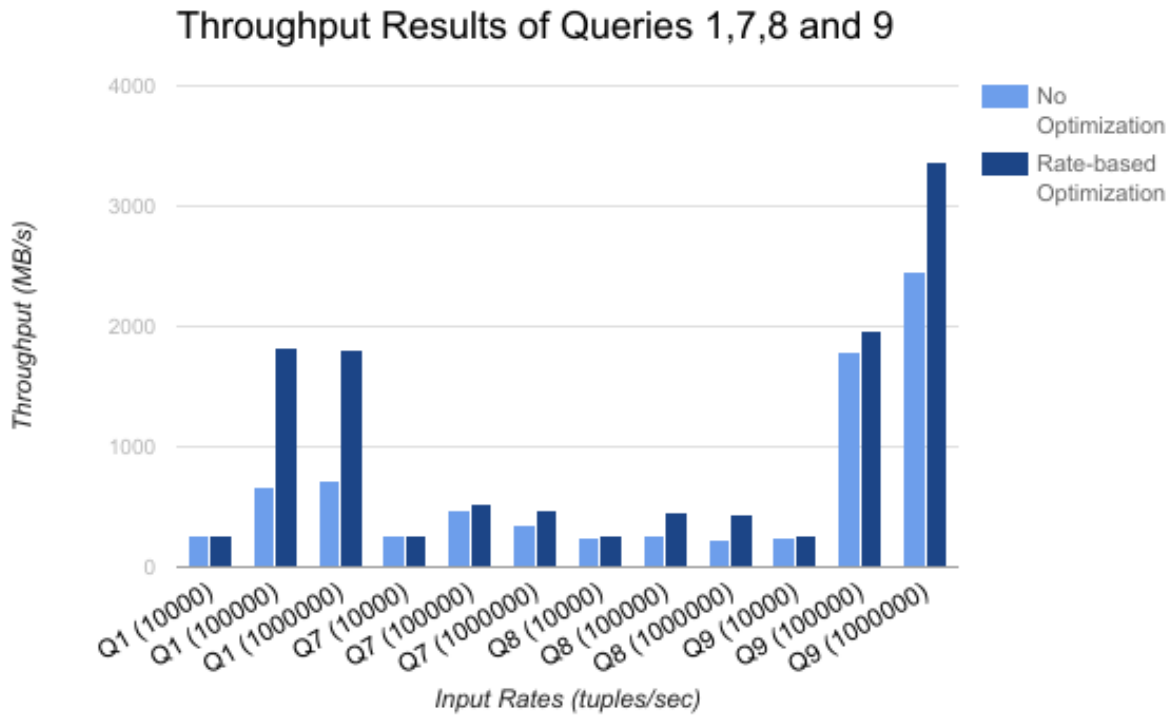


Fig. 6.3.a, Throughput Results of Queries 1, 7, 8 and 9.

As far as the 10th query is concerned, it helped us test our optimization procedure against rules that require dynamic optimization, using the rate-based cost model. We noticed that the proper reordering of consecutive Filter operators can result in greater throughput in some cases, in comparison with the merge of these filters. The following figure presents our results (even 2,22 times more MB/s):

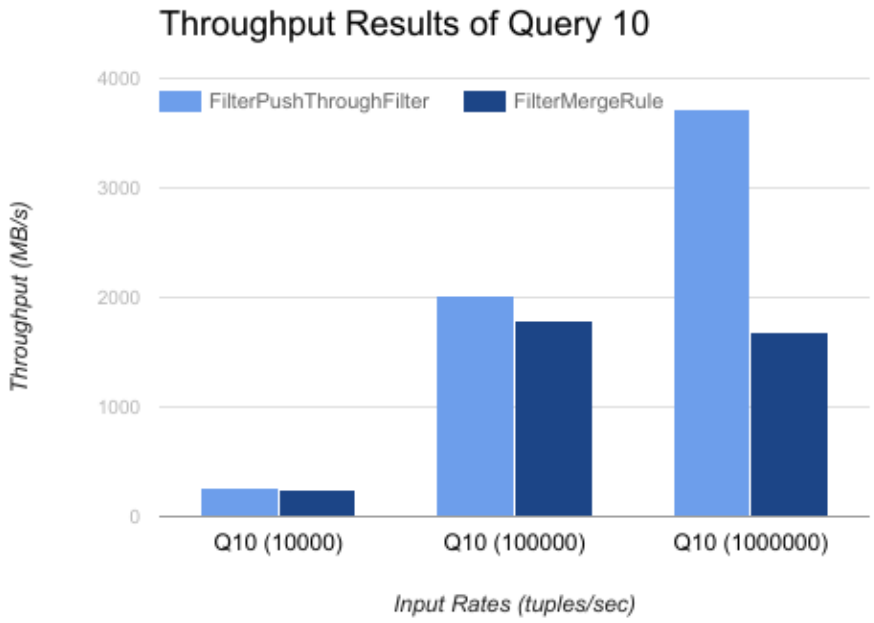


Fig. 6.3.b, Throughput Results of Query 10.

As observed in the graph, when having a filter operator with a complex condition, it is worth to split it in more filters with simpler conditions and take advantage of smaller granularity. By lowering the computation overhead of the initial complex condition, we notice that in this case, the communication cost of having smaller tasks is beneficial and results in greater throughput.

In the second category of queries, that contained Join operators, we didn't observe an increase in the throughput of the different execution plans. The Join operator acts as a bottleneck in the query execution, because of its implementation, and thus we couldn't find a use case in which our plans had different throughput in comparison with the naive implementation. We experimented with the selectivity, the number of threads used and the size of the buffers utilized for the computation of our queries, without noticing any striking difference between the executed plans. As a result, we assume that this is happening because of uneven allocation of the processing load between the worker threads, which is likely to occur in large systems and dynamically varying workloads. In order to overcome this obstacle and make our comparisons for system evaluation, we had to monitor other metrics, such as latency and cpu utilization. The following figures and analysis concern queries Q2, Q3, Q4, Q5 and Q6 (the numbers below the bars in the following charts

correspond to the numbers we gave in our measurements in the tables of the previous section).

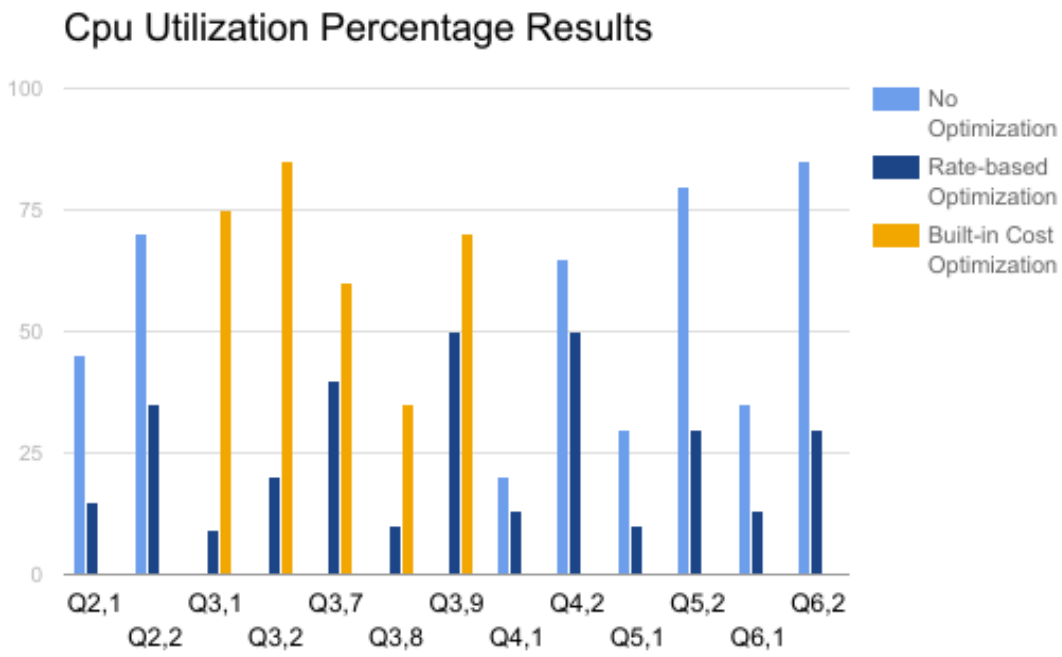


Fig. 6.3.c, CPU Utilizaton Percentage Results of Queries 2, 3, 4, 5 and 6.

In all the use cases of queries 2, 3, 4, 5 and 6, the cpu utilization of the rate-based optimized plan is significantly smaller than the corresponding of either non-optimized plan or optimized with the built-in cost model plan. We experimented with a different number of threads and sizes of buffers and observed that in the rate-based plan, the cpu is underutilized compared to the other plans when having low input rates. Thus, we increased the input rate, in all our use cases and noticed that the rate-based plan can handle even the double input rates without a problem, while both the non-optimized and built-in cost model plans fail to consume the incoming data, causing system failure. This result can be easily explained by the goal we achieve with the rate-based optimization, the decrease of the intermediate data and the query tasks produced from each stage of the computation process. In order to make our results more straightforward, we conducted some experiments for queries 2, 3 and 4 that show that we achieve the same throughput results by utilizing a single thread in the rate-based plan in contrast with the other two options, where at least two threads were required to handle the scheduling and computational tasks:

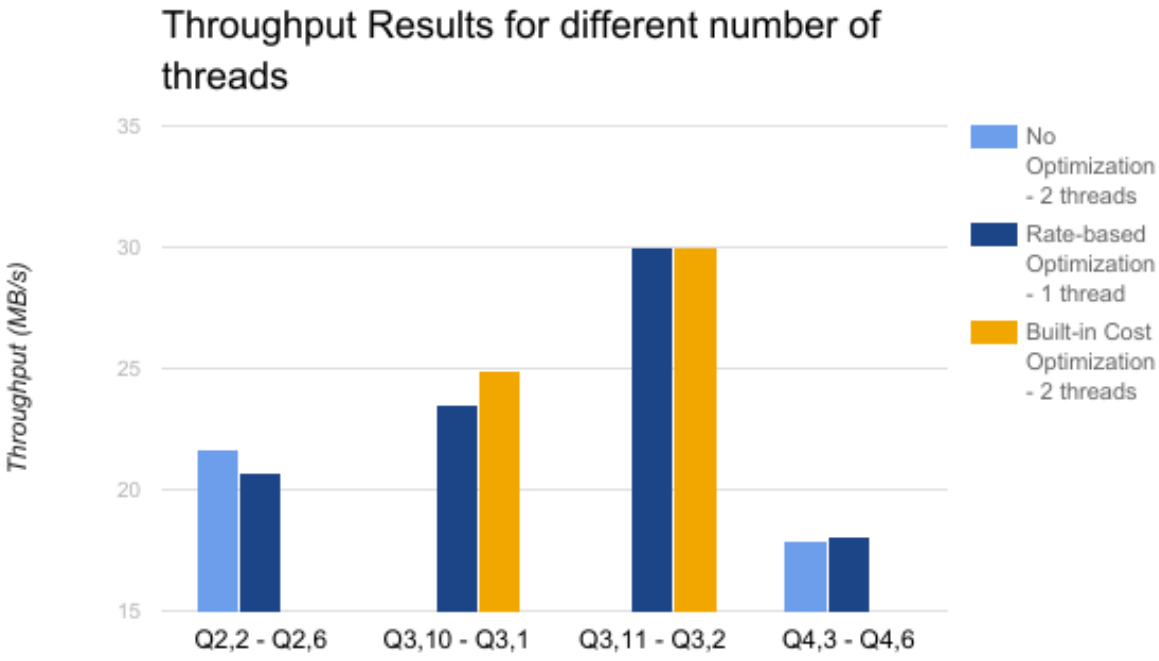


Fig. 6.3.d, Throughput results for different number of threads of Queries 2, 3, and 4.

Although in some of the above experiments the system could utilize less memory for creating the buffers required for the computations when executing the rate-based plans, the static allocation of the fixed sized buffers doesn't allow us to generalize our observation and assume that our cost model apart from using fewer threads, can run with smaller buffer sizes for Unbounded and Circular buffers in SABER.

In all the previous examples, both the built-in and rate-based cost models produce the same plans. Next, we will focus on the Q3, which shows the difference between the two cost models. The Join reordering is an important feature of our rate-based optimization procedure, that affects the execution of complex queries with many joins. Regarding the cpu utilization, the rate-based plan is underutilized with the same system parameters. The first improvement we notice is that as we increase the input rate, the built-in plans become infeasible and result in system's crash (a result which is also apparent from the figure above). The second improvement concerns the latency observed in both cases. In the rate-based plan, we choose to compute earlier joins that produce smaller output results, in association with the input rates, the windows used and the join condition. As a result, we

notice even less than 70% of the built-in cost model's latency in most cases (Figure below), which confirms our expectations.

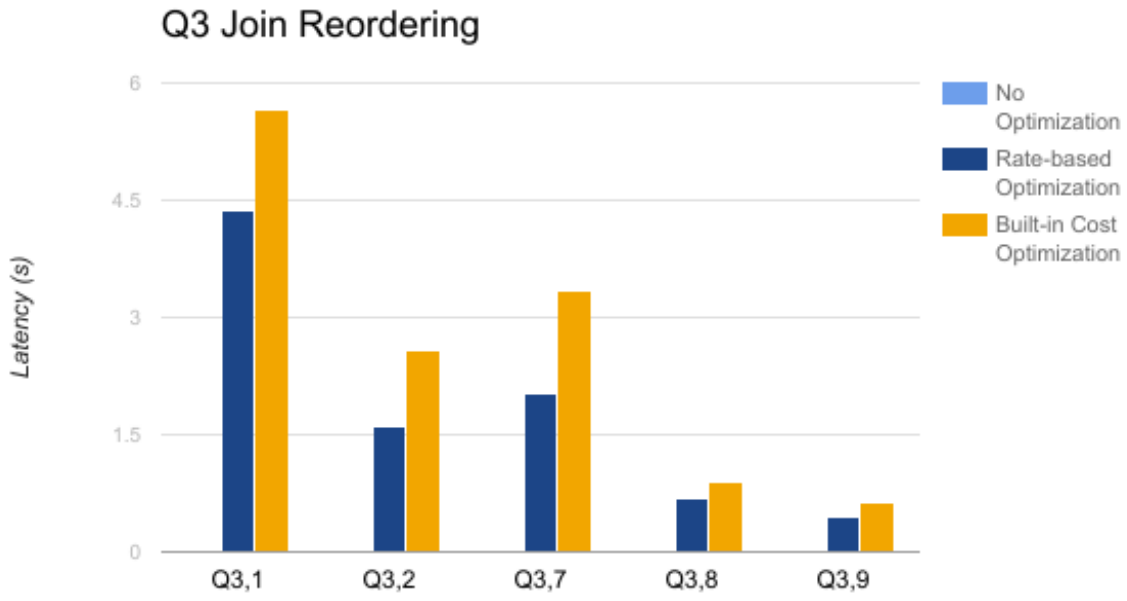


Fig. 6.3.e, Join Reordering Results of Query 3.

In summary, we conclude that the optimization we enforced results in smaller cpu and memory utilization and secures the feasibility of our plans. In the cases that we don't have join operators, we also observe great throughput improvements for our query plans.

Finally, we would like to notice that the rate-based model would have a greater impact in the case we had more than one physical implementations (algorithms) to choose for any of our operators. For example, if the system had at least two different types of Joins, the streaming semantics, which are embedded in the rate-based cost model, would create the optimal plan based on the characteristics of our streaming data sources.

7

Conclusions and future directions

7.1 Conclusions

During this work, we had the opportunity to explore streaming semantics and use our knowledge to introduce a SQL like declarative language to SABER. In addition, we examined the optimization techniques of regular queries, we expanded the built-in optimization logic of Calcite, in order to deal with the requirements of a streaming environment, and we tested their performance with our system on a small server configuration. Here, we will summarize the conclusions and contributions derived from this work:

1. Modern streaming systems require the use of SQL for many reasons. SQL language is declarative, which means that we only need to specify what we want and not how to compute it. Furthermore, it can be effectively optimized and evaluated in both distributed and centralized environments and is the best solution for expressing complex data analytics, which extract valuable information for most industrial and academic use cases. As a result, in RBStream we introduced SQL support to our chosen streaming engine based on a subset of window constructs definable in SQL-99 for OLAP functions. The ease of use of our system against the regular implementation of SABER was apparent and helped us test even further this streaming system against more complex and demanding queries. A simple SQL query with more than three operators is translated in dozens of lines of SABER execution code, with a small overhead of milliseconds only before the query execution for the computation of the optimal plan.
2. Based on SQL operators, we could apply both heuristic and cost-based optimization aiming towards the specific challenges that we face in streaming environments. Our key idea was the introduction of a Rate Based Logical Optimizer to SABER. We decided to use a combination of the heuristic and the dynamic optimizers of Calcite in a phased manner. We separated our optimization procedure in several phases in order to reduce the search space and the time required for the optimization, by taking advantage of the application of relevant subsets of rules. As a result, we

created a procedure more easily sustainable and achieved the best results with the rules we already had. It is very easy to add and remove rules without affecting the rest of the optimization chain. Finally, we examined and evaluated many built-in and custom query optimization techniques in our system, such as predicate push-down, projection pruning, operator merging, join order optimization and filter reordering.

3. Most streaming systems available for commercial use, haven't adopted yet a proper cost model capable of capturing the metrics of a streaming data source and make optimizations based on such metrics. We perceived this need and introduced a rate-based cost model to our system, to deal with streaming characteristics. Previous approaches to rate-based cost model lack the support of aggregate functions. We tried to expand them in order to support efficient estimation of resource consumption, fit our semantics and achieve better optimization for our streaming sources. The main goal of our cost model was the increase of output rate while minimizing the intermediate results and query tasks created from SABER. Our cost model also helps us estimate the impact of optimizations applied on a query plan.
4. Our optimization results in smaller cpu and memory utilization and secures the feasibility of the plans. A great percentage of the intermediate results generated by the non optimized plans are usually dropped because they are not required for the computation of the final output. With the optimization procedure that we apply, we reduce the intermediate results generated and decrease task spawning in SABER, resulting in the reduction of the computational and scheduling overhead of complex queries. In most of the test cases, the rate-based optimization generated a feasible plan, while both the non optimized and the built-in optimized plans generated results that run out of memory and caused a system crash. In addition, in the cases we didn't have join operators, we observed great throughput improvements (from 1,3 to 2,7 times more MB/s), as the output rate of our optimized plan was benefited greatly from our cost model. Finally, we observed from our experiments that the Join operator acts as a bottleneck in the query execution of SABER. We assume that this is happening because of not proper allocation of the processing load between the worker threads. As a result, we had to measure different metrics for these cases, like cpu utilization and latency. More specifically, in the join reordering query, we noticed less than 70% of the latency of the built-in cost model. Finally, we explored the opportunity of merging operators in the same query task or splitting them in order to adjust our task granularity accordingly and achieve greater performance.

5. We present the throughput and latency results to the user with an interactive web interface using Jupyter Notebook and python libraries for real time plots. The user can submit a query and choose from three different plans the one to be executed. These plans are: not optimized plan, optimized plan with built-in optimization and optimized plan with rate-based optimization.

7.2 Future Work

In this section, we will discuss briefly a number of topics that we did not have the opportunity to deal with during this thesis, but we believe are worth investigating in the future.

Introduce more physical implementations of operators to SABER.

At the current version of SABER, we have a single physical implementation (algorithm) for each operator. Although for some operators, like projection, this approach seems acceptable, for others, such as Join, we need to have more than one implementation to deal with different use cases (Hash Join vs B+tree Index Nested Loops Join [24]). In such unpredictable environment, as the one we face at streaming applications, it is important to have the ability to choose the best option for every use case. The introduction of new algorithms and their implementation in SABER (both for cpu and gpu execution) would benefit greater the utilization of our cost model and result in enhanced optimization.

Implement new transformation rules on Calcite.

The built-in transformation rules of Calcite are aiming in relational transformations and don't take advantage of streaming semantics. As a result, a possible solution to this shortcoming would be the implementation of streaming oriented transformation rules, that would be placed in the third phase of our system's optimization procedure. In this phase, we search exhaustively for the optimal plan and it is used for rules applied before enforcing Join reordering.

Construct a more accurate cost model.

Certain modifications are required in order to reshape our cost model and create a more accurate version, specified by SABER. These changes would help us better estimate the

impact of the optimizations and give more optimal plans in complex use cases, based on metrics collected from the execution of SABER.

Try to add adaptivity to the system.

Data stream processing in real life applications is closely related to adaptivity because it has to evolve during its execution. Although dynamic reconfiguration is a desirable feature, stream systems may suffer from the disruption and overhead caused by the reconfiguration. As a result, safe and non-disruptive reconfiguration is still an open problem. It is very challenging to make the proper changes to SABER, in order to support adaptive operators (dataflow routing in Eddies [27]) with the minimum overhead. However, our current optimization procedure could be used as a good starting point for adaptive optimization.

Extend our system to support queries on historical data.

Currently, SABER is designed only for streaming data sources. However, in real life applications, more complex queries and data analysis are required in order to extract valuable information. Thus, we need to query both streaming and historical data (tables). It is difficult to design a unified model that supports analysis on both sources and their combination, with exactly-once semantics.

Support Flow Control: Backpressure mechanism.

It is hard to guarantee fault-tolerant and high performance stream processing, in contrary to batch processing. *"Backpressure refers to the situation where a system is receiving data at a higher rate than it can process during a temporary load spike"* [70]. If backpressure is not dealt with correctly, can lead to exhaustion of resources, or even, in the worst case, data loss. The current version of SABER doesn't support a proper backpressure mechanism, which results in the failure of the system and data loss in the cases that it can't handle the incoming data.

8

Bibliography

[1] "Apache Calcite" https://calcite.apache.org/

[2] A. Koliousis, M. Weidlich, R. C. Fernandez, P. Costa, A. L. Wolf, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in ACM SIGMOD, 2016.

[3] Arvind Arasu , Shivnath Babu , Jennifer Widom, The CQL continuous query language: semantic foundations and query execution, The VLDB Journal — The International Journal on Very Large Data Bases, v.15 n.2, p.121-142, June 2006

[4] "CQL - Continuous Query Language" https://milinda.svbtle.com/cql-continuous-query-language

[5] "The world beyond batch: Streaming 101" https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101

[6] "The world beyond batch: Streaming 102" https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-102

[7] G. Graefe, Volcano - An Extensible and Parallel Query Evaluation System, IEEE Transactions on Knowledge and Data Engineering, v.6 n.1, p.120-135, February 1994

[8] Goetz Graefe , William J. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search, Proceedings of the Ninth International Conference on Data Engineering, p.209-218, April 19-23, 1993

[9] "Introduction to Apache Calcite" http://www.slideshare.net/JordanHalterman/introduction-to-apache-calcite

[10] “Calcite’s DataTypes” https://calcite.apache.org/docs/reference.html#data-types

[11] GRAEFE, G. 1995. The cascades framework for query optimization. IEEE Data Engeneering Bulletin 18, 3 (Sept.), 19-29.

[12] Michael Armbrust, Reynold Xin, Cheng Lian, Yin Yuai, Davies Liu, Joseph Bradley, Xiangrui Meng, Tomer Kaftan, Michael Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in spark. In ACM Special Interest Group on Management of Data, 2015.

[13] “Streaming SQL with Apache Calcite” http://www.slideshare.net/julianhyde/streaming-sql-with-apache-calcite

[14] “Calcite Streaming” https://calcite.apache.org/docs/stream.html

[15] S. Babu, J. Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. Stanford University Technical Report 2002-52, November 2002.

[16] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In VLDB, 2013.

[17] “Data in Flight” http://cacm.acm.org/magazines/2010/1/55738-data-in-flight/fulltext

[18] “Package org.apache.calcite.plan.volcano” https://calcite.apache.org/apidocs/org/apache/calcite/plan/volcano/package-summary.html

[19] “Package org.apache.calcite.rel.rules” https://calcite.apache.org/apidocs/org/apache/calcite/rel/rules/package-summary.html#package.description

[20] Milinda Pathirage, Julian Hyde and Yi Pan. SamzaSQL: Scalable Fast Data Management with Streaming SQL. In Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International, 04 August 2016.

[21] "Package org.apache.calcite.plan" https://calcite.apache.org/apidocs/org/apache/calcite/plan/package-summary.html

[22] "Tumbling Window (Azure Stream Analytics)" https://msdn.microsoft.com/en-us/library/azure/dn835055.aspx

[23] Giedrius Slivinskas , Christian S. Jensen , Richard Thomas Snodgrass, A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering, IEEE Transactions on Knowledge and Data Engineering, v.13 n.1, p.21-49, January 2001

[24] J. Kang, J. Naughton, S. Viglas. Evaluating Window Joins over Unbounded Streams. To appear in Proc. Int. Conf. on Data Engineering, 2003.

[25] Stratis D. Viglas , Jeffrey F. Naughton, Rate-based query optimization for streaming information sources, Proceedings of the 2002 ACM SIGMOD international conference on Management of data, June 03-06, 2002, Madison, Wisconsin

[26] Ahmed M. Ayad , Jeffrey F. Naughton, Static optimization of conjunctive queries with sliding windows over infinite streams, Proceedings of the 2004 ACM SIGMOD international conference on Management of data, June 13-18, 2004, Paris, France

[27] Ron Avnur , Joseph M. Hellerstein, Eddies: continuously adaptive query processing, Proceedings of the 2000 ACM SIGMOD international conference on Management of data, p.261-272, May 15-18, 2000, Dallas, Texas, USA

[28] Cammert, M., Krämer, J., Seeger, B., Vaupel, S.: An approach to adaptive memory management in data stream systems. In: Proceedings of the International Conference on Data Engineering, ICDE. Atlanta (2006)

[29] Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. The design of the Borealis stream processing engine. In CIDR (2005).

[30] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, et al. STREAM: the stanford data stream management system.

[31] Sirish Chandrasekaran , Owen Cooper , Amol Deshpande , Michael J. Franklin , Joseph M. Hellerstein , Wei Hong , Sailesh Krishnamurthy , Samuel R. Madden , Fred Reiss , Mehul A. Shah, TelegraphCQ: continuous dataflow processing, Proceedings of the 2003 ACM SIGMOD international conference on Management of data, June 09-12, 2003, San Diego, California

[32] Tyler Akidau , Robert Bradshaw , Craig Chambers , Slava Chernyak , Rafael J. Fernández-Moctezuma , Reuven Lax , Sam McVeety , Daniel Mills , Frances Perry , Eric Schmidt , Sam Whittle, The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing, Proceedings of the VLDB Endowment, v.8 n.12, August 2015

[33] J. Naughton, D. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara Internet query system. IEEE Data Engineering Bulletin, June 2001.

[34] “Kappa Architecture” http://milinda.pathirage.org/kappa-architecture.com/

[35] “Questioning the Lambda Architecture” https://www.oreilly.com/ideas/questioning-the-lambda-architecture

[36] Mohamed F. Mokbel , Ming Lu , Walid G. Aref, Hash-Merge Join: A Non-blocking Join Algorithm for Producing Fast and Early Join Results, Proceedings of the 20th International Conference on Data Engineering, p.251, March 30-April 02, 2004

[37] A. M. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In SIGMOD, pages 419–430, 2004.

[38] "Cost-based optimization in Hive" https://cwiki.apache.org/confluence/display/Hive/Cost-based+optimization+in+Hive

[39] "Drill's Planner Phases" https://github.com/apache/drill/blob/master/exec/java-exec/src/main/java/org/apache/drill/exec/planner/PlannerPhase.java

[40] "Apache Storm" http://storm.apache.org/

[41] "Apache Spark" http://spark.apache.org/

[42] "Apache Flink" https://flink.apache.org/

[43] "Apache Samza" http://samza.apache.org/

[44] "Esper" http://www.espertech.com/esper/

[45] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In PLDI, pages 363–375, 2010.

[46] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In VLDB, 2013.

[47] "Using GROUP BY with ROLLUP, CUBE, and GROUPING SETS" https://technet.microsoft.com/en-us/library/bb522495(v=sql.105).aspx

[48] "Execution Plans for GROUPING SETS, ROLLUP, and CUBE" https://technet.microsoft.com/en-us/library/bb522631(v=sql.105).aspx

[49] "Session Windowing in Flink" http://data-artisans.com/session-windowing-in-flink/

[50] P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," Proceedings of the ACM SIGMOD Conference, pp. 23-34 (MayJune 1979).

[51] L. Golab and M. T. Ozsu. Issues in data stream ¨ management. SIGMOD Record, 32(2):5–14, 2003

[52] "Why we need SQL for data stream processing and real-time streaming analytics" http://sqlstream.com/2017/02/sql-for-real-time-streaming-analytics/

[53] "Cost-based Query Optimization in Apache Phoenix using Apache Calcite Slides" http://www.slideshare.net/julianhyde/costbased-query-optimization-in-apache-phoenix-using-apache-calcite

[54] B. Babcock, S. Babu, M. Dater, and R. Motwanti, "Models and Issues in data stream systems," in Proc. PODS, 2002, pp. 1–16.

[55] "Complex Event Processing" https://en.wikipedia.org/wiki/Complex_event_processing

[56] "Apache Hive" https://hive.apache.org/

[57] "Apache Kylin" http://kylin.apache.org/

[58] “Apache Drill” https://drill.apache.org/

[59] “Apache Phoenix” https://phoenix.apache.org/

[60] “Apache Kafka” https://kafka.apache.org/

[61] V. Koukis, C. Venetsanopoulos, and N. Koziris. Okeanos: Building a Cloud, Cluster by Cluster. IEEE Internet Computing, 17(3):67–71, 2013

[62] “Jupyter Notebook” http://jupyter.org/

[63] “Nigel's performance Monitor for Linux” http://nmon.sourceforge.net/pmwiki.php

[64] “Online analytical processing” https://en.wikipedia.org/wiki/Online_analytical_processing

[65] “Realtime Trending Analysis with Approximate Algorithms” https://pkghosh.wordpress.com/2014/09/10/realtime-trending-analysis-with-approximate-algorithms/

[66] “Introducing streaming k-means in Apache Spark 1.2” https://databricks.com/blog/2015/01/28/introducing-streaming-k-means-in-spark-1-2.html

[67] “Top-down parsing” https://en.wikipedia.org/wiki/Top-down_parsing

[68] “Context-free language” https://en.wikipedia.org/wiki/Context-free_language

[69] “The Evolution of Massive-Scale Data Processing slides” https://docs.google.com/presentation/d/1sEXJAni1MN34IOLtSclUfKy3C5ppUaBMtqCo1SqK6mc/present?slide=id.g63ca2a7cd_0_527

[70] “How Apache Flink™ handles backpressure” https://data-artisans.com/how-flink-handles-backpressure/

[71] J. Kwiatkowski, "Evaluation of Parallel Programs by Measurement of its Granularity", Proceedings of PPAM'01 International Conference Naleczow Poland, pp. 145-153, 2001.

[72] “Lambda architecture” https://en.wikipedia.org/wiki/Lambda_architecture

[73] “Relational Algebra” https://www.tutorialspoint.com/dbms/relational_algebra.htm

[74] “Apache Impala” https://impala.incubator.apache.org/

[75] “Apache Presto” https://prestodb.io/

[76] “Writing SQL on Streaming Data with Amazon Kinesis Analytics – Part 1” https://aws.amazon.com/blogs/big-data/writing-sql-on-streaming-data-with-amazon-kinesis-analytics-part-1/

[77] “Why we need SQL like query language for Real-Time Streaming Analytics?” http://srinathsview.blogspot.gr/2015/02/why-we-need-sql-like-query-language-for.html

[78] “Chapter 5: Large-Scale Dataflow Engines” http://www.redbook.io/ch5-dataflow.html

[79] “5 reasons why Spark Streaming’s batch processing of data streams is not stream processing” http://sqlstream.com/2015/03/5-reasons-why-spark-streamings-batch-processing-of-data-streams-is-not-stream-processing/

[80] “Continuous Queries on Dynamic Tables: Analyzing Data Streams with SQL” https://data-artisans.com/blog/continuous-queries-on-dynamic-tables-analyzing-data-streams-with-sql

[81] “Calc Operator” https://github.com/apache/calcite/blob/master/core/src/main/java/org/apache/calcite/rel/core/Calc.java

[82] “HIVE 0.14 COST BASED OPTIMIZER (CBO) TECHNICAL OVERVIEW” https://hortonworks.com/blog/hive-0-14-cost-based-optimizer-cbo-technical-overview/