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

Multi-objective query optimization for massively parallel processing in Cloud Computing

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΜΙΧΑΗΛ Κ. ΓΕΩΡΓΟΥΛΑΚΗ ΜΙΣΕΓΙΑΝΝΗ

Επιβλέπουσα : Βασιλική Καντερέ
Επίκουρη Καθηγήτρια ΕΜΠ

Αθήνα, Νοέμβριος 2021

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

Multi-objective query optimization for massively parallel processing in Cloud Computing

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΜΙΧΑΗΛ Κ. ΓΕΩΡΓΟΥΛΑΚΗ ΜΙΣΕΓΙΑΝΝΗ

Επιβλέπουσα : Βασιλική Καντερέ
Επίκουρη Καθηγήτρια ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Νοεμβρίου 2021

....	....	....
Βασιλική Καντερέ	Laurent D’Orazio	Συμεών Παπαβασιλείου
Επίκουρη Καθηγήτρια ΕΜΠ	Καθηγητής Univ. Rennes	Καθηγητής ΕΜΠ

Αθήνα, Νοέμβριος 2021

....
Μιχαήλ Κ. Γεωργουλάκης Μισεγιάννης

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

Abstract

Data processing has become a hot topic lately, as large volumes of data that need to be analyzed are produced every minute. The transition to the big data era was made easier with the commercial rise of cloud computing, and the use of massively parallel processing frameworks like Apache Spark for its processing in a parallel and distributed manner. Query optimization is a traditional DBMS optimization problem, where the query optimizer selects the optimal way to execute a query. Cloud computing features like its pricing policy led us to tackle query optimization in cloud environments as a multi-objective optimization problem, considering the objectives of execution time and monetary cost.

In this thesis, we propose a baseline query optimizer system architecture for efficient and multi-objective query optimization in a cloud-like environment. Components of this system are implemented, and it is used as a basis in our experiments.

Working with Apache Spark allows us to benefit from parallel processing and gain useful insights about processing big data in a distributed, cloud-like environment. However, trying to solve multi-objective query optimization problems using Spark comes with a significant limitation, as the optimizer of Spark SQL, Catalyst, is mostly based on heuristics and not cost based estimations. As a result, it is difficult to consider alternative query plans to compare and apply query optimization techniques that have been successfully used in relational databases.

To overcome this limitation, we reimplemented a state of the art cost model for Spark SQL from scratch to provide theoretical estimations for the costs of alternative query execution plans. Its accuracy is evaluated with large scale experiments, and an additional formula is presented and integrated into the cost model that gives an estimation for the monetary cost of a query plan in Amazon EC2, based on its execution time and computing resources used. The cost model and the formula allow us to provide solutions for multi-objective query optimization problems.

After implementing a baseline query optimization system, we move to integrate a state of the art query optimization technique, multi-objective parametric query optimization in our contribution and observe its relevance, as it is an optimization technique evaluated in a relational database. In this technique, a query is modeled as a function of a set of parameters, which must be sensitive factors for the optimization objectives.

KeyWords

Query Optimization, Cloud computing, multi-objective optimization, parametric optimization, massively parallel processing frameworks, cost model, serverless computing, Apache Spark, Spark SQL, Catalyst, HDFS, Apache Hive

Περίληψη

Η επεξεργασία δεδομένων έχει εξελιχθεί σε ένα φλέγον ζήτημα τα τελευταία χρόνια, καθώς τεράστιες ποσότητες δεδομένων που χρήζουν επεξεργασίας παράγονται καθημερινά. Η μετάβαση στην εποχή των Big Data διευκολύνθηκε με την εμπορική ανάπτυξη του υπολογιστικού νέφους, και τη χρήση περιβαλλόντων παράλληλης επεξεργασίας όπως το Apache Spark για την επεξεργασία των δεδομένων με παράλληλο και κατανεμημένο τρόπο. Η βελτιστοποίηση ερωτημάτων είναι ένα παραδοσιακό πρόβλημα βελτιστοποίησης των βάσεων δεδομένων, όπου ο βελτιστοποιητής καλείται να επιλέξει το βέλτιστο πλάνο εκτέλεσης για ένα δεδομένο ερώτημα. Χαρακτηριστικά του υπολογιστικού νέφους όπως τα μοντέλα χρέωσης που διαθέτει για τη χρήση των πόρων του, μας οδηγούν στο να αντιμετωπίσουμε τη βελτιστοποίηση ερωτημάτων ως ένα πρόβλημα με πολλαπλά κριτήρια, λαμβάνοντας υπόψη το χρόνο εκτέλεσης και το κόστος σε χρήματα των διαφορετικών πλάνων εκτέλεσης.

Στη διπλωματική εργασία αυτή, προτείνουμε μία πρότυπη αρχιτεκτονική για έναν βελτιστοποιητή ερωτημάτων που θα δουλεύει αποτελεσματικά και με πολλαπλά κριτήρια σε ένα περιβάλλον υπολογιστικού νέφους. Μέρη του συστήματος αυτού υλοποιήθηκαν, ενώ χρησιμοποιήθηκε και σαν βάση επαλήθευσης κάποιων από τα πειράματα μας.

Δουλεύοντας με το Apache Spark μπορούμε να επωφεληθούμε από την παράλληλη επεξεργασία και να αποκομίσουμε χρήσιμα συμπεράσματα για την επεξεργασία μεγάλων δεδομένων σε ένα κατανεμημένο περιβάλλον, παρόμοιο με τις πλατφόρμες του υπολογιστικού νέφους. Όμως, η επίλυση προβλημάτων βελτιστοποίησης ερωτημάτων με πολλαπλά κριτήρια χρησιμοποιώντας το Spark έχει μία σημαντική δυσκολία, καθώς ο βελτιστοποιητής του Spark SQL, ο Catalyst, λειτουργεί κυρίως με βάση κάποιους κανόνες βελτιστοποίησης και όχι παρέχοντας εκτιμώμενες τιμές χρόνου εκτέλεσης για τα διαφορετικά πλάνα εκτέλεσης. Ως αποτέλεσμα, είναι δύσκολο να λάβουμε υπόψη και να συγκρίνουμε διαφορετικά πλάνα εκτέλεσης και να εφαρμόσουμε με επιτυχία μεθόδους βελτιστοποίησης που έχουν χρησιμοποιηθεί σε σχεσιακές βάσεις δεδομένων.

Για να αντιμετωπίσουμε τη συγκεκριμένη πρόκληση υλοποιήσαμε από την αρχή ένα υπάρχον πρότυπο μοντέλο κοστολόγησης για την Spark SQL, με το οποίο θα μπορούμε να παρέχουμε εκτιμήσεις για το χρόνο εκτέλεσης διαφορετικών πλάνων. Η ακρίβεια του επαληθεύτηκε με πολυάριθμα και μεγάλα πειράματα, ενώ προτάθηκε και εισήχθη στο μοντέλο κοστολόγησης και ένας τύπος που παρέχει εκτιμήσεις για το κόστος κάθε πλάνου εκτέλεσης σε χρήματα αν το εκτελούσαμε στην πλατφόρμα Amazon EC2, με βάση το χρόνο εκτέλεσης του και τους υπολογιστικούς πόρους που χρησιμοποιεί. Το μοντέλο κοστολόγησης μαζί με τον τύπο αυτό μας επιτρέπουν να λύσουμε προβλήματα βελτιστοποίησης με πολλαπλά κριτήρια σε μία πλατφόρμα παράλληλης επεξεργασίας.

Στη συνέχεια, προχωράμε στην ένταξη στο σύστημα μας μιας άλλης πρότυπης τεχνικής βελτιστοποίησης ερωτημάτων, της παραμετρικής βελτιστοποίησης πολλαπλών κριτηρίων. Επαληθεύουμε την αξιοπιστία και την ποιότητα της τεχνικής στο περιβάλλον εκτέλεσης μας, καθώς είναι μία τεχνική που έχει χρησιμοποιηθεί μόνο σε σχεσιακές βάσεις δεδομένων. Με την τεχνική αυτή, τα ερωτήματα μοντελοποιούνται ως συναρτήσεις ενός αριθμού παραμέτρων, που πρέπει να είναι ευαίσθητοι παράγοντες όσον αφορά τα κριτήρια βελτιστοποίησης μας.

Λέξεις κλειδιά

Βελτιστοποίηση ερωτημάτων, Υπολογιστικό νέφος, Βελτιστοποίηση με πολλαπλά κριτήρια, Παραμετρική βελτιστοποίηση, Περιβάλλον Παράλληλης επεξεργασίας, Μοντέλο κοστολόγησης, Υπολογιστικό Μοντέλο Serverless, Apache Spark, Spark SQL, Catalyst, Apache Spark, HDFS, Apache Hive

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω πολύ τους δύο επιβλέποντες της διπλωματικής μου, τον κ. D’Orazio για τον χρόνο και την καθοδήγηση που μου παρείχε από την Γαλλία καθ’όλη τη διάρκεια των τελευταίων μηνών, και την κ. Καντερέ για την καθοδήγηση και στήριξη στη δουλειά μου πάντα με έναν γενικότερο ερευνητικό προσανατολισμό.

Επίσης, θα ήθελα να ευχαριστήσω από καρδιάς όλους τους φίλους μου που ήταν δίπλα μου όλα αυτά τα χρόνια, με μία ιδιαίτερη μνεία στο παρεάκι της Πάτρας που μου έδωσε την ενέργεια που χρειαζόμουν στα τελευταία και πιο δύσκολα στάδια της εργασίας μου.

Τέλος, θα ήθελα να ευχαριστήσω από καρδιάς την οικογένεια μου, τους γονείς μου και την αδελφή μου, για την αμέριστη στήριξη τους καθ’όλη τη διάρκεια των σπουδών μου και όχι μόνο.

Acknowledgements

Experiments presented in this diploma thesis were carried out using the Grid’5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

Content

Figures and Tables List

Εκτενής Περίληψη

Εισαγωγή

Σκοπός Εργασίας

Ο σκοπός της συγκεκριμένης εργασίας είναι η πρόταση για έναν βελτιστοποιητή ερωτημάτων ο οποίος θα λειτουργεί πάνω από μία αρχιτεκτονική τύπου Cloud, επιτυγχάνοντας αποτελεσματική βελτιστοποίηση με πολλαπλά κριτήρια.

Ερευνητικό Πρόβλημα

Όπως υποδηλώνει και ο τίτλος, το ερευνητικό πρόβλημα της εργασίας είναι η βελτιστοποίηση ερωτημάτων (query optimization). Πιο συγκεκριμένα, η βελτιστοποίηση ερωτημάτων είναι η διαδικασία επιλογής ενός βέλτιστου πλάνου εκτέλεσης ενός ερωτήματος με βάση τις διαθέσιμες επιλογές που μπορούν να γίνουν στα διάφορα επιμέρους στάδια της εκτέλεσης του ερωτήματος. Ασχοληθήκαμε με το πρόβλημα αυτό λαμβάνοντας επιπλέον υπόψη τους διαθέσιμους υπολογιστικούς πόρους που έχουμε σε μία πλατφόρμα cloud, η ποσότητα και το μέγεθος των οποίων μπορούν να έχουν μεγάλη επιρροή στα κριτήρια βελτιστοποίησης.

Καθώς το περιβάλλον ενασχόλησης μας είναι το Cloud, προσαρμοζόμαστε στα χαρακτηριστικά του και υποστηρίζουμε ότι η βελτιστοποίηση ερωτημάτων σε τέτοιο περιβάλλον δεν μπορεί να λυθεί ως ένα πρόβλημα ενός μόνο κριτηρίου βελτιστοποίησης, καθώς η τιμολόγηση που υπάρχει στην κάθε πλατφόρμα για παράδειγμα, οδηγεί στην ύπαρξη ενός άλλου πολύ σημαντικού κριτηρίου βελτιστοποίησης, του χρηματικού κόστους. Σε αυτή την περίπτωση, το πρόβλημα μας είναι δύο κριτηρίων, καθώς αναζητούμε για έναν συμβιβασμό ο οποίος εξασφαλίζει μικρό χρόνο εκτέλεσης και χαμηλό κόστος. Συνεπώς, το πρόβλημα με το οποίο ασχολούμαστε είναι η βελτιστοποίηση ερωτημάτων πολλαπλών κριτηρίων σε περιβάλλον Cloud.

Χρήσιμες Έννοιες

Cloud & Serverless

Το υπολογιστικό νέφος (Cloud computing) αναφέρεται στην κατ’απαίτηση ενοικίαση υπολογιστικών πόρων μέσω του διαδικτύου. Τέτοιοι πόροι μπορεί να είναι CPUs, εξυπηρετητές, αποθηκευτικός χώρος, ή βάσεις δεδομένων, οι οποίοι επιτρέπουν στους χρήστες αλλά και στις οργανώσεις που τα χρησιμοποιούν να επικεντρωθούν στα πιο ουσιαστικά κομμάτια της δουλειάς τους, ελαχιστοποιώντας τα καθήκοντα και το κόστος του IT, καθώς το στήσιμο ενός συστήματος (CPUs, χώρος αποθήκευσης), δεν αφορά πλέον αυτούς.

Δύο από τα πιο σημαντικά χαρακτηριστικά του Cloud, είναι η μεγάλη ευελιξία/ελαστικότητα του, καθώς και το μοντέλο πληρωμής με βάση τη χρήση των υπηρεσιών του.

Η ελαστικότητα του Cloud είναι πολύ σημαντική για τον χρήστη, καθώς οι διάφοροι πόροι/υπηρεσίες μπορούν με απόλυτη ευελιξία να ενοικιαστούν ή να απελευθερωθούν, αυξάνοντας ή μειώνοντας το μέγεθος και την ποσότητα τους, ανάλογα με τις ανάγκες του χρήστη, ή το φόρτο εργασίας μίας εφαρμογής του.

Όσον αφορά το μοντέλο πληρωμής, η δυνατότητα να πληρώνονται οι πόροι σε μία short-term βάση (π.χ. οι επεξεργαστές ανά ώρα, ο χώρος αποθήκευσης ανά μέρα), και να ελευθερώνονται όταν δεν χρειάζονται πια, σημαίνει ότι οι χρήστες μπορούν να γλιτώσουν σημαντικά έξοδα υποδομών, ενώ οι πάροχοι μπορούν να βελτιστοποιήσουν τη χρονοδρομολόγηση των πόρων που προσφέρουν.

Το Cloud διαθέτει 3 μοντέλα υπηρεσιών. Αρχικά, υπάρχει το Infrastructure as a service (IaaS), το οποίο έχει τις ίδιες παροχές με ένα data center, δίνοντας τη δυνατότητα στους χρήστες να νοικιάζουν και να διαχειρίζονται υπολογιστικούς πόρους, αποθηκευτικό χώρο, κ.α. με τη μορφή εικονικών μηχανών. Επιπλέον, το Platform as a service (PaaS) παρέχει μια ολοκληρωμένη πλατφόρμα με εργαλεία για την ανάπτυξη και τη διαχείριση εφαρμογών. Τέλος, το Software as a service (SaaS) παρέχει στους χρήστες μία εφαρμογή, την οποία διαχειρίζεται πλήρως ο πάροχος της πλατφόρμας SaaS, και στην οποία ο χρήστης έχει πρόσβαση μέσω ενός API και ενός περιηγητή ιστού.

Τα είδη του Cloud διακρίνονται σε δημόσιο, ιδιωτικό, και υβριδικό, ανάλογα με το επίπεδο προσβασιμότητας που έχουν στο κοινό.

Μία πρόσφατη εξέλιξη του Cloud, είναι το Serverless. Οι serverless πλατφόρμες επιτρέπουν σε προγραμματιστές να γράφουν τον κώδικα τους σε αυτές, και η πλατφόρμα να τον εκτελεί με όσους πόρους χρειάζονται. Οι προγραμματιστές δεν χρειάζεται να έχουν κρατήσει από πριν κάποιους πόρους, ούτε να έχουν στήσει έναν δικό τους server (εξ ου και το serverless), ενώ το γεγονός ότι χρεώνονται μόνο για τους πόρους που χρησιμοποιούνται όταν εκτελείται ο κώδικας τους, υποδηλώνει ότι τα οικονομικά οφέλη μπορούν να είναι σημαντικά. Το μοντέλο υπηρεσιών που περιγράφτηκε ονομάζεται Function as a service (FaaS), και είναι αυτό που συνήθως αναφερόμαστε όταν μιλάμε για serverless. Στο μοντέλο αυτό, οι χρήστες γράφουν τον κώδικα τους σε μορφή συναρτήσεων (functions), οι οποίες είναι και οι μονάδες εκτέλεσης.

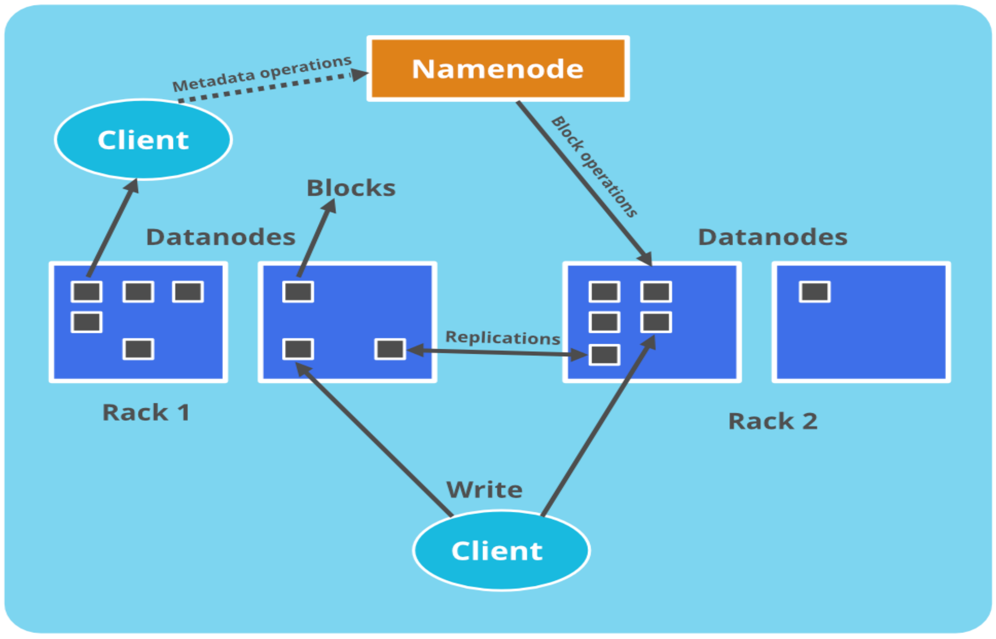
Παρά τα σημαντικά οφέλη του (απλό στη χρήση, οικονομικό, ελαστική χρήση), έχει και κάποια σημαντικά μειονεκτήματα, που δεν του έχουν επιτρέψει ακόμη να φτάσει τις επιδόσεις που θα μπορούσε. Κάποια από αυτά είναι η μικρή διάρκεια ζωής των συναρτήσεων (στο AWS Lambda [26] δεν υποστηρίζονται συναρτήσεις που διαρκούν πάνω από 15 λεπτά). Ακόμη, οι συναρτήσεις δεν μπορούν να επικοινωνούν μεταξύ τους ενώ δεν υποστηρίζονται ακόμη καλά ετερογενείς πόροι, με τις περισσότερες FaaS πλατφόρμες να επιτρέπουν στους χρήστες να χρησιμοποιούν CPU και μία ποσότητα RAM. Πολλές προσπάθειες έχουν γίνει για να αντιμετωπιστούν αυτά τα μειονεκτήματα. Μία αξιόλογη δουλειά είναι το Pocket [21], που έχτισε πάνω στο Apache Crail [19].

Διαχείριση Μεγάλων Δεδομένων

Τα μεγάλα δεδομένα είναι ένας τεράστιος όγκος δεδομένων τον οποίο δεν μπορούμε να διαχειριστούμε με παραδοσιακούς τρόπους, π.χ. σχεσιακές βάσεις δεδομένων. Ορίζονται ως οποιοσδήποτε μεγάλος όγκος δεδομένων που έχει τα εξής χαρακτηριστικά:

- Volume(όγκος): πολύ μεγάλη ποσότητα δεδομένων
- Velocity(ταχύτητα): πολύ μεγάλη ταχύτητα παραγωγής δεδομένων
- Variety(ποικιλία): μεγάλη και ευρεία ποικιλία δεδομένων

Οι εφαρμογές που χρησιμοποιούν big data έχουν, όπως είναι λογικό, μεγάλες απαιτήσεις σε υπολογιστικούς πόρους. Το Cloud είναι ιδανικό για τη διαχείριση τους, λόγω της ελαστικότητας του αλλά και της τιμολόγησης του. Ανάλογα με την εφαρμογή, ο χρήστης πρέπει να πάρει αποφάσεις για το που θα αποθηκεύσει τα δεδομένα του, πως θα τα διαχειριστεί και πως θα τα επεξεργαστεί.



Εικόνα 1. Αρχιτεκτονική HDFS

Το Apache Hadoop περιλαμβάνει μία συλλογή από εργαλεία λογισμικού για τη διαχείριση μεγάλης ποσότητας δεδομένων σε συμπλέγματα υπολογιστών (clusters). Περιλαμβάνει το κομμάτι της αποθήκευσης, το HDFS, και το κομμάτι της επεξεργασίας, το MapReduce. Το HDFS είναι ένα κατανεμημένο σύστημα αρχείων, που εγγυάται αξιόπιστη αποθήκευση μεγάλων δεδομένων. Το HDFS αποτελείται από έναν master κόμβο, που διαχειρίζεται το σύστημα αρχείων και κρατάει την τοποθεσία του κάθε data block, καθώς και πολυάριθμους κόμβους δεδομένων (datanodes), όπου τα δεδομένα αποθηκεύονται με τη μορφή blocks. Το MapReduce είναι μία υπολογιστική δομή για επεξεργασία μεγάλων συνόλων δεδομένων με έναν παράλληλο και κατανεμημένο αλγόριθμο σε ένα σύμπλεγμα υπολογιστών.

Το Apache Spark είναι μία δομή ανάλυσης για επεξεργασία δεδομένων μεγάλης κλίμακας, αντίστοιχο με το MapReduce. Είναι μια κατανεμημένη υπολογιστική δομή που όπως και το MapReduce χωρίζει τις διεργασίες σε κόμβους εργάτες, όμως οι επιδόσεις του είναι σημαντικά καλύτερες, καθώς χρησιμοποιεί RAM για να αποθηκεύει δεδομένα στην cache και να τα επεξεργάζεται ενώ οι εφαρμογές ακόμη τρέχουν, αποθηκεύοντας τα ενδιάμεσα αποτελέσματα μίας διεργασίας στη μνήμη, και αποφεύγοντας έτσι την επιπλέον καθυστέρηση από τις I/O διεργασίες στο HDFS. Θεωρείται πλέον μία καλύτερη εναλλακτική από το MapReduce, αλλά είναι και μία πρότυπη προσθήκη στο οικοσύστημα του Hadoop, καθώς έχει απόλυτη συμβατότητα με το HDFS αλλά και το Apache Hive. Ο πυρήνας του Spark είναι το Spark Core, που αποτελεί την προγραμματιστική διεπαφή του Spark, ενώ υπάρχει και το Spark SQL, που είναι ένα συστατικό του Spark που λειτουργεί πάνω από το Spark Core, και εισήγαγε τα DataFrames, που είναι η οργάνωση μίας κατανεμημένη συλλογής δεδομένων σε στήλες. Αποτελεί την SQL διεπαφή του Spark και υποστηρίζει δομημένα και ημι-δομημένα δεδομένα.

Βελτιστοποίηση Ερωτημάτων

Μιλήσαμε για τη διαχείριση μεγάλου όγκου δεδομένων σε περιβάλλον cloud. Τα συστήματα διαχείρισης δεδομένων χωρίζονται στις σχεσιακές βάσεις δεδομένων, αλλά και στις NoSQL που χειρίζονται κάθε είδους δεδομένα ανεξαρτήτως δομής. Ακόμη κι έτσι όμως, οι πιο πολλές NoSQL βάσεις έχουν και SQL διεπαφή, για διευκόλυνση των χρηστών.

Οι γλώσσες ερωτημάτων (όπως η SQL) χρησιμοποιούνται για να κάνουν κλήσεις σε βάσεις δεδομένων για ανάγνωση, εισαγωγή, ενημέρωση ή διαγραφή δεδομένων. Η επεξεργασία ερωτημάτων είναι από τις πιο σημαντικές διεργασίες που συμβαίνουν σε μία βάση, όπου υψηλού επιπέδου ερωτήματα μεταφράζονται σε low level εκφράσεις. Περιλαμβάνει τέσσερα βήματα, την συντακτική ανάλυση του ερωτήματος, τη μετάφραση του σε ένα δέντρο λογικών εκφράσεων, την βελτιστοποίηση του με αποτέλεσμα ένα δέντρο βέλτιστων φυσικών εκφράσεων, και τέλος την εκτέλεση και επαλήθευση του ερωτήματος. Εμάς μας ενδιαφέρει το τρίτο στάδιο, της βελτιστοποίησης.

Η κλασσική βελτιστοποίηση ερωτημάτων, περιλαμβάνει την σύγκριση των διαφορετικών φυσικών πλάνων εκτέλεσης με βάση ένα κριτήριο βελτιστοποίησης, που συνήθως είναι ο χρόνος εκτέλεσης του ερωτήματος. Η επιλογή του βέλτιστου πλάνου είναι ένα απαιτητικό πρόβλημα βελτιστοποίησης [87]. Το σύνολο των πιθανών πλάνων εκτέλεσης περιλαμβάνει την εξέταση όλων των πιθανών τρόπων απόκτησης των δεδομένων (πλήρης σάρωση πίνακα, αναζήτηση ευρετηρίου), καθώς και όλων των πιθανών τρόπων συνένωσης 2 πινάκων (sort merge join, hash join) και των αποφάσεων χρονοπρογραμματισμού (π.χ. σειρά των συνενώσεων).

Η βελτιστοποίηση, ή αλλιώς μετατροπή του λογικού πλάνου σε ένα φυσικό, μπορεί να γίνει είτε με τη χρήση κανόνων (rule-based), είτε με βάση το κόστος της κάθε επιλογής (cost-based). Στη δεύτερη περίπτωση, που μπορεί να μας δώσει πιο σωστές επιλογές όταν γίνεται σωστά, χρειαζόμαστε ένα μοντέλο κοστολόγησης (cost model), που είναι ο πυρήνας ενός βελτιστοποιητή ερωτημάτων. Τα μοντέλα κοστολόγησης αποτελούνται από 2 μέρη. Το πρώτο είναι το λογικό, που είναι υπεύθυνο για την εκτίμηση του όγκου των δεδομένων που εμπλέκονται σε μία λειτουργία, το οποίο και εξαρτάται μόνο από τα δεδομένα που βρίσκονται στη βάση. Το δεύτερο μέρος είναι υπεύθυνο για την εκτίμηση του κόστους των διαφορετικών αλγορίθμων που μπορούν να χρησιμοποιηθούν για την εκτέλεση του ερωτήματος.

Μία γνωστή τεχνική βελτιστοποίησης ερωτημάτων, είναι η παραμετρική βελτιστοποίηση ερωτημάτων (ΠΕ), που υιοθετεί μία διαφορετική προσέγγιση στο πως μοντελοποιείται ένα πλάνο εκτέλεσης. Συγκεκριμένα, κάθε πλάνο εκτέλεσης ενός ερωτήματος συσχετίζεται με μία συνάρτηση κόστους c: $Rn-> R$ αντί για μία σταθερή τιμή c, εκπροσωπώντας το κόστος ενός πλάνου εκτέλεσης ως μία συνάρτηση n παραμέτρων, των οποίων οι τιμές δεν είναι ακόμη γνωστές. Οι παράμετροι στην ΠΕ μπορεί να είναι για παράδειγμα η επιλεξιμότητα ενός κατηγορήματος (predicate selectivity), ή ο διαθέσιμος χώρος αποθήκευσης. Ο στόχος της ΠΕ είναι η εύρεση ενός βέλτιστου πλάνου εκτέλεσης για κάθε πιθανό συνδυασμό των τιμών των παραμέτρων. Ένα σημαντικό πλεονέκτημα της ΠΕ, είναι ότι η βελτιστοποίηση συμβαίνει πριν την εκτέλεση του ερωτήματος, με αποτέλεσμα να μην υπάρχει χρόνος βελτιστοποίησης κατά τη διάρκεια της εκτέλεσης του ερωτήματος, παρά μόνο αντιστοίχιση του ερωτήματος σε έναν συνδυασμό παραμέτρων, και επιλογή του βέλτιστου πλάνου που έχει υπολογιστεί πριν την εκτέλεση του ερωτήματος.

Ένα παράδειγμα βελτιστοποιητή ερωτημάτων, τον οποίο θα χρησιμοποιήσουμε και στη συνέχεια, είναι ο Catalyst, ο βελτιστοποιητής της Spark SQL. Λειτουργεί κατά βάση με κανόνες για τη βελτιστοποίηση, όντας παράλληλα επεκτάσιμος και επιτρέποντας την προσθήκη νέων κανόνων. Κάποιες επιλογές γίνονται με βάση εκτιμήσεις κόστους (π.χ. η επιλογή των join), παρ' όλα αυτά το γεγονός ότι παραμένει κυρίως rule based θεωρείται μειονέκτημα, το οποίο οδήγησε τον Baldacci και τον Golfarelli να παρουσιάσουν ένα μοντέλο κοστολόγησης για την Spark SQL [3], με σκοπό να είναι το πρώτο βήμα για να μετατραπεί ο Catalyst σε έναν cost based βελτιστοποιητή .

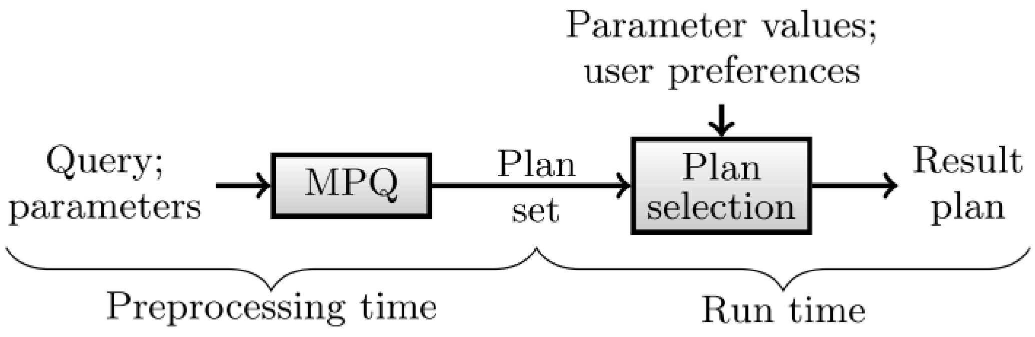
Βελτιστοποίηση πολλαπλών κριτηρίων

Η βελτιστοποίηση πολλαπλών κριτηρίων περιλαμβάνει προβλήματα όπου αποφάσεις πρέπει να παρθούν με βάση παραπάνω από μία συναρτήσεις στόχου που πρέπει να βελτιστοποιηθούν ταυτόχρονα. Όταν οι συναρτήσεις στόχου είναι αντιφατικές (δηλαδή η αύξηση του ενός μειώνει το άλλο πχ.), δεν υπάρχει μοναδική λύση, παρά ένα σύνολο από βέλτιστες λύσεις, γνωστές και ως βέλτιστες κατά Pareto.

Στο περιβάλλον cloud, το οποίο και μας ενδιαφέρει, τα 2 βασικά προβλήματα βελτιστοποίησης πολλαπλών κριτηρίων, είναι η βελτιστοποίηση ερωτημάτων, και η βελτιστοποίηση πόρων, που περιλαμβάνει την βελτιστοποίηση της κατανομής και του χρονοπρογραμματισμού των πόρων σε μία πλατφόρμα Cloud.

Όσον αφορά τη βελτιστοποίηση ερωτημάτων, που είναι και το ερευνητικό πρόβλημα που μας απασχολεί, σε μία παράλληλη (π.χ. IaaS) πλατφόρμα, αυτή συμβαίνει ως εξής. Ο χρήστης υποβάλλει το ερώτημα του σε μία γλώσσα υψηλού επιπέδου (π.χ. SQL). Το ερώτημα λαμβάνεται από τους master κόμβους του δικτύου, και ο βελτιστοποιητής υπολογίζει ένα αναλυτικό πλάνο εκτέλεσης για το ερώτημα με βάση ένα μοντέλο κοστολόγησης, και το αποσυνθέτει σε υποερωτήματα, τα οποία αναθέτονται στους κόμβους εργάτες με βάση κάποιο κριτήριο εξισορρόπησης του φόρτου. Ο κάθε εργάτης υπολογίζει το υποερώτημα του και επιστρέφει το αποτέλεσμα στον master. Ο master ενώνει όλα τα υποαποτελέσματα, και παράγει την απάντηση του ερωτήματος.

Η βελτιστοποίηση ερωτημάτων πολλών κριτηρίων (ΠΒ), είναι μία γενίκευση της κλασσικής βελτιστοποίησης ερωτημάτων, και αναπαριστά κάθε πλάνο εκτέλεσης ερωτήματος με έναν πίνακα κόστους c ∈ Rn , που περιγράφει το κόστος του πλάνου εκτέλεσης με βάση καθεμία από τις μετρικές βελτιστοποίησης. Ο στόχος της ΠΒ είναι η εύρεση ενός συνόλου από βέλτιστα κατά Παρέτο πλάνων, για τα οποία κανένα δεν είναι καλύτερο από τα υπόλοιπα με βάση όλα τα κριτήρια βελτιστοποίησης.



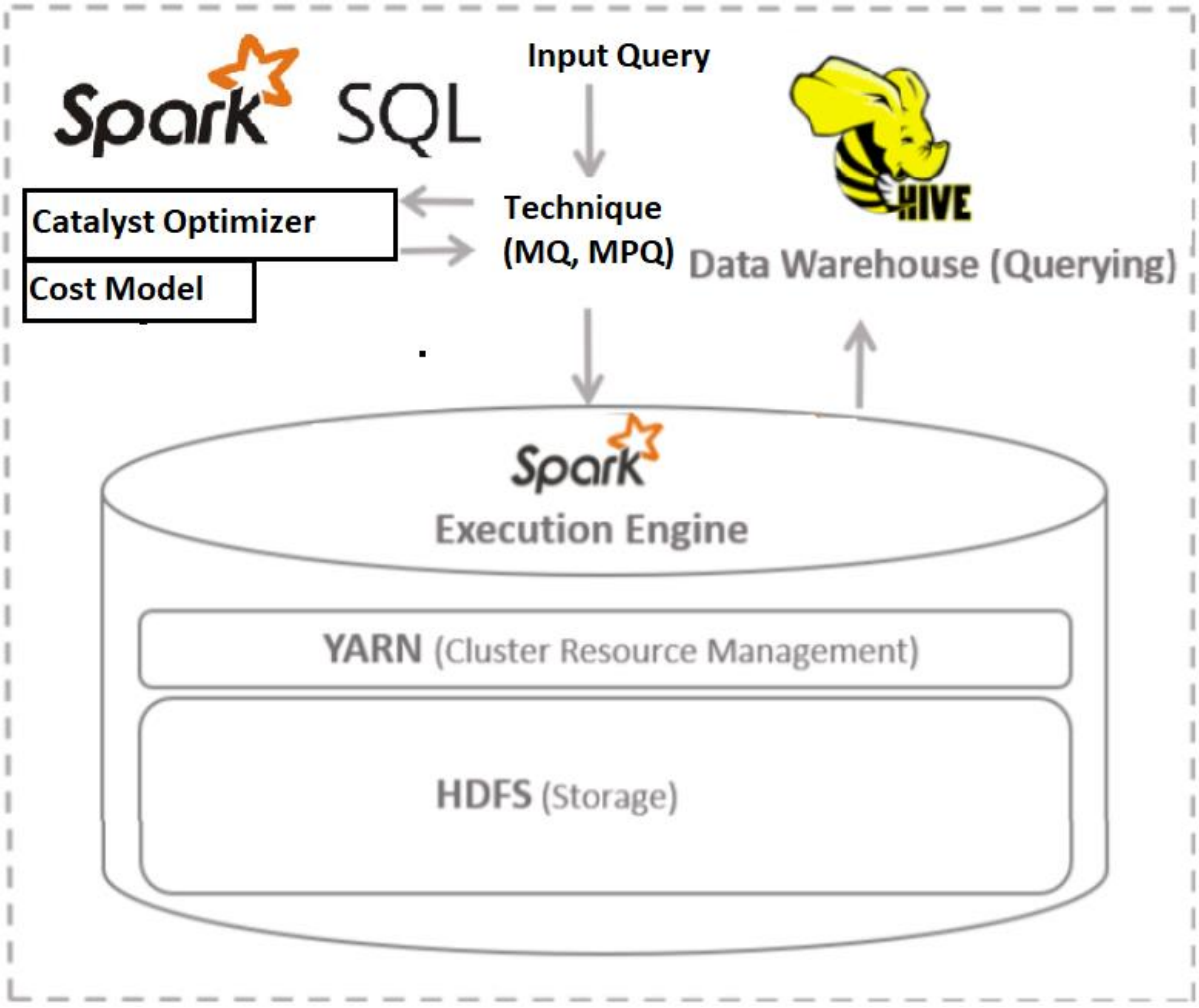
Εικόνα 2. Η λογική της ΠΠΒ.

Το 2014, ο Trummer και ο Koch παρουσίασαν την παραμετρική βελτιστοποίηση ερωτημάτων πολλαπλών κριτηρίων (ΠΠΒ) [1], που ήταν μία προσπάθεια γενίκευσης του προβλήματος βελτιστοποίησης αυτού, συνδυάζοντας υπάρχουσες τεχνικές της ΠΒ και της ΠΕ.

Στην ΠΠΒ το κόστος ενός πλάνου εκτέλεσης αναπαρίσταται ως μία διανυσματική συνάρτηση c : $R^n \rightarrow R^m$, λαμβάνοντας έτσι υπόψη πολλαπλές παραμέτρους, αλλά και πολλαπλά κριτήρια βελτιστοποίησης. Ο στόχος της ΠΠΒ είναι η παραγωγή ενός συνόλου από βέλτιστα κατά Παρέτο πλάνα για κάθε πιθανό συνδυασμό παραμέτρων. Όπως και στην ΠΕ, όλα τα πλάνα εκτέλεσης υπολογίζονται και συγκρίνονται πριν την εκτέλεση του ερωτήματος.

Πρόταση

Στόχος αυτής της εργασίας είναι και η πρόταση μίας αρχιτεκτονικής για αποτελεσματική και πολλαπλών κριτηρίων βελτιστοποίηση ερωτημάτων σε περιβάλλον Cloud. Η προτεινόμενη αρχιτεκτονική μας παρατίθεται στην ακόλουθη εικόνα.



Εικόνα 3. Προτεινόμενη αρχιτεκτονική.

Αρχικά, για το πειραματικό σκέλος της εργασίας μας χρησιμοποιήσαμε το Grid ‘5000, μία ευέλικτη και μεγάλης κλίμακας πλατφόρμα για έρευνα σε όλους τους τομείς της επιστήμης των υπολογιστών. Τη χρησιμοποιήσαμε στα πρότυπα μίας πλατφόρμας IaaS, κάθε φορά ενοικιάζοντας τους πόρους που χρειαζόμασταν για τα πειράματα μας.

Για το κομμάτι της αποθήκευσης, χρησιμοποιήσαμε το HDFS και είχαμε τα δεδομένα μας αποθηκευμένα σε ένα σταθερό αριθμό από κόμβους δεδομένων, ώστε να επωφεληθούμε από την κατανεμημένη φύση του, την ταχύτητα και την αξιοπιστία του. Για το κομμάτι της επεξεργασίας των δεδομένων χρησιμοποιήσαμε το Apache Spark, με τους executors του να είναι η εκτελεστική μονάδα της αρχιτεκτονικής μας. Ανάλογα με τον αριθμό των Spark executors που θέλουμε να χρησιμοποιήσουμε για την εκτέλεση του ερωτήματος, χρησιμοποιούμε τον ίδιο αριθμό κόμβων για υπολογισμό (με κάθε executor να ανατίθεται σε διαφορετικό κόμβο του cluster μας). Όλοι οι executors έχουν τα ίδια χαρακτηριστικά, οπότε η προαναφερθείσα διαδικασία είναι παρόμοια με την ενοικίαση κάποιων πόρων από μία πλατφόρμα όπως η Amazon EC2. Ως βάση δεδομένων, χρησιμοποιήσαμε πίνακες του Apache Hive, καθώς και την SQL διεπαφή του Spark, Spark SQL, για να πραγματοποιούμε ερωτήματα στη βάση μας.

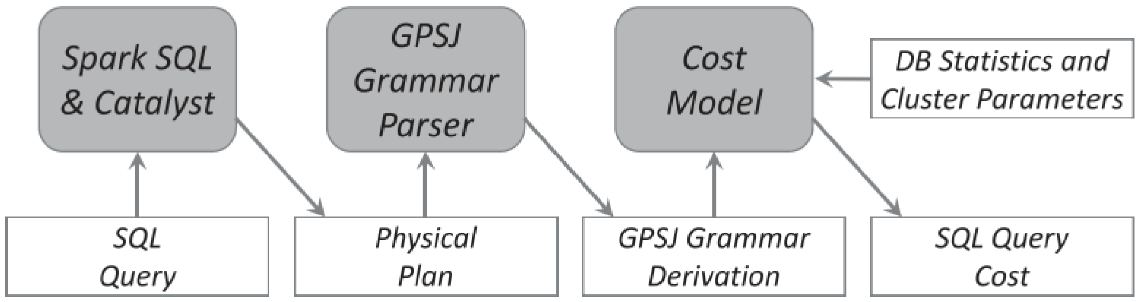
Όσον αφορά τη βελτιστοποίηση ερωτημάτων, η επιλογή της Spark SQL οδηγεί σε μία πρόκληση. Όπως προαναφέραμε, ο βελτιστοποιητής της, Catalyst, έχει κάποια σημαντικά μειονεκτήματα, καθώς λειτουργεί κατά βάση με κανόνες (rule based). Γι' αυτό το λόγο, η αρχιτεκτονική που προτείνουμε θα χρησιμοποιεί το μοντέλο κοστολόγησης που έχει προταθεί για την Spark SQL [3], ώστε να παίρνει αποφάσεις αποτελεσματικά και με εκτιμήσεις κόστους, και για να μπορεί να λαμβάνει υπόψη όλα τα πιθανά πλάνα εκτέλεσης. Ο στόχος μας είναι να εισάγουμε τεχικές βελτιστοποίησης πολλαπλών κριτηρίων που έχουν χρησιμοποιηθεί με επιτυχία σε σχεσιακές βάσεις δεδομένων, σε μία παράλληλη πλατφόρμα όπως είναι το Spark.

Χρησιμοποιώντας το μοντέλο αυτό μας επιτρέπει να προτείνουμε ένα σύστημα όπου η βελτιστοποίηση συμβαίνει μόνο με εκτιμήσεις κόστους, για το χρόνο εκτέλεσης των ερωτημάτων. Όμως όπως προαναφέραμε, εμάς μας ενδιαφέρει η βελτιστοποίηση πολλαπλών κριτηρίων, οπότε προτείνουμε και έναν τύπο για τον υπολογισμό κατ' εκτίμηση του χρηματικού κόστους κάθε πλάνου εκτέλεσης, με βάση τιμές από το Amazon EC2. Το όραμα μας περιλαμβάνει έναν ευέλικτο βελτιστοποιητή ερωτημάτων, ικανό να συνδυάσει διαφορετικές τεχνικές βελτιστοποίησης. Είναι ικανός να εφαρμόσει τόσο ΠΒ, όσο και ΠΠΒ. Θεωρούμε πως αυτό είναι εφικτό με το μοντέλο κοστολόγησης που χρησιμοποιούμε, και το επαληθεύουμε για συγκεκριμένα ερωτήματα, υπολογίζοντας και συγκρίνοντας όλα τα σχετικά πλάνα εκτέλεσης.

Οι συνεισφορές μας προς την κατεύθυνση υλοποίησης της προτεινόμενης αρχιτεκτονικής, είναι οι ακόλουθες:

- Ανακατασκευή του μοντέλου κοστολόγησης για τη Spark SQL από το μηδέν, ώστε να μπορέσουμε να χρησιμοποιήσουμε έναν cost based βελτιστοποιητή ερωτημάτων. Αφού υλοποιήσαμε το μοντέλο, χρειάστηκαν πειράματα μεγάλης κλίμακας και πολλές συγκρίσεις με πειραματικές τιμές από το Spark, για να επαληθευτεί η αξιοπιστία του.
- Προτείναμε έναν τύπο για την εκτίμηση του χρηματικού κόστους των ερωτημάτων της Spark SQL, με βάση την τιμολόγηση του Amazon EC2. Η πολυκριτηριακή φύση των προβλημάτων βελτιστοποίησης σε παράλληλα συστήματα επαληθεύτηκε από τη δημιουργία βέλτιστων κατά Παρέτο πλάνων.
- Εφαρμόσαμε παραμετρική βελτιστοποίηση ερωτημάτων σε περίπλοκα ερωτήματα, χρησιμοποιώντας το μοντέλο κοστολόγησης μας για την παραγωγή και σύγκριση όλων των σχετικών πλάνων. Στο τέλος, ένα σύνολο από βέλτιστα κατά Παρέτο πλάνα παράγεται.

Υλοποίηση και Πειραματική Επαλήθευση μοντέλου κοστολόγησης



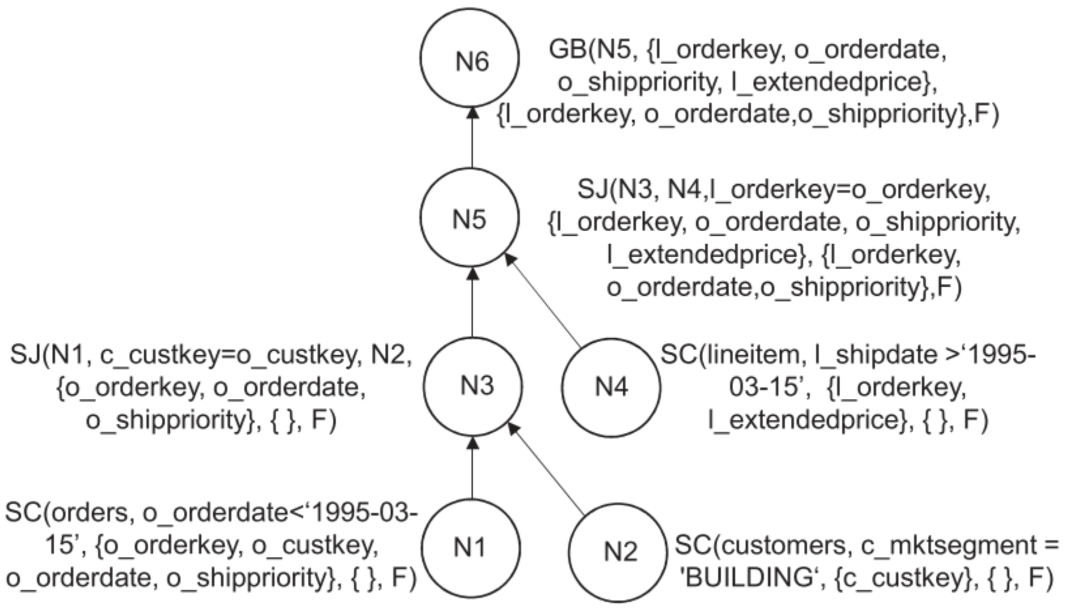
Εικόνα 4. Λειτουργία μοντέλου κοστολόγησης.

Η πρώτη συνεισφορά της εργασίας αυτής είναι η υλοποίηση του μοντέλου κοστολόγησης για την Spark SQL που προτάθηκε από τους Baldacci και Golfarelli. Όπως προαναφέραμε, η Spark SQL βασίζεται στον Catalyst, ο οποίος βασίζεται σε κανόνες κυρίως, αν και κάνει και κάποιες επιλογές με βάση το κόστος. Ο παρακάτω πίνακες συνοψίζει τις ομοιότητες και τις διαφορές των μοντέλων κοστολόγησης (του Catalyst και του Baldacci)

	Catalyst Optimizer	Baldacci SQL Cost Model
Τύποι Ερωτημάτων	Όλα	GPSJ(no UNION ALL, OUTER JOIN)
Επιλογή joins με βάση κόστος	✓	✓
Συλλογή στατιστικών από πίνακες και στήλες	✓	✓
Λαμβάνει υπόψη παραμέτρους του cluster	✗	✓
Βασίζεται στις επιδόσεις του συστήματος και του δικτύου.	✗	✓
Αναλυτικός υπολογισμός εκτίμησης κόστους.	✗	✓

Πίνακας 1. Σύγκριση μοντέλου κοστολόγησης και Catalyst

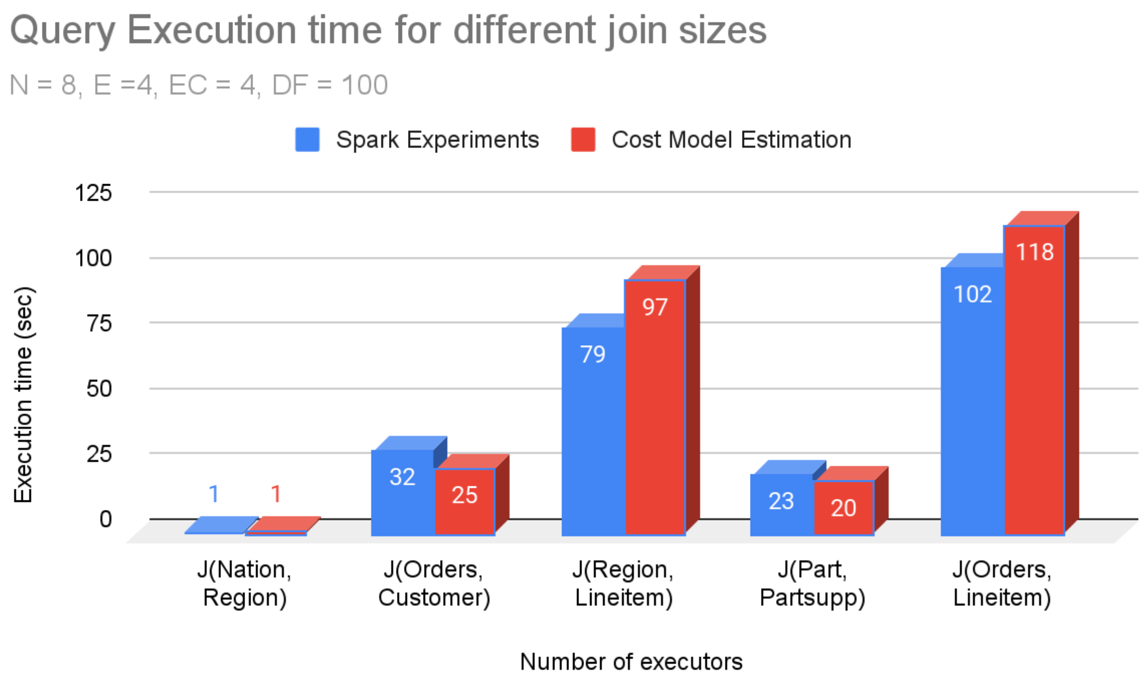
TPC-H QUERY 3
SELECT l_orderkey, o_orderdate, o_shippriority, sum(l_extprice)
FROM customer, orders, lineitem
WHERE c_mktsegment = ‘BUILDING’ AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < date‘1995-03-15’ AND l_shipdate > date‘1995-03-15’
GROUP BY l_orderkey, o_orderdate, o_shippriority


N6
GB(N5, {l_orderkey, o_orderdate, o_shippriority, l_extendedprice}, {l_orderkey, o_orderdate,o_shippriority},F)
N5
SJ(N3, N4,l_orderkey=o_orderkey, {l_orderkey, o_orderdate, o_shippriority, l_extendedprice}, {l_orderkey, o_orderdate,o_shippriority},F)
SJ(N1, c_custkey=o_custkey, N2, {o_orderkey, o_orderdate, o_shippriority}, { }, F)
N3
N4
SC(lineitem, l_shipdate >‘1995-03-15’, {l_orderkey, l_extendedprice}, { }, F)
SC(orders, o_orderdate<‘1995-03-15’, {o_orderkey, o_custkey, o_orderdate, o_shippriority}, { }, F)
N1
N2
SC(customers, c_mktsegment = 'BUILDING‘, {c_custkey}, { }, F)

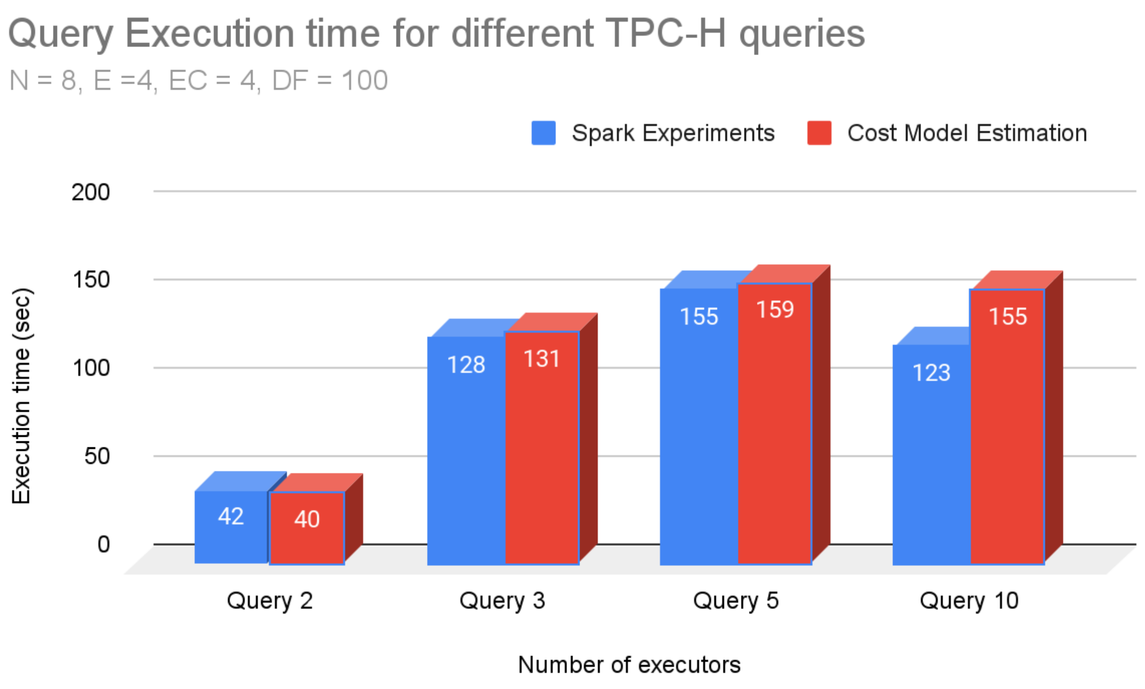
Εικόνα 5. Πλάνο εκτέλεσης στη μορφή δέντρου

Στη συνέχεια, όπως φαίνεται στην εικόνα το πλάνο εκτέλεσης ενός query αναπαρίσταται στη μορφή δέντρου, όπου οι κόμβοι αποτελούν τις ενέργειες που πρέπει να γίνουν. Η αναλυτική εκτίμηση του χρόνου εκτέλεσης έρχεται με το άθροισμα των επιμέρους χρόνων εκτέλεσης του κάθε κόμβου.

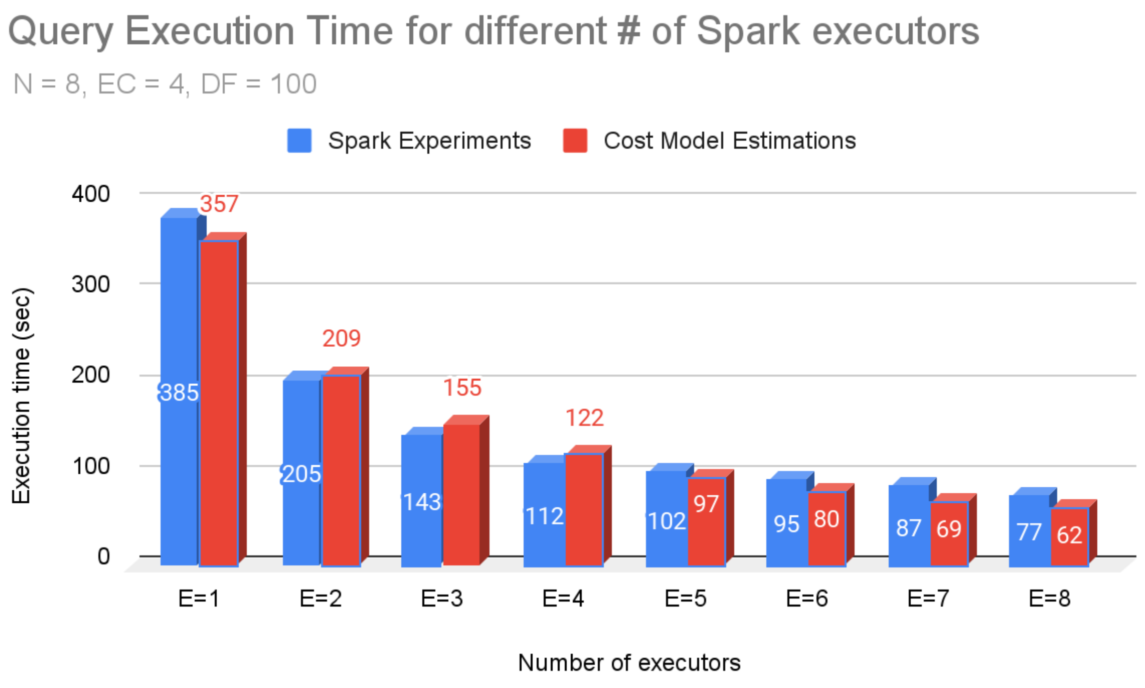
Αφού υλοποιήθηκε το μοντέλο κοστολόγησης, ακολούθησε η πειραματική του επαλήθευση με βάση διάφορους παράγοντες. Η επαλήθευση πραγματοποίηθηκε συγκρίνοντας τις εκτιμήσεις του μοντέλου κοστολόγησης, με αληθινές πειραματικές τιμές από το Spark. Επαληθεύσαμε το μοντέλο κοστολόγησης ως προς 2 παραμέτρους, την ακρίβεια της εκτίμησης, και την επιτυχία της πρόβλεψης (όσον αφορά το βέλτιστο πλάνο εκτέλεσης). Αρχικά, ασχοληθήκαμε με την ακρίβεια της εκτίμησης και πραγματοποιήσαμε τα ακόλουθα πειράματα.



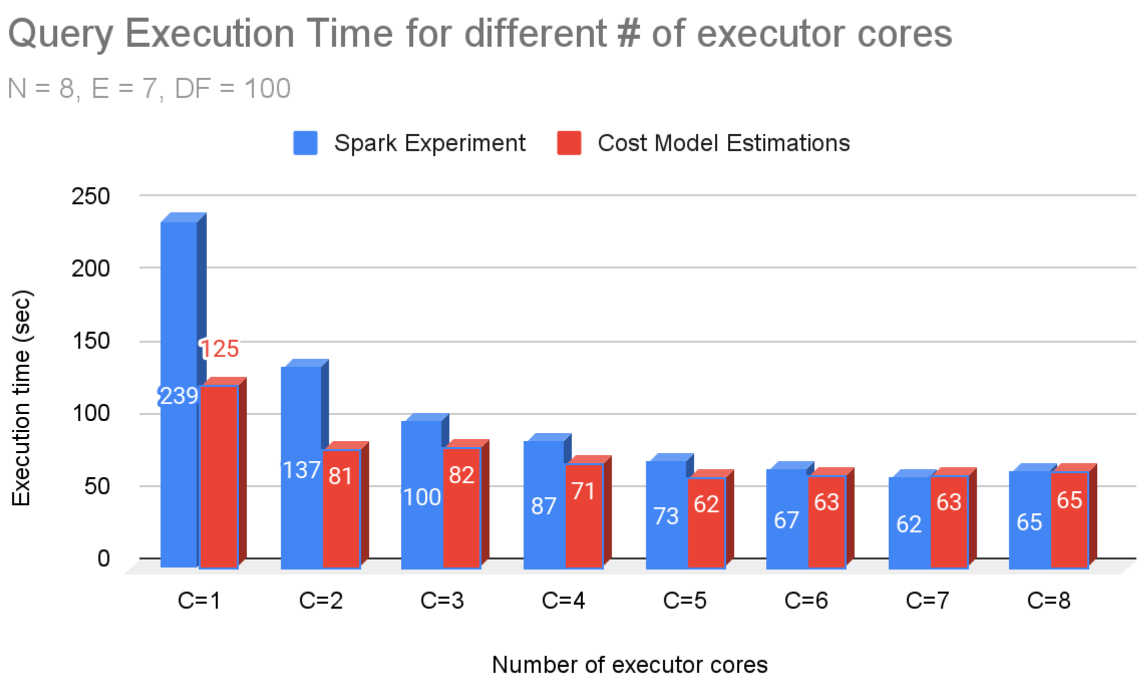
Εικόνα 6. Χρόνοι εκτέλεσης για διαφορετικού μεγέθους joins



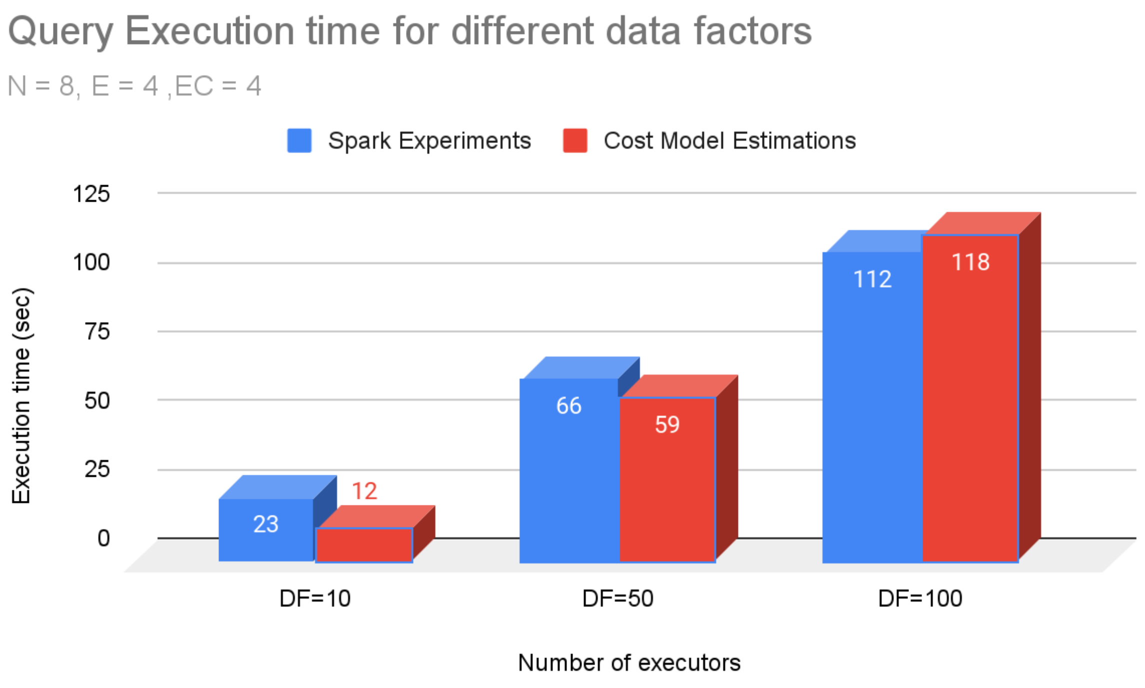
Εικόνα 7. Χρόνοι εκτέλεσης για διαφορετικά ερωτήματα.



Εικόνα 8. Χρόνοι εκτέλεσης για διαφορετικό αριθμό Spark Executors



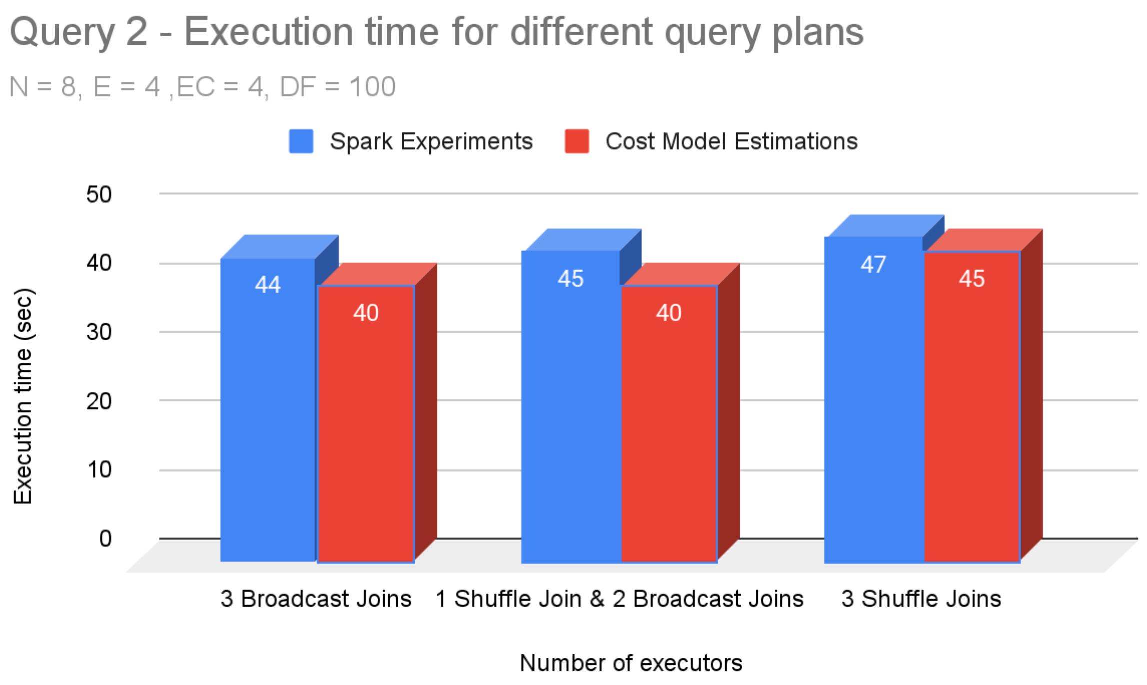
Εικόνα 9. Χρόνοι εκτέλεσης για διαφορετικό αριθμό Spark Executor cores



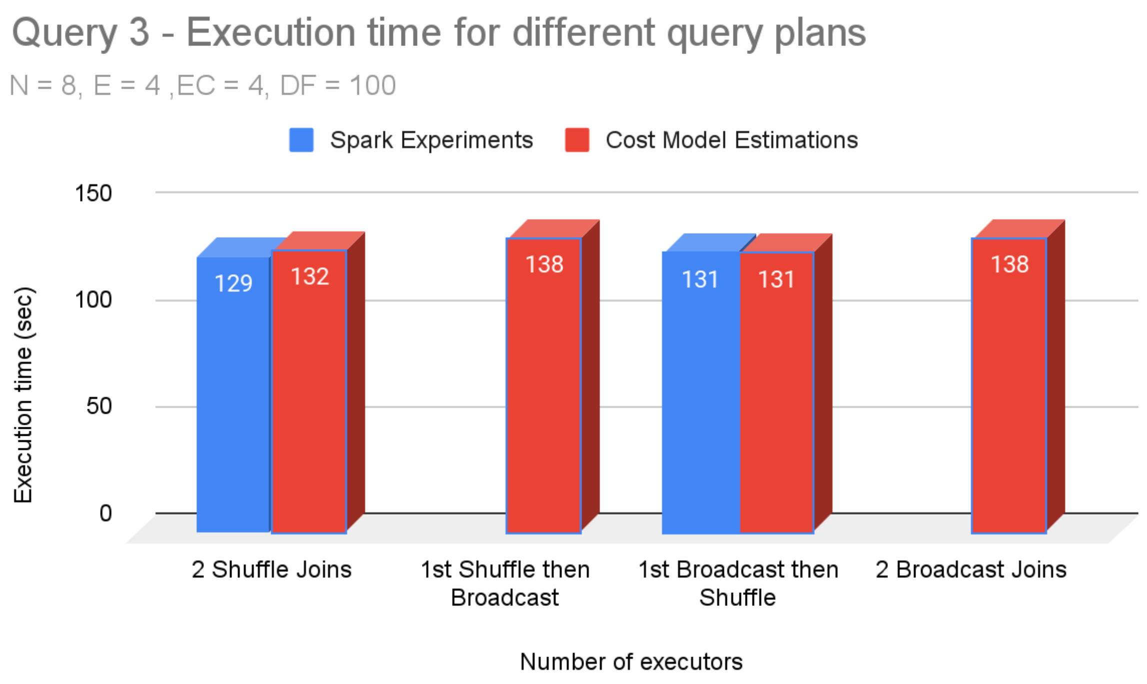
Εικόνα 10. Χρόνοι Εκτέλεσης για διαφορετικό μέγεθος dataset

Το αρχικό συμπέρασμα είναι πως το μοντέλο παρουσιάζει μία αξιοσημείωτη ακρίβεια, ενώ οι όποιες αποκλίσεις βρίσκονται μέσα στα όρια που έχει θέσει ο Baldacci (20%). Ανακρίβειες επίσης μπορεί να οφείλονται και σε στοχαστικότητα κάποιων παραγόντων, όπως πχ η ταχύτητα μετάδοσης στο δίκτυο. Επιπλέον, μελετώντας μεμονωμένα περιπτώσεις ανακρίβειας, παρατηρήσαμε πως το μοντέλο κοστολόγησης τείνει κάποιες φορές να κάνει άστοχες αποφάσεις όταν υπολογίζει τον χρόνο εκτέλεσης των broadcast joins, υπερεκτιμώντας τον χρόνο που χρειάζονται.

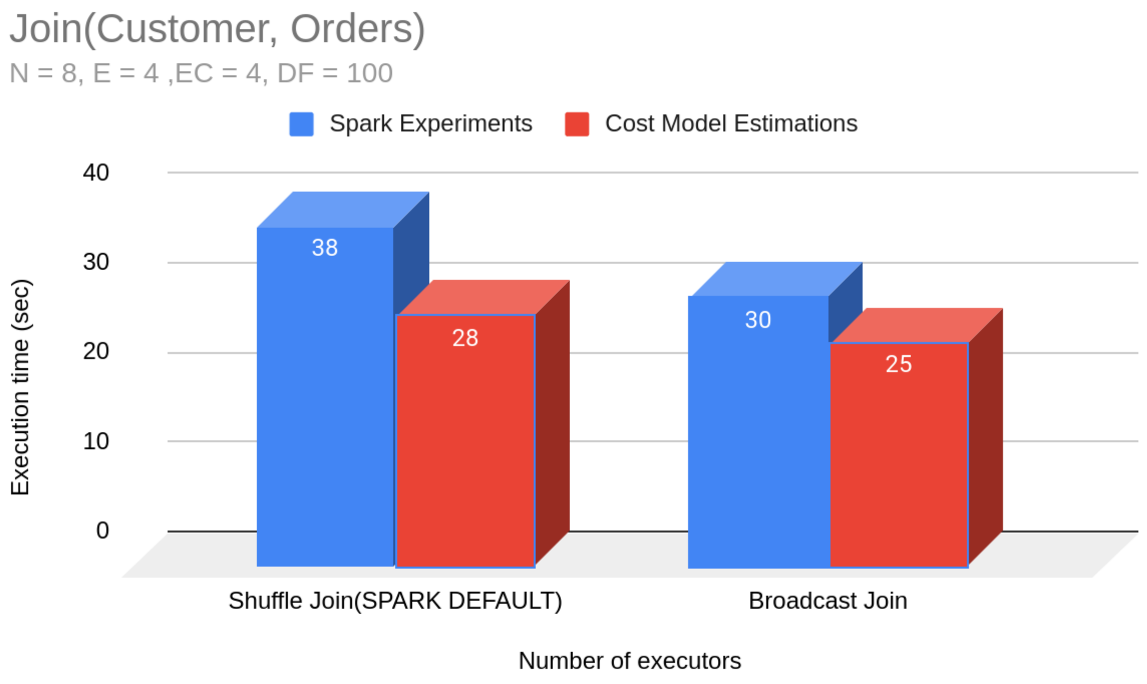
Στη συνέχεια επαληθεύσαμε την ακρίβεια των προβλέψεων, συγκρίνοντας διαφορετικά πλάνα από τα ίδια ερωτήματα.



Εικόνα 11. Διαφορετικά πλάνα εκτέλεσης για το Query 2.



Εικόνα 12. Διαφορετικά πλάνα εκτέλεσης για το query 3

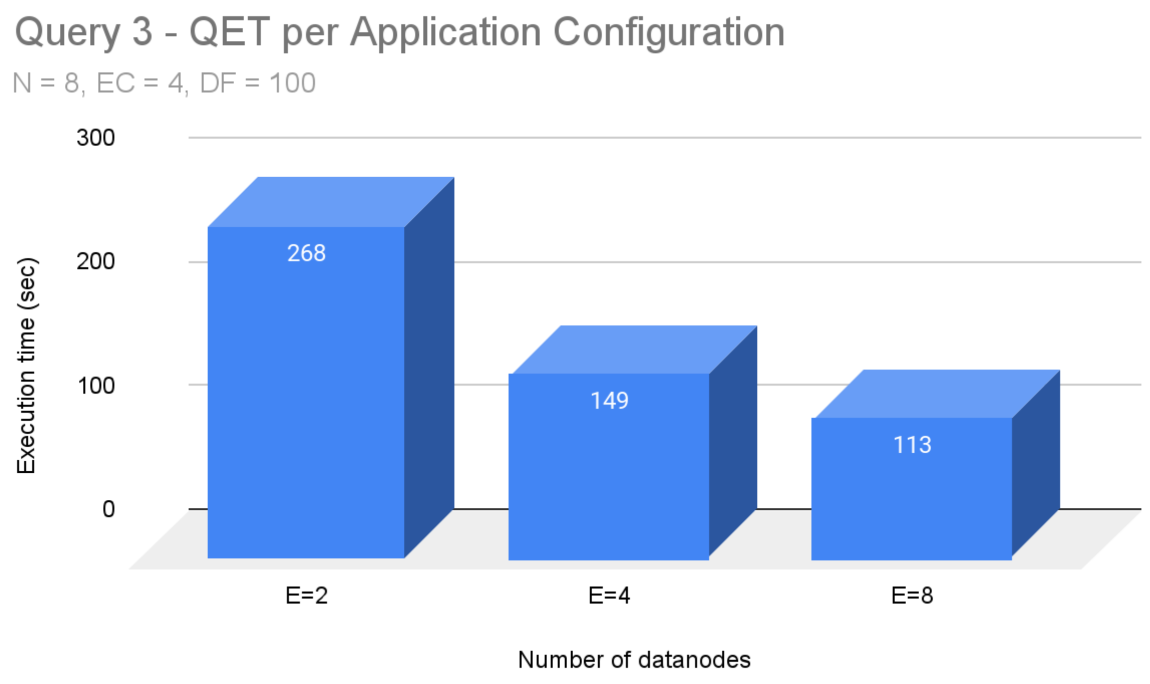


Εικόνα 13. Ο Catalyst κάνει λάθος.

Από τις δύο περιπτώσεις που δείχνουμε εδώ, το μοντέλο μας βρίσκει το βέλτιστο πλάνο στη μία και το χάνει οριακά στη δεύτερη. Παρ'όλα αυτά, είναι φανερό πως ακολουθεί τις τάσεις βελτιστότητας, και είναι σχετικά ακριβές. Στην εικόνα 13, βλέπουμε και μία ξεκάθαρη περίπτωση λάθος επιλογής του Catalyst, την οποία το μοντέλο μας μπορεί και προβλέπει επιτυχώς, προτείνοντας το σωστό πλάνο εκτέλεσης.

Χρηματικό κόστος

.

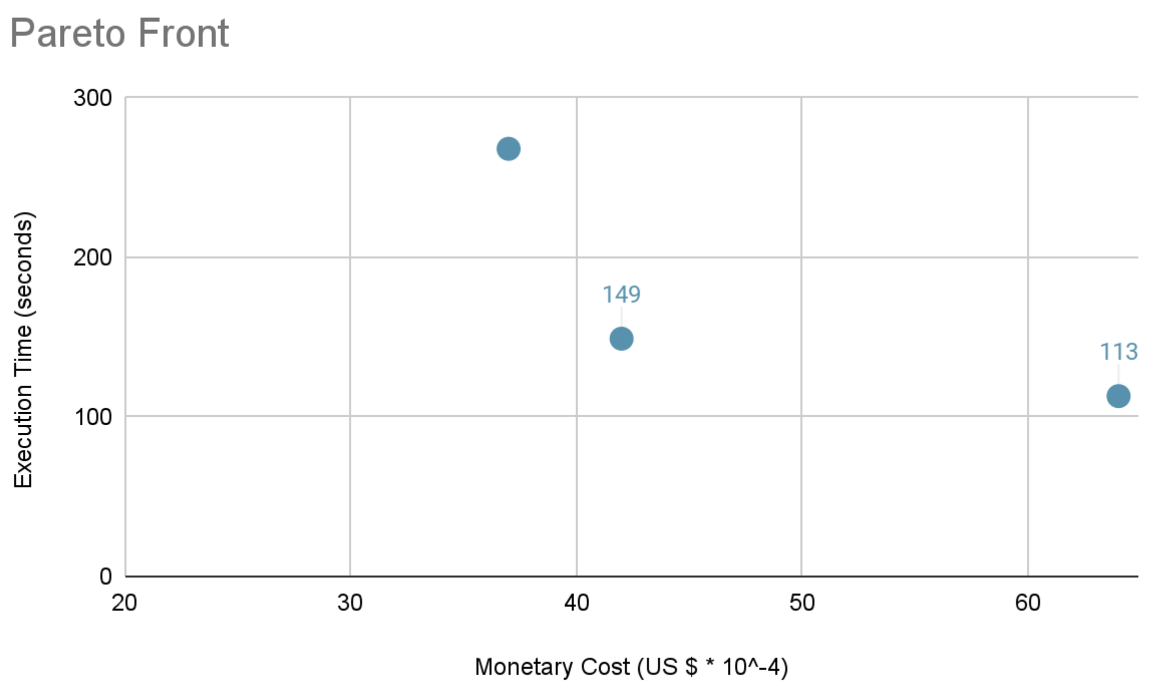


Εικόνα 14. Καλύτερος χρόνος εκτέλεσης για διαφορετικό αριθμό Spark Executors.

Στη συνέχεια, θα εισάγουμε το δεύτερο κριτήριο βελτιστοποίησης, το χρηματικό κόστος. Ως βάση τιμολόγησης επιλέγουμε τις τιμές του Amazon EC2. Ο τύπος που προτείνουμε και χρησιμοποιούμε είναι ο ακόλουθος:

Cost = E * QET (seconds) * cost($/hour)/3600

Όπου E είναι ο αριθμός των executors, και QET ο χρόνος εκτέλεσης του ερωτήματος. Με λίγα λόγια, στην περίπτωση μας τα κριτήρια έχουν σχέση αναλογίας. Χρησιμοποιώντας τον τύπο υπολογίζουμε το κόστος των πλάνων εκτέλεσης, και τα οπτικοποιούμε σε ένα Pareto front.



Εικόνα 15. Σύνολο βέλτιστων κατά Παρέτο πλάνων.

Όπως φαίνεται και από την εικόνα, έχουμε 3 διαφορετικές επιλογές πλάνων εκτέλεσης, το καθένα εκ των οποίων μας δίνει έναν διαφορετικό συμβιβασμό μεταξύ των δύο ζητουμένων. Συνεπώς, καταφέραμε να λύσουμε επιτυχώς ένα πρόβλημα βελτιστοποίησης ερωτήματος πολλαπλών κριτηρίων, σε μία παράλληλη πλατφόρμα όπως το Spark.

Υλοποίηση και επαλήθευση ΠΒΒ

Η τελευταία συνεισφορά της διπλωματικής είναι η εφαρμογή της παραμετρικής βελτιστοποίησης ερωτημάτων. Για να το πετύχουμε αυτό, ορίσαμε ως παραμέτρους την επιλεξιμότητα κάποιων από τα κατηγορήματα ενός ερωτήματος.

Οι διαφορές στην επαλήθευση που είχαμε από τον Trummer παρατίθενται στον πίνακα παρακάτω:

	Η δική μας υλοποίηση	Η υλοποίηση του Trummer	
Είδη ερωτημάτων	GPSJ(no UNION ALL, OUTER JOIN, EQUI JOINS)	Όλα	✗
Επιλογή ερωτημάτων	TPC-H Benchmark	Τυχαία γεννήτρια	✗
Υπολογισμός κόστους πλάνου	Άθροισμα υποπλάνων	Άθροισμα υποπλάνων	✓
Κριτήρια βελτιστοποίησης	Χρόνος εκτέλεσης, χρηματικό κόστος	Χρόνος εκτέλεσης, χρηματικό κόστος	✓

Παράμετροι	Επιλεξιμότητα κατηγορημάτων	Επιλεξιμότητα κατηγορημάτων	✓
Τρόποι πρόσβασης σε δεδομένα	1 (Πλήρης σάρωση πίνακα)	2(Πλήρης σάρωση, αναζήτηση ευρετηρίου)	✗
Διαθέσιμα είδη joins	2 (Shuffle Hash, Broadcast)	2(Single node, shuffle hash)	
Βάση κοστολόγησης	Amazon EC2 general purpose medium instance	Amazon EC2 general purpose medium instance	✓
Επιλογή συστήματος	Spark SQL + μοντέλο κοστολόγησης	Benchmark γραμμένο ως java application, με cost model	✗

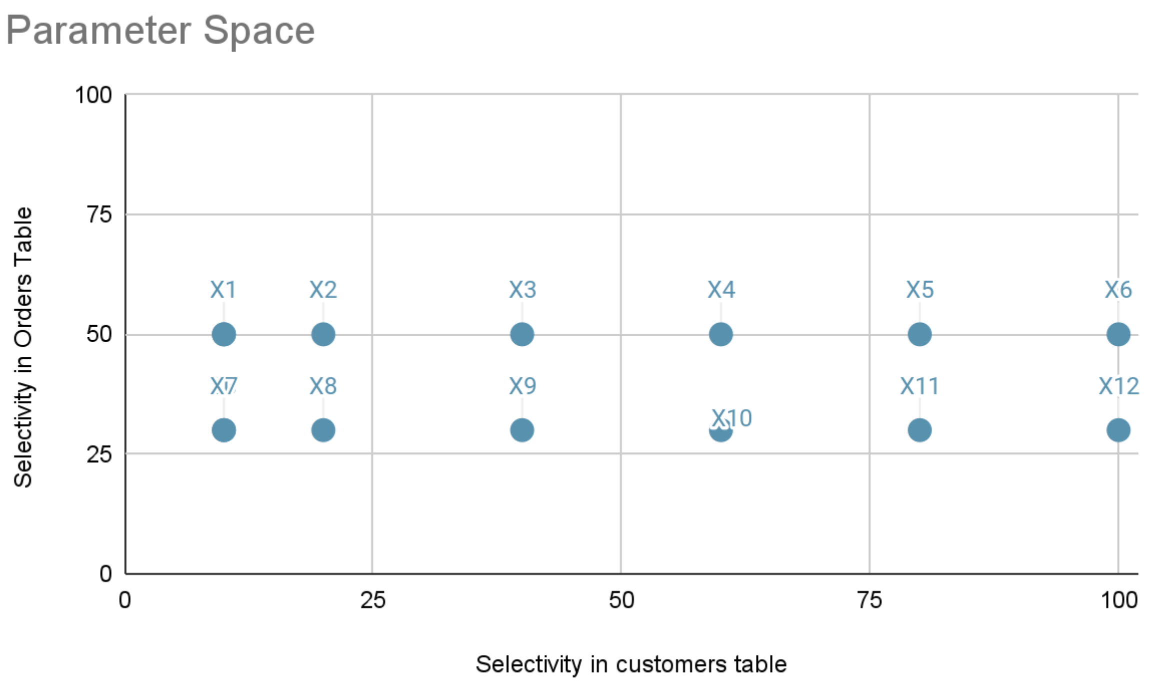
Πίνακας 2. Διαφορές υλοποίησης

Επιλέξαμε να παρουσιάσουμε την βελτιστοποίηση του ακόλουθου ερωτήματος, όπου με γαλάζιο και κίτρινο είναι τα κατηγορήματα των οποίων η επιλεξιμότητα είναι οι παράμετροι μας.

TPC-H QUERY 3
SELECT l_orderkey, o_orderdate, o_shippriority, sum(l_extprice)
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < date'1995-03-15' AND l_shipdate > date'1995-03-15' AND c_custkey <15000000
GROUP BY l_orderkey, o_orderdate, o_shippriority

Ο χώρος παραμέτρων που θα χρησιμοποιήσουμε είναι ο P1 = [0.1,0.2,0.4,0.6,0.8,1], P2 = [0.5,0.3], όπου P1 είναι το κίτρινο κατηγόρημα και P2 το γαλάζιο. Θα λάβουμε υπόψη όλα τα σχετικά πλάνα εκτέλεσης, για 3 διαφορετικές ρυθμίσεις του Spark, με 2, 4, και 8 executors.

Μία ψηφιακή απεικόνιση του χώρου των παραμέτρων μας είναι η ακόλουθη.



Εικόνα 16. Χώρος Παραμέτρων

Για την περίπτωση του ερωτήματος που παρουσιάσαμε, υπάρχουν 4 διαφορετικοί συνδυασμοί joins (η σειρά που πρέπει να γίνουν τα 2 joins είναι προφανής οπότε λαμβάνουμε υπόψη μόνο το είδος του join).

PL1: Join Types -> Shuffle Join, Shuffle Join

PL2: Join Types -> Broadcast Join, Broadcast Join

PL3: Join Types -> Shuffle Join, Broadcast Join

PL4: Join Types -> Broadcast Join, Shuffle Join

Για τις 3 διαφορετικές ρυθμίσεις του Spark, θα δούμε τις εκτιμήσεις του μοντέλου κοστολόγησης μας, για καθένα εκ των 4 αυτών πλάνων, σε όλο τον χώρο των παραμέτρων μας.

E=2, EC=4

P1/P2	0.3				0.5			
PLAN	PL1	PL2	PL3	PL4	PL1	PL2	PL3	PL4
0.1	210.5	195.9	198.4	208.0	211.9	196.2	199.8	208.2
0.2	211.1	196.5	199.0	208.6	212.9	197.7	201.3	209.2
0.4	212.4	199.0	201.6	210.0	214.6	202.0	205.7	211.1
0.6	213.7	202.3	204.8	211.2	216.6	208.7	212.0	213.4
0.8	215.0	206.9	209.1	212.9	218.5	217.3	220.3	215.8
1.0	216.4	212.2	214.2	214.4	220.4	227.8	230.3	218.4

E=4, EC=4

P1/P2	0.3				0.5			
PLAN	PL1	PL2	PL3	PL4	PL1	PL2	PL3	PL4
0.1	127.5	122.7	123.5	126.8	128.0	122.9	124.1	126.9
0.2	127.9	123.0	123.9	127.1	128.5	123.5	124.7	127.3
0.4	128.5	124.2	125.1	127.7	129.4	125.6	126.8	128.2

0.6	129.0	125.8	126.6	128.2	130.2	128.7	129.8	129.2
0.8	129.7	127.9	128.6	129.0	131.1	132.8	133.7	130.3
1.0	130.3	130.4	131.0	129.7	132.0	137.8	138.4	131.6

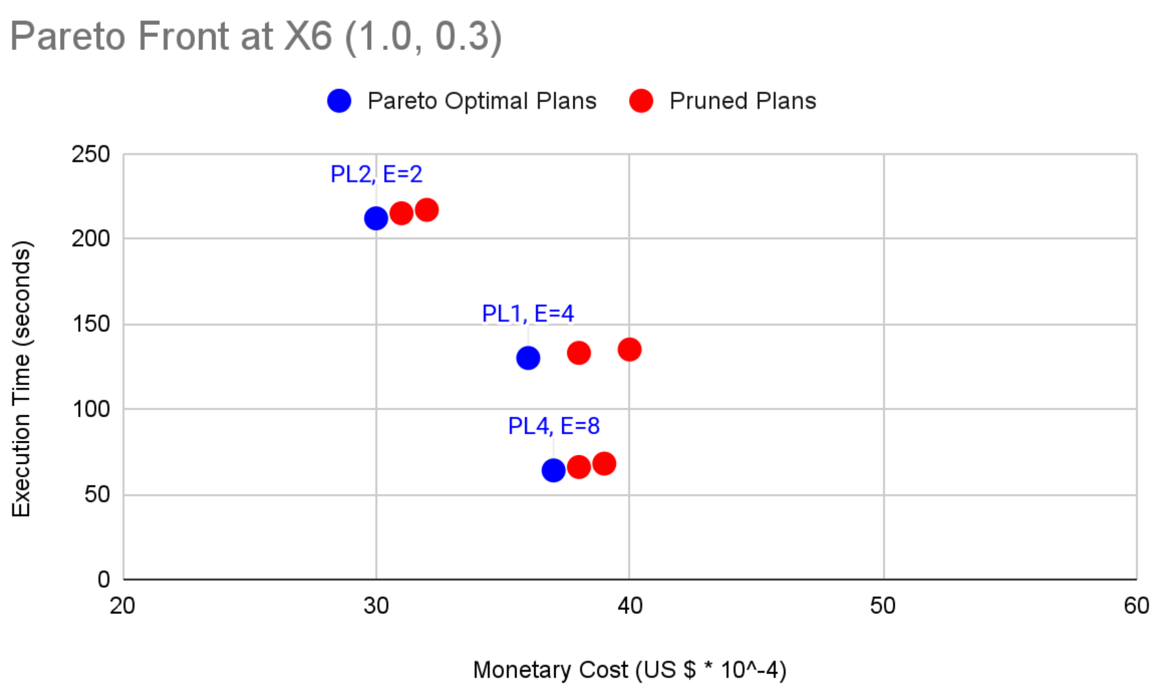
E = 8 EC = 4

P1/P2	0.3				0.5			
PLAN	PL1	PL2	PL3	PL4	PL1	PL2	PL3	PL4
0.1	62.6	61.4	61.6	62.3	62.7	61.4	61.7	62.4
0.2	62.7	61.5	61.7	62.5	62.9	61.8	62.1	62.6
0.4	63.0	62.1	62.3	62.8	63.3	62.7	63.0	63.0
0.6	63.3	62.8	63.0	63.1	63.8	64.2	64.4	63.5
0.8	63.6	63.8	64.0	63.4	64.1	66.1	66.2	64.0
1.0	63.8	65.0	65.1	63.8	64.5	68.5	68.6	64.6

Πίνακας 3. Χρόνος εκτέλεσης για διαφορετικά πλάνα και διαφορετικές ρυθμίσεις Spark

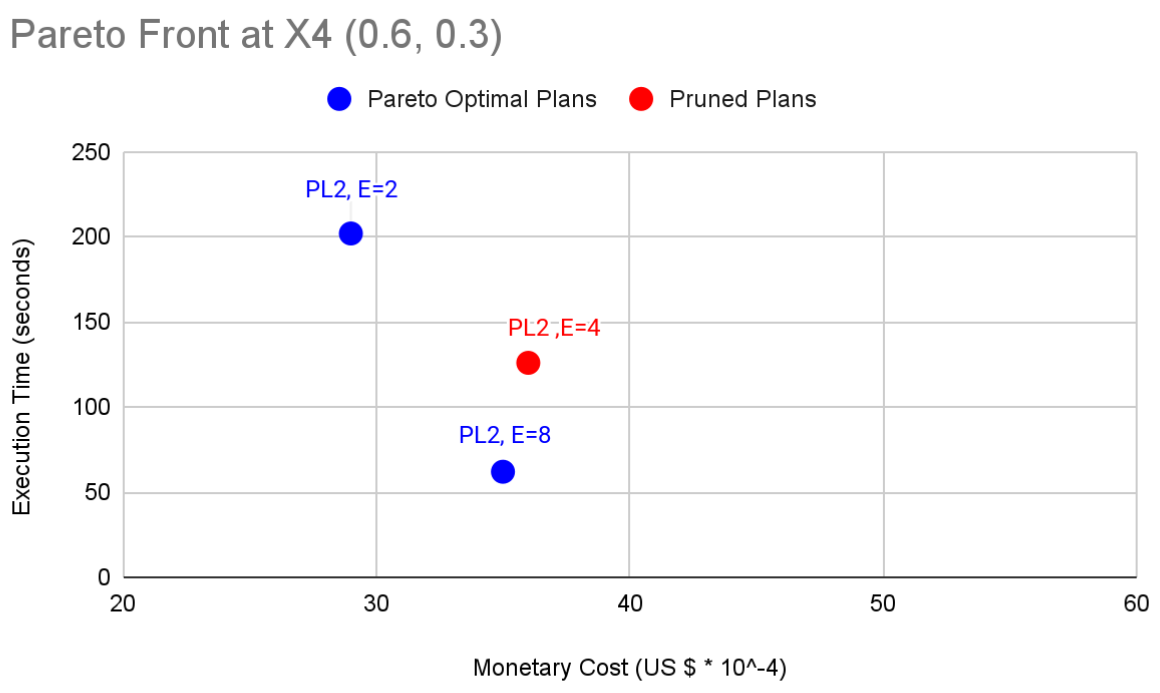
Όπως φαίνεται, η συμπεριφορά της Spark SQL μεταφέρεται επιτυχώς στο μοντέλο κοστολόγησης. Όσο μεγαλώνει η επιλεξιμότητα ενός κατηγορήματος, τα πλάνα με shuffle join τείνουν να προτιμώνται από αυτά με broadcast join, που είναι ένα ιδανικό join μόνο όταν ο ένας εκ των 2 πινάκων που ενώνεται είναι πολύ μικρός.

Είναι φανερό πως κάθε συνδυασμός παραμέτρων οδηγεί σε διαφορετικούς συμβιβασμούς όσον αφορά την εκτέλεση του ερωτήματος. Ακολουθούν 2 Pareto Fronts από τον χώρο των παραμέτρων μας.



Εικόνα 17. Pareto front στο X6

Ο συνδυασμός X6 είναι μία ενδιαφέρουσα περίπτωση, καθώς είναι η μοναδική από τους συνδυασμούς παραμέτρων όπου τα 3 βέλτιστα πλάνα εκτέλεσης, είναι όλα και διαφορετικά φυσικά πλάνα εκτέλεσης, δηλαδή περιλαμβάνουν διαφορετικούς συνδυασμούς joins.



Εικόνα 18. Pareto front στο X4

Σε αυτή την περίπτωση, ένα από τα 3 βέλτιστα πλάνα (για κάθε ρύθμιση του Spark), είναι χειρότερο από ένα άλλο τόσο όσον αφορά το χρηματικό κόστος, όσο και όσον αφορά το χρόνο εκτέλεσης. Ως αποτέλεσμα, δεν συμπεριλαμβάνεται στο σύνολο των βέλτιστων κατά Pareto πλάνων (το συμπεριλάβαμε στην εικόνα με κόκκινο).

Συμπεράσματα

Ένα προφανές συμπέρασμα από αυτό το σύνολο πειραμάτων είναι ότι παράμετροι όπως η επιλεξιμότητα κατηγορημάτων μπορεί να έχουν μεγάλη επιρροή όσον αφορά το ποιο πλάνο εκτέλεσης είναι βέλτιστο. Από εκεί και πέρα, δείξαμε πως η συγκεκριμένη τεχνική βελτιστοποίησης, η οποία ως τώρα είχε επαληθευτεί μόνο σε σχεσιακές βάσεις δεδομένων, μπορεί να επεκταθεί και να χρησιμοποιηθεί με επιτυχία σε πλατφόρμες παράλληλης επεξεργασίας όπως το Spark, βοηθώντας τον χρήστη να επωφεληθεί από σημαντική εξοικονόμηση χρόνου, μιας και η διαδικασία της βελτιστοποίησης πραγματοποιείται πριν την εκτέλεση κάποιου ερωτήματος.

Chapter 1

1.Introduction

1.1 Aim of thesis

The aim of this thesis is the proposal of a query optimizer operating on a cloud-like architecture, for efficient and multi-objective query optimization.

The query optimization problem is tackled and techniques for solving the problem are discussed and evaluated. The proposed architecture for the query optimizer was based on tools of the Hadoop ecosystem for data storage and processing, alongside the Grid ‘5000 platform for deployment of computing and storage resources, which enabled us to operate numerous complex and time-consuming experiments in the form of SQL queries. The system was used and evaluated with large scale experiments, alongside implementation of components of the query optimizer like its cost model, and multiple optimization techniques.

1.2 Problem definition

As implied by the title, the optimization problem discussed in this thesis is query optimization. More specifically, query optimization is the process of selecting an optimal way to execute a query based on the available operators that can be applied on database data. We tackled query optimization also with regards to the available computing resources of a cloud environment, the amount and size of which can have a huge impact on our optimization objectives. Query optimization can become a very challenging optimization problem when we deal with more complex queries than, for example, typical SELECT attribute FROM TABLE table statements. Operations like table joins, aggregations and generalized projections can result in an almost infinite number of available query execution plans, making it difficult for the query optimizer to find the optimal plan and very time-consuming to compare all available plans.

As our problem setting involves a Cloud environment, we adapt to its features and state that query optimization in the cloud cannot be solved as a single-objective problem, as the pricing of each platform makes monetary cost an equally important optimization goal in most cases. Therefore, we argue that query optimization in the cloud should be at least bi-objective.

In short, we define our optimization problem as multi-objective query optimization in the cloud.

1.3 Motivation

The motivation of this thesis came from reading state of the art works on query optimization and multi-objective optimization problems in general, as well as actually using data processing frameworks like Apache Spark. Reading and experimenting led me to try to answer and explore research questions, like the following:

- How can a query be efficiently and accurately optimized in a Cloud environment?
- How can query optimization be improved in a system like Apache Spark?
- How can a query optimizer cost model be implemented and evaluated?
- From how many different perspectives can a query be optimized in such an environment?
- What parameters of a query have a high influence when it comes to query execution time and other optimization objectives?
- What are the challenges of switching from a cloud to a serverless environment and how does this affect query optimization?

1.4 Thesis Text Structure

The text of the thesis is organized as follows. Chapter 2 introduces the cloud computing landscape, with special reference to serverless architecture. Chapter 3 gives an overview of the tools used for big data management in a Cloud environment, many of which are used in our contribution, like Apache Spark and HDFS. Then, query optimization which is the optimization problem of the thesis is introduced and discussed and the text follows with a link between query optimization and cloud environments, by discussing optimization problems that occur in the cloud. An overview of problems and state of the art solutions is given, and it is argued that the nature of the cloud implies that query optimization in such an environment should be multi-objective.

After describing the state of the art in the research domain, the next chapter includes our vision for a flexible multi-objective query optimizer operating in the Cloud, followed by our proposal of a baseline system architecture which will be used in our experiments to provide proof of concept for our contributions, which are also described.

One of the main contributions of this thesis is the reconstruction of a proposed cost model for Spark SQL [3], and Chapter 8 is dedicated to the description of this implementation. Chapter 9 includes the large-scale experimental evaluation we conducted for the cost model, and also introduces a second objective to our optimization process, which is monetary cost. At that point, we managed to solve multi-objective query optimization problems in a cloud-like environment. The next chapter goes one step further to apply multi-objective parametric query optimization on selected queries, which is a novel technique proposed in 2014 [1], that had not been evaluated in such an environment before.

After presenting and discussing all our experimental results, future research routes and contributions are discussed based on the thesis outcome, and general conclusions are produced.

Chapter 2

2.Cloud Computing & Serverless

2.1 Cloud computing

2.1.1 Introduction

Cloud computing refers to the on demand delivery of computing services through the internet. These services consist of tools, applications and resources. The ability to rent computing resources like servers, storage, databases or networking enables individuals and organizations to concentrate on more essential tasks of their work and minimize up-front IT infrastructure costs, as computation and storage responsibilities are abstracted from their local terminals.

The term cloud was used as a metaphor for the internet, implying that the specifics of how the endpoints of a network are linked are not necessary to understand the diagram. The cloud metaphor was first used for virtualized services by Andy Hertzfeld in 1994 to describe the universe of "places" that mobile agents in the Telescript environment could go: "The beauty of Telescript," says Andy, "is that now, instead of just having a device to program, we now have the entire Cloud out there, where a single program can go and travel to many different sources of information and create a sort of a virtual service.

Cloud computing technologies emerged thanks to the evolution of virtualization, which is the logical division of resources on physical servers across virtual machines (VMs) [5]. The first cloud computing platforms, like Amazon EC2 [6] and Google Cloud Platform [7], leveraged virtualization and used VMs to share large clusters of servers across their clients. Virtualization provides cloud systems with the agility required to speed up IT operations and reduce cost, by increasing utilization of resources. Cloud computing resources (hardware and software) are distributed and stored in multiple locations, each one being a data center.

2.1.2 Characteristics

Cloud computing offers some exciting new aspects and features. The National Institute of Standards and Technology's definition of cloud computing summed up its essential characteristics in the following five: [4]

- On-demand self-service: Cloud users can provision their services automatically and on demand. As a result, there is no need for users to plan far ahead for provisioning, who can instantly rent and lease the services they want.

- Broad network access: The capabilities the cloud offers are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).

- Resource pooling: The provider's computing resources are pooled to serve different consumer needs using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand.

- Rapid elasticity: Elasticity is one of the most important cloud characteristics, as resources can be elastically provisioned and released, in some cases automatically, to scale rapidly up and down to meet workload demands.

- Measured service: Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

In addition to these features, the cloud can provide significant economic benefits to its users, due to a pricing policy commonly applied:

- Pay for use pricing: The ability to pay for the used resources on a short-term basis (e.g. processors by the hour, cloud storage by the day) and release them when they are no longer needed, means that users can have significant cost savings, and providers can optimize the scheduling of their resources.

The elasticity the cloud offers, alongside its pricing policy make it a very attractive choice for companies and individuals, as they are freed from the task of buying and maintaining the resources that they will need to use, need not to worry about overprovisioning or underprovisioning of resources, and can also securely manage their data which can be stored remotely and securely alongside data centers.

2.1.3 Service Models

Cloud providers offer their "services" according to different models, which offer varying levels of abstraction. The three most common service models are the following:

- Infrastructure as a service (IaaS): IaaS provides the same capabilities as data centers (except for their actual maintenance). Users can rent and manage data center infrastructures like compute and storage services in the form of virtual machines (VMs). IaaS provides the virtualization, storage, network, and servers needed, while users are responsible for managing applications, data, runtime, middleware and operating system.

- Platform as a service (PaaS): PaaS provides hardware and software tools for the user to build and manage a customized application. It is mostly targeted for developers who want to focus on their application without having to provision the operating system or the PaaS software itself.

- Software as a service (SaaS): SaaS delivers web applications to its users, which are fully managed by the SaaS vendor. Users typically access the application through an API and a web browser, and do not need to install it locally.

Figure 1 highlights the different levels of abstractions each service model offers, showing with light blue the responsibilities of the user in each service model, and with dark blue what is managed by the cloud provider.

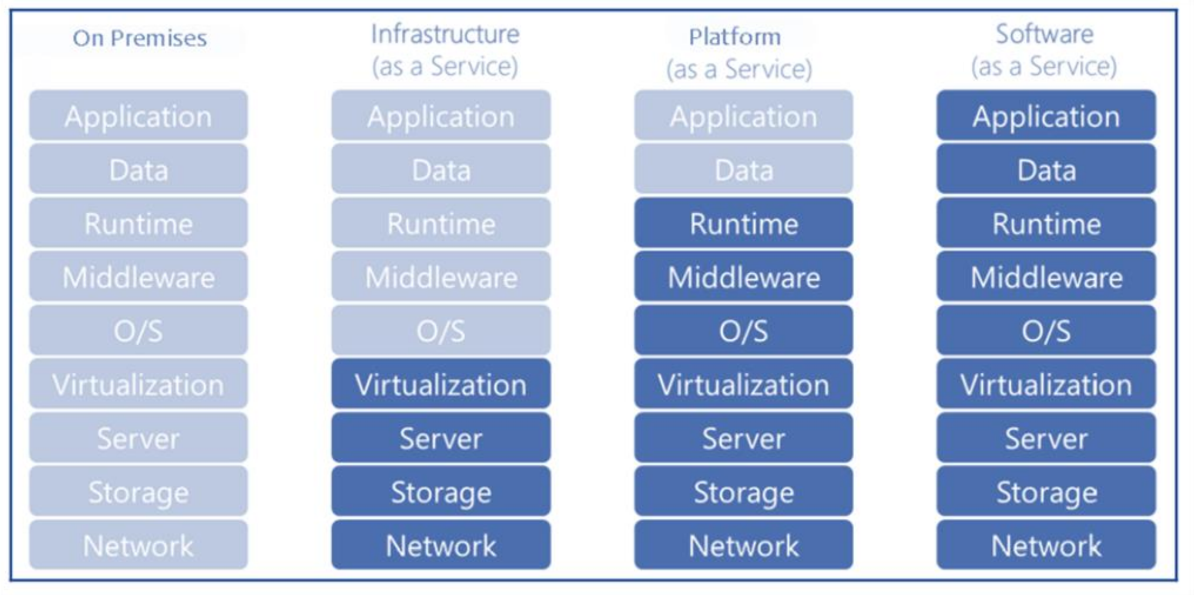


Figure 1. Cloud service models [113]

2.1.4 Cloud Computing Types

There are three main cloud computing deployment types:

- Private clouds: private clouds are internal data centers, of businesses or other organizations not available to the general public and large enough to be able to benefit from the cloud computing advantages discussed in this thesis. It is a popular choice among big businesses as it offers enhanced security and increased control.

- Public clouds: public clouds are the ones available to the general public in a pay for use or even in a free of charge manner, with the service sold being referred to as utility computing. Security concerns are higher in public clouds, as services are shared by multiple customers. The elasticity and the pay for use pricing model make it ideal for small and medium businesses, as well as for individual users. Today, the most notable public cloud providers are the big tech companies, with platforms such as Amazon Web Services (AWS), Microsoft Azure, Google Cloud and IBM Cloud dominating the cloud market.

- Hybrid clouds: hybrid clouds are a mixture of a public cloud and a private environment, offering the benefits of both deployment models.

Other cloud deployment types worth mentioning are community clouds and virtual private clouds:

- Community cloud shares infrastructure between several organizations from a specific community with common concerns and interests (security, jurisdiction, etc.). The costs are spread over fewer users than a public cloud (but more than a private cloud), so the full cost savings potential of the cloud is not realized.
- A virtual private cloud (VPC) is an on-demand configurable pool of shared resources allocated within a public cloud environment. Although located inside a public cloud environment, it provides isolation between the different users of the platform and its resources.

Figure 2. Cloud Deployment Types

2.1.5 Opportunities and use cases

Cloud computing development offers a lot of opportunities. It has resulted in increased parallel processing, scalability, accessibility, high level of security and also integration with heterogeneous data storages. It has resulted in significant cost savings in hardware, storage facilities or other utilities.

Cloud scales on-demand to support all types of workloads, providing resources and services for uninterrupted data management and making it a useful tool for storage and mining of big data [2]. Apart from data analytics, it is also widely used for storage and backup by a lot of different industries (medical, finance) [41,42], and its data replication factor makes it ideal for disaster recovery.

Apart from its applications in the industry and the database community, cloud computing is omnipresent in our everyday lives. It is used for all kinds of activities, from safe storage of photos in Dropbox, to music streaming in Spotify, writing code and training machine learning models in Google Colab or processing text and photos on an online editor.

2.2 Serverless Computing

2.2.1 Introduction

Recently, the cloud evolved with the emergence of serverless computing platforms. Serverless computing offerings include platforms in the cloud, where developers can simply upload their code, and the platform executes it for them at any scale. Developers are not concerned about provisioning, or operating a real server (where the term "serverless" originates), and the fact that they are charged only for computing resources used when their code is invoked, enables them to benefit even more from fine-grained billing, as they will never have to pay for idle resources.

With serverless, virtualization moved one step ahead. After VMs virtualized physical hardware and containers virtualized the operating system itself, serverless platforms virtualized the runtime or the process itself, allowing multiple isolated functions (or lambdas) to share the same runtime. Figure 3 illustrates these varying virtualization abstraction levels.

Serverless started to attract more attention when Amazon launched AWS Lambda in 2014 [26] Since then, interest is growing. A notable proof of the recent serverless popularity came from Google search trends, which showed that queries for the term "serverless" recently surpassed the historic peak of popularity of the phrase "MapReduce". [14]

Today, most public cloud vendors have already introduced serverless. Significant examples are the platforms of big-tech companies, like AWS Lambda [26],Microsoft Azure Cloud Functions [28] and Google Cloud Functions [27]. Furthermore, promising open source serverless platforms [29]-[32] have emerged, tackling numerous serverless limitations and research problems. Their performance is evaluated in an overview work [11]. These platforms shape the current serverless landscape, with the future looking promising and away from servers [5],[14],[15]. Its advantages will be further discussed in $2.2.3.

However, current serverless offerings present significant limitations that do not allow the full potential of serverless to be reached. These limitations are discussed in section $2.2.4, with section $2.2.5 providing an overview of recent research works that aim to provide workarounds to these limitations.

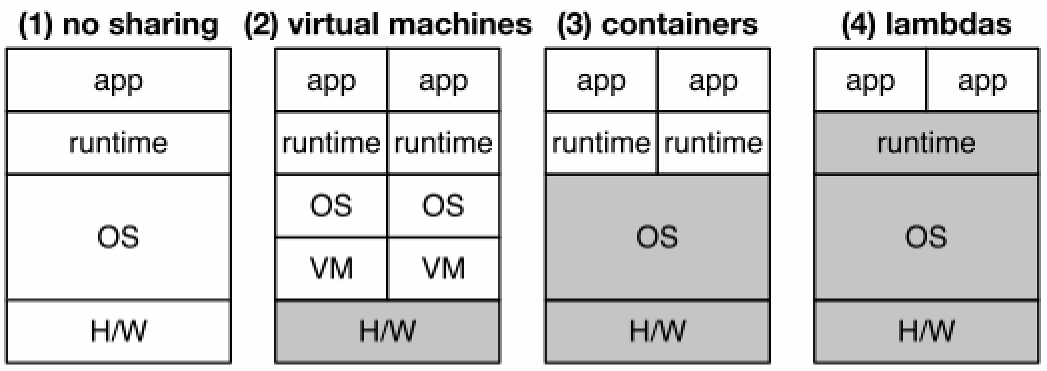


Figure 3. Evolution of virtualization

2.2.2 Service Models

The current serverless ecosystem consists of Function as a service (FaaS) and backend as a service (BaaS) platforms.

- Function as a service (FaaS): FaaS platforms are code execution environments where developers write applications in the form of functions and execute them using cloud compute and storage resources. Programmers can write in high-level languages such as Python or Java. These functions are stateless and its execution time is of limited duration, often a few minutes.

Users are charged at fine-grained time units, typically per hundreds of milliseconds of code execution.

- Backend as a service (BaaS): The term serverless has become synonymous with FaaS platforms today, however FaaS platforms are only one part of the serverless landscape, the other half being BaaS. In comparison to FaaS which aims to provide a general-purpose code execution environment, BaaS platforms often serve specific use cases. They consist of BaaS storage platforms, database platforms and compute platforms. Storage platforms allow users to scale their storage, like Amazon S3 [33] (which predates FaaS platforms), and bill them only for the space they use and the amount of reads and writes they perform. As FaaS platforms are stateless, it is common to use BaaS to store state in a serverless environment. BaaS database platforms ease structured data storage and provide richer query semantics. Compute platforms include Amazon Athena [34], Google BigQuery [35] and Azure Stream Analytics [36] which are tailored for analytic workloads

In this chapter we will mainly discuss FaaS. The following sections describe the advantages and disadvantages of FaaS

2.2.3 Advantages and use cases

Numerous contributions have evaluated FaaS naming its main pros and cons. [5,14,15,16,17]. Its main advantages are the following:

- Easy to use: with FaaS platforms, programmers need not worry about anything else than their code, with management issues handled by the cloud provider

- Cost benefits: the cost savings for its users can be significant thanks to fine-grained billing. Users will only pay for the resources they use and the duration that they use them, in contrast to the server-centric model where users reserve resources regardless of whether they will use them.

- Demand-driven execution: serverless platforms are constantly allocating and deallocating resources for an application based on its workload demands. Thus, the task of scaling of resources does not concern the user anymore, while the provider can more efficiently provide and share his/her resources among its users.

A scenario illustrative of FaaS advantages is one of a web application (common serverless use case) written in a FaaS platform, where a website contains an interactive element linked with a backend API call that triggers a function in the backend. The operation of such an application will be entirely event-driven, with resources being allocated only when a user interacts with the element and a function needs them to execute. If the website has a lot of traffic more resources will be allocated automatically to meet the execution needs, and if it has zero traffic no resources will be allocated (autoscaling). The pay-for-use policy will mean that a FaaS user will never sustain and pay for idle resources.

In a 2018 survey in the serverless community, the popularity of different serverless use cases was analysed. As expected, serving of backend APIs in web applications came out on top, while data processing operations like ETL is the second most popular use case.

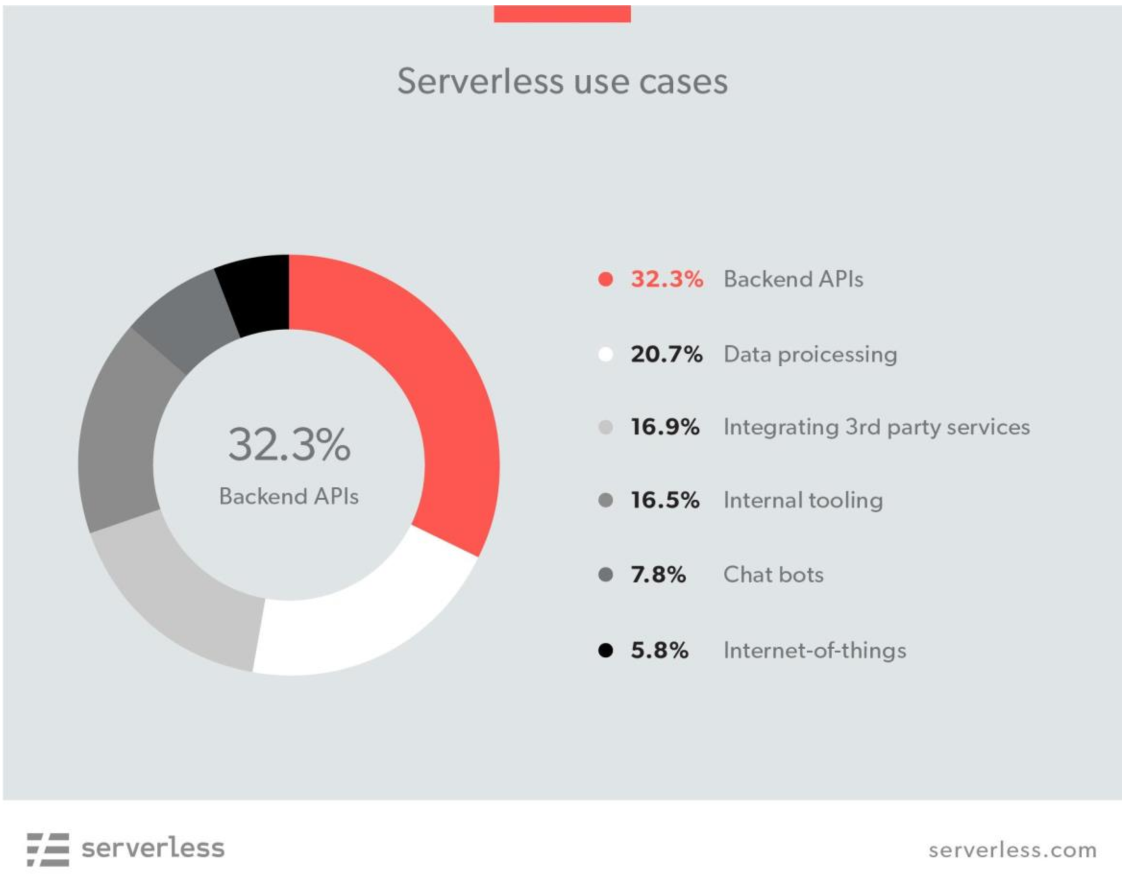


Figure 4. Popular serverless use cases

2.2.4 Limitations

The advantages and the prospects of serverless might be significant, however today's FaaS offerings come with significant limitations, which do not allow it to reach its rich potential.Its main limitations are the following:

- Short function lifespan: Functions have a very short lifespan and after they terminate, their state is lost. In AWS Lambda, this limit is 15 minutes.
- Function communication: Perhaps the more significant FaaS limitation is that functions cannot directly communicate with each other. As a result, exchange of state and intermediate data is not an easy task.
- Cold starts: Instantly starting many functions comes with a significant startup delay
- Resource Heterogeneity: Serverless lacks resource heterogeneity. FaaS offerings today only allow users to provision a time slice of a CPU hyperthread and some amount of RAM. There is no API or mechanism to access specialized hardware like GPUs, FPGAs etc.
- Security Issues: Security is also an open issue in serverless. Existing platforms present some vulnerabilities due to increased co-residency, as well as higher leak probability because of

increased network communication. This happens because data is disaggregated from functions and stored in a serverless data store.

A research team from UC Berkeley [14] states that these limitations mean that at the moment, FaaS is attractive only for a limited number of tasks such as massively parallel applications which exploit its autoscaling character and its functions need not communicate with each other, function orchestration and interactive data analytics.

2.2.5 FaaS v IaaS

The concept of the FaaS service model sounds more similar to IaaS from the 3 traditional cloud service models, but there is a key difference. In IaaS, the developer selects resources from a pool of available options, rents them for the time he needs them, and if his needs change during the process, he can rent more, or release some of his resources. In FaaS, the cloud provider takes care of this process. FaaS is autoscaling, and has a pay-for-what-you-use charging policy.

Serverless might be gaining ground, however at the moment the areas where FaaS performs better and overcomes traditional, VM-based IaaS are still limited. In the Lambada contribution [23], Muller et al. compare IaaS and FaaS in a Data Analytics task, highlighting the cost gain by sporadic resource use alongside the ability to service analytic queries, as the advantages of FaaS over Iaas. They go on to pinpoint interactive queries on cold data as a great fit for FaaS. Jiang et al. [38] implemented LambdaML, a platform enabling a fair comparison between FaaS and IaaS on ML training tasks, concluding that serverless is the best choice for models with reduced communication that quickly converge. They conclude that although FaaS can prove much faster than IaaS, it can never be much cheaper.

2.2.6 Existing Work

The significant limitations of serverless has led to a lot of research work, aiming to overcome these limitations with some promising contributions.

Communication between functions has been perhaps the hottest issue from the above. An obvious alternative involves communication through BaaS storage like Amazon S3 [23][25], which will come with a higher latency without heuristics, and also raises some security issues. Other works include an ephemeral data layer [20], like Pocket [21], which presents an ephemeral data store that automatically scales to meet the demands of serverless applications. This was achieved by the use of Apache Crail [39], an I/O architecture tailored to best integrate fast networking and storage hardware into distributed data processing platforms, taking full advantage of modern hardware capabilities. Another route was followed by Boxer [22], which enabled direct function-to-function communication in existing public cloud infrastructure, through a conventional TCP/IP network stack.

Cloudburst [24] is a completely new FaaS platform that achieves logical disaggregation and physical colocation of computation and state, and in contrast with other works, allows programs written in traditional programming languages to keep a consistent state across function compositions. Cloudburst achieves this via a combination of an autoscaling key-value store (providing state sharing and overlay routing) and mutable caches co-located with function executors (providing data locality). The system is built on top of Anna [40], a low latency autoscaling key-value store implemented by the same research team (UC Berkeley).

Lambada [23] presented an interesting workaround for the "cold start" problem, by parallelizing the worker invocation process. They achieved this by implementing a structure where after the first set of workers is invoked, a second generation follows as each worker invokes a number of other workers, until the scaling process is completed. That way startup latency is significantly reduced.

2.3 Conclusion

In this chapter, an overview of the cloud and serverless computing landscape was presented, with the pros and cons of each, and discussion on state of the art platforms and works on both systems. In the next chapter, we will explain why the cloud can be a good fit for big data handling, and introduce tools and frameworks used.

Chapter 3

3.Big Data Processing in the Cloud

3.1 Introduction

In the previous chapter, we mentioned that the Cloud can be a useful tool for storage and processing of big data. In this chapter we are going to introduce big data and present some state of the art frameworks that are used for its storage and processing. The distributed data storage and large scale parallel processing that can be achieved with these frameworks makes them the backbone of modern Cloud platforms. The number of frameworks used in big data is very large, so the chapter will focus on a subset of those that leverage massively parallel processing and are based on the MapReduce paradigm and Hadoop ecosystem.

3.1.1 Big Data

Big data is a field that deals with very large datasets, whose storage, processing and analysis cannot easily be done with the use of relational databases. Big data can be better described by the following three characteristics, also known as “The 3 Vs of Big Data” [47]:

1. Volume: when considering big data, the quantity of generated and stored data is very large, usually more than a few TBs. As a result, it requires significant computing resources for the data to be processed and stored
2. Variety: an important characteristic of big data is heterogeneity, meaning that the type and nature of data can range from structured, semi-structured or even unstructured data collected via numerous different sources.
3. Velocity: the speed of data generation and processing of big data is very high, and their processing in real time is important, as it will then enable even faster data flow.

Apart from these 3 characteristics, there are some extra “Vs” that are sometimes included to describe Big Data, making the number of “Vs” 5 or 6. One of them is veracity, which refers to the reliability and quality of data, indicating that Big Data does not just have to be large, but it also requires reliability in order to reach valuable conclusions by analyzing it. Value is another characteristic, which refers to the importance of the output gained by big data processing. Finally, variability is another big data characteristic, which refers to the variety of the data, alongside the transformations that occur during its processing, sometimes resulting in the changing of the data format.

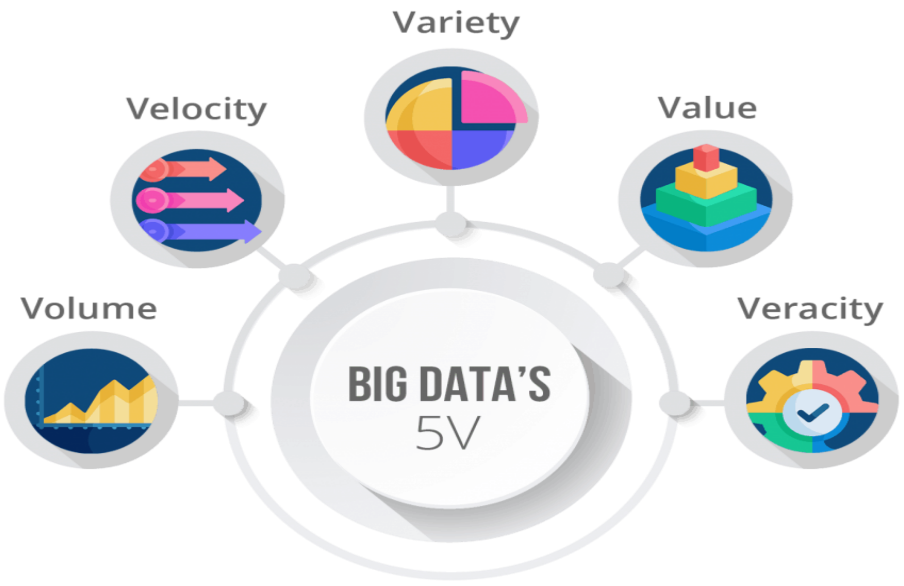


Figure 5. Five "Vs" of big data [48]

We mentioned three different data formats:

- Structured data: structured data is the easiest data to handle, as it is tabular data represented by columns and rows in a database. Data of this form is stored in relational databases.
- Semi-structured data: semi-structured data is not organized like structured data, but it has a basic structure. Semi-structured data consist of data stored in JSON files and key-value stores.
- Unstructured data: Unstructured data is the main focus of big data processing, as it is data not organised in a pre-defined data model. Video and audio files, raw text files, as well as binary files are regarded as unstructured data.

3.1.2 Big Data Processing in the Cloud

Big Data characteristics mean that big data applications have significant requirements in resources. Some features of the Cloud, like elasticity and pay for use pricing, make it a suitable place to run Big Data applications [2,49]. Depending on the application requirements, the data architect has to make some important decisions:

- Storage: Big data volume and velocity means that applications require large and scalable storage space. Cloud offers elastic storage resources on demand and reduces IT costs in many use cases. Therefore, the data architect has to find a suitable cloud provider considering pricing, scalability and availability.
- Database: Big data variety implies that each application might have different types of data to handle. Depending on the form of the data (mainly whether it is structured or not), the data architect must select an appropriate database management system, either relational, or NoSQL. In $3.3.1 the two different management systems will be compared.

- Data processing: Big data volume makes its processing a demanding procedure, in order to obtain its value. Furthermore, its velocity makes real time data processing essential in certain applications. Therefore, the data architect needs to use an appropriate data processing framework in order to process big data with high speed and if needed, in real time.

In the following subsections, state of the art frameworks that are used for storage ($3.2), database management ($3.3) and processing ($3.4) of big data will be presented. These frameworks are used by the world's largest tech companies. The reader will also be introduced to the Hadoop ecosystem.

3.2 Big Data Storage

3.2.1 Apache Hadoop Distributed File System (HDFS)

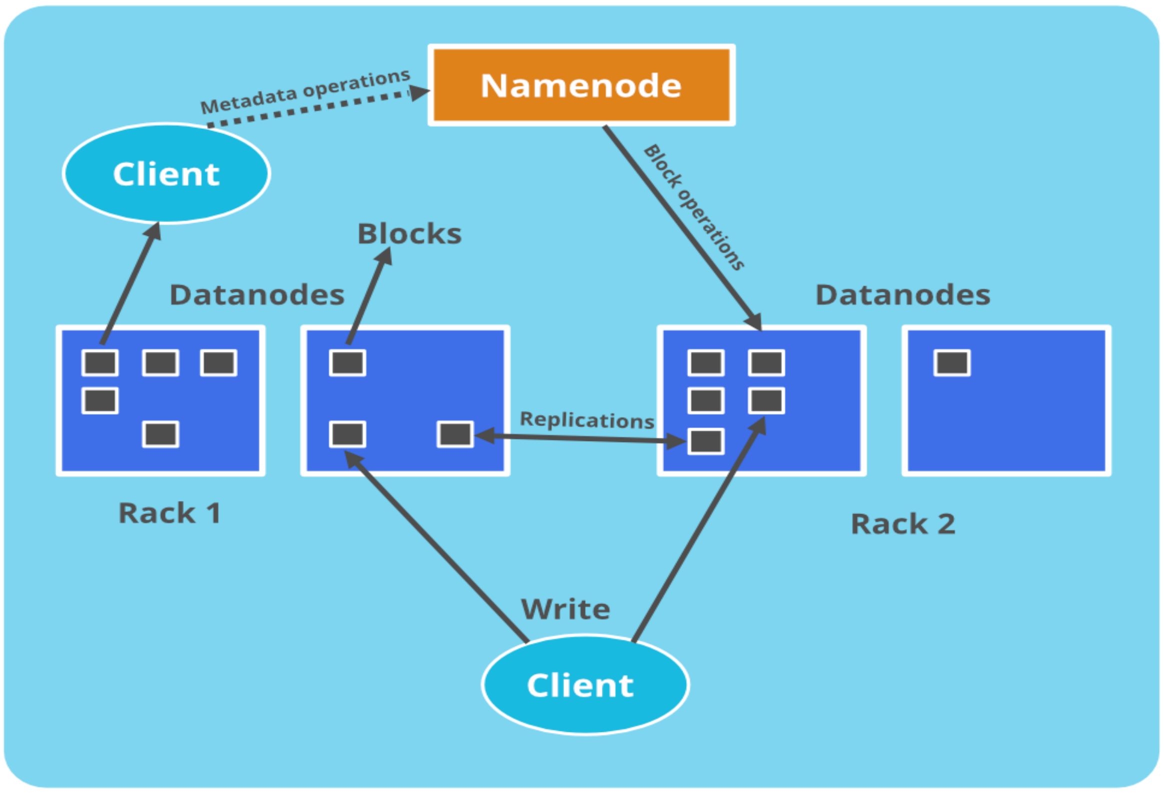


Figure 6. HDFS Architecture [64]

Apache Hadoop [50] consists of a collection of software tools to handle massive amounts of data in computer clusters. Hadoop consists of a storage part, HDFS, and a processing part, the MapReduce programming model, which will be discussed in $3.4. HDFS is a distributed and scalable file system, providing redundant and reliable massive data storage. It usually consists of a single master node (namenode), which manages the filesystem and has the metadata and the location of each data block, and a cluster of nodes for storage (datanodes), where data is stored in the form of blocks. (Figure 6). It is able to store very large files and datasets (in the range of terabytes) and achieves fault tolerance and reliability by replicating the data across different nodes. Its nature makes it ideal for storing big data. It is inspired by the Google File System paper [59], which aided Doug Cutting to develop an open-source implementation of its Map-Reduce framework, and he named it Hadoop, after his son’s toy elephant. Hadoop is widely used by big companies, and it is estimated that more than half of the Fortune 50 companies are using it. In 2010, Facebook had claimed to have the biggest Hadoop cluster in the world, with 21 PB of data distributed across 2000 nodes [60].

3.3 Big Data Databases

3.3.1 NoSQL and SQL

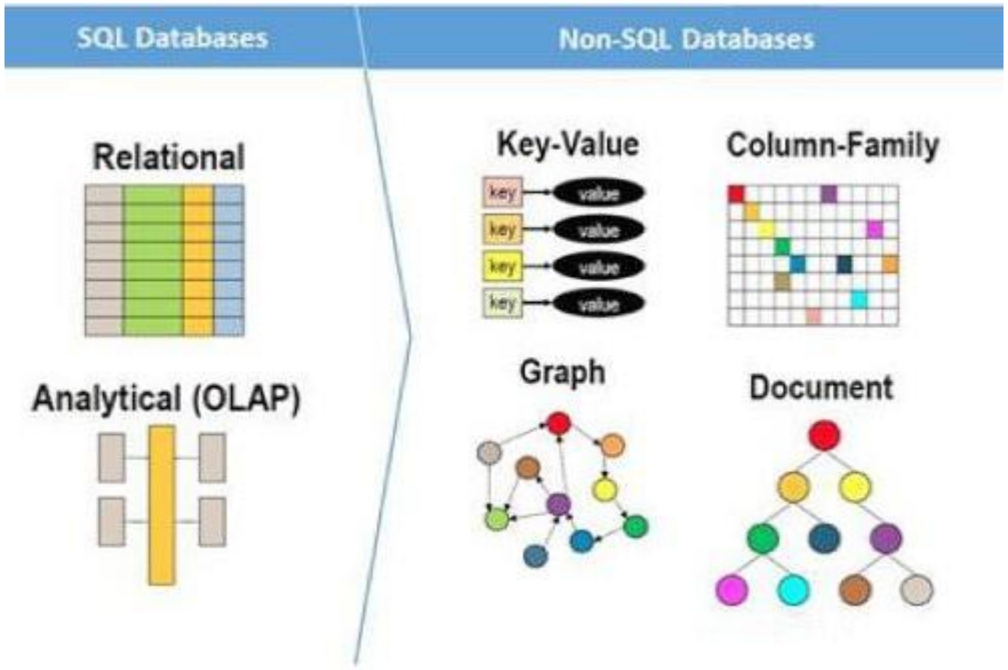


Figure 7. SQL and NoSQL databases [65]

SQL is the most widely used language for data management. It is a useful tool for relational database management systems (RDBMS), or for stream processing in relational data stream management systems (RDSMS). It is state of the art in handling structured data where the relations between the data tables are known, and it is based on relational algebra. RDBMSs has been a popular choice for storing data in databases since the 1980s. With the emergence of big data however, RDBMSs have started losing ground, as their relational nature can prove a significant limitation. Unstructured or semi-structured data need to be structured first in order to be added to an RDBMS. Moreover, when the volume of data grows, relations between data can become very complicated. Finally, RDBMSs cannot cope with data being generated in a high velocity, as it is designed for steady data retention, making it problematic and expensive when rapid data growth occurs. The inability of an RDBMS to operate well with data of high volume,variety and velocity makes it a bad alternative for big data management.

The aforementioned limitations led to a rise in popularity of NoSQL databases, which are purpose built databases for specific data models, and provide flexible schemas. NoSQL databases can ingest large volumes of data at low cost. NoSQL databases are schema-free so the data does not have to be structured, and it can be collected in the database and be structured later. NoSQL as a result supports all forms of data (structured, semi-structured, unstructured), making it a good fit for handling a good variety of data files, like videos or text files. NoSQL databases are also scalable, as they are designed to scale out by using distributed clusters of hardware. This enables them to provide high availability and fault tolerance when handling large volumes of data. NoSQL databases consist of:

1. Key-value stores: A key-value store is a hash table where a unique ID (key), points to a specific element (value).
2. Column-family stores: Similar to key-value stores, but in this case keys point to a specific row, not element. Columns are organised in column families.
3. Document store: Each key points to a certain document, which is a specific collection of data.
4. Graph store: Data is organised in a graph model, which can be used in parallel by several nodes.

In the following 2 subsections, 2 state of the art NoSQL databases will be presented that are widely used in the industry. They are just a few of the available solutions for database management, based on the MapReduce paradigm.

3.3.2 Apache Cassandra

Apache Cassandra [52] is a widely used, distributed NoSQL database management system. It is designed to handle large volumes of data across clusters, offering high availability and no single point of failure. Data replication across nodes provides fault tolerance and low latency across the cluster. Data is modeled in column families consisting of columns and organised by row keys. Cassandra comes with a query language (CQL), a simple alternative interface to SQL and it is also compatible with the Hadoop ecosystem, offering MapReduce and Apache Hive support.

Originally developed by Facebook, it was open sourced in 2009 [45] and is now managed by the Apache foundation and used by Netflix, Twitter, Reddit and Cisco, among other companies handling big data. [61]

3.3.3 Apache HBase

Apache HBase [53] is another open source, distributed NoSQL database management system, modeled after Google’s BigTable. It is part of the Hadoop ecosystem and it runs on top of HDFS, providing a fault-tolerant way of storing large quantities of sparse data. HBase data model organises data in tables having cells organised by row keys and column families. Each cell contains a value and a timestamp. Timestamps make HBase a consistent database, making it a CP system according to the CAP theorem. [46]

3.4 Big Data Processing Frameworks

3.4.1 Apache Hadoop MapReduce

Apache Hadoop MapReduce (MR) [57] is a computing framework for processing and generating big data sets with a parallel, distributed algorithm on a cluster. It is the core of the Hadoop data analysis framework.

MapReduce processing depends on two steps:

- Map: In the map procedure filtering and sorting of data from database tables is carried out. After the map stage, the processing results are passed to the Reduce stage.

- Reduce: In the reduce phase, a summary operation is performed, where aggregation functions are calculated and data is grouped.

MapReduce runs its tasks in parallel, managing communication and data transfer between its datanodes and providing redundancy and fault tolerance. However, due to the number of MR operations on HDFS (calculations, backup, download/upload, shuffling), disk I/O and data movements can create significant overheads. As a result, MR is gradually replaced by Spark and Flink for more data-intensive and interactive applications, and is now mainly used for offline data processing.

3.4.2 Apache Spark

Apache Spark [51] is a unified analytics engine for large scale data processing. It is an open source Apache project, which was developed in UC Berkeley in 2009. It is a distributed computing framework, which much like MapReduce splits up large tasks across different worker nodes. However, it can perform much faster as it uses RAM to cache and process data while applications are running, which enables it to store the intermediate results of a task in memory and avoid the extra HDFS I/O latency. Its “memory” feature makes it a good fit for iterative algorithms and applications like data mining and machine learning. Its architectural foundation lies in the resilient distributed dataset (RDD), a read-only multiset of data items distributed over a cluster of machines. Spark has different types of computing modes, like interactive queries, or stream processing. Spark is considered as a better alternative to MapReduce, however it is also considered a state of the art module to Hadoop, thanks to its compatibility with HDFS and Apache Hive (see $3.5).

Spark Core is the heart of the framework, and it consists of an application programming interface which offers functionalities of distributed task dispatching, scheduling and basic I/O functionalities.

Spark SQL is a component that can be used on top of Spark Core which introduced DataFrames which is a data abstraction over RDDs where a distributed collection of data is organized into named columns. It provides support for structured and semi-structured data.

Spark Streaming is another Spark component that uses Spark Core's scheduling capability to perform streaming analytics. It receives data in small batches and performs RDD transformations on them.

3.4.3 Apache Flink

Apache Flink [56] is an open-source stream-based distributed data processing framework. Its core component is a dataflow streaming execution engine, similar to Spark Streaming. Flink provides quality memory management and can also prove very fast, as it allows iterative processing to execute solely on one node, without having to process each step independently on the cluster. Flink works really well when repeated passes need to be made on the same data values, which makes it ideal for machine learning use cases.

3.4.4 Apache Storm

Apache Storm [55] is another open source distributed computing framework, tailored for real-time big data processing. It is a scalable and fault-tolerant framework, and its use cases include real time analytics, online machine learning, continuous computation, distributed RPC and ETL.

Storm became open source after being acquired by Twitter in 2011 [43] which still uses it for real-time data processing. With the emergence of Spark and Flink, Storm use has significantly decreased, it remains however a reliable choice for real-time data analysis.

3.5 Other Hadoop tools

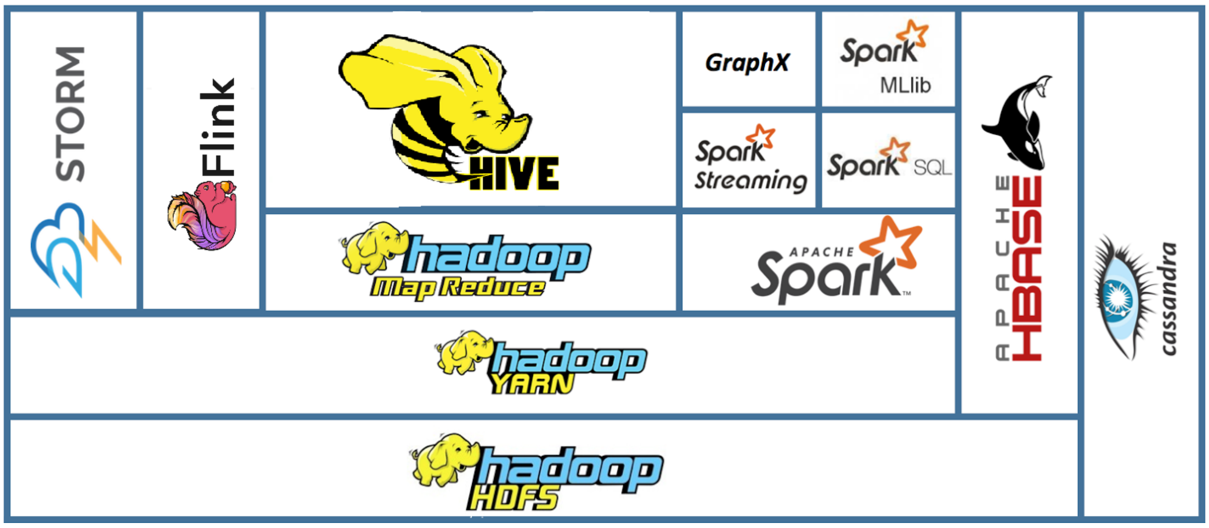


Figure 8. Architecture of the Hadoop ecosystem [62]

In this subsection, two more tools of the Hadoop ecosystem that will be addressed later in the thesis will be described briefly. Figure 8 shows how the different frameworks discussed in this chapter are connected. The figure shows, from the bottom moving up, the storage layer, resource management layer and application layer.

3.5.1 Apache Yarn

Yarn [58], also known as Yet Another Resource Negotiator is another component of the Hadoop ecosystem that works as a large-scale, distributed operating system for big data applications. It is responsible for the resource management and job scheduling to meet application needs. In a system architecture, it is located between HDFS and the processing engines used to run applications.

3.5.2 Apache Hive

Apache Hive [54] is a data warehouse software project built on top of Apache Hadoop, providing data query and analysis via an SQL-like interface. It can query data stored in various databases and file systems that integrate with Hadoop. It uses various SQL optimizations like partition pruning or predicate pushdown, enabling it to process more raw data in bulk than Spark SQL or CQL.

It was developed by Facebook, and is now used by many companies like Netflix and the Financial Industry Regulatory Authority (FINRA) [63].

3.6 Conclusion

In this chapter an overview of state of the art systems for handling big data was given. In the next chapter, a particular database optimization challenge will be presented, query optimization. In chapter 5, query optimization will also be tackled with regards to the optimization goals of a cloud environment.

Chapter 4

4.Query Optimization

4.1 Query Processing

4.1.1 Introduction - Query Processing Steps

In the previous chapter, we talked about handling large datasets in a cloud environment. Database management systems are classified into relational databases which model data in a structured way and use SQL for writing and querying data, and non-relational databases (NoSQL) which handle all types of data and use different query languages, although most popular NoSQL databases have SQL-like interfaces.

Query languages are computer languages used to make queries in databases and information systems. A database query is a request to insert or access data from a database to read it, update it, or delete it. Query processing is one of the most important processes happening in a DBMS, where high-level queries are translated into low-level expressions.

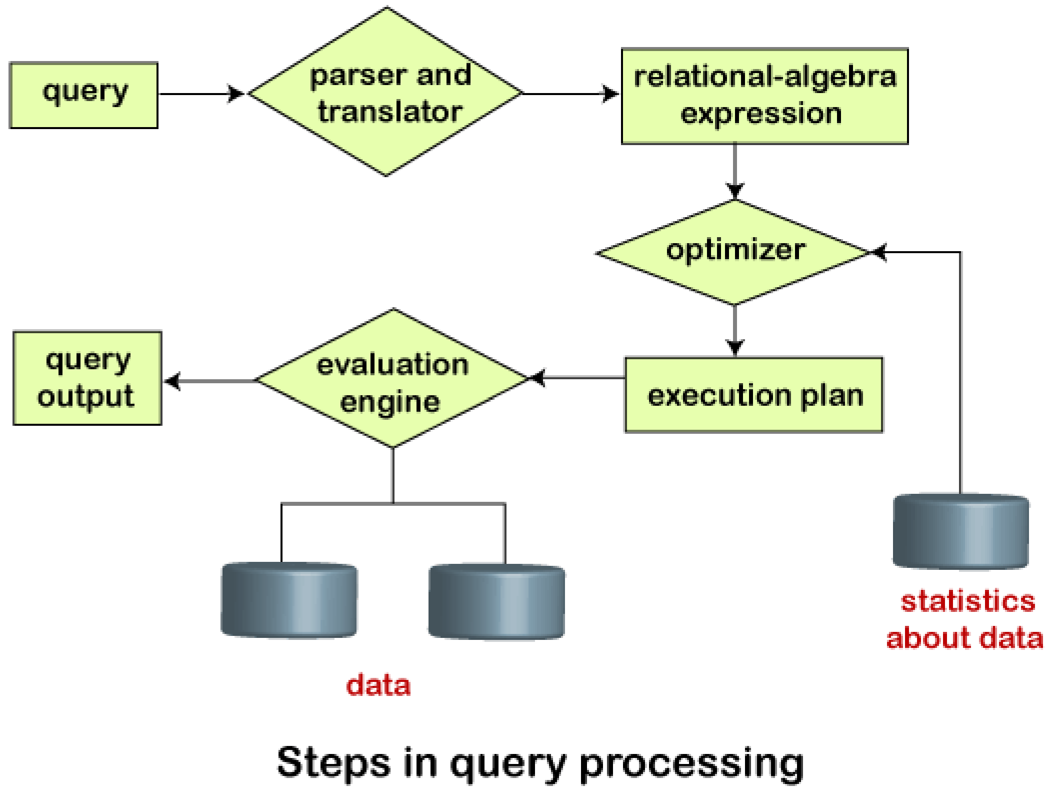


Figure 9. Query Processing Steps in an RDBMS [66]

After a user submits a high-level query language (in our case we mainly discuss SQL) query, its processing consists of five essential steps [67]:

1. Parsing: In the parsing phase, the submitted query is checked for syntax errors, based on the query language grammar. The parser verifies the attributes' and tables' names and creates a 'parse-tree', a syntax tree for the query.
2. Translation: If the query is parsed successfully, the translation phase transforms the query from SQL (high-level), to relational algebra instructions that indicate the operations that must be performed on the data in order to solve the query (e.g. selection, project, join). The output of this step is a tree containing all the logical operations that need to be applied for the processing of the query (logical tree).
3. Optimization: After the first steps, the translated relational query is given as an input to the query optimizer. The optimizer uses statistics and data collected from the database for each table and attribute to generate one or more query execution plans for a query, each of which may be a mechanism used to run a query. A query execution plan specifies the execution order for the operations described in the algebraic tree, as well as the operators used. The most efficient query execution plan is selected and used to run the query, usually based on a cost model. The output of this step is a query plan in the form of a tree containing all the physical operators needed to process the query (physical tree), which is later sent to the execution engine.
4. Evaluation/Execution: The query execution plan returned by the optimizer is executed by the query evaluation/execution engine, and the answer to the query is returned.

4.2 Query Optimization & Techniques

4.2.1 Introduction - Classical Query Optimization (CQ)

In this subsection we will focus on query optimization, one of the most challenging tasks during query processing.

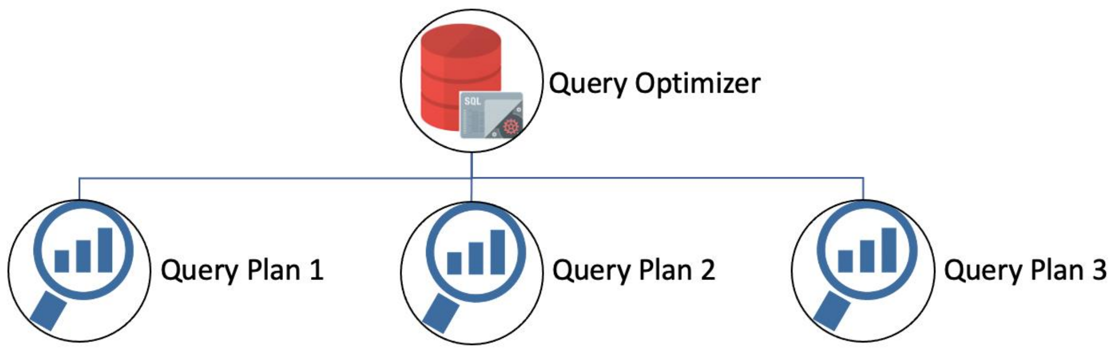


Figure 10. Query Optimizer

Classical query optimization assumes that query plans are compared according to one single cost metric, usually execution time, and that the cost of each query plan can be calculated without uncertainty. Given a query, there are multiple query plans that can be used for its execution, and selection of an optimal query plan is a challenging optimization problem [87]. For example, finding the optimal join order in a query, which is a task of query optimization, is an NP-complete problem.

The set of possible query execution plans is formed by examining all possible data access paths (e.g. index access, full table scan), as well as all possible join techniques (e.g. merge join, hash join) and scheduling decisions (e.g. join order) for the operators used. By applying some operators on the logical plan returned by the parser, a physical execution plan is produced. The query plan search space can become almost infinitely large for SQL queries of high complexity. A query optimizer can either use rules to select a query plan, or cost functions. Today, most query optimizers are cost-based, as rule-based estimations are usually less accurate.

4.2.2 Rule based query optimization

In rule based query optimization (RBO), the conversion of the logical plan to a physical plan is based on rules. For instance, the selection of join operators can depend on the size of the smallest table joined, and vary from shuffle join to replicated join in a distributed system scenario, if it surpasses a size threshold. Although RBO is not preferred in modern optimizers, there are cases where rule based choices can be of use. For instance, in the case of nested queries, a rule-based approach based on a query graph model might not consider all the range of possible query plans, but can prove useful in that use case for reducing the optimization overhead of using an algorithm to examine the space of alternative plans. A well known optimizer that still remains mostly rule based is Catalyst [3], the optimizer module of Spark SQL, which will be discussed in $4.4.5.

4.2.3 Cost Based Query optimization

In cost based query optimization (CBO), costs are used to estimate the runtime cost of executing a query, in terms of the number of I/O operations required, CPU path length, amount of disk buffer space, disk storage service time, and other factors which provide an accurate estimation of the query execution time, which is the most common optimization objective.

The choice of an execution plan is the result of numerous factors, such as table and attributes statistics, database and system characteristics, cost formulas, and algorithms to cope with the big number of possible execution plans. There is a tradeoff between the amount of time spent looking for the best query plan and the quality of the choice, so in cases where the space of alternative plans is very big, exhaustive evaluation of query plans will probably result in high latency. A common technique in CBO is cardinality estimation, where the optimizer takes into account table cardinalities and estimations for the selectivity of predicates, to calculate the cardinality reduction after each operation. The translation of cardinalities into cost estimation is achieved through the use of cost functions, which are specified in the most important component of cost based optimizers, the cost model.

4.2.4 Cost Models

Database cost models are the core of query optimizers. They consist of two parts, the logical and the physical one. [83] The logical part is responsible for estimating the data volumes involved in the operations. It depends only on the data stored in the database, the query operators, and the order in which they will be executed. It is independent from the choice of algorithm and implementation to carry each operation (e.g. selection of join operator). The physical part is provided with the estimated data volumes from the logical plans, and is responsible for estimating the cost of different algorithms and implementations for each operation. The optimizer uses this information and executes the query accordingly..

4.3 Parametric Query Optimization (PQ)

Parametric Query Optimization (PQ) [69] takes a different approach in how the cost of a query plan is modeled. It associates each query plan with a cost function c: $R^n -> R$ instead of a constant value c, representing the cost of a query plan as a function of n parameters, whose value is not yet known at optimization time. PQ parameters may represent selectivity of predicates, or the amount of available buffer space at query execution time [1]. The optimization goal is to find a plan set that contains an optimal query plan for each possible combination of parameter values. The main advantage of PQ over CQ, is that query optimization, which can prove a computationally expensive operation, is avoided at runtime and it happens in a pre-processing phase. After the parameters of a query are specified, query optimization simply involves the selection of the relevant plan from the aforementioned plan set.

One branch of PQ algorithms decomposes a PQ problem into a set of non-parametric CQ problems. This approach has the advantage that an existing query optimizer for CQ can be used for PQ with only small changes to the optimizer's code. Such PQ approaches are called non-intrusive. **[70]** Many non-intrusive approaches to PQ are based on parameter space decomposition **[70,71,72]**, where they repeatedly use a standard optimizer to generate optimal plans for fixed parameter values, thus decomposing the parameter space into different regions where a single plan is optimal for each region. Another branch of PQ algorithms **[70]** is based on dynamic programming, like the CQ algorithm by Selinger **[67]**. They use several data structures and corresponding manipulation functions. Ioannidis et al. **[69]** use randomized algorithms for PQ, which cannot offer worst-case guarantees on generating complete plan sets.

4.4 Query Optimizers

In this subsection, an overview of the evolution of query optimizers will be presented, based on a relevant tutorial paper [79].

4.4.1 EXODUS

The EXODUS optimizer [75] marked the beginning of an era for rule based query optimizers. Its architecture was based on code generation from declarative rules of logical and physical algebra, and it introduced the division of a query optimizer into independent modular components.

4.4.2 Volcano

The Volcano work followed [76], and it combined improved extensibility with an efficient search engine based on dynamic programming and memorization. Extensibility was achieved by generating optimizer source code from data model specifications and by modeling costs as well as logical and physical properties into abstract data types. Effectiveness was achieved by permitting exhaustive search, which will be pruned when the user chooses it. Efficiency was achieved by combining dynamic programming with directed search based on physical properties, introducing a new search algorithm called directed dynamic programming. The choice when and how to use heuristic transformations or cost-based optimization is not prescribed or "wired in."

Volcano was regarded as a very good fit for object-oriented and scientific database systems.

4.4.3 Cascades

Cascades [77] leveraged Volcano advantages (modularity, extensibility, dynamic programming) [76], offering a number of new features in order to overcome its limitations. First of all, modeling of predicates and other operations as part of query and plan algebra was introduced. Specific operators could now be both logical and physical, appearing both in the optimizer input and output. Moreover, new operators like sorting and merging exploration were inserted as normal operators, being inserted into the execution plan based on rules. In Volcano, they were special operators not appearing in rules.

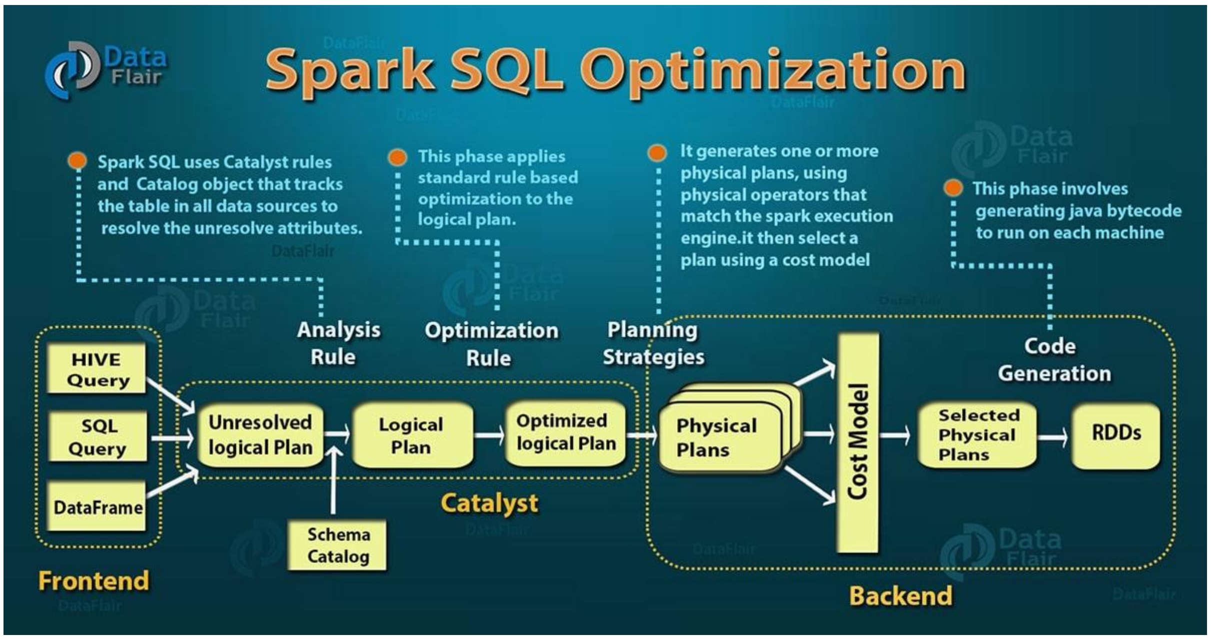
4.4.4 Calcite

Figure 11. Apache Calcite

Building on ideas from Cascade, Apache Calcite [78] is a foundational software framework that provides query processing, optimization, and query language support to many popular open-source data processing systems over heterogeneous data sources. Its main component consists of a modular and extensible query optimizer. It is currently the most widely adopted optimizer for big-data analytics in the Hadoop ecosystem, as it has been adopted by Apache Hive [54].

One of the main advantages of the Calcite optimizer is that each of its components is pluggable and extensible. Users can add relational operators, rules, cost models and statistics.In addition, Calcite supports multiple planning engines. As a result, the optimization can be broken down into phases handled by different optimization engines depending on which one is best suited for the stage. Calcite also provides an adapter for Apache Cassandra [52].

4.4.5 Catalyst


Data Flair
Spark SQL Optimization
Spark SQL uses Catalyst rules and Catalog object that tracks the table in all data sources to resolve the unresolve attributes.
This phase applies standard rule based optimization to the logical plan.
It generates one or more physical plans, using physical operators that match the spark execution engine.it then select a plan using a cost model
This phase involves generating java bytecode to run on each machine
Analysis Rule
Optimization Rule
Planning Strategies
Code Generation
HIVE Query
SQL Query
DataFrame
Unresolved logical Plan
Logical Plan
Optimized logical Plan
Physical Plans
Cost Model
Selected Physical Plans
RDDs
Schema Catalog
Catalyst
Frontend
Backend
Figure 12. Catalyst Architecture

In the core of Spark SQL lies the Catalyst optimizer, which uses many rules to optimize queries for efficient execution, while being extendable and allowing new rules to be added easily, if needed. These rules are applied to a data structure called a tree, the main data type in this optimizer. Before a query is executed, it passes through four phases of the catalyst optimizer. These phases are analysis(parsing), logical optimization, which is done on the analyzed logical plan; physical planning, during which physical operators are applied to the optimized logical plan; and code generation, in which the physical plan is converted into Java bytecode to run on each node of the cluster.

Spark SQL can also be used to execute queries over multiple data sources as in Calcite. However, although the Catalyst optimizer in Spark SQL also attempts to minimize query execution cost, it lacks the dynamic programming approach used by Calcite and risks falling into local minima [78].

4.4.5.1 Cost Model for Spark SQL

The fact that Catalyst still relies on a rule-based optimizer (although it makes some cost-based choices when choosing the join algorithms) is considered a significant limitation which led Baldacci et al. to introduce a cost model for Spark SQL [3], covering the class of GPSJ (Generalized Projection/ Selection/Join) queries and allowing an accurate prediction of the cost of each query to be made. A GPSJ query is a query composed only of joins, selection predicates and aggregations. As the authors described it, it is a first step towards turning Catalyst into a fully cost-based optimizer and to compare the execution cost of different physical plans even when adaptive execution is considered. Their cost model will be discussed in more detail in Chapter 7, where we will also reconstruct it and evaluate its accuracy.

In another recent work **[80]**, an algorithm was implemented to reduce unnecessary memory consumption and overhead during the shuffling process of Catalyst.

4.5 Other Query Optimization directions

4.5.1 Multi-query Optimization

In the previous subsections, the definitions and techniques presented concern single-query optimization, where queries are optimized one at time. A different approach is multi-query optimization, which has the task of generating an optimal combined execution plan for a collection of multiple queries [85][86]. Compared to single-query optimization, multi-query optimization can exploit commonalities between queries, for example by computing common query sub-expressions once and reusing them, or by sharing scans of relations from disk and the cache.

4.5.2 Query Re-optimization

When describing query processing steps in $4.1, we showed that the standard approach includes optimization happening before execution. The optimizer combines statistics and cost models to yield an optimal plan, and execution happens instantly after that. However, when the complexity of queries increases, the quality of the choice degrades, as finding the optimal plan is very time-consuming. This problem is tackled by query re-optimization [87], which involves progressively optimizing the query, executing parts of the plan at a time, and continuously optimizing the plan after each part is executed, in the light of new information. As a result, plan quality increases and the optimization overhead decreases.

4.6 Conclusion

In this chapter, an analytical description of the query optimization process was given, as well as examples of techniques and optimizers. Until now, when we were talking about query optimization we were assuming that it had one objective, which usually is, or depends on query execution time. In the next chapter, we will see in detail how query optimization can have multiple objectives, and explore techniques and related work using multi-objective query optimization in a cloud environment, which will be the core of our proposal.

Chapter 5

5.Multiobjective Optimization

5.1 Introduction

Multi-objective optimization (MOO) is an area of multiple criteria decision making that concerns mathematical optimization problems involving more than one objective function that need to be optimized simultaneously. MOO has applications in many fields of science like engineering, economics and logistics where optimal decisions need to be taken in the presence of trade-offs between two or more conflicting objectives. When the objective functions are conflicting, no single solution simultaneously optimizes each objective, and solutions are called nondominated, or Pareto optimal. The number of Pareto optimal solutions that can be considered equally good can be very big. In this chapter, we will focus on MOO problems concerning data management.

5.2 MOO Definitions

5.2.1 Mathematical Definition - Pareto Front

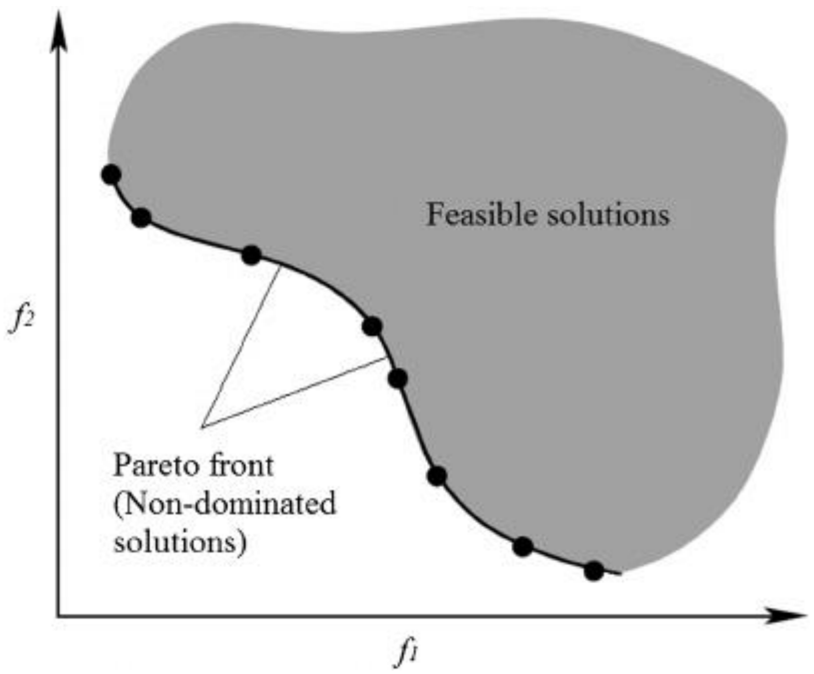


Figure 13. Pareto Front

In mathematical terms, a multi-objective optimization problem can be formulated as:

$$\min(f_1(\vec{x}), f_2(\vec{x}), \ldots, f_k(\vec{x}))$$
$$\text{s.t. } \vec{x} \in X,$$

where k is the number of objectives and the set X is the feasible set of decision vectors. A solution $\underline{x}*$ (and its output $\underline{f}(\underline{x}*)$) is called Pareto optimal if there does not exist another solution that dominates it. The boundary defined by the set of all points mapped from the Pareto optimal set (Fig.13) is called the Pareto front or Pareto frontier.

Therefore, the Pareto front is a set of nondominated solutions, chosen as optimal, where no objective can be improved without sacrificing at least one other objective. On the other hand a solution $\underline{x}$ is referred to as dominated by another solution $\underline{x}*$ if, and only if, $\underline{x}$ is equally good or better than $\underline{x}*$ with respect to all objectives.

5.2.2 MOO Types

There usually exist numerous Pareto optimal solutions for MOO problems, so solving such a problem is not as straightforward as it is for single-objective optimization problems. Solving MOO problems can be done in different ways. Some methods convert the MOO problem to a single-objective one. These methods are called scalarized. Other methods have the goal of approximating a representative set of Pareto optimal solutions, each one providing a different trade-off between the objectives. [107]

In some cases decision making plays an important role in the solution, as solving an MOO problem requires finding a single Pareto optimal solution that best meets the decision maker's preferences.

MOO problems can be tackled with methods belonging in four different classes: [88]

- No preference methods belong in the only class where no decision making is involved. The goal is to find a relevant set of Pareto optimal solutions providing different trade-offs.
- In a priori methods, the decision maker has provided his objectives and then a solution best satisfying these preferences is found.
- A posteriori methods combine the two aforementioned techniques, as a representative set of Pareto optimal solutions is first found, and then it is up to the decision maker to select one of them.
- In interactive methods, the decision maker iteratively searches for his/her preferred solutions. In each iteration he/she is presented with a set of Pareto optimal solutions, and then gives feedback to the system, as to how the solutions could be improved. Then, new Pareto optimal solutions are produced based on the information provided by the decision maker. This way, the DM learns about the feasibility of his/her preferences and concentrates only on relevant solutions. It is up to the decision maker to stop the search and pick a solution.

5.3 Optimization Problems in the Cloud

5.3.1 Introduction

Some of the unique characteristics of the cloud, like the pay-as-you-go pricing scheme and the elasticity it provides, create numerous optimization opportunities. Most of them involve IaaS platforms, where the different resource allocation options mean that one combination of resources will be better than another, depending on user objectives. When considering data management, query processing and resource management are two procedures containing optimization problems that have been tackled a lot from the research community. We will first describe them and then explore how they can be solved in a multiobjective environment.

5.3.2 Query Optimization

Query optimization in databases was discussed in chapter 4. When considering query processing in a cloud environment, it has to be taken into account that in most cases, a distributed data management system is used, like Cassandra or HTable **[52][53]**. Furthermore, optimization in IaaS requires considering more factors than in a traditional cluster, like future resource availability, release of resources etc. In IaaS, a query is processed like this: a user submits a query in a high level language, like SQL. This query is submitted and received from the master nodes. The optimizer then calculates a detailed query execution plan (QEP) for the query based on a cost model and decomposes it into sub-queries which are assigned to worker nodes based on some load balancing criterion. Each worker evaluates its sub-query and returns the results to the master. The master merges all the sub-results and produces the answer to the query.

5.3.3 Resource Allocation, Task Scheduling

When it comes to resource management, IaaS platforms come with a pool of available resources which can be rented by their users to execute their tasks. The optimization problem that arises is broken into two sub-problems:

- Resource allocation: The resource configuration a user chooses for her application from the alternatives the cloud vendor provides will determine how efficiently the application will be executed. Overprovisioning or underprovisioning for a specific task could result in higher monetary costs, or lower execution time respectively.
- Task Scheduling: Scheduling tasks to available resources is a difficult optimization problem, due to the infinite space of alternative schedules.

5.4 MOO Goals

The aforementioned optimization problems can be tackled either as single objective optimization problems (SOO), or multi-objective optimization problems (MOO).In SOO problems, the goal is to find the solution that minimizes or maximizes a certain metric (eg. QEP with minimum monetary cost in query optimization). In MOO, more than one metric is considered, and the optimal solution involves a tradeoff between the two. The different optimization goals can be the following:

- Execution time: In query and resource optimization, execution time usually translates to query execution time. In most research MOO works, it is the primal optimization objective. The aim is to maintain low execution time without ignoring other objectives.

- Monetary cost: Monetary cost is an equally important factor in optimization problems in the cloud. Resources are rented based on a pricing model and depending on the user's budget constraints, cost can be traded off other objectives **[95][94][103][112]**.

- Energy Consumption: An optimisation metric closely related to monetary cost (monetary cost is dependent on the energy cost of the resources rented) is energy consumption. While users mostly aim to reduce latency and cost, cloud providers yearn for reducing the energy consumption in their data centers, which is one of the major costs in the cloud service environment. **[99][101]**

- Result Precision: Accuracy is another optimization goal, sometimes more important than cost. For instance in services where crucial decisions have to be taken quickly (e.g. a hospital with a medical patient dataset), there is no room for inaccurate results. In these cases, execution time can be traded off against result precision, aiming to have high accuracy and low latency. **[1][92]**

5.5 Multi-objective Query Optimization

5.5.1 Multi-objective Query Optimization (MQ)

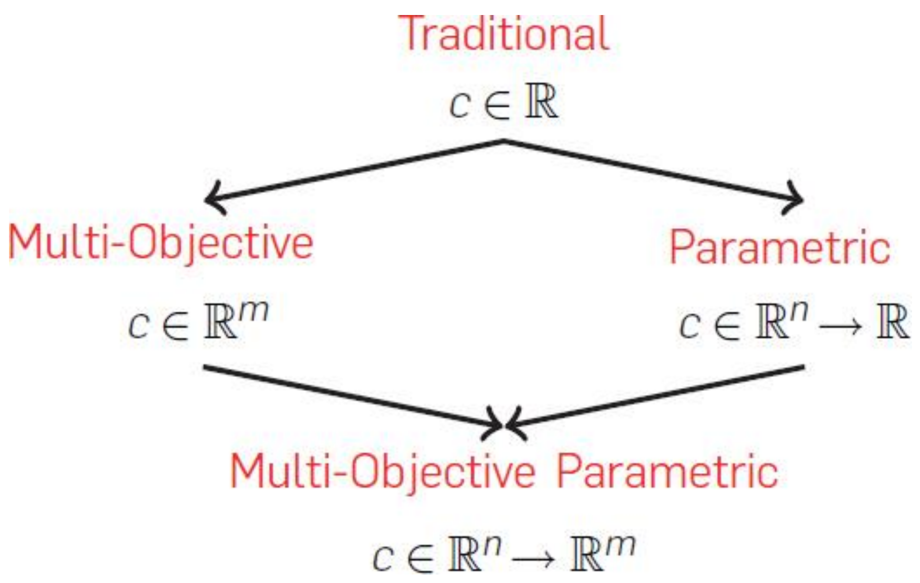
Multi-objective Query Optimization (MQ) generalizes the CQ optimization model and associates each query execution plan with a cost vector $\mathbf{c} \in R^n$which describes the cost of the plan according to multiple cost metrics. The optimization goal is to find a set of Pareto-optimal query plans, meaning that for a given query plan in the Pareto front, no other plan has better cost according to all the optimization metrics at the same time.

MQ algorithms have to compare the different plans according to n cost metrics. The optimization goal here is to find a plan that represents the best tradeoff between the conflicting metrics, with user preferences taken into account. The Selinger single-objective CQ algorithm **[67]** has been generalized to MQ **[89]**. In this work, plans are compared according to multiple cost metrics during pruning and non Pareto-optimal plans are discarded.

Other MQ algorithms are tailored for specific cost metrics or user preferences **[90],[91],[92]**. Some of them support multiple metrics if the query cost can be represented as a weighted sum over the cost of its sub-plans. Another branch of MQ algorithms separates multi-objective optimization from join ordering. For example, they produce a time-optimal join tree first and configure operators within it according to the other cost metrics later. Algorithms for multi-objective data flow optimization **[93]** are not applicable to query optimization with join reordering.

5.5.2 Multi-objective Parametric Query Optimization (MPQ)

In 2014, Trummer and Koch presented multi-objective parametric query optimization **[1]**, which was an attempt to generalize the database query optimization problem by combining and generalizing multi-objective query optimization (MQ) and parametric query optimization (PQ).


Traditional
$c \in \mathbb{R}$
Multi-Objective
$c \in \mathbb{R}^m$
Parametric
$c \in \mathbb{R}^n \to \mathbb{R}$
Multi-Objective Parametric
$c \in \mathbb{R}^n \to \mathbb{R}^m$
Figure 14. Query Optimization Variants

In MPQ, the cost of a query plan is represented as a vector-valued function c : $R^n \rightarrow R^m$. This allows to model multiple parameters as well as multiple cost metrics, thus combining MQ and PQ cost functions (Figure 14). As in PQ, a parameter in MPQ may represent any quantity that influences the cost of query plans and is unknown at optimization time. The objective of MPQ is to generate a complete set of relevant plans, containing a plan p∗ for each possible plan p and each point in the parameter space x such that p∗ has at most the same cost as p at x according to each cost metric. In other words, the objective is to find a Pareto front for all points in the parameter space. All relevant query plans are generated in advance (like in PQ), and no query optimization is required at runtime. Although MPQ combines MQ and PQ, it is impossible to use MQ or PQ algorithms in MPQ, as MQ algorithms do not support parameters and PQ algorithms do not support multiple cost metrics.

Π

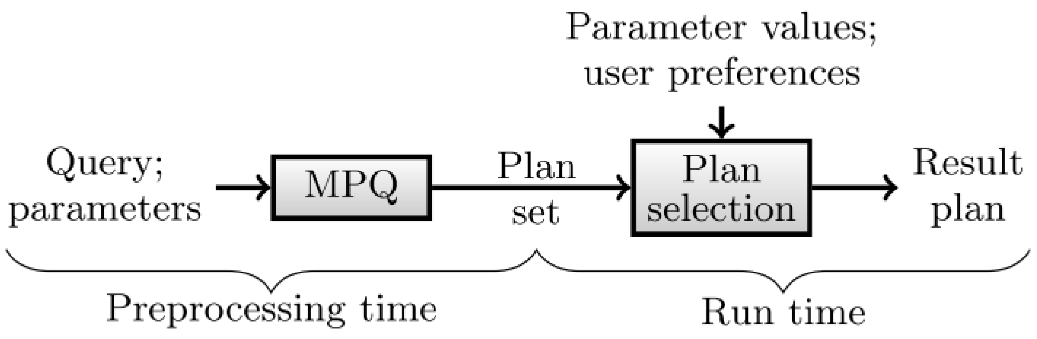


Figure 15. MPQ Context

Let p1 and p2 be two query plans that produce the same result. Plan p1 dominates plan p2 in all points of the parameter space in which p1 has at most the same cost as p2 according to each cost metric. The function Dom(p1, p2) ⊆ X yields the parameter space region where p1 dominates p2:
$\mathrm{Dom}(p1, p2) = \{x \in X \mid \forall m \in M : c^m(p1, x) \leq c^m(p2, x)\}$, where c(p,x) is the cost function for plan p

P ⊆ P(Q) is a Pareto plan set (PPS) iff it contains for each possible plan p∗ ∈ P(Q) and each parameter vector x ∈ X at least one plan plan that dominates p∗ for x:

$$\forall p* \in P(Q)\ \forall x \in X\ \exists p \in P : x \in Dom(p, p*)$$

An MPQ problem is defined by a query Q, a parameter space X, and a set of cost metrics M. Any Pareto Plan Set for Q is a solution to the MPQ problem.

Trummer & Koch also introduced the first 2 MPQ algorithms, the first being the Relevance Region Pruning Algorithm (RRPA) for MPQ, which associates each query plan with a relevance region(RR) in the parameter space, which is used to detect irrelevant plans. The algorithm is generic for different types of cost functions and runs exhaustively.

The second algorithm presented was PWL-RRPA, where data structures were used to represent cost functions and relevance regions (RR). Relevange region of a query plan is the set of parameter combinations for each a given plan is relevant. It is a specialized version of RRPA for piecewise linear (PWL) cost functions. PWL-RRPA was evaluated in Postgres, using two optimization metrics (execution time and monetary cost). Standard formulas were used to estimate join time; monetary cost was estimated according to the pricing system of Amazon EC2.

5.6 MOO Techniques

In order to respond to the optimization problems we discussed in $5.3, a lot of other MOO techniques have been used, either providing the user with a Pareto optimal query plan/front based on his preferred trade-off (in query optimization), or responding with a front of Pareto optimal cluster configuration (in resource optimization). We present these techniques here, alongside the challenges they are tackling and their optimization metrics, all implemented for cloud computing applications and scenarios:

- Weighted Sum Model: With the use of a weighted sum model (WS), a multi-objective optimization problem is transformed into a single objective one, by distributing the weight of every objective [94,96]. It calculates the optimal plan for a number of weight distributions, and returns a Pareto plan set (PPS). A disadvantage of WS techniques is the poor coverage of the Pareto frontier, which makes it difficult to find the optimal tradeoff for our objectives. [97]

- MOO Algorithms: Another common choice is a mathematical programming-based model where an algorithm is implemented and run repeatedly. After each loop, a Pareto optimal solution is produced. This algorithm can be exhaustive [1][93], linear programming-based [98], probabilistic [93], greedy [93], or incremental, meaning that they avoid regenerating query plans when being invoked several times for the same query. [99]

- Evolutionary Algorithms: Evolutionary multi-objective optimization (EMO) algorithms are another popular approach for calculating Pareto optimal solutions. Non-dominated Sorting Genetic Algorithm-II (NSGA-II) is probably the most popular choice [100,101] or basis for new algorithms [102,103], with interesting alternatives having considered particle swarm optimization [104], simulated annealing [106],and ant colony optimization [105]. The fact that genetic algorithms calculate entire sets of solutions make it possible for the entire Pareto front to be approximated. However, in some cases the convergence of the algorithm (and consequently the Pareto optimality of the solution set) cannot be guaranteed and measured. [107]

- Machine Learning: Some works on scheduling have also benefited from recent machine learning techniques like based modeling approaches, which can automatically learn a predictive model for each of the objectives from the behavior of user tasks [95]. Artificial Neural Networks (ANN) have also been used to predict the resources needed based on estimated completion time and energy consumption. [101] Reinforcement learning techniques have also been hot in MOO query optimization works, especially when tackling query re-optimization. [108,109]

Most of the works studied in this section tackle the optimization dipole between query latency and monetary cost, however there have also been multi-objective optimization works aiming to reduce energy consumption alongside latency [101,99], and also works mentioning result precision in their use case scenarios [1], or having error percentage constraints alongside time constraints. [92]

The target environment of most research works is IaaS platforms. However, there have also emerged works studying scheduling in FaaS environments [100], and more are expected to appear in the future, as Serverless is steadily gaining ground in the Cloud landscape.

Finally, the majority of the research works have been evaluated in a bi-objective way, having two optimization metrics. However, some of the works are by nature many-objective, having used up to 9 optimization metrics. [89]

5.7 Existing Cloud MOO Works

In the previous section, we presented different MOO techniques used in works tailored for cloud computing applications. In this section, we present some exemplary systems using the presented techniques as well as the use cases that they support:

- Weighted Sum Model: Normalized Weighted Sum Model (NWSM) [94,96] is a user-interactive multi-objective query optimization strategy based on a modified weighted sum model and is used to achieve MOO in a mobile-cloud database environment, able to take into account 3 cost objectives: query latency, monetary cost, and mobile device energy consumption. By using NWSA, the user assigns weights of importance to each of the 3 objectives, which are multiplied with the different metric costs to produce the final cost estimation of a query execution plan (QEP). In the end, the QEP with the lowest score is selected and executed.

- MOO Algorithms: In $5.5.2 we introduced multi-objective parametric query optimization [6] where an exhaustive algorithm which executes before query run time for every combination of a number of parameters (size of tables, type of joins) to cover the space of query execution plans, taking into account multiple objectives. Real life applications of this technique involve querying Cloud databases with a time-money tradeoff, or executing embedded SQL queries with a time-result precision tradeoff. In another work concerning dataflow schedule optimization in the Cloud [93], several greedy, probabilistic, and exhaustive optimization algorithms are used to explore the space of alternative schedules, before choosing an optimal one based on a time-money tradeoff. More recently, Kllapi et al. [98] tackled dataflow optimization with index interleaving algorithms aiming to utilise idle slots in the dataflow execution schedule and build indexes in parallel, without extra delay or cost.

- Evolutionary Algorithms: Christoforou et al. [100] proposed a new resource management approach in a FaaS platform, based on intelligent techniques and genetic algorithms. Three different algorithms were used to find a set of near-optimal solutions to the MOO problem. These solutions could be of use for developers going serverless looking to select an efficient resource allocation scheme. The results were promising, although works on scheduling in FaaS environments are not very common yet [111]. In another MOO work aiming to reduce energy consumption and makespan, [101] the DVFS technique is used to reduce energy consumption, which enables the VMs to run at different blends of frequencies and voltages. The MOO is solved with the help of NSGA-II for energy consumption and latency. Finally, Artificial Neural Networks were also used in the genetic algorithms optimization procedure to predict the available resources based on task characteristics.

- Machine Learning: Song et al. tackled the MOO cloud problem by building an optimizer using a novel approach [95] to incrementally transform a MOO problem to a set of constrained optimization (CO) problems which can be solved individually. The coverage (of the Pareto frontier) and efficiency challenges are met by the use of different Progressive Frontier (PF) algorithms, which were also implemented in a Spark workload. The optimizer also used recent ML modeling approaches to learn a predictive model for each MO objective based on the runtime behavior of a user task, with no requirement of using a query plan. Evaluation showed that the optimizer can outperform state-of-the-art optimizers, like Ottertune. Handaoui et al. introduced ReLeaSER [109], a reinforcement learning strategy for optimizing the utilization and allocation of ephemeral resources in the cloud, with regards to the SLA. A dynamic safety margin is set for each resource metric. The RL strategy learns from prediction errors and improves the height of the safety margins. As a result, SLA violation penalties are reduced, and Cloud providers can increase potential savings.

- Cost Models: Karampaglis et al. [112] introduced the first proposal for a bi-objective query cost model suitable for queries executed over a multi-cloud environment. It successfully provides estimates of both the expected running time and the monetary cost associated with a query. The cost model can also be applied to DAG data flows, apart from QEPs, which make it easily extensible.

5.8 Serverless for MOO

Multi-objective query and resource optimization have been the basis of numerous research works, which will be analysed in the following sections. The majority of these works are based on IaaS platforms where the user's decisions on resources are by themselves an optimization problem.

As FaaS built on some of the cloud's existing advantages, its unique characteristics would provide some optimization opportunities if it was selected as an execution environment for MO query and resource optimization. Some of them are:

- Even more cost, energy savings: Serverless characteristics means that tackling MOO problems in FaaS creates more optimization opportunities in regards to monetary cost and energy consumption. Users will pay less, as they will only pay when their code (e.g. a query) is actually executed and never for idle resources. Cloud providers will save energy by allocating their resources in an event-driven way, and will also benefit economically as the fact that their resources are released when idle is a big step for optimization of resource allocation and scheduling.

- Handling of massively parallel tasks: Massively parallel applications are a good fit for serverless platforms, as they exploit serverless autoscaling character, and requests need not communicate with each other. When considering data management and querying in the cloud, distributed database queries using distributed set processing can be a good fit, as their tasks can be parallelized and almost no data shipping between serverless functions will be required.

- Response to interactive analytics: Serverless can also be a good fit for data intensive applications like interactive analytics, and subsequently is suitable for optimizing interactive queries [23].

However, the limitations of serverless mentioned in chapter 2 means that the serverless paradigm also comes with significant obstacles, concerning MOO in data management scenarios:

- Data Shuffling: Data shuffling is an essential part in distributed query processing, being the most commonly used communication pattern to transfer data across stages. In serverless, as direct communication and data transfer between functions is difficult and its stateless compute units have no local storage, intermediate data between stages needs to be stored in storage systems like Amazon S3, and accessing it each time comes with an extra overhead increasing latency and cost. Slow data shuffling is a big obstacle for query execution, and limits query optimization opportunities.

- Lack of control over resources: FaaS platforms are easy to use, as users only have to worry about their application and code. However, the fact that they have no control over computing resources and task scheduling means that the optimization challenges they offer are also abstracted by the users and rely on FaaS vendors methods.

- Short term tasks only: Serverless functions have a short lifespan. In AWS Lambda **[26]**, this limit is 15 minutes. As a result, they cannot be used for long-running tasks like complex analytical queries, limiting even more the queries that can leverage the serverless paradigm.

- Security Issues: Security is also an open issue in serverless. Existing platforms present some vulnerabilities due to increased co-residency, as well as higher leak probability because of increased network communication. This happens because data is disaggregated from functions and stored in a serverless data store.

In conclusion, serverless might be gaining ground, however at the moment the areas where FaaS performs better and overcomes traditional, VM-based IaaS are still limited. Multi-objective query and resource optimization in IaaS has been a hot research topic with numerous contributions.

5.9 Conclusion

In this chapter, multi-objective optimization was defined and numerous interesting works concerning multi-objective query and resource optimization were presented. In the following chapter, our vision of a state of the art architecture for a query optimizer will be presented, with regards to some of the works presented here, and in chapter 7 we will give an analytic description of our proposal.

Chapter 6

6.Proposal - System architecture

6.1 Vision - From Cloud to Serverless

In this chapter, we will propose an architecture for efficient and multi-objective query processing in the cloud. Firstly, we will present our vision of an architecture for efficient data processing in the cloud, with its core being a multi-objective and flexible query optimizer, which can be broken down in four objectives:

- Tackle query optimization problems efficiently in a cloud environment, with respects to its unique characteristics. As a result, we tackle the optimization problem with multiple optimization objectives, related to the use of computing and storage resources on demand. These objectives may include query execution time, monetary cost, energy consumption.

- Generalize query optimization in the cloud, by combining state of the art optimization methods (discussed in Chapters 4 and 5), and adapting them in a massively parallel environment, in contrast with relational DBMSs, which had been their primary use case environment until now.

- Explore the opportunities offered by a fast emerging subset of the cloud, FaaS platforms. No serverless query optimizer has yet been implemented. Serverless NoSQL databases like Amazon DynamoDB may use indexing for optimizations [110], however they operate without a query optimizer. Couchbase [12] recently introduced one of the first cost-based optimizers for NoSQL document databases [13] but does not operate serverlessly. We aim to simulate a serverless environment and evaluate the opportunities and challenges that occur in regards to multi-objective query optimization.

- Come up with a generic and flexible optimizer solution, not limited to a certain technique or implementation system, but being able to take into account different optimization objectives, different optimization techniques, and even different query execution environments (e.g. Cloud provider, Cloud federations, Serverless provider).

In the following subsections, we will propose an architecture for data and query processing, including the implementation of certain parts of the aforementioned query optimizer, in the direction of the described vision. In the following chapters we will also present our actual contribution and experimentally evaluate its proof of concept.

6.2 Proposed System Components

In this section we will present the components of the proposed system architecture, alongside examples of frameworks and platforms that could be used for its implementation. Our main focus when envisioning the system involved traditional IaaS cloud platforms, however we took many decisions based on serverless features, to enable extensibility to different platforms.

6.2.1 Deployment

Figure 16. Grid ‘5000

The core of our proposed system architecture will be Grid ‘5000, a large-scale and flexible platform for experiment-driven research in all areas of computer science with a focus on parallel and distributed computing, including Cloud, HPC and Big Data and AI. Our computing and storage resources will be deployed through Grid ‘5000 where numerous cores and computing nodes are available and grouped in homogeneous clusters, featuring various technologies (e.g. PMEM, GPU, SSD). We will use Grid 5000 in the manner of a cloud IaaS platform, each time renting the resources we are going to need for our experiments.

6.2.2 Storage

In the cloud computing system we envision, data will be stored in a distributed data store, with a constant number of nodes used for storage. We also aim to logically disaggregate computation and storage, like in the Serverless paradigm, but without necessarily preventing physical colocation. [14]. Storage and computation resources can scale independently, providing the system with the necessary flexibility introduced in serverless, and colocation can increase performance.

In our proposed system, data is stored in a constant number of datanodes inside an HDFS filesystem. Logical disaggregation is achieved by separating the process of scaling resources for storage and computation. Physical colocation is achieved by initializing our computational units in the same nodes used for data storage. We deploy the number of nodes we want through Grid ‘5000, and we choose HDFS to store them in order to benefit from its distributed nature, speed and resiliency through replication. It is also the main framework used for storage by most big tech companies offering cloud services.

6.2.3 Computation

As we mentioned before, deployed resources will be used both for computing and for storage purposes. We envision a system where running an application would come with automatic allocation and deallocation of a number of nodes for computation specified by the user. As this is difficult to implement on a deployment platform like Grid ‘5000, we rented a number of nodes and each time used as many as we wanted for computation, assuming that our vision can be extended to a cloud platform where the first step of renting all the possible nodes for computation beforehand would be avoided. With the use of Apache Yarn, the Hadoop ecosystem resource scheduler, the executors are initialized in different nodes of our cluster.

6.2.4 Data Processing

The big data processing framework of our proposed architecture is Apache Spark, with Spark executors being the computational unit of our architecture. Depending on the number of Spark executors we want to use for the execution of a query, we utilize the same number of nodes for computation (each executor allocated in a different node). All the Spark executors have the same characteristics (e.g. vcores), so the described process is similar to allocating a number of instances with the same characteristics in a platform like Amazon EC2.

We chose Spark as our data processing framework due to its distributed and massively parallel features, its compatibility with HDFS, its SQL interface which we used, and its wide adoption in industry.

6.2.5 DBMS

For database management, we store our dataset tables as Hive tables in Apache Hive, the data warehouse introduced in chapter 3. We use Spark for processing, and its SQL interface, Spark SQL, to query our data. Thanks to our choice to store our data as Hive tables, SparkSQL can use the HiveMetastore to get the metadata of the data stored in HDFS. This metadata enables SparkSQL to do better optimization of the queries that it executes, with Spark being the query processor.

6.2.6 Query Optimization

As we use Spark SQL for querying our data, query optimization inevitably relies on Spark SQL optimizer, the Catalyst, introduced in chapter 4.

6.2.6.1 Cost Model

One of Catalyst’s main limitations is that it remains largely rule-based, having a very simple cost model covering only specific operations on data. This is why a cost model for Catalyst was recently proposed **[3]**. Our vision includes integrating multi-objective optimization techniques that have successfully been used in RDBMSs, into Spark which is a parallel processing framework. To be able to use a fully cost-based optimizer without leaving Spark, we propose a system using the proposed cost model for query optimization. To support our proposal, we reimplemented the cost model, by coding it in a Python script. In the next chapter we will further describe how we reimplemented it, and in Chapter 9 we will experimentally evaluate it.

6.2.6.2 Multi-objective optimizer

Using the aforementioned cost model allows us to propose a system where optimization becomes fully cost based, and we will be using an optimizer accurately estimating the execution time of alternative query plans, selecting the optimal one. However, our envisioned system is not limited to single-objective optimization, therefore we will also propose a formula for estimating the monetary cost of a query plan in our architecture, basing our estimations on the hourly cost of using Amazon EC2 generic medium instance. This will allow us to use a bi-objective cost model, enabling us to tackle MQ problems.

6.2.6.3 MPQ

In chapter 5 we presented MQ, and also introduced another set of multi-objective query optimization problems, MPQ, where each query is defined as a function of parameters. MPQ is a technique that can prove much faster than MQ, as the fact that the query optimization process happens before runtime in a preprocessing time, means there is zero optimization overhead during runtime. However, it is challenging to address MPQ problems accurately, as the parameters picked have to be sensitive, and also an exhaustive estimation of the cost of all relevant plans has to be done in the preprocessing step, in order for the selection of the Pareto-optimal plans to be relevant. MPQ also requires all the queries received as input to obey to a certain template (e.g. being of the form SELECT * FROM TABLE t WHERE pred, with pred being the only varying parameter of the query).

Our proposed optimizer is a flexible one, aiming to use different optimization methods. It is able to consider MPQ, alongside traditional MQ. We propose and argue that this is possible with the cost model we are using, and evaluate it for certain queries using a script calculating and comparing all relevant query plans. In our proposed optimizer architecture, the decision maker will have the choice of using MQ or MPQ, depending on his use case.

6.2.7 Serverless

In $6.1, we mentioned the vision of a first serverless query optimizer, which can alternate between different execution environments, in our case being cloud or serverless. Our contribution will not include any serverless components, however our system architecture and the use of the Hadoop ecosystem makes it easily extensible, and leaves the door open for validating it in a serverless environment in the future. In chapter 2 we mentioned Pocket [21], an ephemeral data store for serverless data analytics. This contribution leveraged Apache Crail [39], an I/O architecture tailored to best integrate fast networking and storage hardware into distributed data processing platforms, in order to deal with communication between functions. Crail has also been the core of Spark-IO [19], a fast distributed storage system, containing two I/O plugins: a shuffle engine and a broadcast module. Both plugins inherit all the benefits of Crail such as very high performance (throughput and latency) and multi-tiering (e.g., DRAM and flash).

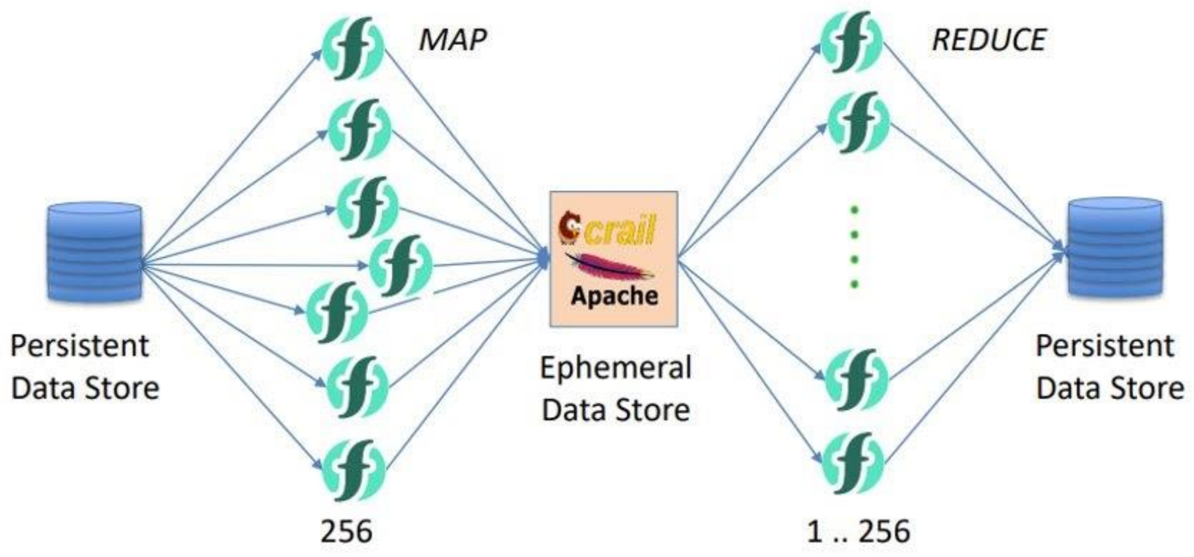


Figure 17. Apache Crail

Using Spark-IO enables us to simulate a serverless-like environment, as the data sharing limitation is controlled by Apache Crail, the compute and storage services are disaggregated, and the resources used for computation are able to scale up and down on demand. Therefore, by using Apache Crail as an ephemeral data layer and basing the monetary cost estimations of a query plan on values for example from Amazon S3, which is a BaaS platform, we can show proof of concept for multi-objective query optimization in a serverless platform. As we discussed in Chapter 2, depending on the application, choice of IaaS or FaaS as an execution environment might be more efficient. A generic query optimizer could also allow the user to select the execution environment, and execute his/her application in a Cloud (e.g. EC2), or Serverless (e.g. Lambda) environment.

6.3 System Architecture

Visualized, our proposed system architecture can be summed up in the following figure.

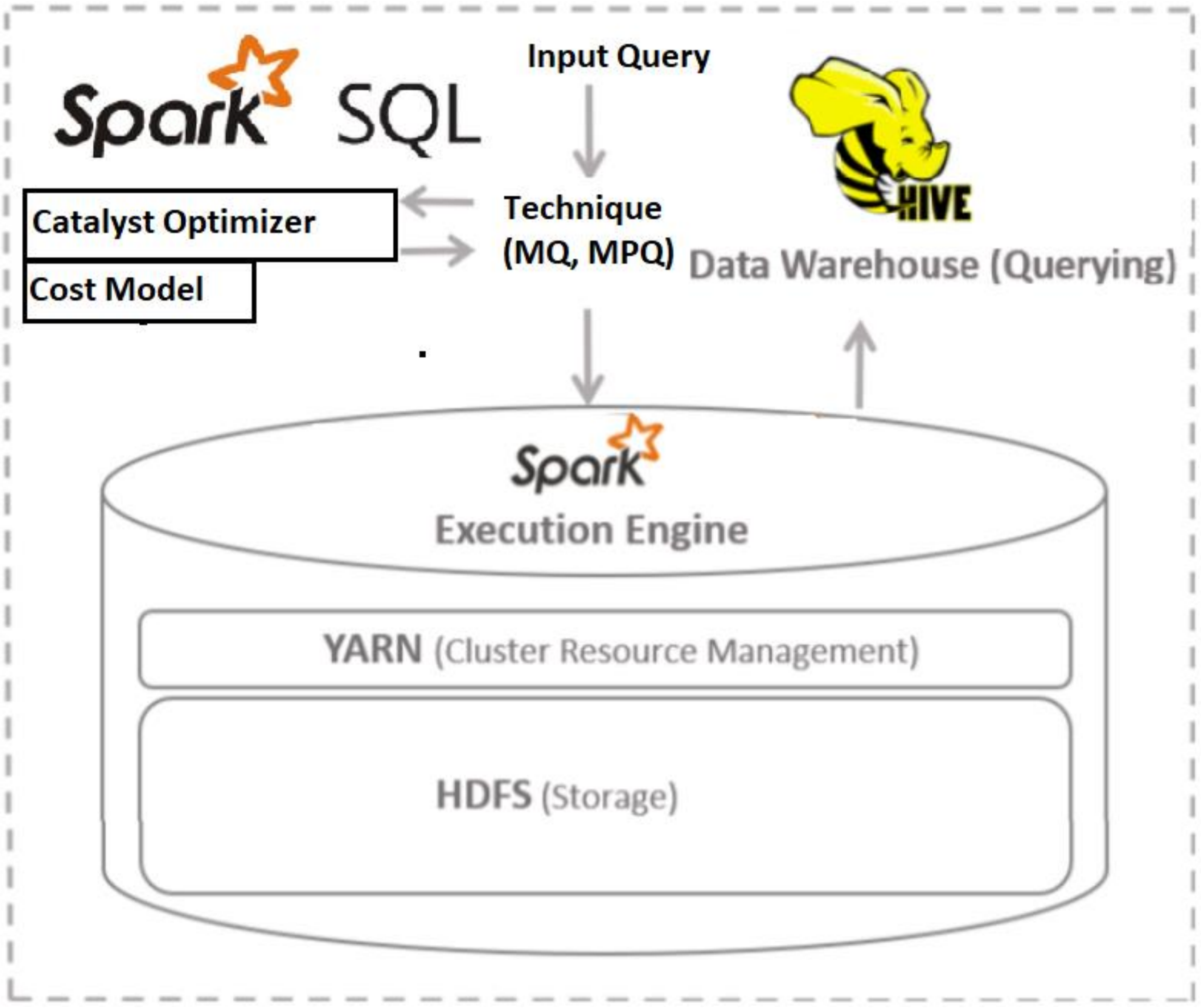


Figure 18. Proposed system architecture

In short, a user submits a query written in SQL. The query is parsed and translated by the Spark SQL optimizer. The optimizer uses the reimplemented cost model to select the optimal physical execution plan. Then, the user gets to choose how many Spark executors he will allocate for the execution of the query. Selection of many executors will mean the execution is quick, but more costly, whereas a low number of executors will mean slower but cheaper execution. After the application configuration is specified, YARN schedules the tasks that have to be done, and allocates resources for each executor to operate. After the execution environment has been set up, physical operations described in the physical plan are applied to the data. Data is stored in the form of Hive tables, which are stored in HDFS.

If we aimed one step further and implemented the more generic query optimizer, which could alternate between a cloud and serverless environment, Crail would be integrated in our system as an in-between layer on top of HDFS and under Spark, as shown in the figure below.

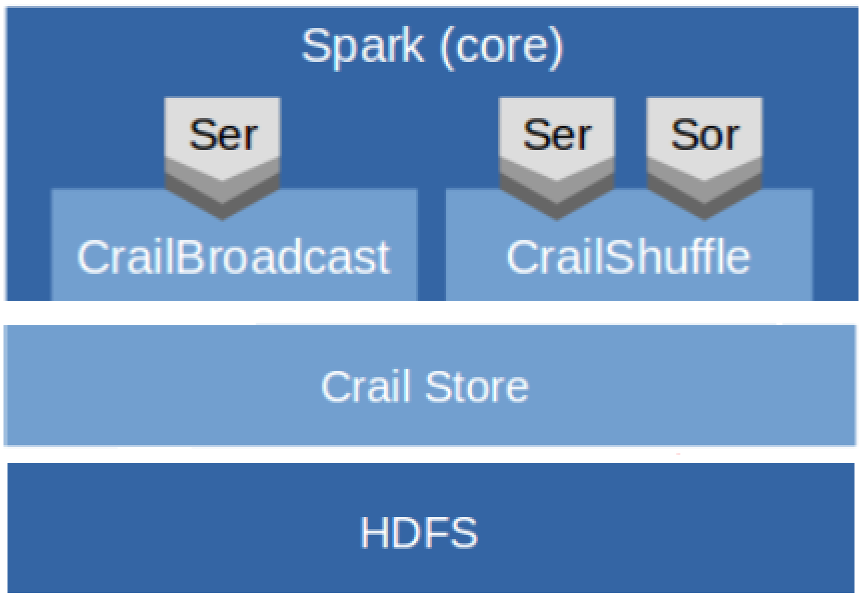


Figure 19. Integrating Crail into our architecture

This addition to our architecture, apart from enabling us to consider the serverless factor, would also speed up execution, as it provides significant optimizations in the way local and remote storage resources are used in a high-speed network deployment. The whole architecture would look like this:

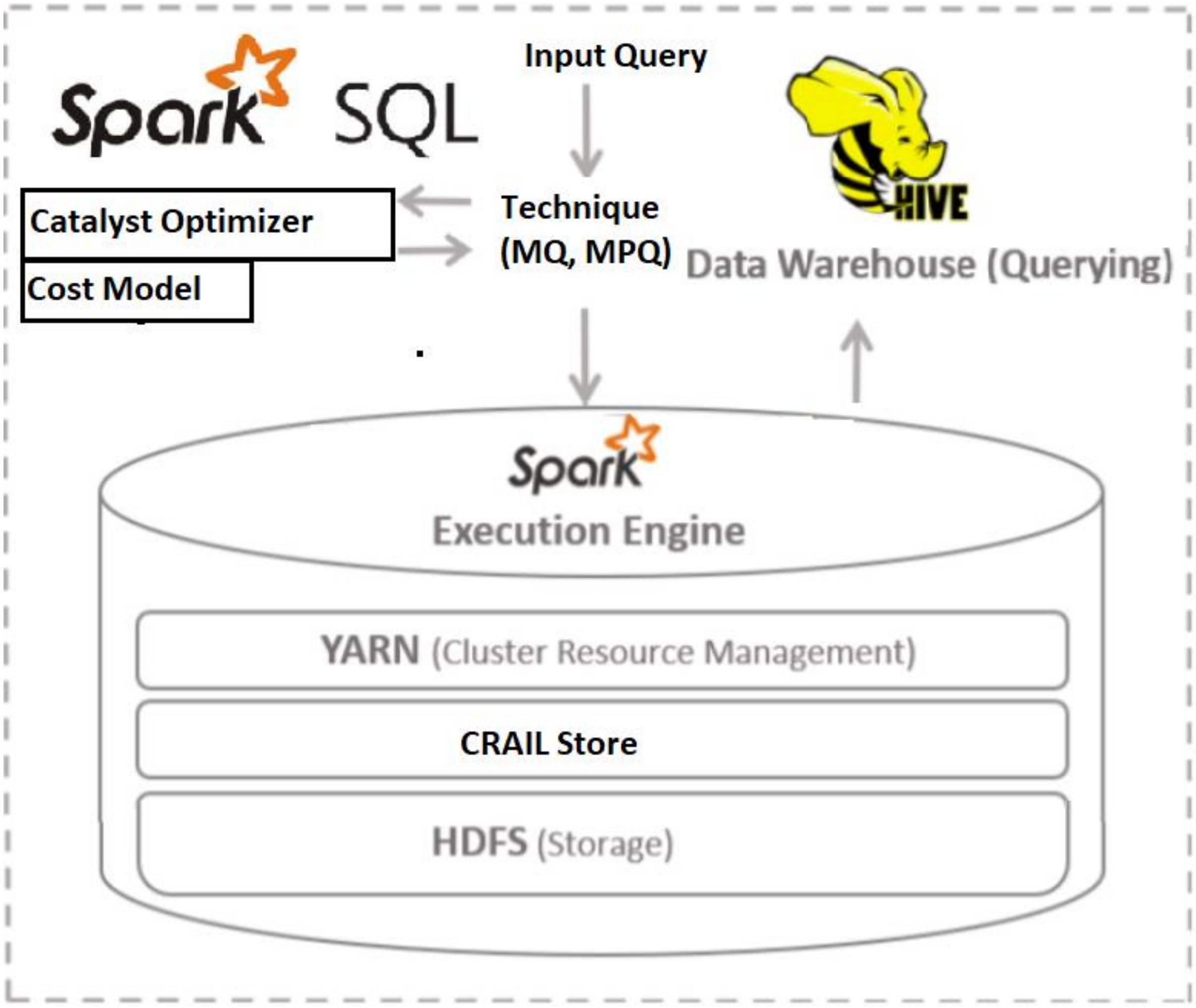


Figure 20. Proposed Architecture with Crail

6.4 Contributions

Our contributions towards implementing the proposed system include the following:

- Re-implementation of the aforementioned Spark SQL cost model, in order to work with a fully cost based query optimizer. As reconstructing the cost model was a challenging job and required large scale experiments and comparisons in Spark, we will describe our implementation in detail in the following chapter and validate it in Chapter 8.
- A formula for estimating the monetary cost of Spark SQL queries, based on the pricing scheme of Amazon EC2. The multi-objective nature of the query optimization problems in the described system was validated by the creation of Pareto fronts for given queries.
- We applied parametric query optimization on complex queries. We created scripts, operating on top of our cost model to run the preprocessing step needed and exhaustively compare all relevant query plans. In the end, a Pareto Plan Set of optimal plans is produced. By doing so, we aimed to prove that parametric query optimization does not have to be limited to query optimization in relational databases, and evaluated this with illustrative examples (Chapter 9).

6.5 Conclusion

This chapter described our proposed architecture for a query optimizer framework, and also listed the contributions that we made for this purpose. In the next chapters, the contributions are described in detail and validated.

The experimental part of our thesis consisted of six steps. First, we implemented from scratch the state of the art Spark SQL cost model($7.2) [3]. Then, we evaluated the estimation and prediction accuracy of the cost model with experiments on a large scale cluster using Spark (Chapter 8). The next step was to introduce a second objective, monetary cost and present a formula to estimate each query execution plan cost based on the Amazon EC2 pricing scheme ($8.2). After those steps, we have an operational multiobjective cost model, and proceed to integrate and validate MPQ (Chapter 9).

Chapter 7

7.Cost Model Implementation

7.1 Introduction and Motivation

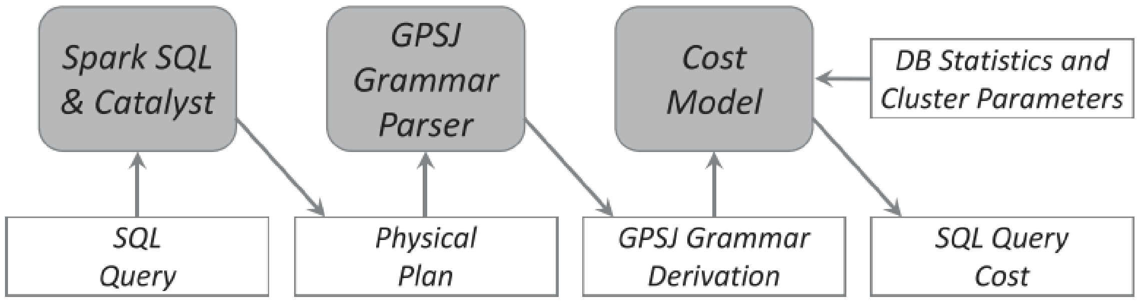


Figure 21. Description of Cost Model process

In this section, our first contribution is described, which is the aforementioned Spark SQL cost model, which we implemented in a Python script. Its key components will be presented, alongside parts of our code. Its differences with the existing Spark SQL optimizer, Catalyst, and its cost model will be presented to show why we preferred it and finally, challenges that occurred during its implementation will be noted. The next chapter contains the experimental evaluation of the cost model.

Before describing the cost model, it is relevant to state our motivation for its use. We mentioned that we aim to achieve multi-objective query optimization, by using different techniques in a Cloud-like environment, with Spark being the data processing framework. In order to do this and use such techniques, we need to be able to produce all relevant query plans for a given query and compare them for all optimization goals. This is impossible to do with the Spark SQL optimizer, which only uses a very basic cost model and each time deterministically produces one query plan, without it being able to provide full estimations for the execution time of a query. The cost model is necessary to be able to consider and evaluate which one of these plans is optimal, by exhaustively comparing all relevant plans and producing a set of Pareto optimal plans. Thus, we decided to reimplement Baldacci and Golfarelli's Spark SQL cost model which we regarded as a very promising contribution, being able to estimate the actual execution time of a query with about 20 percent error. Moreover, Spark is a flexible and easily extensible and configurable framework, which allows us to produce more query plans, apart from the one selected as optimal, which will prove very useful in the experimental evaluation of the cost model.

It is important to note that the contribution is not independent from Catalyst, but can be used to improve it, if integrated in the optimizer itself. Our re-implementation of the cost model helped us provide theoretical estimations for different query plans with high accuracy, but as we implemented a cost model and not a whole optimizer, optimization has to be done manually, if the query plan that is considered optimal by the cost model, can also be produced in Spark SQL. Integration of the cost model in the Catalyst is left as a future plan, like shown in the figure. However, proof of the potential improvements it can offer is provided.

7.2 Cost Model and Catalyst

As we mentioned in Chapter 5, Spark SQL's core is Catalyst, its extensible query optimizer. Although it makes some cost-based choices, it is mainly a rule based optimizer with a very simple cost model. Catalyst collects tables and columns statistics for optimization, but it can only make cost estimations for specific subparts of the query execution, like join selection. The Spark SQL cost model that was discussed in the previous chapter provides more than Catalyst when it comes to estimating the execution time for a query execution plan, as highlighted by the table below.

	Catalyst Optimizer	Baldacci SQL Cost Model
Query types	ALL	GPSJ(no UNION ALL, OUTER JOIN)
Cost based join selection	✓	✓
Tables & Columns statistics	✓	✓
Considers cluster parameters	✗	✓
Based on system disk & network performance	✗	✓
Analytic estimation of QET	✗	✓

Table 1. Comparison of Catalyst and Spark SQL Cost Model

First of all, the cost model only covers the class of Generalized Projection, Selection, Join (GPSJ) queries, which are a subset of SQL queries, as use of operators like UNION ALL or OUTER JOIN are not supported. Furthermore, it benefits from statistics collected from the database tables and table columns like the Catalyst. Apart from that, it bases its cost estimations on cluster parameters like the disk access time and the network throughput of the system, while also considering CPU times for data serialization and compression are also considered. Finally, it also takes into account parameters of each Spark application that may influence query execution time, which include the number of Spark executors and number of cores within each executor.

The cost model is capable of analytically estimating the execution time of the five essential RDD transformations that may occur in Spark SQL GPSJ queries:

- Table scan (SC): The table is fully scanned.
- Table scan & broadcast (SB): The smallest table involved in a broadcast joined has to be broadcast before the join operation occurs
- Shuffle hash join (SJ): This join strategy involves moving data with the same value of join key in the same executor node.

- Broadcast hash join (BJ): Also known as map-side join. In this join strategy, a copy of one of the join relations is sent to all the worker nodes, to save shuffling costs. It is useful in cases where a large relation is joined with a smaller one.

- Group by (GB)

Estimation of the aforementioned RDD transformation costs is enabled by calculating each action's subtasks, like time needed to read, write, shuffle and broadcast data. A physical query execution plan consists of several RDD transformations, and it can be represented as a tree whose nodes apply operations to one or more input tables. The Figure below shows a possible physical query execution plan for the following query, which contains 3 joins.

TPC-H QUERY 3
SELECT l_orderkey, o_orderdate, o_shippriority, sum(l_extprice)
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < date'1995-03-15' AND l_shipdate > date'1995-03-15'
GROUP BY l_orderkey, o_orderdate, o_shippriority

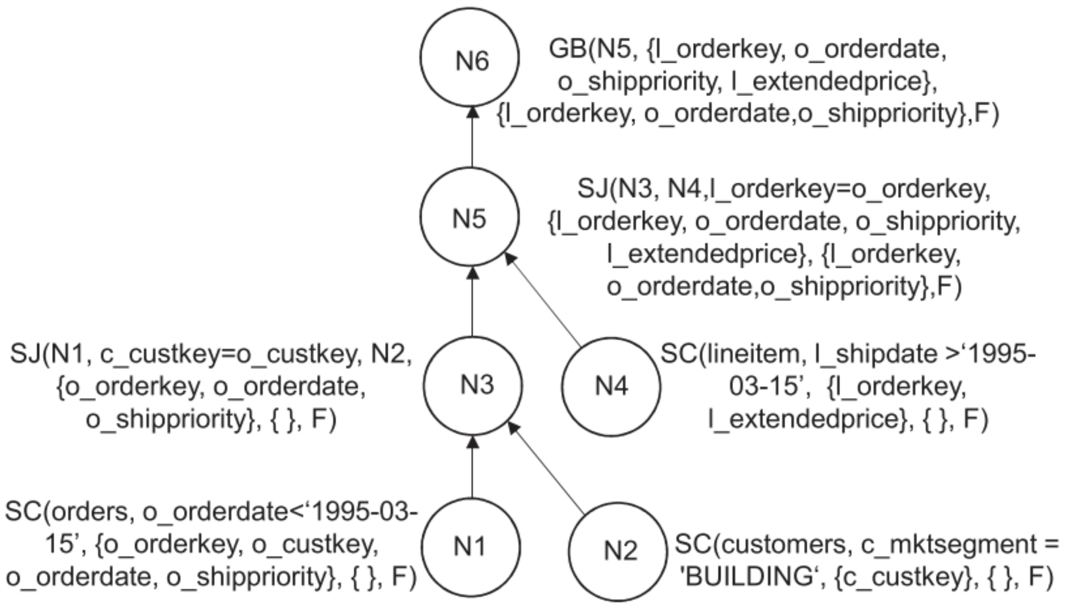

N6
GB(N5, {l_orderkey, o_orderdate, o_shippriority, l_extendedprice}, {l_orderkey, o_orderdate,o_shippriority},F)
N5
SJ(N3, N4,l_orderkey=o_orderkey, {l_orderkey, o_orderdate, o_shippriority, l_extendedprice}, {l_orderkey, o_orderdate,o_shippriority},F)
SJ(N1, c_custkey=o_custkey, N2, {o_orderkey, o_orderdate, o_shippriority}, { }, F)
N3
N4
SC(lineitem, l_shipdate >'1995-03-15', {l_orderkey, l_extendedprice}, { }, F)
SC(orders, o_orderdate<'1995-03-15', {o_orderkey, o_custkey, o_orderdate, o_shippriority}, { }, F)
N1
N2
SC(customers, c_mktsegment = 'BUILDING', {c_custkey}, { }, F)

Figure 22. Query execution plan in the form of a tree.

As we can see in the figure, scan operations are always leaf nodes in the execution tree, as they deal with the physical storage where the tables are located. Join operations are inner nodes of the tree, and group by, if present, will be the last action to be carried out, that is why it will be located in the root of the tree. The full cost of the query execution plan is calculated by summing up the time needed to execute each node of the tree.

7.3 Implementation

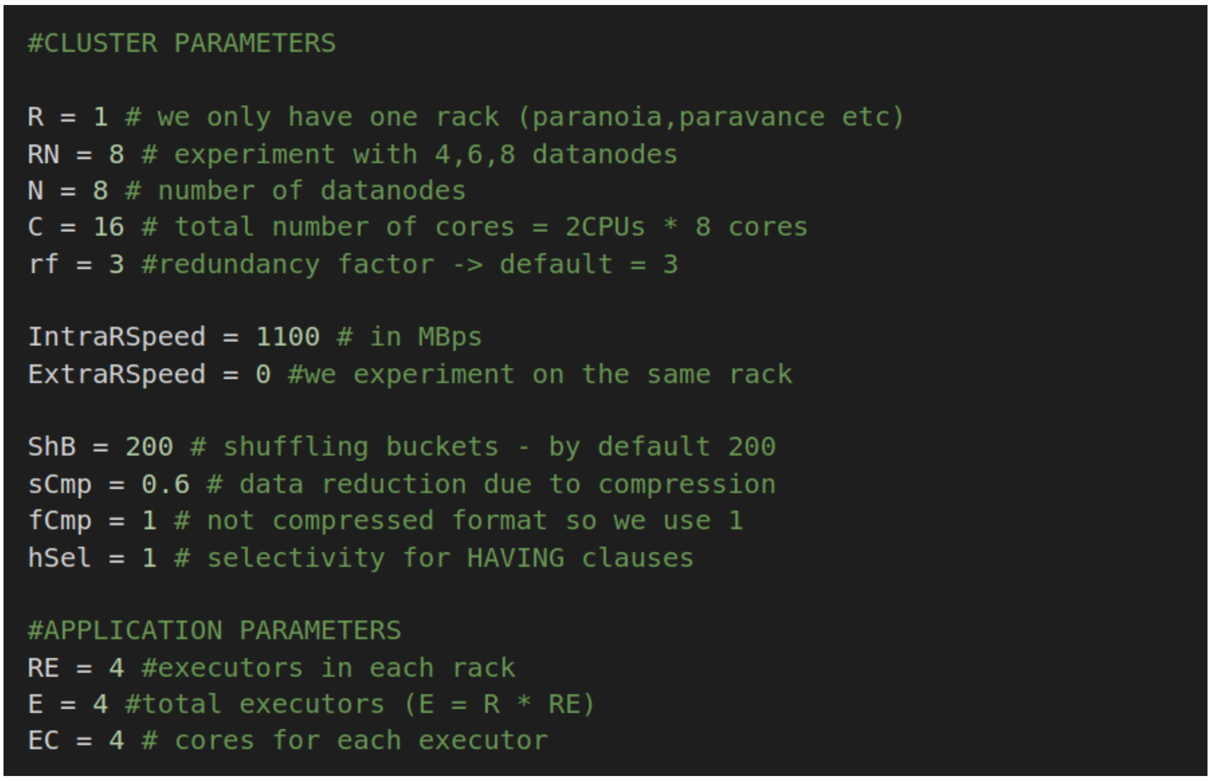
7.3.1 Configuration Parameters

We implemented the cost model on a Python script, using the formulas described in the paper for each one of the five RDD transformations, which will be included below. We tuned the model by using the relevant aforementioned cluster, application and dataset characteristics.

For our experiments and estimations we used the TPC-H benchmark dataset [10], scaled to 10,50 and 100GBs. TPC-H is a decision support benchmark that consists of business-oriented queries and concurrent data modifications. It illustrates decision support systems examining large volumes of data, includes high complexity queries, and answers critical business questions.

We repeated experiments in clusters of different sizes, of 4, 6 and 8 nodes with the same system characteristics, in order to have a homogeneous environment. In all the experiments, HDFS replication factor was 3 (rf=3).

In order for the cost model to be tuned, certain parameters needed to be known, which we will define here:



```
#CLUSTER PARAMETERS

R = 1 # we only have one rack (paranoia,paravance etc)
RN = 8 # experiment with 4,6,8 datanodes
N = 8 # number of datanodes
C = 16 # total number of cores = 2CPUs * 8 cores
rf = 3 #redundancy factor -> default = 3

IntraRSpeed = 1100 # in MBps
ExtraRSpeed = 0 #we experiment on the same rack

ShB = 200 # shuffling buckets - by default 200
sCmp = 0.6 # data reduction due to compression
fCmp = 1 # not compressed format so we use 1
hSel = 1 # selectivity for HAVING clauses

#APPLICATION PARAMETERS
RE = 4 #executors in each rack
E = 4 #total executors (E = R * RE)
EC = 4 # cores for each executor
```

Figure 23. Cluster and Application Parameters

Nodes (N): overall number of nodes composing the cluster. In our case, it is also the number of HDFS datanodes where our dataset is stored

Cores (C): number of CPU cores available on each node. In our case C=16 as we worked with nodes having 2 CPUs Intel Xeon E5-2630 v3 and 8 cores/CPU.

Data factor (DF): size of the TPC-H dataset used in a specific experiment. The value of this variable is translated to the total size of the dataset used in GBs.

rf (Redundancy factor): HDFS redundancy factor, the amount of times data is replicated across our cluster

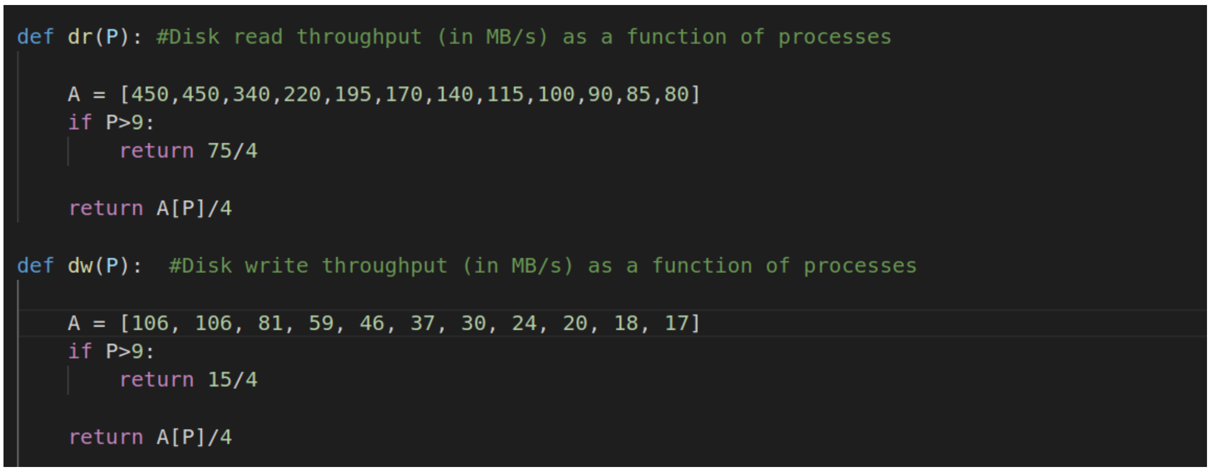

```python
def dr(P): #Disk read throughput (in MB/s) as a function of processes

    A = [450,450,340,220,195,170,140,115,100,90,85,80]
    if P>9:
        return 75/4

    return A[P]/4

def dw(P):  #Disk write throughput (in MB/s) as a function of processes

    A = [106, 106, 81, 59, 46, 37, 30, 24, 20, 18, 17]
    if P>9:
        return 15/4

    return A[P]/4
```

Figure 24. Disk read and write throughput values

Disk read throughput (δr): disk read throughput as a function of concurrent processes.

Disk write throughput (δw): disk write throughput as a function of concurrent processes

Network throughput between nodes (ρi): network throughput between nodes as a function of concurrent processes.

7.3.1.1 System Characteristics Calculation

In order to obtain the disk read and write throughput, the dd command was used, a Linux command which monitors the reading and writing performance of a Linux disk device. The commands used and the results can be seen below for the same test in the cluster we used and in my local PC. These results are valid for the systems used when no other processes are running.

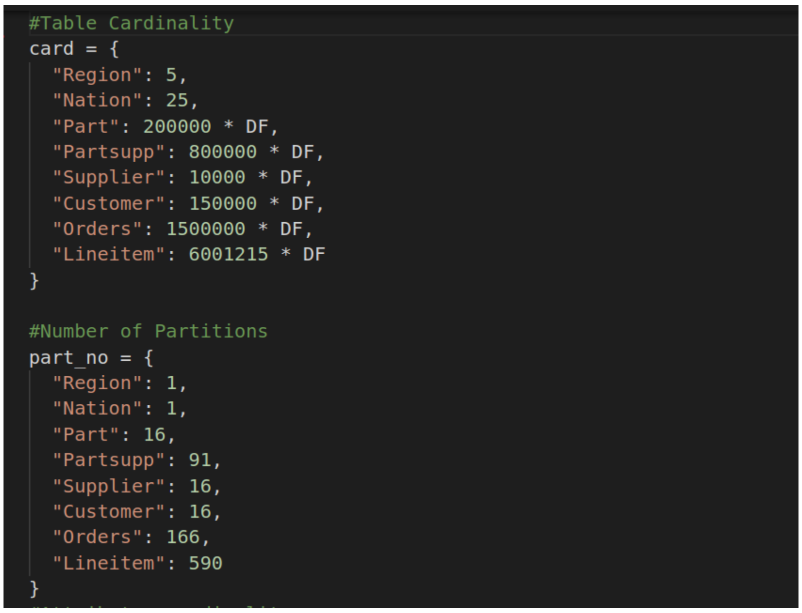
```
Write test with DD
$ dd if=/dev/zero of=benchfile bs=4k count=200000 && sync; rm benchfile
Local (PC) results : 399 MB/s, 481 MB/s, 277 MB/s
Cluster (Grid '5000) results : 106 MB/s, 105 MB/s, 106 MB/s, 106 MB/s

Read test with DD
$ dd if=/dev/zero of=/dev/null && sync
Local (PC) results : 598 MB/s, 575 MB/s, 610 MB/s
Cluster (Grid '5000) results : 447 MB/s, 452 MB/s, 451 MB/s, 457 MB/s
```

The actual values used in the cost model are significantly smaller, as CPU times for data serialization/deserialization and compression in Spark are taken into account by the cost model. By applying some tests to calculate the overhead of these operations for data of a certain size, we make the assumption that the disk read and write throughput, when serialization and compression is taken into account, is 25% of the values calculated with dd.

As for the average network throughput, it was calculated using the linux command iperf3, which is used for performing real-time network throughput measurements. It is a powerful tool for testing the maximum achievable bandwidth in IP networks. We ran the command simultaneously on two nodes, one being the server and one the client, sent 10 GBs of data from one node to another, and calculated the network throughput to be around 9.42 Gbps.

7.3.1.2 Application and Data Parameters

Apart from the cluster parameters, which are constant for each set of experiments, each Spark application is configured with a different number of:
Executors (E): the number of worker processes in charge of running individual tasks in a given Spark job. They run on worker nodes, in our case in HDFS datanodes. The executors are in charge of carrying out the operations on data.
Executor cores (EC): number of cores for each executor allocated to Spark application

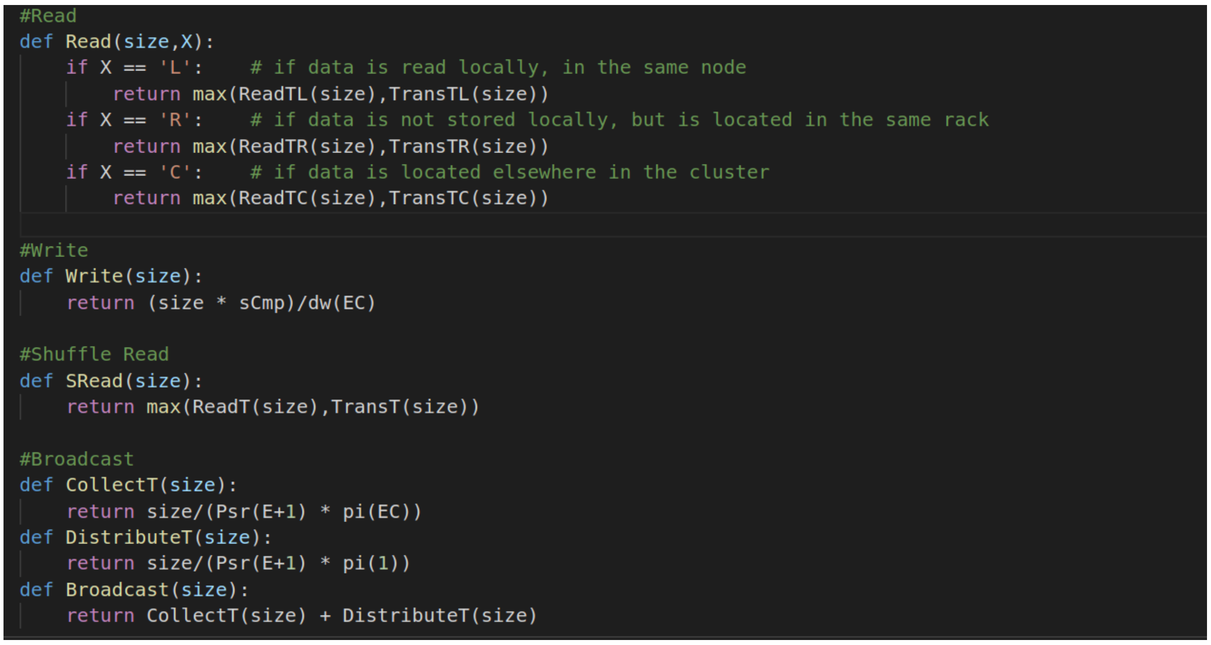


```
#Table Cardinality
card = {
  "Region": 5,
  "Nation": 25,
  "Part": 200000 * DF,
  "Partsupp": 800000 * DF,
  "Supplier": 10000 * DF,
  "Customer": 150000 * DF,
  "Orders": 1500000 * DF,
  "Lineitem": 6001215 * DF
}

#Number of Partitions
part_no = {
  "Region": 1,
  "Nation": 1,
  "Part": 16,
  "Partsupp": 91,
  "Supplier": 16,
  "Customer": 16,
  "Orders": 166,
  "Lineitem": 590
}
```

Figure 25. Dataset characteristics

Finally, some additional characteristics of the dataset needed to be collected, such as table and attribute size, cardinality, and number of table partitions. We obtained these characteristics by running specific queries to obtain the size and cardinalities of tables and attributes in Spark, and some of its values can be shown in the figure above..

7.3.2 Cost Model Basic Bricks



```python
#Read
def Read(size,X):
    if X == 'L':    # if data is read locally, in the same node
        return max(ReadTL(size),TransTL(size))
    if X == 'R':    # if data is not stored locally, but is located in the same rack
        return max(ReadTR(size),TransTR(size))
    if X == 'C':    # if data is located elsewhere in the cluster
        return max(ReadTC(size),TransTC(size))

#Write
def Write(size):
    return (size * sCmp)/dw(EC)

#Shuffle Read
def SRead(size):
    return max(ReadT(size),TransT(size))

#Broadcast
def CollectT(size):
    return size/(Psr(E+1) * pi(EC))
def DistributeT(size):
    return size/(Psr(E+1) * pi(1))
def Broadcast(size):
    return CollectT(size) + DistributeT(size)
```

Figure 26. Python functions estimating cost of basic operations on data

Before going deeper in the cost model, deconstructing the internals of a Spark application is necessary. First of all, data is distributed across the cluster in a set of Resilient Distributed Datasets (RDDs). Each RDD is a collection of immutable and distributed elements, named partitions, that can be processed in parallel. These partitions can either come from a storage (in our case HDFS) or be the result of a previous operation (held in application memory).

Spark operators are classified in Transformations and Actions. Transformations are carried out in-memory on each RDD partition, whereas actions either return a result to the driver, or imply a shuffling to combine data distributed in different RDD partitions. Tasks are distributed over the cluster and executed in parallel. Given a SQL query, Catalyst is responsible for translating it into a set of jobs. Precisely modeling Spark actions and transformations would be very challenging, so the cost model focuses on a set of basic bricks that determine transformations and actions cost. for which it provides cost estimates. These basic bricks are the following four:

- **Read**: the time needed to read an amount of data. As Spark applies locality principles, it loads RDD partitions from the "closest" available position. As a result, reading time varies depending on the location of the RDD partition, which can either be the local disk, the disk of another node in the same rack, or the disk of a node in another rack. In our case, we did not experiment with nodes located in different racks.

- **Write**: the time needed to write an amount of data in memory. Apart from the disk write throughput, write latency also depends on the number of executor cores, as more executor cores usually speed up the time needed for an executor to write to the disk.

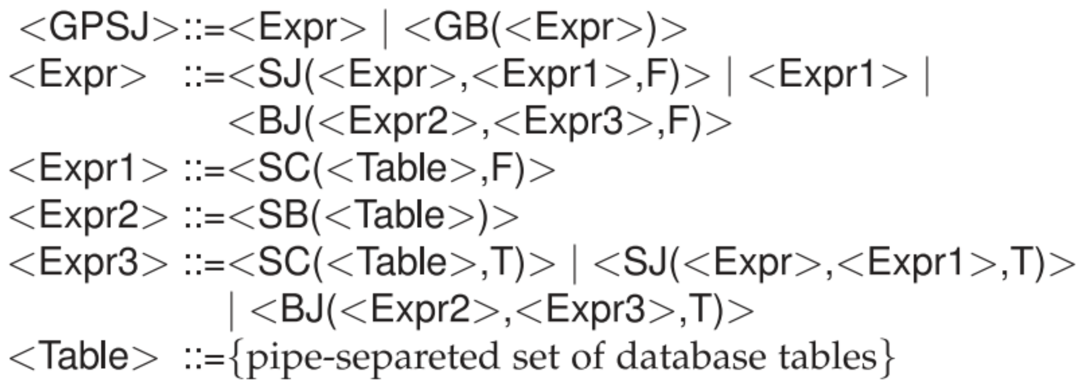
- **Shuffle Read**: the time needed to read a data bucket and transmit it to the executor which will handle it. When performing a shuffle read, Spark generates #SB tasks which are in charge of processing the #SB buckets previously created during the shuffle write phase (e.g. the result of a shuffle join). The reading of a data bucket and its transmission to the executor, happen in pipeline, so the loading time is computed as the maximum of the times needed to carry out the two operations

- **Broadcast**: The time needed to collect and distribute (broadcast) the partitions of an RDD to the available executors for further processing.

Implementations of these functions in our Python script can be shown in the figure above (Figure 26), although not all sub functions involved are shown.

7.3.3 RDD Transformations

Spark provides a rich set of operators to manipulate RDDs. Five Spark RDD transformations can be applied to GPSJ queries, which we mentioned in $8.2 (Table scan (SC), table scan and broadcast (SB), shuffle join (SJ), broadcast join (BJ), and group by(GB)). A Backus-Naur representation of the grammar of GPSJ queries in regards to these five RDD transformations can be shown below (Figure).

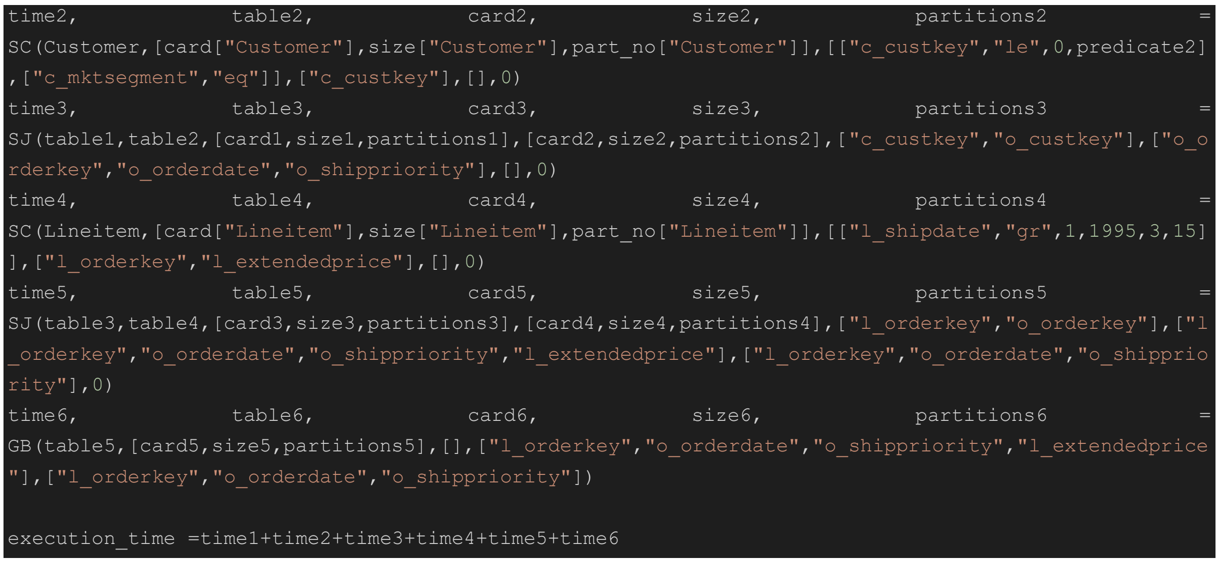


```
<GPSJ>::=<Expr> | <GB(<Expr>)>
<Expr>   ::=<SJ(<Expr>,<Expr1>,F)> | <Expr1> |
            <BJ(<Expr2>,<Expr3>,F)>
<Expr1> ::=<SC(<Table>,F)>
<Expr2> ::=<SB(<Table>)>
<Expr3> ::=<SC(<Table>,T)> | <SJ(<Expr>,<Expr1>,T)>
            | <BJ(<Expr2>,<Expr3>,T)>
<Table>  ::={pipe-separeted set of database tables}
```

Figure 27. Backus-Naur representation of the grammar of GPSJ queries [83]

Consequently, our cost model can provide estimates for these 5 RDD transformations, by using the basic bricks described in the previous subsection. Each RDD transformation is modeled as a function, which receives a data table as an input, as well as the table's cardinality size and partitions, as well as any filtering and selection predicates. It returns the time needed to execute the transformation, the columns of the table returned, the cardinality and the size of the table, as well as the number of partitions of the output. To achieve this, we also implemented functions that return the cardinality reduction of a join or a projection based on the formulas described in the original paper.

```
time1,              table1,             card1,              size1,              partitions1              =
SC(Orders,[card["Orders"],size["Orders"],part_no["Orders"]],[["o_orderdate","le",1,predicate,3,15]],
["o_orderkey","o_orderdate","o_shippriority"],[],0)
```



```
time2,            table2,            card2,            size2,            partitions2            =
SC(Customer,[card["Customer"],size["Customer"],part_no["Customer"]],[["c_custkey","le",0,predicate2]
,["c_mktsegment","eq"]],["c_custkey"],[],0)
time3,            table3,            card3,            size3,            partitions3            =
SJ(table1,table2,[card1,size1,partitions1],[card2,size2,partitions2],["c_custkey","o_custkey"],["o_o
rderkey","o_orderdate","o_shippriority"],[],0)
time4,            table4,            card4,            size4,            partitions4            =
SC(Lineitem,[card["Lineitem"],size["Lineitem"],part_no["Lineitem"]],[["l_shipdate","gr",1,1995,3,15]
],["l_orderkey","l_extendedprice"],[],0)
time5,            table5,            card5,            size5,            partitions5            =
SJ(table3,table4,[card3,size3,partitions3],[card4,size4,partitions4],["l_orderkey","o_orderkey"],["l
_orderkey","o_orderdate","o_shippriority","l_extendedprice"],["l_orderkey","o_orderdate","o_shipprio
rity"],0)
time6,            table6,            card6,            size6,            partitions6            =
GB(table5,[card5,size5,partitions5],[],["l_orderkey","o_orderdate","o_shippriority","l_extendedprice
"],["l_orderkey","o_orderdate","o_shippriority"])

execution_time =time1+time2+time3+time4+time5+time6
```

Figure 28. Series of RDD operations using our cost model script

Running the transformations included in the physical plan of a query using our script, looks like the figure above (Figure 28). The query plan of the figure is the one shown in the physical tree in $8.2 for query 3 of the TPC-H benchmark. To obtain the execution time of the query plan, we just have to add the execution time of each RDD transformation included in it.

7.4 Conclusion

This chapter described in detail the cost model that we reimplemented, providing insights on its architecture and screenshots of the code that we wrote to implement it. After implementing it and successfully providing some query execution time estimations that made sense, we had to test it by comparing its estimations for a given query physical plan, with the actual execution time of running this query plan in Spark SQL. The experimental evaluation of the cost model will be described in detail in the following chapter.

Chapter 8

8.Cost Model Evaluation

8.1 Experimental Setup & Assumptions

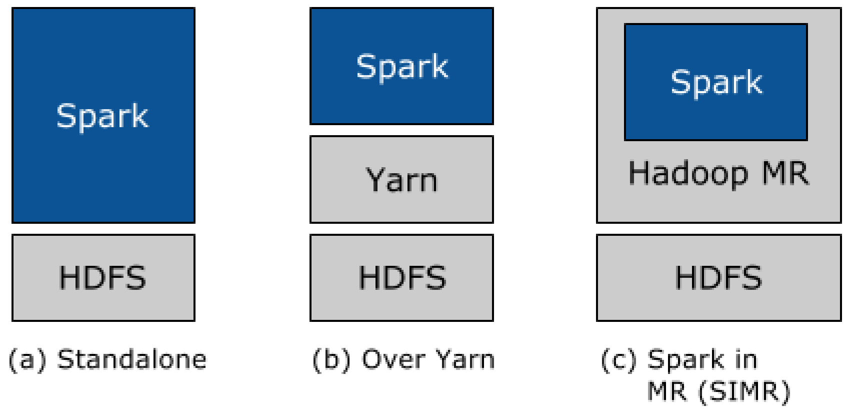


Figure 29. Different Spark Architectures

We conducted our experiments using Apache Spark's SQL component, in an architecture similar to b from Figure 29. Before we present our experimental results, it is important to note the assumptions that our cost model does, and the assumptions that we made in our experimental process. First of all, the cost model is not generic, as it only covers the class of GPSJ queries. Furthermore, data in the TPC-H datasets is uniformly distributed, so our cost model makes the assumption of uniform data distribution, for example when estimating the selectivity of a predicate. Finally, we assumed that each node of the cluster can host one Spark executor, and we did not experiment with multiple executors co-located in one node. This assumption was taken from the serverless paradigm, where each function invoked is independent and not co-located with each other. This assumption also highlights the parallelization benefits offered by adding more nodes for computation in a cluster. Finally, the results shown in some diagrams might be the mean execution time of several queries, however each query is executed independently as the cost model simulates single-query optimization and not multi-query optimization. Multi-query execution might be more relevant in real life use cases, where queries happen on hot data and caching improves mean execution time. However, the status quo in query optimizer cost models is to assume that every query starts with a cold cache [8], which is what Baldacci & Golfarelli cost model does too. Assuming that data is read from disk and not from memory might not be relevant in general and lead to inaccurate cost estimations in some cases, however this will not be addressed in this contribution.

Finally, in our experiments we observed a Spark-Yarn overhead, which is the time needed for Yarn to set up the cluster and allocate resources to each Spark executor. We calculated the overhead by running some queries of negligible complexity, and found it to be around 21 seconds. We tuned the experimental results with the cost model estimations by removing 21 seconds from each experimental value. We did this so that we can focus on the “clean” query execution time, and observe patterns, as well as error estimations better.

Apart from that overhead, we also assume that we excluded any exogenous factors affecting the cluster performance. As the clusters of Grid5000 are used by many students every day, we conducted most of our experiments at night in a cluster with very low network traffic, which we validated with the use of the command oarstat, which informs us of how many users have rented resources in this cluster.

A query optimizer cost model can be evaluated in two ways. One way is to evaluate the accuracy of execution time estimations ($8.2.1), to validate that for a given query plan, the cost model makes an accurate calculation. The other way is to judge it solely by its prediction, and selection of query plan. In other words, produce all relevant execution plans for a query and evaluate experimentally that the fastest plan the cost model shows is actually the fastest one ($8.2.2), regardless of how accurate the execution time value is by itself.

8.2 Cost Model Experimental Evaluation

8.2.1 Estimation Accuracy

As we mentioned before, the implemented cost model covers the class of GPSJ queries. In our first experiments, we wanted to observe the impact of join relation sizes in the execution time. For this experiment, we used a default cluster size of 8 nodes (N=8), a default number of 4 Spark executors, as well as 4 executor cores (EC=4) and a TPC-H dataset of data factor 100 (100GBs). The tables used were the following:

Region	**0.5 KB**
Nation	**2 KB**
Customer	**2.4 GB**
Part	**2.5 GB**
Partsupp	**12 GB**
Orders	**17 GB**
Lineitem	**77 GB**

Table 2. TPC-H Table Sizes

We first considered a join between the two smallest tables, which would obviously have negligible execution time. However, it helped us calculate the overhead caused by Spark and Yarn while the Spark driver negotiates resources with Yarn. We also considered two “medium” joins, between tables Part and Partsupp, and Orders and Customer. These joins include one table with similar size (2.4 and 2.5 GBs), and another which is 5 GBs bigger in the case of Orders being used. This helped us gain conclusions regarding the effect of increasing the size of one table in a join on execution time. Finally, we considered 2 joins with the bigger table Lineitem. In one case, we joined it with the smallest possible table, Region, in a classic broadcast join case, and in the second case we joined it with the largest available table, Orders, to observe the highs that the execution time would reach. In the two joins where

Region is involved, the query plan chosen by Catalyst involves a broadcast join, and in all other cases it involves a Shuffle Hash Join. This allowed us to evaluate accuracy for query plans containing both types of joins.

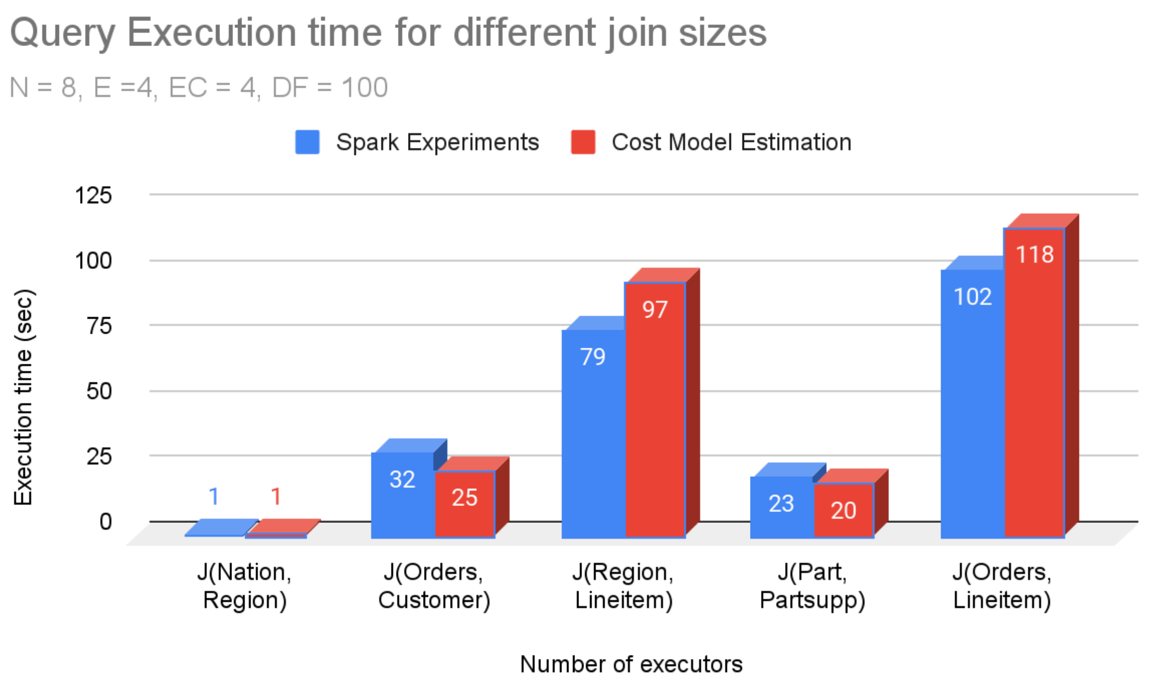


Figure 30. Execution times for different join sizes

The first observation from this set of experiments is that the cost model is not inaccurate. The mean error percentage of these experiments is 16.1%, which is within the upper bound of 20%, which is the average error percentage of the cost model calculated in the paper.

The next observation is that there is an obvious dependence between the join size (sum of table sizes involved) and the execution time. To clarify this, we decided to run the same join experiment for different data factors, to observe if we can measure the impact of the table size. In these experiments, we used the two biggest tables with 3 different data factors (DF=10,50,100).

Orders	**1.7,8.5,17 GB**
Lineitem	**7.7,38.5,77 GB**

Table 3. Experiment tables sizes.

Figure 30. Execution times for different join sizesFigure 30. Execution times for different join sizes

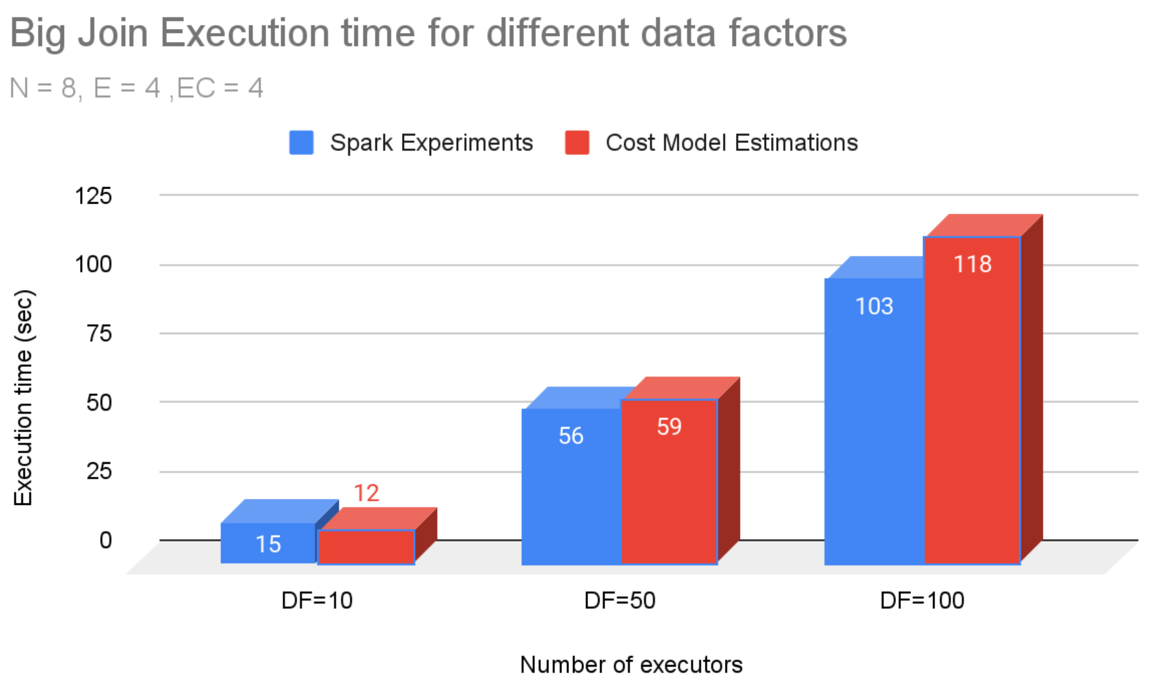


Figure 31. Join Query Execution times for different data factors

After this experiment, it is quite obvious that join size has a linear relation to execution time. For every more GB that has to be joined, around 1 seconds of overhead is added. However, we should note that this mainly applies to shuffle hash joins, as if one table is very small, broadcasting it might save significant time and break this linear relation. Accuracy comparisons between different execution plans for the same query will be more thoroughly tackled in the following subsection. The error percentage in these experiments is 14.2%, remaining in acceptable levels. We observe that as the join size increases, our cost model tends to slightly overestimate the execution time of the query. Basically, the cost model assumes an absolute data factor execution time linear dependence (equal to ExecTime = DF * 1.18 for this specific query and environment), whereas the experiments show that although the trend is similar, doubling the data factor does not exactly double execution time.

A possible explanation is that there is some constant overhead in some Spark action or transformation whose influence is less when the execution time of this action increases. After all, we mentioned in the previous chapter that not all components of each RDD transformation are calculated by the cost model, but only some key operations (the basic bricks of the cost model), so this simplification probably slightly affects accuracy. This overestimation probably happens in the cost estimation of joins. After the following experiments, we will isolate the parts of the cost model and evaluate their behaviour in comparison to Spark experiments.

After observing the impact of the join sizes, our next experiments involved some high complexity queries coming from the TPC-H benchmark. We chose 4 queries for our experiments, Queries 2,3,5 and 10. Our choice was mostly based on the structure of the queries, as all of them are queries involving a number of joins, tackle different tables from the dataset, and their optimal execution plan differs. 3 of the 4 queries involve a group by operation, so we will get to evaluate this part of the cost model. Query 3 is a great baseline query involving 3 tables and 2 join operations, easily reconfigurable in terms of selectivity of its predicates. Query 10 is like an extended Query 3 considering one more join which is better operated with a broadcast join, and also enables a comparison between the 2. Finally, Query 5 is the most complex query, involving 5 different join operations, considering 6 different tables from the dataset. This query will enable us to test the cost model when it comes to estimating the cost of a query with multiple stages (12 for this query). This is necessary as cardinality and size reductions have to be accurately calculated by the cost model so that it remains accurate in later stages of the query. Query 2 considers 4 tables of the dataset which are relatively small, favoring the broadcast option in many cases, while it also involves a nested query. Although nested queries are not taken into account by the cost model, we calculate its cost independently and add it to the remaining query costs of query 2. In conclusion, these 4 queries have a combination of features that covers small and big numbers of tables and joins, include group by operations, and also have tables of varying sizes favoring different join operators and query plans.

Apart from its query features, they all answer certain business questions from the TPC-H dataset, and can be considered for real life scenarios. This is their description taken from the TPC-H manual:

- Minimum Cost Supplier Query (Q2) This query finds which supplier should be selected to place an order for a given part in a given region.
- Shipping Priority Query (Q3): This query retrieves the 10 unshipped orders with the highest value.
- Local Supplier Volume Query (Q5) This query lists the revenue volume done through local suppliers.
- Returned Item Reporting Query (Q10) The query identifies customers who might be having problems with the parts that are shipped to them.

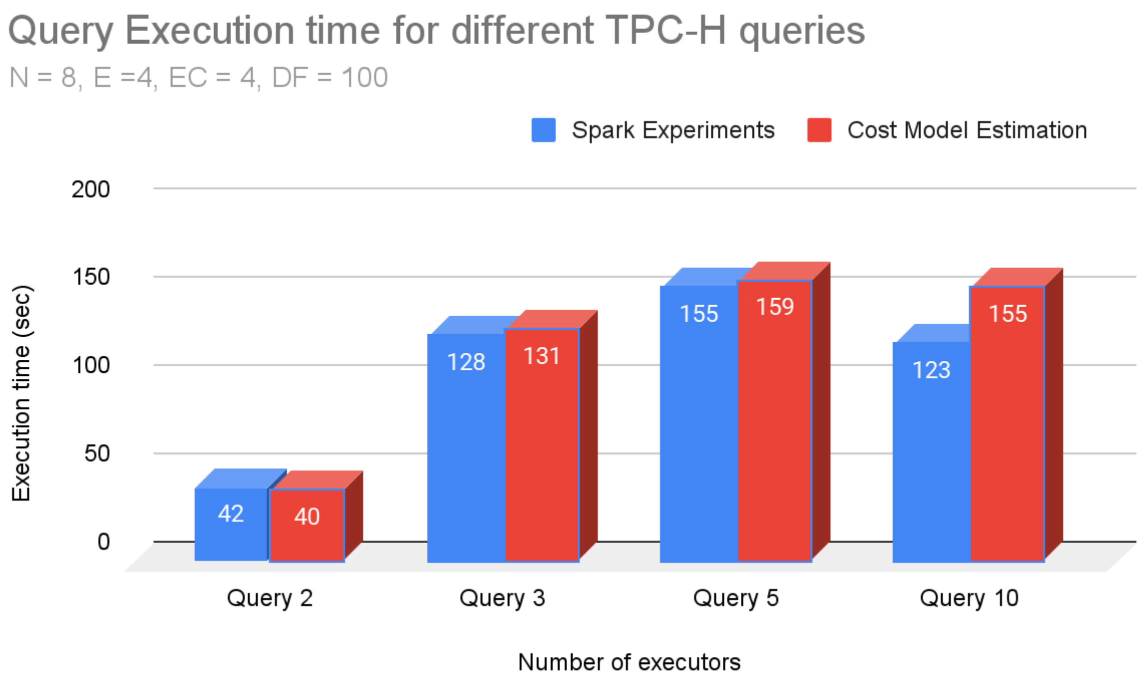


Figure 32. Query Execution times for different TPC-H Queries

The performance of the cost model estimations is ranked positive once more, with the average error percentage being only 7.7%. There is not a specific pattern behind the accuracy of the cost model for this experiment. For 3 of the 4 queries the estimation is very close to the experimental value, but for Query 10, there is a 20% overestimation. Query 10 includes the biggest tables of the TPC-H dataset and 4 join operations respectively.

Before moving on to our experiments, we analyze query 10 and compare each RDD transformation execution time with the cost model estimation, to see where the overestimation happened.

# of execution step	Cost Model Estimation	Experimental Value
1.SC(ORDERS)	22 sec	22 sec
2.SC(CUSTOMER)	7 sec	4 sec
3.SJ(1,2)	4.5 sec	4 sec
4.SC(LINEITEM)	97 sec	87 sec
5.SJ(3,4)	6.6 sec	5.4 sec
6.SB(NATION)	~0 sec	~0 sec
7.BJ(5,6)	15.7 sec	2 sec
8.GB()	3.8 sec	1 sec
Total	155 sec	123 sec

Table 4. Step by step execution times of Query 10

From this table we notice that the main inaccuracies of the cost model come in stages 4 and 7, with 7 being the most significant one, as in general SC operations tend to be accurate. By going deeper in the cost model, we realize that this inaccuracy has to do with an overestimation of the size of the table produced by the join in stage 5, that leads to the broadcast join taking more time than expected. Later on in this chapter, we will examine if this inaccuracy is generalized in other cases and draw further conclusions.

Our next experiments aimed at observing the cost model behaviour for a varying number of Spark executors (E), in a 8-datanode cluster(N=8) and with a default value of 4 executor cores(EC=4). The values shown are the mean execution time of the 4 TPC-H queries we are examining, each one ran 3 times.

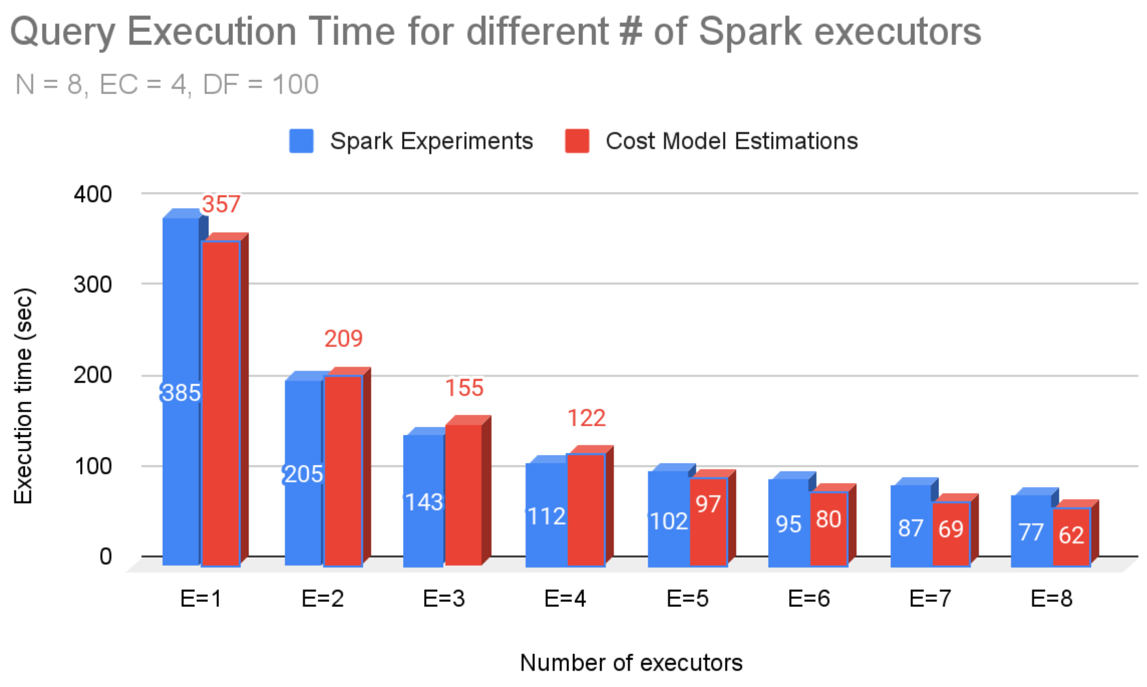


Figure 33. Query Execution times for different number of Spark executors.

The mean error percentage of this experiment was 12.3%, also within the bounds of Baldacci and Golfarelli's experiments. It is also worth noting that they conducted a similar experiment in their work which we show below, in order to compare our performance with theirs. The et(q) line is the estimated execution time from the cost model, and the t(q) line is the actual execution time from their experiments. For their experiment, they used a 7-datanode cluster (N=7), 4 spark executor cores (EC=4), and the same TPC-H data factor (DF = 100).

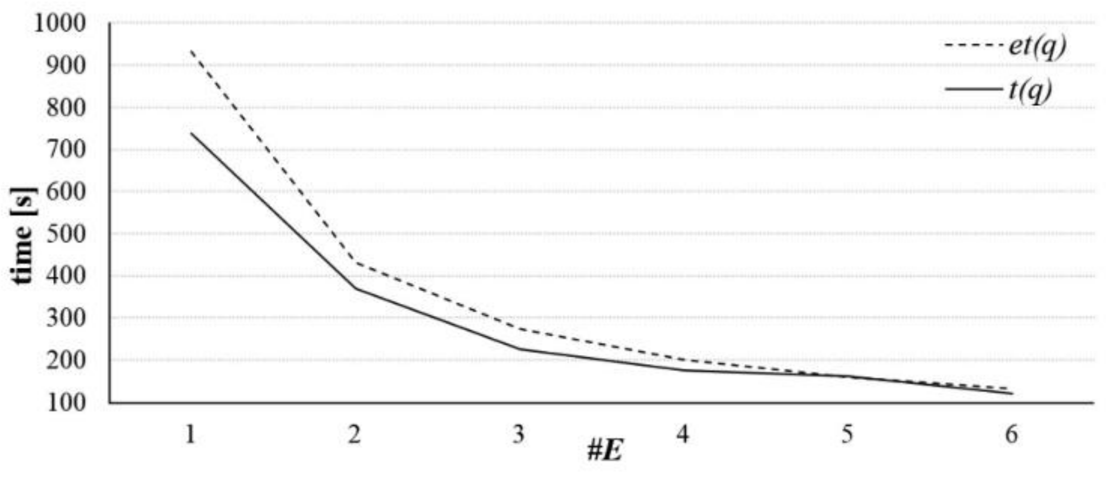


Figure 34. Query Execution times for different number of Spark executors from original paper.

The comparison of the two diagrams shows a similar error percentage, but also some differences. The values of their experiment are double from ours, as they ran different queries and as they state, they used some TPC-H queries as their basis to construct even more complex queries. As a result, it is no surprise that the mean execution value is higher.

Moreover, their cost model tends to overestimate the execution time, at least for low numbers of executors, from 1 to 4. Our cost model's behaviour is different, as it underestimates the execution time for E=1, overestimates for medium E values (2-4), and then for the biggest E values starts underestimating the execution time. This difference in behaviour also partly has to do with the selection of queries, as from the previous experiment where we tackled each query independently, we observed that for more complex joins (involving many and big tables and joins), our cost model slightly overestimates too. If we emphasized on these queries the behaviour would be more similar to theirs. As for the reason behind the differences, a cause could be the divergence between the real disk read and write throughput, as well as the network throughput, as it is obviously impossible to estimate and predict these values with complete accuracy, especially when multiple processes are involved and their value is not constant in each experiment. Finally, differences in theoretical and experimental values for varying E values also arise due to the imperfections of the cost model. As we mentioned before, it has a 20% error percentage and by observing the experimental values of the cost model authors', we see that the cost model is more accurate for E=4,5,6, which are the best choices for their 7 datanode cluster, when it comes to achieving the lowest execution time. Therefore, their cost model seems to be better tuned for optimal cluster configurations, and less accurate for "worse" configurations (e.g. with only 1 executor), where the behaviour is also more unpredictable. The cost model seems to assume a constant rate of execution time reduction each time that an executor is added, while the experimental values show that for few executors, adding one has a significant impact, and for 5 and more executors the decrease rate is much lower.

Our follow-up experiment observed the impact of varying the executor cores in a 8-datanode cluster, this time with a default value of 7 executors (E=7).

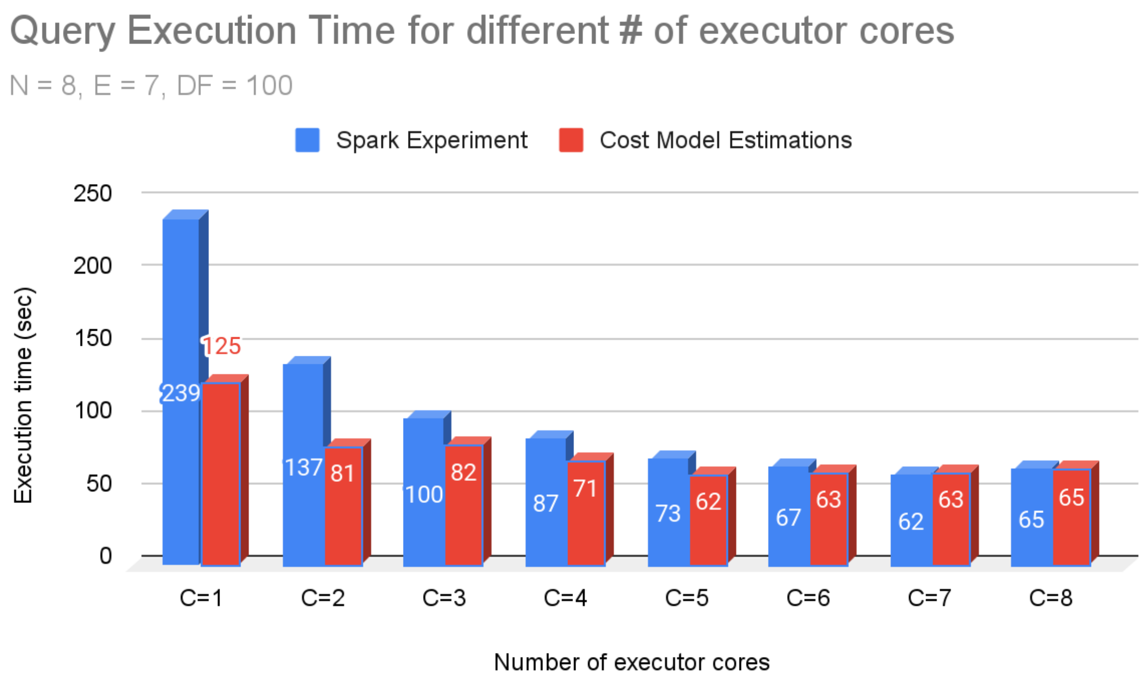


Figure 35. Query Execution times for different number of Spark executor cores

The error percentage in this experiment is higher than the previous ones (27.9%), however it is due to the first two experiments, for 1 and 2 executor cores. Without taking them into account, the error percentage is 10.4%. Before we make our conclusions, we include again the same experiment conducted by the cost model authors. Same as before, they used a cluster with N=8 and a default value of 6 executors (E=6), whereas we used 7. The same dataset is used.

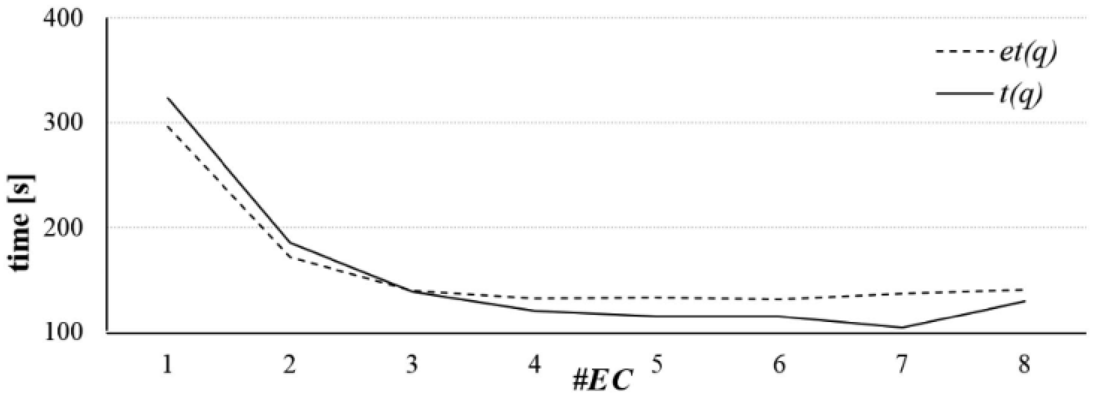


Figure 36. Query Execution times for different number of Spark executor cores from original paper

The first conclusion from the comparison is the same as before, that their experimental values are slightly higher than ours, little less than double. The explanation here is quite different, and has to do with the use of 7 executors. As we also observed in the previous experiment, our cost model underestimates the execution time for a high number of executors, failing to take into account that for many computational resources (executors and cores), HDFS I/O throughput worsens, and the additional overhead results in higher execution times. As a result, after 5 executor cores, a saturation is observed and adding more cores does not result in lower execution time. Slight errors in system features might also have an impact in the inaccuracies observed, and once more, the absolute values are different because the queries considered are different.

Apart from that, the general behaviour this time is similar, with our cost model underestimating the execution time for low EC values, and then stabilising from C=5 and more, and in the end even overestimating a little. Our experimental values are also very similar, with the optimal choice for EC being 7.

In the next experiment, we examined again the impact of the data factor, this time for the TPC-H queries, and for a configuration with default values N=8,E=4,EC=4, for the TPC-H support benchmark dataset scaled to 10, 50 and 100 GBs.

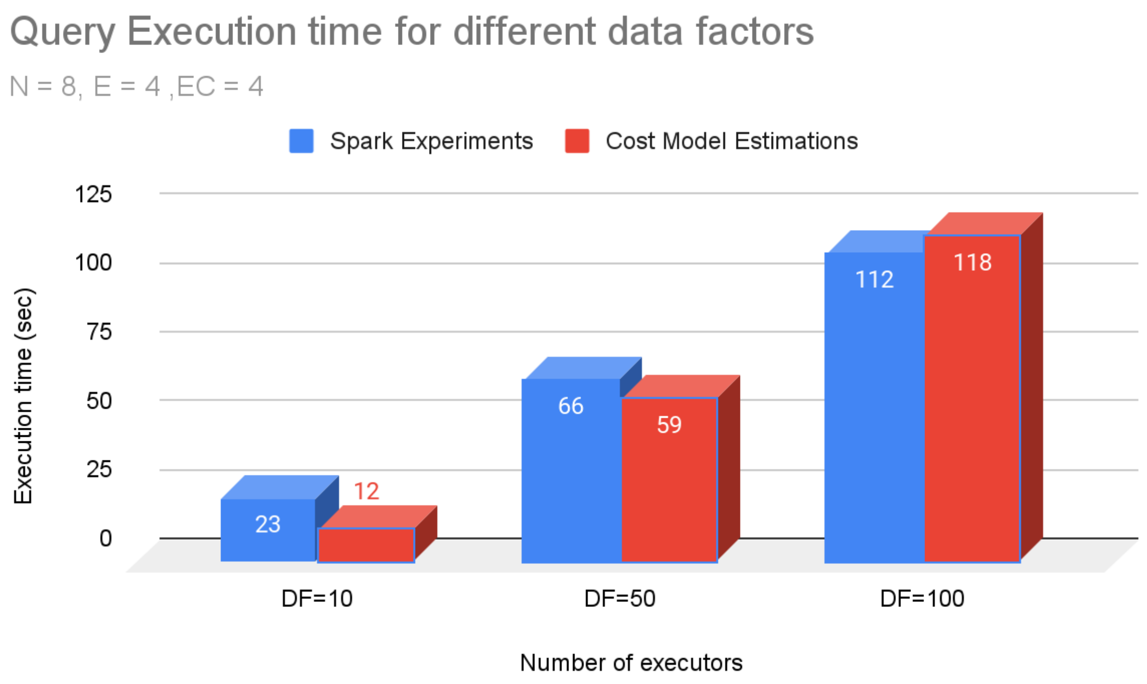


Figure 37. Query Execution times for different data factors.

The results confirm our previous conclusions, with the data size having a linear relation with the execution time. The inaccuracies of the cost model seem to be related with some overhead/optimization in some Spark RDD transformation/action that occurs during joins which it does not take into account. Its estimates are 100% linear, with each 10GBs added to the dataset resulting in 12 seconds of extra execution time.

8.2.2 Prediction Accuracy

IIn this subsection, the prediction accuracy of the cost model is evaluated when it comes to which query execution plan has the lowest execution time for a given query. The cost model can give estimates for 2 different scan RDD operation types (simple scan, scan & broadcast) and join types (shuffle hash join, broadcast join). However, the scan types are dependent on the join types (there is no point in scanning and broadcasting a table if no broadcast join is performed after), so for each join there are 2 different strategies, resulting in 2 different cost models. For a query with n join operations, the cost model can examine 2^n possible join combinations and (n-1)! possible join orderings.

In order to evaluate the prediction accuracy of the query, it is necessary to be able to produce different physical plans for a given query in Spark. We achieved this by disabling broadcast joins in cases we wanted to produce Shuffle join-only queries, using the command spark.conf.set("spark.sql.autoBroadcastJoinThreshold","-1"), by varying the broadcast join threshold of Spark SQL using the command spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 102400000), for example, to set it at 100 MBs. Finally, we also used the command spark.conf.set("spark.sql.join.preferSortMergeJoin", False) to force Spark to prefer Shuffle hash joins over sort merge joins.

Before making our comparisons, it is important to have in mind each join's best uses. When a broadcast join is used, it joins the two tables by first broadcasting the smaller one to all workers, and then evaluating the join criteria with the executor's partition of the other table. It is a very good option for cases where a small table is joined with a bigger one, as broadcast joins require minimal data shuffling. Above a table size, however, bottlenecks in network and memory usage tend to make it a non-reliable choice, in comparison with shuffle-based joins. Shuffle joins, on the other hand, aim to gather the same keys from both tables in the same partition, and then hash the smallest table and perform a hash join within the partition. Shuffle Hash Join's performance is the best when the data is distributed evenly with the key you are joining and you have an adequate number of keys for parallelism.

With that in mind, we will first examine some specific use cases where the one join operator is the obvious choice. In the first case, with a join of a very big table (77GB) with a very small one (1 KB), where broadcast join is the obvious choice, and then with a join of two big tables (77GB & 17 GB) where choosing a broadcast join would result in huge overheads.

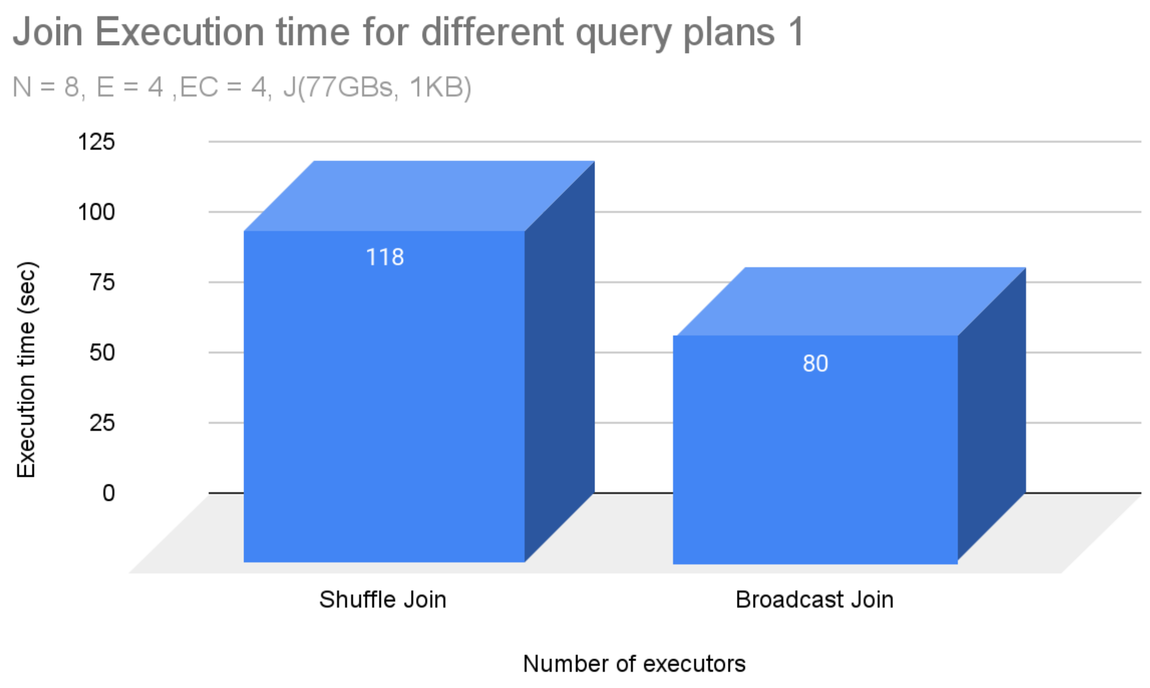


Figure 38. Query Execution times for different query plans in trivial broadcast join case.

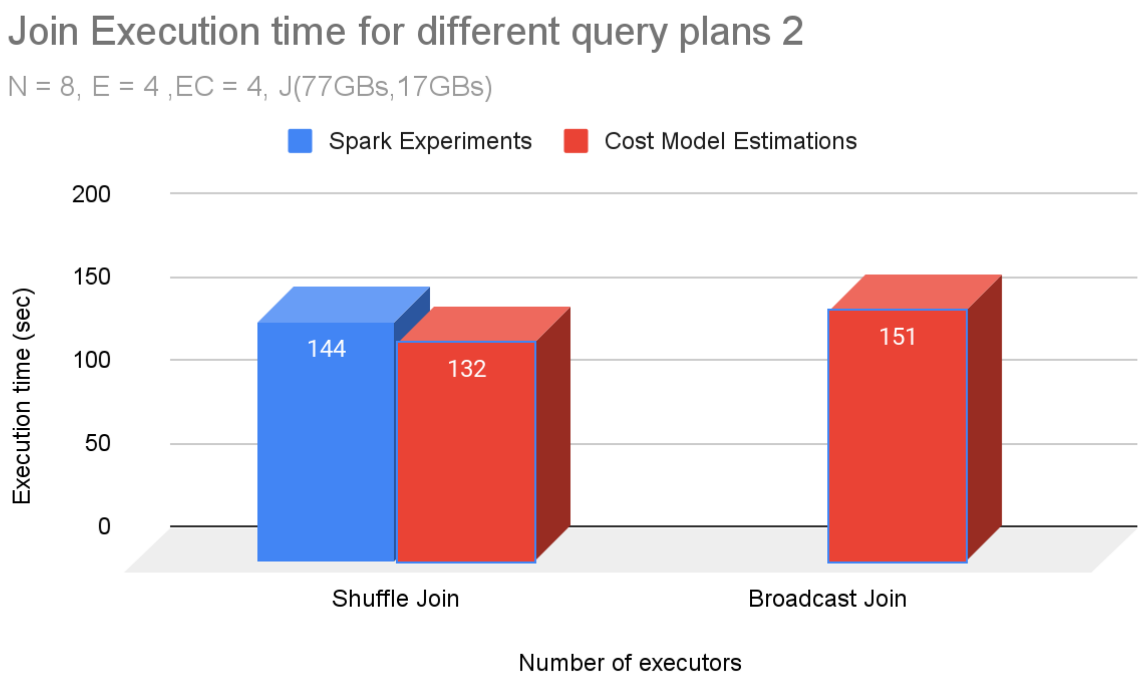


Figure 39. Query Execution times for different query plans in trivial shuffle join case.

The cost model makes the correct prediction in both cases. In the first case, broadcasting the small table saves time from the shuffling phase and is significantly slower than the shuffle join operator. In the second case, broadcasting a 17GB table to every executor is an overkill compared to shuffle joining, which is observed by the cost model estimation. There is no experimental value in the broadcast join case as Spark does not allow a table of more than a specific size to be broadcasted, due to network costs. In conclusion, the cost model makes correct decisions in trivial cases.

Our next step consisted of evaluating different query plans for TPC-H queries, to evaluate the cost model's prediction. We present here 2 cases, of queries 2,3, and 5.

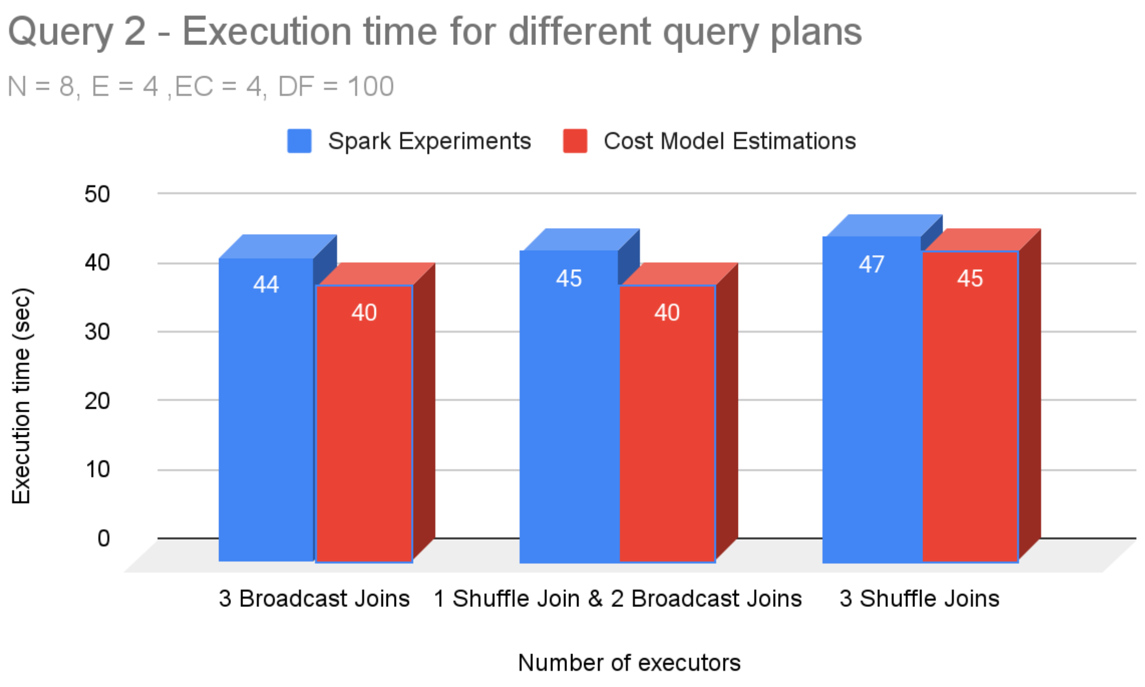


Figure 40. Query Execution times for different query 2 query plans.

Query 2 involves a nested query with 3 joins of very small size, which are all operated with a broadcast join. Outside the nested query, query 2 consists of 3 more joins. The optimal execution plan is to operate the 1st join with a shuffle join operator, and the following 2 with broadcast joins, as due to filtering and projections the size of the RDD has been reduced significantly, and the size of the one table in the joins is quite small. As shown from the experimental values, forcing a shuffle join for all operations results in slightly worse execution time, whereas operating all joins with a broadcast operator is also a good enough execution plan. The cost model gives equally good estimates for the two best execution plans. It is worth noting that in the case of this query, the time differences between the query plans are not significant. However, even like this, the cost model makes a correct prediction. In our next experiment, we produced all relevant plans for query 3 using the cost model.

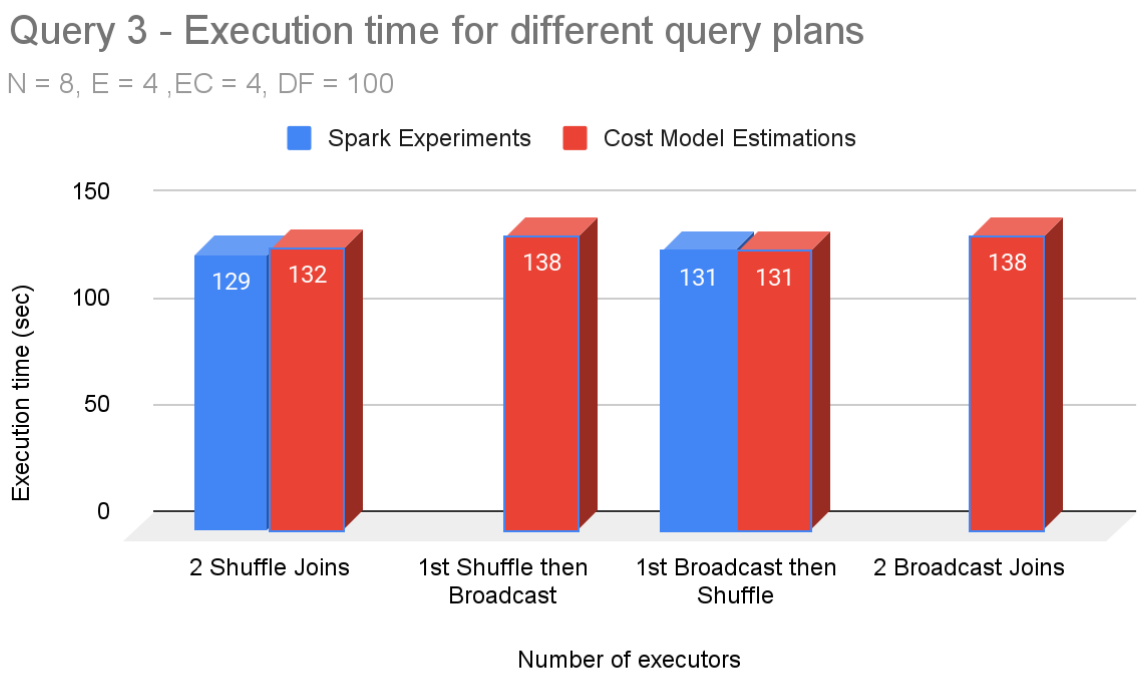


Figure 41. Query Execution times for different query 3 query plans.

The best option experimentally was the use of 2 shuffle joins, as the tables included in the join operations are not very small, with the first join containing tables of 17 GBs and 1.2 GBs, and the second containing a 77GB one and the output of the first join, which is estimated to be around 100 MBs. Once more, the missing experimental values could not be produced, as Spark would not broadcast the 2nd table, due to its size. for the two query plans missing. The cost model correctly points two the two optimal plans with lower values than the other two. It chooses the 2nd best query plan, preferring a broadcast join for the 1st join, however once more the difference between the two optimal plans is very small.

Finally, we will show cases where our cost model can successfully predict the optimal query plan, which is not selected by Catalyst. In the figure in the following figure, Catalyst chooses a query plan performing a shuffle hash join, as the smaller table transmitted is above its threshold for broadcast joining a table (which is by default 10 MBs).

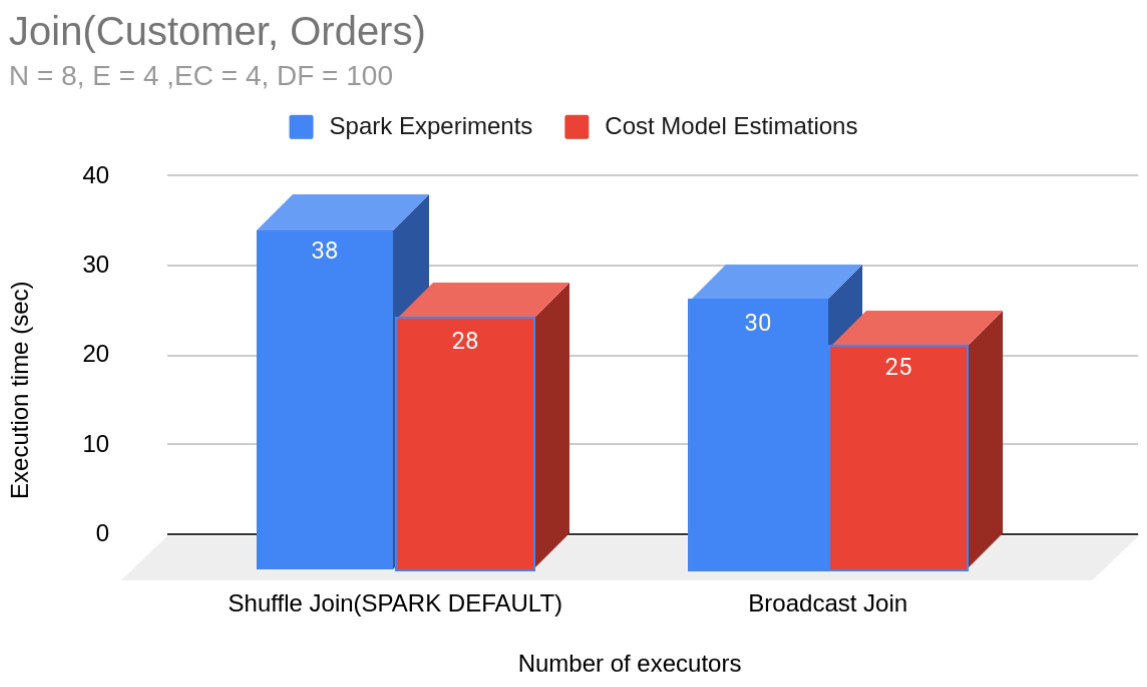


Figure 42. Execution times for specific join query - Catalyst makes wrong choice

We can see that broadcast joining results in significantly faster execution time, and although the cost model does not estimate this significance accurately, it predicts correctly that broadcast join will be the optimal choice. As a result, this is a case where using the cost model manually to evaluate which query plan is the best and then forcing Spark to produce this plan, would result in saving time.

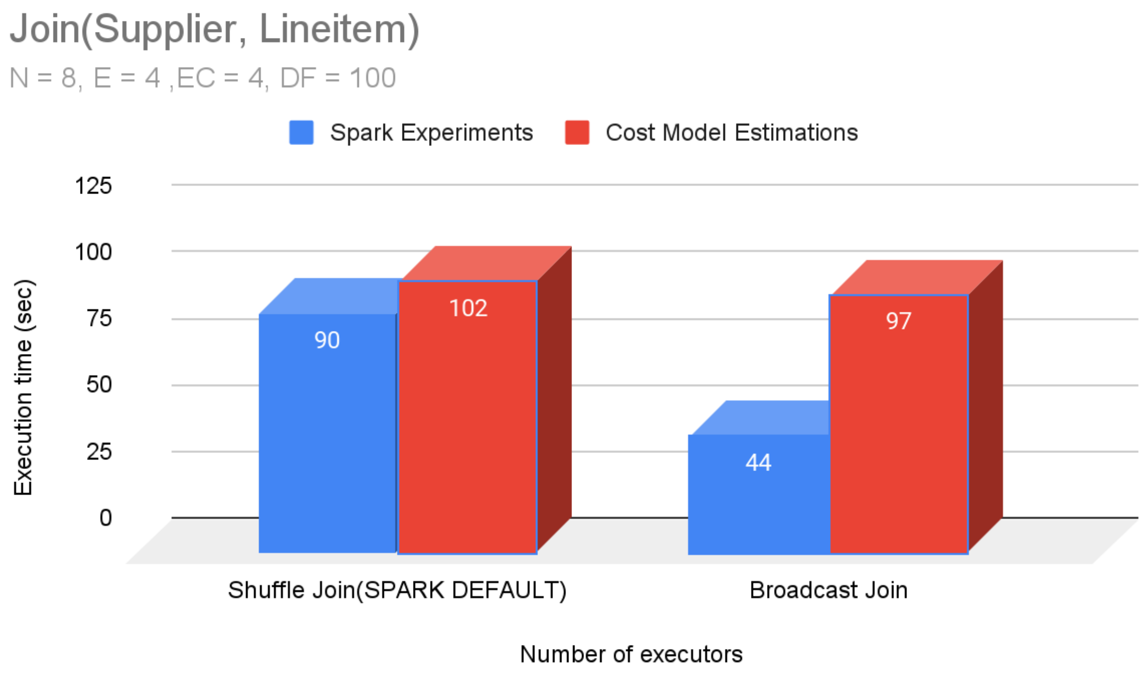


Figure 43. Execution times for specific join query - Catalyst makes very wrong choice

In this figure, we see another case where Catalyst makes a very bad choice concerning the join operator, as it predictably picks a Shuffle Hash join (smaller table above 10 MB threshold), whereas picking a broadcast join would result in less than half of the execution time achieved by broadcast joining. This happens due to the gain of not shuffling the bigger table and focusing only on the broadcast of the smaller one, which is slightly above the Spark threshold for broadcast joins.

The cost model is able to predict that broadcast join is a better choice, however it definitely underestimates the benefits gained from this choice. The reason behind this has to do with the estimation of the time needed for scanning the bigger table, Lineitem. In the case of the cost model, the times for scanning the two tables are the same, as the GPSJ grammar presented in the previous chapter obeys. This phase requires fetching the table from HDFS, to retrieve the RDD partitions and send them to the executors needed to process them. In the case of our cluster, avoidance of the shuffling phase of the bigger table, as well as co-location of data with the executors in some cases eliminated the execution time. The cost model is able to spot the difference in the execution time of the join operation, however the high constant time estimated for scanning the two tables (around 90 seconds), does not leave much room for optimization. The architecture of the cost model is incapable of understanding the optimizations that are inherited from the latter to the first stages in cases like this.

Apart from join selection, the cost model is able to accurately estimate the best order of joins to be performed in the query. However, in the TPC-H queries the join order is optimized beforehand, and we did not conduct large scale experiments to evaluate its accuracy when it comes to join orderings. However, the fact that it is accurate in estimating the in-between stages sizes and cardinalities of the tables means that it works well in principle, avoiding big joins.

In conclusion, our implementation of the cost model is accurate in many cases, when it comes to selecting a join strategy. It makes correct decisions for trivial cases, and even in more complex queries it is able to spot the trend offered by different query plans, although not always picking the optimal one. Furthermore, there are cases like the latter where it strongly overestimates the execution time of a query plan, usually involving broadcast joins. Importantly, it is also able to spot cases where the Catalyst makes wrong query plan selection, proving that its integration in Spark can be a relevant step for turning the Catalyst into a fully cost-based optimizer.

8.3 Monetary cost calculation

After implementing the cost model which provides an estimate for the query execution time, it is necessary to calculate its monetary cost, to be able to consider multiobjective optimization.

Our assumption of one worker per node, as well as the highest parallelization offered by more nodes for storage and execution, means that different numbers of executors in the Spark application will come with different best-case scenarios for running a query. In the following diagram, we show the best execution time achieved by our experiments, in each one of the 3 different numbers of executors (with

2,4, and 8 nodes), when running query 3.

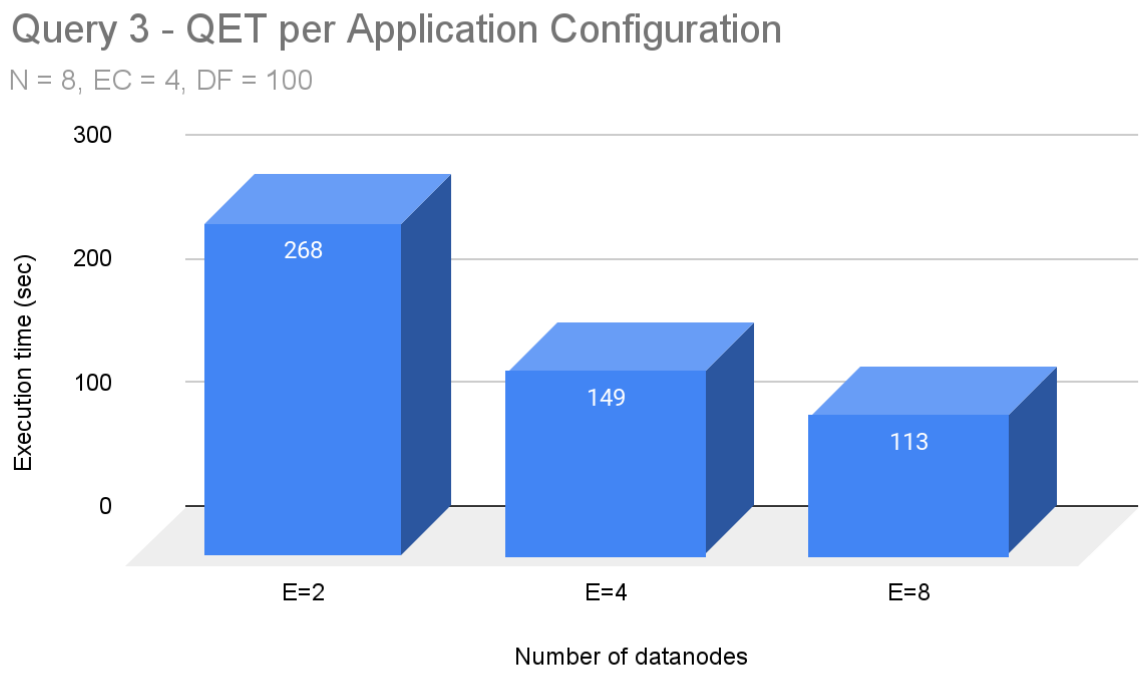


Figure 44. Best execution time for different application configurations

For the pricing of the query plans, Trummer and Koch used the Amazon EC2 pricing system when evaluating MPQ, so we will do the same [9]. Although the prices have changed since 2014, our contribution is up to date. Each Spark executor using EC executor cores, is similar to a vCPU instance with a similar number of cores. From the list of available instances on Amazon EC2, we select the generic medium instance a1.medium, a single 8-core CPU instance with 2 GB of RAM, and it will be the only one used in our estimations, as we assume use of homogeneous resources.

By observing the Spark & Yarn execution logs, we noticed that all the executors involved in the Spark applications are set up with insignificant time delay after query execution begins, and are terminated at the end of the query. Therefore, it is a fair assumption to make that each executor is used for a runtime equal to the runtime of the query. With this assumption in mind, we generated the following formula for estimating the monetary cost of running a single query in Spark, using the Amazon EC2 cloud environment and a1.medium instances:

Cost = E * QET (seconds) * cost($/hour)/3600

with QET being the query execution time, cost being the hourly cost of using a1.medium (0.0255$), and E being the number of executors.

For the 3 execution plans in the diagram above, their cost is calculated as follows:

- Cost 1: 2 * 0.0255 * 268/3600 = 0.0037$ (2*a1.medium, 2 vCPUs)
- Cost 2: 4 * 0.0255 * 149/3600 = 0.0042$ (4*a1.medium, 4 vCPUs)
- Cost 3: 8 * 0.0255 * 113/3600 = 0.0064$ (8*a1.medium, 8 vCPUs)

By observing the monetary cost values, we can see that we have a Pareto front for a multi-objective resource optimization problem, as no cluster configuration results in a query cost that is worse both in means of execution time, and money. The Pareto front is visualised below:

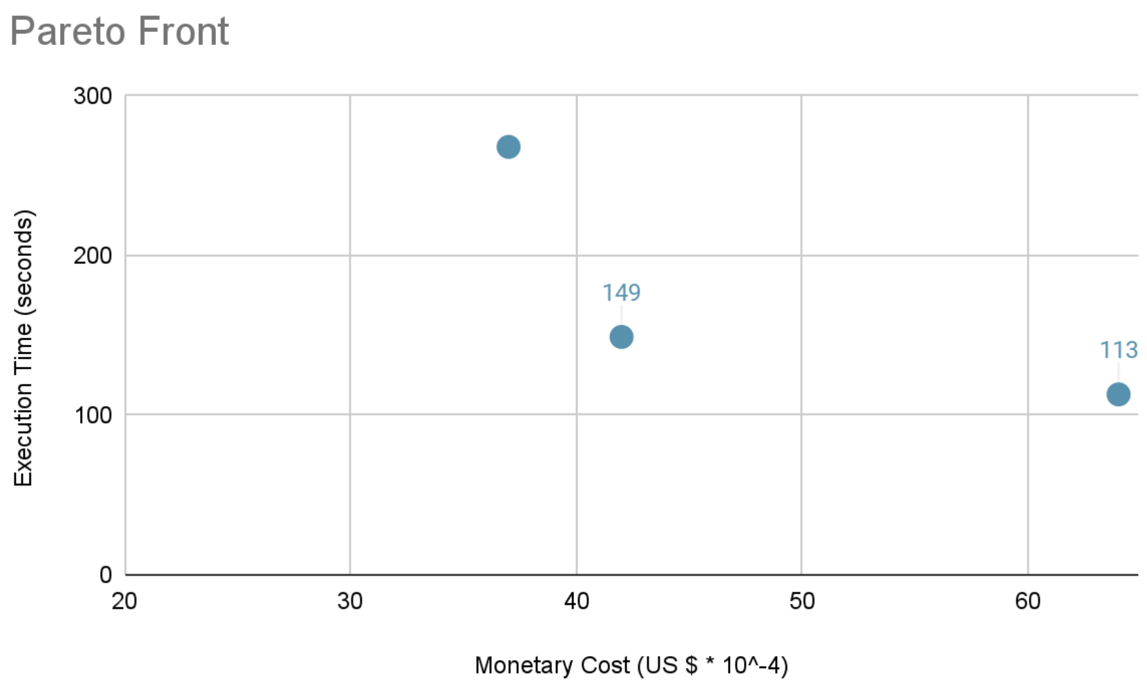


Figure 45. Pareto Front of the optimal query plans

Therefore, the combination of the cost model and the monetary cost estimation formula is able to achieve MQ, and provide a Pareto-front with optimal query plans, offering different tradeoffs between execution time and monetary cost, when run in a hypothetical cloud environment like Amazon EC2.

8.4 Conclusions

After evaluating the cost model experimentally, it proved to be a good first step for cost based estimation of query plans' execution time in Spark SQL. The implemented cost model has room for improvement, but is within the error percentage limits of Baldacci and Golfarelli, showing that its reimplementation is valid. After evaluating its estimation and prediction accuracy, we went on to provide a formula for monetary cost estimation of a query plan, and we proved that using the cost model, we can achieve multi-objective query optimization and scheduling of executors in a Massively Parallel Processing environment.

Chapter 9

9.MPQ Evaluation

9.1 Introduction

In the last section, we validated our implementation of Baldacci and Golfarelli's cost model [3] and produced a formula for estimating the monetary cost of a query plan based on its runtime and application characteristics. In this section, we will use the cost model and the formula as a basis to apply a form of Trummer's generic algorithm to a different form of the MPQ problem. By doing this, we aim to tackle a scenario of an MPQ problem in the cloud from another perspective, without a relational DBMS but using an MPP framework. With the use of such a framework (Spark), we simulate the setting of a cloud environment. With our experiments we aim to show that MPQ algorithms can be applied in this different setting, highlight the advantages of using MPQ, and also prove that query optimization depends on several query characteristics.

9.1.1 Parameters and Objectives

As we mentioned before, MPQ problems model a query optimization problem by taking into account multiple parameters of the query as well as multiple optimization goals. In their original paper, Trummer & Koch mentioned two example scenarios where MPQ problems occur. One of them involved a cloud computing setting, with optimization goals being execution time and monetary cost, and the parameters being the selectivity of a number of query predicates. In a second scenario, embedded SQL queries were tackled, with optimization goals being execution time and result precision, and the parameter being the available buffer size at runtime.

An example of an MPQ problem similar to the first scenario is shown in the figure below (Fig.), where we have an SQL query template of the form:
SELECT * FROM Table WHERE P1 AND P2

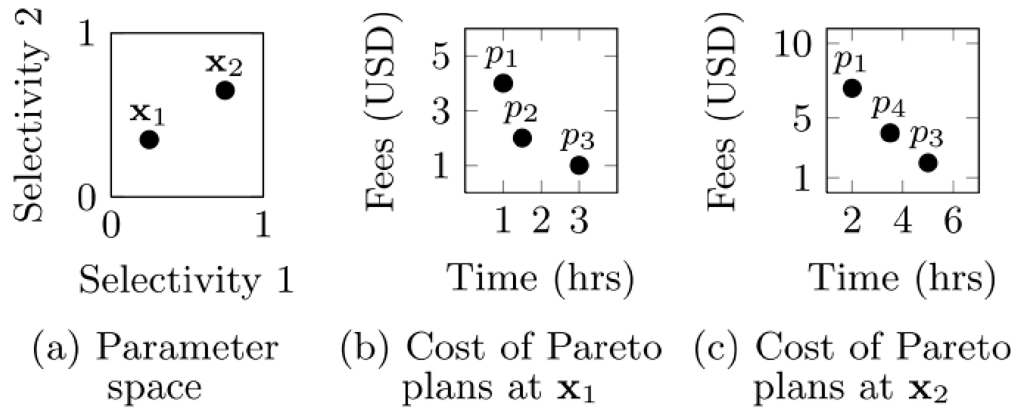


Figure 46: Two parameter combinations, each yields a set of Pareto-optimal plans [1]

In this case, the parameters are the selectivities of predicates P1 and P2. As we can see from the figure, different values for the selectivities of the query predicates lead to different plans being Pareto-optimal, as well as different values for the two metrics of each query plan. After the query parameters are specified, users can select a trade-off, realized by an alternative query plan from the relevant Pareto front. All relevant query plans (Pareto-optimal somewhere in the parameter space) have been calculated and compared in terms of both optimization goals in a preprocessing step.

Our evaluation scenario will be similar to the aforementioned one, consisting of two optimization goals (query execution time and monetary cost), as well as a varying number of parameters, each one being the selectivity of a query predicate.

9.1.2 Validation Differences

There are some significant differences in the architectures and the assumptions between our validation of an MPQ algorithm, and the validation route followed in the original paper. Some of them are highlighted in the following table.

	Our Evaluation	Original MPQ paper Evaluation	
Query types	GPSJ(no UNION ALL, OUTER JOIN, EQUI JOINS)	ALL	✗
Query selection	TPC-H Benchmark	Randomly Generated	✗
Plan cost calculation	Sum of sub plans (independence)	Sum of sub plans (independence)	✓
Join Estimation Formulas	Standard	Standard	✓

Optimization Objectives	Execution time, monetary cost	Execution time, monetary cost	✓
Parameters	Predicate Selectivities	Predicate Selectivities	✓
Available Scan Types	1 (Full scan)	2(Full scan, index seek)	✗
Available Join Types	2 (Shuffle Hash, Broadcast)	2(Single node, shuffle hash)	
Pricing basis	Amazon EC2 general purpose medium instance	Amazon EC2 general purpose medium instance	✓
System Choice	Chapter 9 Cost Model & Spark SQL Catalyst	Benchmark written as Java application with cost model	✗

Table 5. Validation differences between our and the original MPQ algorithms

As one can see there are some similarities in the validation process, and also some differences. For example, in the original paper, for the evaluation of the MPQ algorithms, 2 scan types were considered for data tables, a full scan and an index seek. In our case, we use the only one available in Spark, which is a full scan.

The most important difference, however, has to do with the implementation system, with Trummer and Koch creating a benchmark which compares all relevant plans for randomly generated queries with desired parameter characteristics. The comparison is done based on estimations for the execution time and monetary cost of each query plan. These estimations are based on characteristics of the query like the number of I/O operations needed, the number of bytes transmitted in the network (for the execution time), and the hourly cost of the Amazon EC2 general purpose medium instance (for the monetary cost).

In our case, the basis was once more queries from the TPC-H benchmark. An important difference has to do with the algorithm used to produce the Pareto Plan Set, as our assumption that our two optimization objectives are dependent changes the approach. This will be more thoroughly discussed in $9.3.

9.2 Parameter Space

In MPQ problems, it is necessary that the parameter value domain is known in advance. In this chapter, we will analyze Query 3 from the TPC-H benchmark, with an additional predicate. We chose query 3 as it is a query including only 2 joins and 3 tables, making it audience friendly. Moreover, it was a query where sensitivity of the selectivities of the predicates was high, and can work as an illustrative example.

Apart from this, it can work as a real life scenario for a shipping company. It is a query that retrieves the shipping priority and potential revenue of the orders having the largest revenue among those that have not yet been shipped. In a very big company where thousands of orders arrive each day, exhaustively examining all relevant query plans each time can lead to significant optimization overhead. In that case, it would be useful to calculate in a preprocessing step all relevant query plans for a number of query templates, so that query optimization overhead is avoided. As a result, decisions will be taken more quickly. The form of query 3 we are using is the following:

TPC-H QUERY 3
SELECT l_orderkey, o_orderdate, o_shippriority, sum(l_extprice)
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING' AND c_custkey = o_custkey AND l_orderkey = o_orderkey AND o_orderdate < date'1995-03-15' AND l_shipdate > date'1995-03-15' AND c_custkey <15000000
GROUP BY l_orderkey, o_orderdate, o_shippriority

The parameter space we will consider is P1 = [0.1,0.2,0.4,0.6,0.8,1], P2 = [0.5,0.3], with P1 being the selectivity of the query predicate in yellow and P2 being the selectivity of the query predicate in light blue. P1 filters the percentage of customers to be considered for the query, which can make sense if customers are classified in different types. P2 filters the order date before which an unshipped order was sent which also makes sense as a predicate, as older orders are of higher priority. For each parameter combination, we will examine all possible query plans, for 3 different application configurations (E=2,E=4,E=8), assuming that there are 3 different options of using resources in an IaaS platform (like Amazon EC2) to execute the query. All relevant query plans for a query involve every possible combination of join order, scan operator (we only have one) and join operator being applied. In this example query, we examined 18 query plans for each parameter combination and application configuration, totaling 648 query plan checks.

The parameter space can be visualized in a 2D figure as follows.

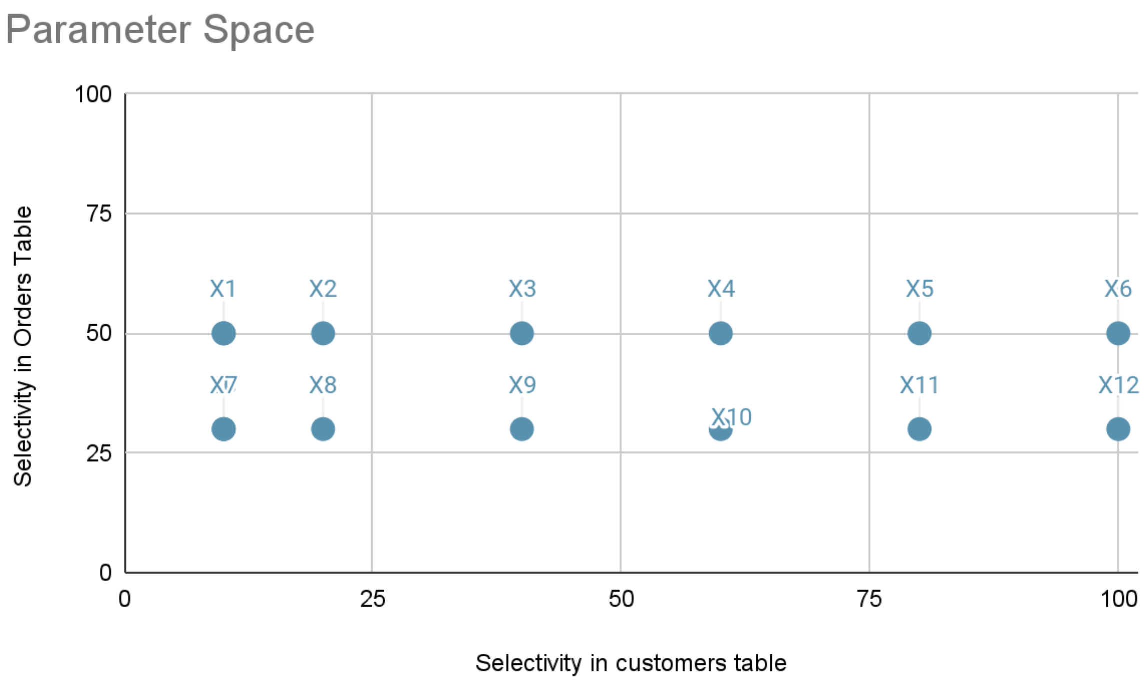


Figure 47. Parameter Space of our experiments

9.3 MPQ Algorithm - Pareto optimality

9.3.1 Algorithm

Our MPQ algorithm will be a more simple version of the two MPQ algorithms Trummer and Koch presented in their original paper. This is due to the assumption that we make, that monetary cost is dependent, among other factors, on execution time. This means that for a specific application configuration and parameter combination, there exists a single optimal physical plan. As a result, there does not exist a tradeoff between different physical plans, but between different application configurations, as using more executors for a query reduces runtime and increases the monetary cost. Therefore, for each parameter combination, if we examine N different application configurations, we only have to compare the N query plans, and prune those that are not Pareto-optimal. As a result, a different MPQ problem is tackled which enables the algorithm to be logically split in two step, the first one being single-objective and the second one multi-objective. This allows us to significantly reduce its execution time, by avoiding exhaustive comparisons of the relevance regions of all possible query plans:

- In the first step, for all parameter combinations and for each application configuration, all relevant query plans are produced and compared according to a single metric, query execution time. For each parameter combination and each application configuration, the algorithm yields a single optimal query plan (the fastest).
- In the second step, Algorithm 1 from the original MPQ paper is used to compare the remaining query plans according to their monetary cost and execution time, and calculate the relevance region of each plan. The output of the algorithm is a Pareto Plan Set, which contains all query plans with a non-empty relevance region.. Relevange region of a query plan is the set of parameter combinations for each a given plan is relevant, as we defined it in chapter 9.

The difference in the MPQ problem that we tackle is that the main choice the user has to make is not primarily influenced by the query physical plan. Basically, he chooses the system in which his query will be executed. Therefore, we take an approach to MPQ similar to IaaS (or FaaS) scheduling optimization problems, to better evaluate its use in cloud use cases.

9.3.2 Results

For the query and the parameter space we introduced in $10.2.1, 648 query plans were checked. In this section, we will present the results of the algorithm. Firstly, the algorithm iteratively checks all plans for all possible join orders. The table involved in the query are the following:

Customer	**2.4 GB**
Orders	**17 GB**
Lineitem	**77 GB**

Table 6. Table sizes

The query involves 2 joins. In the default case, Customer is joined with Orders, and the join product is joined with the bigger table Lineitem. Changing the order of these joins is possible, however it is obviously a bad choice, as joining the bigger tables first will result in a very big first join. This is proven from the cost model too, as the mean execution time for joining orders and lineitem first, is double than joining customer and orders first. In conclusion, the default join order is the optimal, and our algorithm prunes all plans with the worse one.

When the query plan involves broadcast joins, there is theoretically the choice of which table to broadcast. However, it is obvious that the join table has to be the smallest one, so our algorithm also prunes those plans. This leaves us with 1 join order, 4 different join combinations and 3 different application configurations for each parameter combination. In the table below, we will consider only these plans and not the ones we mentioned above (different join order, different broadcast choices), in order to save space and highlight the most interesting comparisons.

We will define the physical plans with each join combination as follows:

PL1: Join Types -> Shuffle Join, Shuffle Join

PL2: Join Types -> Broadcast Join, Broadcast Join

PL3: Join Types -> Shuffle Join, Broadcast Join

PL4: Join Types -> Broadcast Join, Shuffle Join

We will show the results of the algorithm, which are no other than the estimation times of our cost model for each parameter configuration, application configuration, and physical plan. In each case, we have highlighted the fastest query plan with green.

E=2, EC=4

P1/P2	0.3				0.5			
PLAN	PL1	PL2	PL3	PL4	PL1	PL2	PL3	PL4
0.1	210.5	195.9	198.4	208.0	211.9	196.2	199.8	208.2
0.2	211.1	196.5	199.0	208.6	212.9	197.7	201.3	209.2
0.4	212.4	199.0	201.6	210.0	214.6	202.0	205.7	211.1
0.6	213.7	202.3	204.8	211.2	216.6	208.7	212.0	213.4
0.8	215.0	206.9	209.1	212.9	218.5	217.3	220.3	215.8
1.0	216.4	212.2	214.2	214.4	220.4	227.8	230.3	218.4

E=4, EC=4

P1/P2	0.3				0.5			
PLAN	PL1	PL2	PL3	PL4	PL1	PL2	PL3	PL4
0.1	127.5	122.7	123.5	126.8	128.0	122.9	124.1	126.9
0.2	127.9	123.0	123.9	127.1	128.5	123.5	124.7	127.3
0.4	128.5	124.2	125.1	127.7	129.4	125.6	126.8	128.2
0.6	129.0	125.8	126.6	128.2	130.2	128.7	129.8	129.2
0.8	129.7	127.9	128.6	129.0	131.1	132.8	133.7	130.3
1.0	130.3	130.4	131.0	129.7	132.0	137.8	138.4	131.6

E = 8 EC = 4

P1/P2	0.3				0.5			
PLAN	PL1	PL2	PL3	PL4	PL1	PL2	PL3	PL4
0.1	62.6	61.4	61.6	62.3	62.7	61.4	61.7	62.4
0.2	62.7	61.5	61.7	62.5	62.9	61.8	62.1	62.6
0.4	63.0	62.1	62.3	62.8	63.3	62.7	63.0	63.0
0.6	63.3	62.8	63.0	63.1	63.8	64.2	64.4	63.5
0.8	63.6	63.8	64.0	63.4	64.1	66.1	66.2	64.0
1.0	63.8	65.0	65.1	63.8	64.5	68.5	68.6	64.6

Table 7. Execution time of different query plans for 3 different application configurations

It is easy to see that varying the selectivity of the predicates, as well as changing the number of Spark executors has a significant impact on the optimality of the plans.
The relevance regions of the 4 physical plans are the following:
E=2 EC=4
RR(PL1) = []
RR(PL2) = [(0.1,0.3),(0.1,0.5),(0.2,0.3),(0.2,0.5),(0.4,0.3),(0.4,0.5),(0.6,0.3),(0.6,0.5),(0.8,0.3),(1.0,0.3)]
RR(PL3) = []
RR(PL4) = [(0.8,0.5),(1.0,0.5)]

E=4 EC=4
RR(PL1) = []
RR(PL2) = [(0.1,0.3),(0.1,0.5),(0.2,0.3),(0.2,0.5),(0.4,0.3),(0.4,0.5),(0.6,0.3),(0.6,0.5),(0.8,0.3)]
RR(PL3) = []
RR(PL4) = [(0.8,0.5),(1.0,0.3),(1.0,0.5)]
E=8 EC=4
RR(PL1) = [(1.0,0.3),(1.0,0.5)]
RR(PL2) = [(0.1,0.3),(0.1,0.5),(0.2,0.3),(0.2,0.5),(0.4,0.3),(0.4,0.5),(0.6,0.3)]
RR(PL3) = []
RR(PL4) = [(0.6,0.5),(0.8,0.3),(0.8,0.5)]

A lot of conclusions can be made from these relevant regions. First of all, PL3's relevance region is empty. PL3 included a shuffle hash join first and a broadcast hash join second. However, for the parameter combinations examined, the size of the smallest table in the first table is always less than the size of the smallest table on the second join. As a result, it is obvious that broadcast join will be a better fit for the first join, so excluding it from the first and including it in the second will never be optimal. Furthermore, the regions show that the lower the selectivity of each predicate and the number of executors, the more fit broadcast join is for the concerned join. This makes sense as lower selectivity will mean the join inputs will be of lower cardinality and size, making it more preferable to broadcast the smaller one. Less executors also make broadcast joins preferable, as the smaller table will have to be broadcasted to less nodes, so less data will be transmitted through the network.

Finally, we can also observe a pattern, which shows that going from 0 to 1 selectivity, the optimal plan sequence is PL2 -> PL4 -> PL1. This is a logical conclusion, as in the PL2 broadcast joins all relations and PL1 shuffle joins all relations, with PL4 acting as an intermediate where the smaller join is performed with a broadcast join operator and the bigger one with a shuffle join operator.

Although the validation of the cost model was included in the last chapter, we also include the results from the same set of experiments for E=4 and EC=4 and half of the parameter combinations performed in Spark SQL, to see if the same pattern will appear..

E = 4 EC = 4 (Spark SQL)

P1/P2	0.5		
PLAN	PL1	PL2	PL4
0.1	125.6	122.1	125.7
0.2	127.3	124.8	125.2
0.4	128.2	124.8	125.0
0.6	128.3	N/A	126.3
0.8	128.7	N/A	129.1
1.0	129.2	N/A	129.9

Table 8. Spark experimental values for different query plans.

As it seems, the behaviour observed in Spark SQL is captured by the cost model. As the selectivity increases, PL2 is replaced by PL4 as optimal, and with 100% selectivity PL1 is the optimal one. Wherever the cost model picks a different plan, we mark it with red. Above a table size, Spark does not proceed to broadcast the table as it considers it an overkill. This constraint is not taken into account by our cost model which continues to give estimations no matter the size, and that's why it fails the [0.6,0.5] prediction. Similarly, in the case of [1.0,0.5], the cost model seems to underestimate the overhead of broadcasting customer table. We will not further analyze the cost model as this was thoroughly done in the last chapter, but included this table to show that the process was repeated with experiments in Spark, which showed similar behaviour.

Apart from these observations, it is obvious that each parameter combination provides different trade-offs for query execution. We will now present some of the results of the second part of our algorithm, where each parameter combination yields a set of Pareto optimal plans.

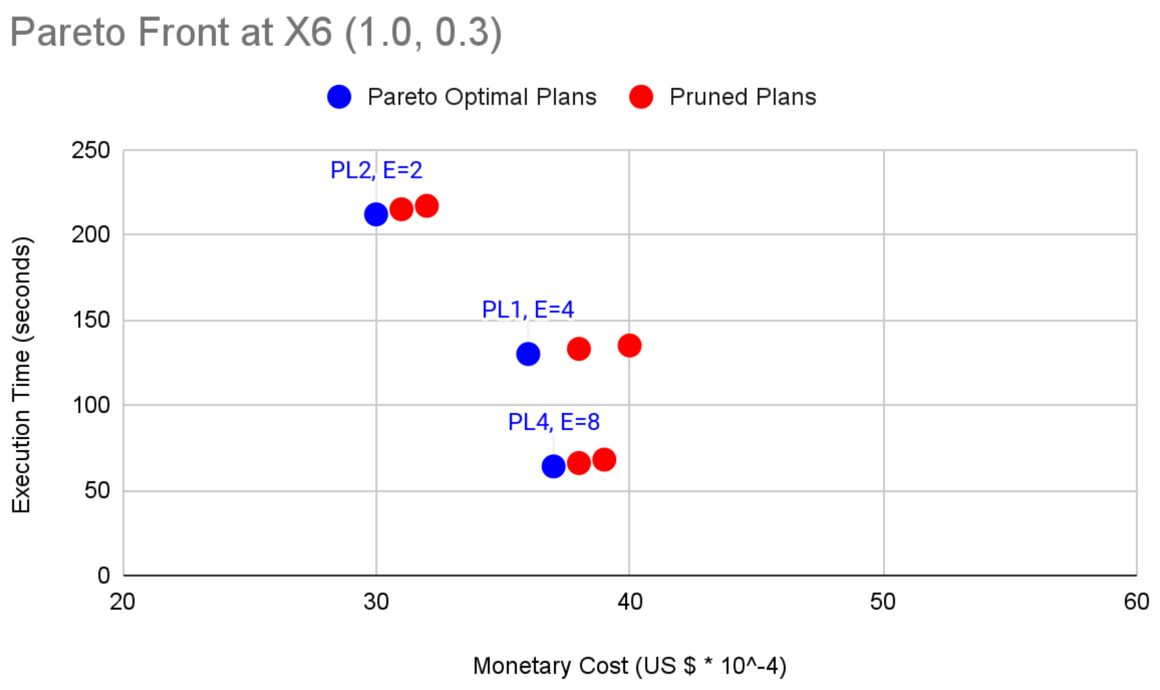


Figure 48. Pareto Front at X6

First, we visualised the Pareto Front for the parameter combination (1.0, 0.3). It is an interesting case, as it is the only one from the combinations examined where 3 different physical query plans are Pareto Optimal. With red we have also included some non-optimal Pareto plans which were pruned by our algorithm, to illustrate the difference in time and cost.

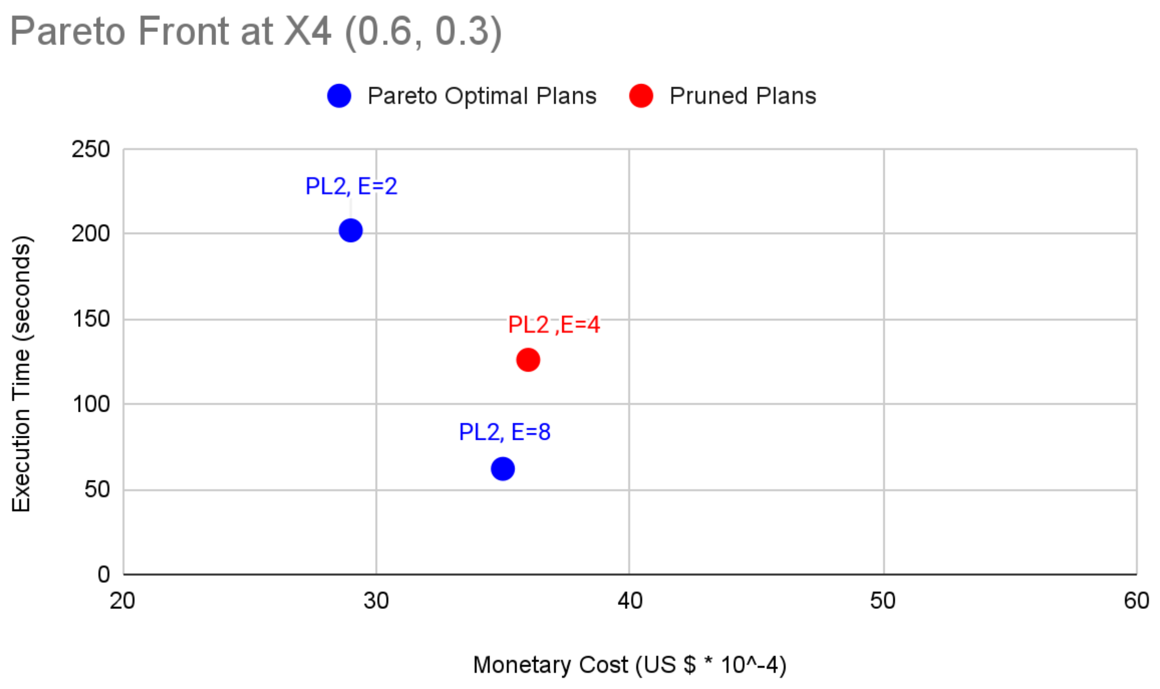


Figure 49. Pareto Front at X4

In this parameter combination (0.6,0.3), we only included the three plans that are optimal for each Spark application configuration. We included this case as it is one where the optimal plan of the configuration with E=4, is dominated by the plan with E = 8, and as a result, it is pruned. The reason this happens has to do with the speedup that 8 executors can offer, in comparison to 4. It is very significant and in this case, the execution time of PL2, E=8 is less than half of the runtime of PL2, E=4. This scenario, with the plan using 4 Spark executors being dominated by the plan using 8 Spark executors happens in numerous cases from the ones we examined.

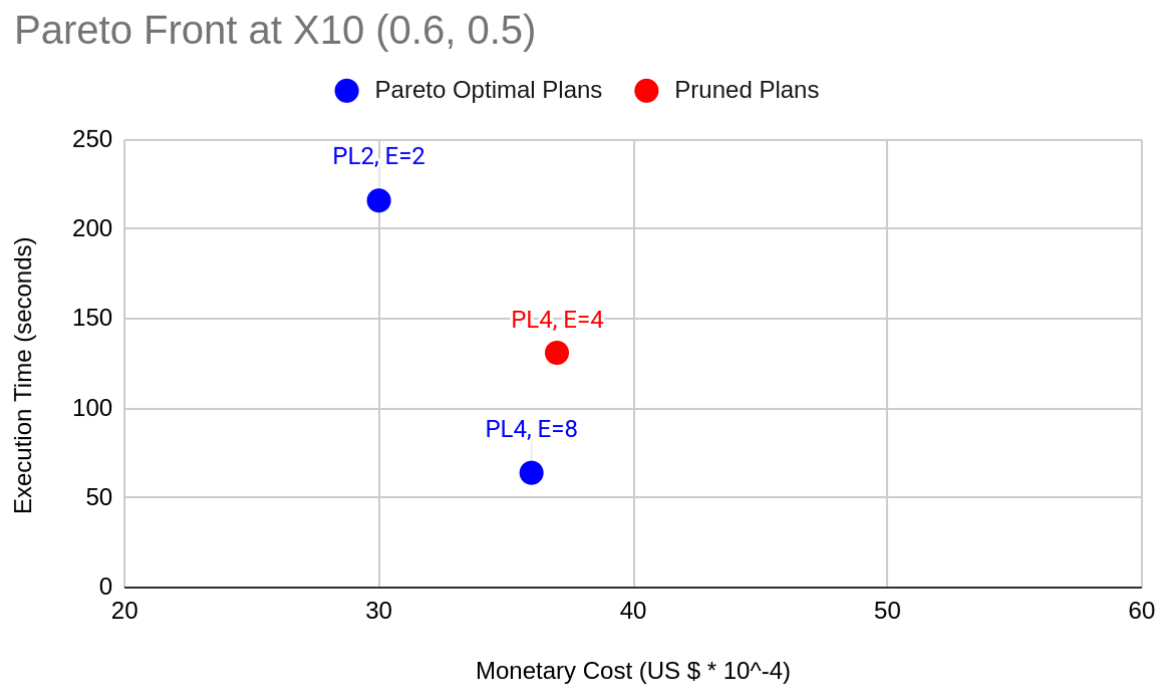


Figure 50. Pareto Front at X10

For each one of the 12 parameter combinations, a different set of Pareto optimal query plans is produced. We will not include the remaining 9 ones for space reasons.

9.4 Conclusions

The obvious conclusions of this set of experiments include that factors such as parameter selectivity can have a huge influence on which query plan is the optimal from the available ones. In cases of large scale data processing, optimization time can prove to be very time consuming as the number of available plans is very large. MPQ happens before runtime, and pays off by avoiding runtime query optimization. We presented a scenario with a shipping company querying its dataset to find unshipped orders with the highest revenue. We argued that it can be a scenario where decisions must be taken quickly and is a good fit for MPQ, as the company can benefit from avoiding query optimization time. We tackled MPQ in a different data management setting, using a MPP instead of a relational DBMS and showed its proof of concept.

We used the selectivity of 2 join predicates as a parameter and performed the preprocessing step ourselves, producing the necessary Pareto Plan sets, with the Spark SQL cost model presented in Chapters 8 and 9. We also evaluated this process with experiments in Spark SQL, which confirmed that the choices guided by the Pareto Plan Set are relevant. With this we showed that optimization problems can be tackled as MPQ problems in the cloud, and users can benefit from MPQ algorithms.

Our evaluation was less complex than the one in the original MPQ paper, but could be extended in the future, for example by considering optimizations presented in Hadoop++ **[18]** to make index seek techniques available in Spark and significantly alter the query optimization landscape and trends shown.

Chapter 10

10.Conclusion, Future Work

10.1 Summary and Conclusions

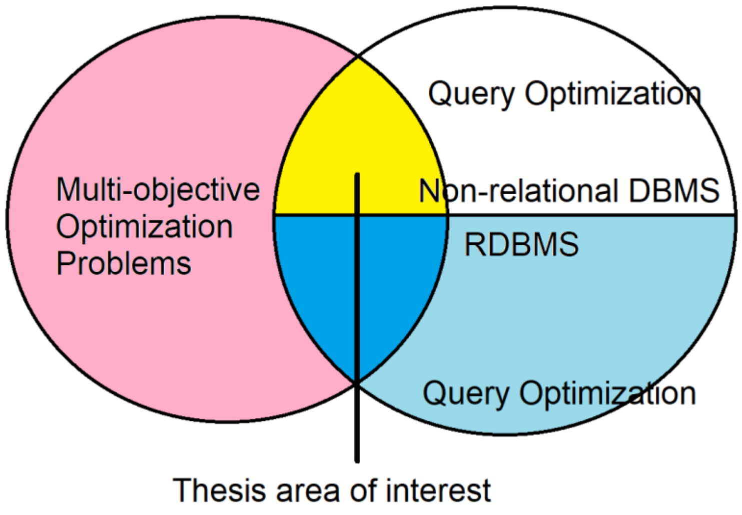


Figure 51. Thesis Research Area

In this thesis, the research area of multi-objective query optimization was studied, with emphasis on non-relational databases (Figure). Massively parallel processing frameworks (like Apache Spark) based on the MapReduce paradigm, commonly used in big data processing, was the non-relational alternative considered. In the thesis we proposed an architecture for a hybrid query optimizer operating in an environment like the aforementioned, able to alternate between two state of the art multi-objective query optimization techniques (MQ and MPQ).

As both of these techniques require exhaustive comparisons of alternative query plans, and MPQ has only been evaluated in a RDBMS-like environment, in order to integrate them in Apache Spark a state of the art cost model was required, as the optimizer of Spark SQL comes with significant limitations, and is not fully cost based.

Our first contribution was the reimplementation of a state of the art theoretical cost model for Spark SQL, which would enable us to compare the alternative plans based on their execution time. We conducted large scale experiments on Spark to evaluate the cost model estimation and prediction accuracy. The evaluation showed that the cost model operates with an error percentage within the bounds mentioned in the original paper of the cost model, so we were able to use the cost model to apply the query optimization techniques we wanted.

Before we applied the optimization techniques, it was necessary to provide a cost model for a second optimization objective, which in our case was monetary cost of the query plan, a very important factor when working in a cloud computing platform. Thus, our second contribution was to extend the cost model to provide an estimate for the monetary cost of a query plan, based on its execution time and the pricing scheme of Amazon EC2.

Our next step was to define the optimization opportunities in our execution environment. Similar to an IaaS cloud platform, where renting more computational resources speeds up the execution of an application but increases its monetary cost, Spark configuration settings include number of executors allocated for the execution of a given query, as well as number of cores included in each executor. We repeatedly ran queries for different application configurations (different numbers of executors and cores), used the cost model to find an optimal query plan for each configuration, and then compared the optimal plans, and generated a Pareto front. In conclusion, we successfully modeled query optimization in Spark as an a posteriori multi-objective query optimization problem, by applying MQ.

The next part of our contribution involved integrating MPQ to this system. We modeled queries as parametric functions, with the parameters being selectivities of query predicates, and proved that query execution times are actually highly dependent on these parameters, with different values resulting in different query plans. We then used our cost model to apply (multi-objective) parametric query optimization, and successfully produced the relevant Pareto Plan Sets, evaluating this optimization technique in a different environment, with an alternative version of the optimization problem, and also described in detail the algorithm we used to solve the MPQ problem, which is a variant of one of the original two MPQ algorithms implemented by Trummer and Koch.

10.2 Future Work

There are many different research routes that can be followed to build on the contributions and experiments conducted in this thesis. First of all, the cost model that we reimplemented proved that it can be accurate and make better choices than Spark in certain cases. This cost model has only been described theoretically and has been evaluated outside Spark. As a result, integrating this cost model in Spark SQL Catalyst (which is possible as it is an extensible optimizer), and turning it into a fully cost-based query optimizer, although challenging, would be a very important contribution and significantly improve Spark SQL performance. To do this would also require conducting even more experiments in Spark, and to carefully observe cases of misestimations and inaccuracies, to see if we can further optimize it before integrating into the Catalyst.

When it comes to multi-objective optimization, in this work we considered two optimization metrics, query execution time and monetary cost. In the future, it is our goal to consider even more cost metrics, like energy consumption and result precision. We also aim to introduce another optimization goal, data security.

Security of data is essential in an era when every person and organization stores terrabytes of confidential data in the cloud. Optimizing a query plan in terms of security could be introduced by including two or more different filesystem storage options, like storing our data in HDFS, or in a blockchain file system, like IPFS. An execution plan with the data stored in IPFS will provide more security, while definitely being more expensive. Modeling security is an additional challenge as it is not a countable metric, so each filesystem type could be assigned a constant security value ranging from 0 to 1. A query execution plan accessing data over HDFS would have an average security value (eg 0.5), whereas if the data were in IPFS, the security value could be higher (eg 0.8), leaving the door open for including more filesystems in the future.

Resource heterogeneity is also an issue that could be addressed in the future, as in our work we assumed use of heterogeneous resources (the medium instance of Amazon EC2). However, Amazon EC2 is also a system providing specialized computing resources like GPUs [81] and FPGAs [84]. Thus, addressing query optimization in an heterogenous cluster would be a topic of interest.

Finally, the final contribution of the thesis, MPQ evaluation is examined as to possible further developments. MPQ algorithms have the significant advantage of having no query optimization overhead, as optimization is carried out in a preprocessing step. Its disadvantage might be an amount of inaccuracy, when the parameter space is not dense enough and the optimizer does not make optimal choices for the values in between of the parameter combinations taken into account. In the future, we aim to make a detailed comparison between MQ and MPQ, and observe whether the MPQ benefits are that significant. This comparison could happen in an actual web application operating over a cloud platform and using the proposed optimizer architecture.

We also aim to more extensively evaluate MPQ algorithms in different operation environments, involving more complex decision making (e.g. more scan, join types), and creating a large benchmark to evaluate our results and gain more insights about MPQ benefits in a cloud environment, similar to Trummer and Koch evaluation method.

In our thesis, we regularly mentioned serverless computing, but it did not play an essential part in our experimental part, as apart from certain assumptions that gave our architecture some features similar to serverless, as well as improvements on them (e.g. logical disaggregation and physical colocation of storage and compute resources), we did not consider a serverless architecture. In the future, we aim to integrate Apache Crail into our system to observe its impact, and possibly implement the proposed hybrid optimizer system, where the optimizer could pick the optimization environment of its choice (Cloud or Serverless). This would enable us to simulate the first ever serverless query optimizer and make an in depth analysis of the challenges and opportunities offered by query optimization in a serverless environment.

References

[1] I. Trummer and C. Koch. Multi-objective parametric query optimization. In *The VLDB Journal* 26, 1, 107–124, 2017.

[2] M. Muniswamaiah, T. Agerwala, and C. Tappert, Big Data in cloud computing review and opportunities, *International Journal of Computer Science and Information Technology*, vol. 11, no. 4, 43–57, 2019.

[3] L. Baldacci and M. Golfarelli. A cost model for SPARK SQL. In *IEEE Transactions on Knowledge and Data Engineering*, 31, 5, 819–832, 2019.

[4] P. M. Mell and T. Grance, "The NIST definition of cloud computing," 2011.

[5] A. Khandelwal, A. Kejariwal, and K. Ramasamy, Le Taureau: Deconstructing the Serverless Landscape; A Look Forward. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2641–2650, 2020.

[6] Amazon EC2. https://aws.amazon.com/ec2/

[7] Google Cloud. https://cloud.google.com/

[8] The SQL Server Query Optimizer. https://www.red-gate.com/simple-talk/databases/sql-server/performance-sql-server/the-sql-server-query-optimizer/

[9] Amazon EC2 On-Demand Pricing. https://aws.amazon.com/fr/ec2/pricing/on-demand/

[10] TPC-H Benchmark. http://www.tpc.org/tpch/

[11] S. K. Mohanty, G. Premsankar and M. di Francesco, An Evaluation of Open Source Serverless Computing Frameworks. *In 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 115-120, 2018.

[12] Couchbase. https://www.couchbase.com/

[13] Cost Based Optimization with Couchbase json database. https://blog.couchbase.com/cost-based-optimization-with-couchbase-json-database/

[14] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[15] E. Jonas, J. Schleier-Smith, V. Sreekanti et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. 2019.

[16] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2018.

[17] I. Baldini et al., Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing. Springer*, 1-20, 2017.

[18] J. Dittrich, J. Quiane-Ruiz, A. Jindal, Y. Kargin, V. Setty, J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing), In Proceedings of the VLDB Endowment, 3,. 515-529, 2010.

[19] Crail-spark-io. https://github.com/zrlio/crail-spark-io

[20] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi. Understanding Ephemeral Storage for Serverless Analytics. In *Proceedings of the USENIX Annual Technical Conference*, 789–794, 2018.

[21] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th Symposium on Operating Systems Design and Implementation*, 427–444, 2018.

[22] M. Wawrzoniak, G. Alonso. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *CIDR*, 2021.

[23] I. Müller, R. Marroquín, and G. Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *SIGMOD*, 2020.

[24] V. Sreekanti, C. Wu, X. C. Lin, J. S.-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful Functions-as-a-Service. In *The Computing Research Repository*, 2020.

[25] M. Perron, R. C. Fernandez, D. DeWitt, and S. Madden. Starling: A scalable query engine on cloud function services. In *SIGMOD*, 2020.

[26] AWS Lambda. https://aws.amazon.com/lambda/

[27] Google Cloud Functions. https://cloud.google.com/functions

[28] Azure Functions. https://azure.microsoft.com/en-us/services/functions/

[29] OpenFaaS. https://github.com/openfaas/faas

[30] Fission. https://fission.io/

[31] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *HotCloud*, 2016.

[32] OpenWhisk. https://openwhisk.apache.org

[33] Amazon S3. https://aws.amazon.com/s3/

[34] Amazon Athena. https://aws.amazon.com/athena/

[35] Google BigQuery. https://cloud.google.com/bigquery

[36] Azure Stream Analytics. https://azure.microsoft.com/en-us/services/stream-analytics/

[37] 2018 Serverless Community Survey: Huge growth in serverless usage. https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/

[38] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu and C. Zhang. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data*, 2021.

[39] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou and I. Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. In *IEEE Data Engineering Bulletin 40*, 1, 38–49. 2017.

[40] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A kvs for any scale. In *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[41] NetApp. https://www.netapp.com/industries/healthcare/

[42] Medsphere. https://www.medsphere.com/

[43] A Storm is coming: more details and plans for release. https://blog.twitter.com/engineering/en_us/a/2011/a-storm-is-coming-more-details-and-plans-for-release

[44] SQL vs. NoSQL. https://www.thorntech.com/sql-vs-nosql/

[45] Facebook releases Cassandra as open source. https://perspectives.mvdirona.com/2008/07/facebook-releases-cassandra-as-open-source/

[46] Brewers CAP Theorem on distributed systems. https://www.royans.net/2010/02/brewers-cap-theorem-on-distributed.html

[47] Big Data: The 3 Vs explained. https://bigdataldn.com/intelligence/big-data-the-3-vs-explained/

[48] What is Big Data?. https://www.auraquantic.com/what-is-big-data/

[49] S. Zhelev, A. Rozeva. Big data processing in the cloud - Challenges and platforms. In *2017 AIP Conference Proceedings*, 1910, 2017.

[50] Apache Hadoop. https://hadoop.apache.org/

[51] Apache Spark. https://spark.apache.org/

[52] Apache Cassandra. https://cassandra.apache.org/_/index.html

[53] Apache HTable,
https://hbase.apache.org/2.0/devapidocs/org/apache/hadoop/hbase/client/HTable.html?is-external=true

[54] Apache Hive. https://hive.apache.org/

[55] Apache Storm. https://storm.apache.org/

[56] Apache Flink. https://flink.apache.org/

[57] MapReduce. https://en.wikipedia.org/wiki/MapReduce

[58] Apache Yarn. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[59] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *SOSP*, 29–43. 2003

[60] Facebook has the world's largest Hadoop cluster!.
http://hadoopblog.blogspot.com/2010/05/facebook-has-worlds-largest-hadoop.html

[61] NoSQL at Netflix. https://netflixtechblog.com/nosql-at-netflix-e937b660b4c

[62] Apache Hadoop Ecosystem. http://blog.newtechways.com/2017/10/apache-hadoop-ecosystem.html

[63] Use case study of Hive/ Hadoop. https://www.slideshare.net/evamtse/hive-user-group-presentation-from-netflix-3182010-3483386

[64] How Does Namenode Handles Datanode Failure in Hadoop Distributed File System?. https://www.geeksforgeeks.org/how-does-namenode-handles-datanode-failure-in-hadoop-distributed-file-system/

[65] SQL and NoSQL databases. https://www.researchgate.net/figure/SQL-and-NoSQL-databases_fig3_299535734

[66] Query Processing in DBMS. https://www.javatpoint.com/query-processing-in-dbms

[67] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD Conf.*, 23–34, 1979.

[68] A. Swami and K. B. Schiefer. On the estimation of join result sizes. In *Proc. 4th Int. Conf. Extending Database Technol.: Adv. Data- base Technol.*, 287–300. 1994.

[69] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. In *PVLDB*, 1997.

[70] A. Hulgeri and S. Sudarshan. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In *PVLDB*, 2002.

[71] P. Bizarro, N. Bruno, and D. DeWitt. Progressive Parametric Query Optimization. In *Trans. on KDE*, 2009.

[72] S. Ganguly. Design and Analysis of Parametric Query Optimization Algorithms. In *PVLDB*, 1998.

[73] D. Kossmann. The State of the Art in Distributed Query Processing, In *ACM Comput. Survey* 32, 4, 422–469, 2000.

[74] A. F. Cardenas. Analysis and performance of inverted data base structures. In *Commun. ACM*, 18, 5, 253–263, 1975.

[75] G. Graefe and D. J. DeWitt, The EXODUS Optimizer Generator, In *Proc. ACM SIGMOD* Conf., 1987.

[76] G. Graefe and W.J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. of the ICDE*, 209–218, 1993.

[77] G. Graefe 1995. The Cascades Framework for Query Optimization. In *IEEE Data Eng. Bull*, 1995.

[78] E. Begoli, J. Camacho-Rodruguez, J. Hyde, M.J. Mior, D. Lemire. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data*, 221–230, 2017.

[79] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS*, 1998.

[80] M. Chawla, V. Baniwal. Optimization in the catalyst optimizer of Spark SQL. In *Turkish Journal of Electrical Engineering and Computer Sciences*, 26, 5, 2489-2499, 2018.

[81] Amazon EC2 P2 Instances, https://aws.amazon.com/ec2/instance-types/p2/

[82] Spark SQL Optimization – Understanding the Catalyst Optimizer. https://data-flair.training/blogs/spark-sql-optimization/

[83] S. Manegold, P. Boncz, M.L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*, 2002.

[84] Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/

[85] T. Sellis. Multiple-Query Optimization. In ACM Transactions on Database Systems (TODS), 13, 1, 23-52, 1988.

[86] P. Roy, S. Sudarshan. Multi-Query Optimization. In: *Encyclopedia of Database Systems. Springer, Boston, MA.*, 2009.

[87] N. Laurens. Exploring Query Re-Optimization, 2021.

[88] M. Emmerich, A. Deutz. Multiple Objective Decision Making — Methods and Applications. In *Natural Computing*, 17, 3, 585-609, 2018.

[89] I. Trummer and C. Koch. Approximation Schemes for Many-Objective Query Optimization. In *SIGMOD Conf.*, 2014.

[90] Z. Xu, Y. C. Tu, and X. Wang. PET: Reducing Database Energy Cost via Query Optimization. In *PVLDB*, 2012.

[91] Z. Abul-Basher, Y. Feng, and P. Godfrey. Alternative Query Optimization for Workload Management. In *Database and Expert Systems Applications*, 2012.

[92] S. Agarwal, A. Iyer, and A. Panda. Blink and It's Done: Interactive Queries on Very Large Data. In *PVLDB*, 2012.

[93] H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. E. Ioannidis. Schedule Optimization for Data Processing Flows on the Cloud. In *SIGMOD Conf.*, 2011.

[94] F. Helff. Interactive Multi-Objective Query Optimization in Mobile-Cloud Database. 2016.

[95] F. Song, K. Zaouk, C. Lyu, A. Sinha, Q. Fan, Y. Diao, P. Shenoy. Boosting Cloud Data Analytics using Multi-Objective Optimization. 2021.

[96] F. Helff. Weighted sum model for multi-objective query optimization for mobile-cloud database environments. In *CEUR Workshop Proceedings*, 1558, 2016.

[97] I. Das, J.E. Dennis. A closer look at drawbacks of minimizing weighted sums of objectives for Pareto set generation in multicriteria optimization problems. In *Structural Optimization* 14, 63–69, 1997.

[98] H. Kllapi, I. Pietri, V. Kantere, Y. Ioannidis. Automated Management of Indexes for Dataflow Processing Engines in IaaS Clouds. In *Advances in Database Technology - EDBT*, 169-180, 2020.

[99] I. Trummer, C. Koch. An Incremental Anytime Algorithm for Multi-Objective Query Optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1941-1953, 2015.

[100] A. Christoforou, A. Andreou. An effective resource management approach in a FaaS environment. In *CEUR Workshop Proceedings*, 2330, 2-8, 2019.

[101] A. Sathya Sofia, P. GaneshKumar. Multi-objective Task Scheduling to Minimize Energy Consumption and Makespan of Cloud Computing Using NSGA-II. In Journal of Network and Systems Management, 26, 2, 463-485, 2018.

[102] G. Peng. Multi-objective Optimization Research and Applied in Cloud Computing. Proceedings - 2019 IEEE 30th International Symposium on Software Reliability Engineering Workshops, ISSREW 2019, 97-99, 2019.

[103] T. Le, V. Kantere, L. d'Orazio. Dynamic estimation and Grid partitioning approach for Multi-Objective Optimization Problems in medical cloud federations. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2020.

[104] F. Azimzadeh and F. Biabiani. Multi-objective job scheduling algorithm in cloud computing based on reliability and time. In 2017 3rd International Conference on Web Research, ICWR 2017, 96-101, 2017.

[105] A. A. Priyanto, Adiwijaya, W. Maharani. Implementation of ant colony optimization algorithm on the project resource scheduling problem. In *International Conference on Telecommunication (ICTel),* 2009.

[106] G. Guoning, H. Ting-Lei and G. Shuai. Genetic simulated annealing algorithm for task scheduling based on cloud computing environment. In *Proceedings of the International Conference on Intelligent Computing and Integrated Systems*, 2010.

[107] M. Emmerich, A. Deutz. A tutorial on multiobjective optimization: fundamentals and evolutionary methods. In *Natural Computing*, 3, 17, 585-609, 2018.

[108] C. Wang, L. Gruenwald, L. d'Orazio and E. Leal. Cloud Query Processing with Reinforcement Learning-Based Multi-objective Re-optimization. In *C. Attiogbé, & S. Ben Yahia (Eds.), Model and Data Engineering - 10th International Conference, MEDI 2021*, *Proceedings*, 141-155, 2021.

[109] M. Handaoui, J. Dartois, J. Boukhobza, O. Barais and L. d'Orazio, ReLeaSER: A Reinforcement Learning Strategy for Optimizing Utilization Of Ephemeral Cloud Resources. In *2020 IEEE International Conference on Cloud Computing Technology and Science,* 2020.

[110] Amazon DynamoDB Creating an Index.
https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SQLtoNoSQL.Indexes.Creating.html

[111] M. HoseinyFarahabady, J. Taheri, Z. Tari, A. Zomaya. A dynamic resource controller for a lambda architecture. In *Proceedings of the International Conference on Parallel Processing*, 332–341, 2017.

[112] Z.Karampaglis, A. Gounaris, Y. Manolopoulos. A Bi-objective Cost Model for Database Queries in a Multi-cloud Environment, In *Proceedings of the 6th International Conference on Management of Emergent Digital EcoSystems*. 2014.

[113] What Is Cloud Computing? Explore The Services And Deployment Models.
https://www.c-sharpcorner.com/article/what-is-cloud-computing-explore-the-services-and-deployment-models/