Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Τεχνικές συμπίεσης για την ανάλυση δεδομένων μεγάλης κλίμακας

Διπλωματική Εργασία

του

Κυριτσά Γεωργίου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εργαστήριο Υπολογιστικών Συστημάτων

Αθήνα, Οκτώβριος 2020

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Τεχνικές συμπίεσης για την ανάλυση δεδομένων μεγάλης κλίμακας

Διπλωματική Εργασία

του

Κυριτσά Γεωργίου

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Σεπτεμβρίου 2020

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
……..	……..	……..
Νεκτάριος Κοζύρης Καθηγητής ΕΜΠ	Γεώργιος Γκούμας Επίκουρος Καθηγητής ΕΜΠ	Δημήτριος Τσουμάκος Αναπληρωτής Καθηγητής Ιονίου Πανεπιστημίου

Εργαστήριο Υπολογιστικών Συστημάτων

Αθήνα, Οκτώβριος 2020

(Υπογραφή)

……

Γεώργιος Κυριτσάς.
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Περίληψη

Στις μέρες μας ο ρυθμός παραγωγής δεδομένων αυξάνεται με γοργούς ρυθμούς, ξεπερνώντας κατά πολύ το ρυθμό αύξησης της υπολογιστικής ισχύος. Η αξιοποίηση αυτού του όγκου δεδομένων μπορεί να οδηγήσει σε βαθύτερη κατανόηση συμπεριφορών και συστημάτων, όπως για παράδειγμα της λειτουργίας των ανθρώπινων κυττάρων ή των κινήσεων του χρηματιστηρίου. Μια ευρέως διαδεδομένη λύση για την επεξεργασία μεγάλου όγκου δεδομένων είναι αυτή των κατανεμημένων συστημάτων, δηλαδή ενός συνόλου διασυνδεδεμένων υπολογιστών, οι οποίοι λειτουργούν σαν ένα ενιαίο υπολογιστικό σύστημα αυξημένων δυνατοτήτων. Μια άλλη λύση που κερδίζει συνεχώς έδαφος είναι η χρήση συστημάτων που χρησιμοποιούν την κύρια μνήμη για την επεξεργασία των δεδομένων. Καθώς η κύρια μνήμη είναι πολύ ταχύτερη από το δίσκο, τα συστήματα αυτά μπορούν να επιτύχουν τάξεις μεγέθους καλύτερες επιδόσεις σε σχέση με τα συμβατικά. Το πρόβλημα είναι ότι η χωρητικότητα της κύριας μνήμης είναι κατά πολύ μικρότερη από αυτή ενός δίσκου.

Σκοπός της παρούσας διπλωματικής είναι να εξετάσουμε τη χρήση της συμπίεσης στον τομέα της ανάλυσης δεδομένων μεγάλης κλίμακας. Εξετάζουμε τους τρόπους με τους οποίους μπορούμε να συμπιέσουμε δεδομένα, ώστε να χωρέσουν στη μνήμη, καθώς και την επίδραση της συμπίεσης στην απόδοση του συστήματος. Έχοντας τα δεδομένα στην κύρια μνήμη, εξαλείφεται ένα σημαντικό κομμάτι καθυστέρησης, αυτό της μεταφοράς δεδομένων από το δίσκο. Προκειμένου να εξετάσουμε αυτή τη προσέγγιση, δημιουργήσαμε το hybrid columnar, ένα σύστημα συμπίεσης δεδομένων και εκτέλεσης ερωτημάτων απευθείας στη μνήμη, χωρίς να έχει προηγηθεί αποσυμπίεση τους. Στο σύστημα αυτό υλοποιήσαμε διάφορες τεχνικές συμπίεσης με σκοπό να μελετήσουμε τη συμπεριφορά τους, τόσο σε χώρο όσο και σε χρόνο, ανάλογα με τα χαρακτηριστικά του συνόλου δεδομένων. Επίσης συγκρίναμε το σύστημα που υλοποιήσαμε, με ένα από τα κυριότερα και ευρέως χρησιμοποιούμενα συστήματα στο χώρο της ανάλυσης δεδομένων, το Parquet.

Λέξεις Κλειδιά

Big Data, ρυθμός παραγωγής δεδομένων, ρυθμός αύξησης της υπολογιστικής ισχύος, κατανεμημένα συστήματα, συστήματα βασισμένα στην κύρια μνήμη (IMDBs), επεξεργασία δεδομένων κατά στήλη, συμπίεση δεδομένων, Apache Spark, Apache Parquet

Abstract

The growth of data being created every year far outpaces the advancements in computing performance and the disparity between them is expected to grow. By analyzing and exploiting the vast amount of data, new insight on systems and behaviors, such as the inner workings of human cells or stock market movements can be gained. Distributed systems are an effective and popular solution for taming the vast amount of data produced. A distributed system is composed of a set of common computers, acting as a single computer with combined computing and storage capacity. Another option that is gaining ground lately is the use of in-memory databases (IMDBs), which use main memory (RAM) as the primary means of data storage. As RAM is much faster than spinning and even solid-state disks, these systems achieve performance orders of magnitude greater than disk-based systems. The downside of this approach is that the capacity of system memory is orders of magnitude smaller than the capacity of a hard disk.

The purpose of this thesis is to evaluate the use of data compression in large scale data processing. We examine ways for data to be compressed in order to fit in main memory and the impact of compression on system performance. Having data reside in main memory, a big bottleneck is eliminated, that of data movement between memory and disk. In order to evaluate this approach, we created hybrid columnar, a system that stores and queries data directly in memory, without prior decompression. In this system we implemented various compression schemes in order to evaluate their performance regarding both time and space, depending on the characteristics of the dataset. We also compare the system we created with Apache Parquet, one of the most established compressed data formats in the field of large-scale data processing.

Keywords

Big Data, growth of data being created, advancements in computing performance, distributed systems, in-memory databases (IMDBs), column stores, data compression, Apache Spark, Apache Parquet

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τους ανθρώπους του εργαστηρίου υπολογιστικών συστημάτων (CSLab) στο σύνολό τους για τις γνώσεις, τις εμπειρίες και την αγάπη για το αντικείμενο που μου μετέδωσαν κατά τη διάρκεια των σπουδών μου στο πολυτεχνείο. Ιδιαίτερα θα ήθελα να ευχαριστήσω τον κ. Νεκτάριο Κοζύρη για την επίβλεψη της εργασίας μου, καθώς και τον Ιωάννη (Γιάγκο) Μυτιλήνη για την καθοδήγηση του σε όλη τη διάρκεια της εργασίας. Θα ήθελα επίσης να ευχαριστήσω την οικογένειά μου για την στήριξη που μου παρείχε όλα αυτά τα χρόνια. Τέλος, οφείλω ένα μεγάλο ευχαριστώ στη σύζυγό μου, Φωτεινή, της οποίας η στήριξη και η εμψύχωση σε όλη τη διάρκεια των σπουδών μου ήταν καθοριστική.

Στη Φωτεινή
για την στήριξη και την υπομονή της

Στην κόρη μου
για την ύπαρξή της

Περιεχόμενα

1. Εισαγωγή

1.1. Ρυθμός δημιουργίας δεδομένων

Τα τελευταία χρόνια παρατηρείται ραγδαία αύξηση του ρυθμού παραγωγής δεδομένων. Πόσο μεγάλη είναι αυτή η αύξηση; Έρευνα της IDC για λογαριασμό της DELL EMC αναφέρει πως το 2020 αναμένεται ως ανθρωπότητα να έχουμε συσσωρεύσει 40 zettabytes δεδομένων. Σύμφωνα με την ίδια έρευνα, το 2010 είχαμε συσσωρεύσει μόλις 1.2 zettabytes [1]. Σε νεότερη έρευνα της IDC για λογαριασμό της Seagate [2], προβλέπεται πως από 33 Zettabytes το 2018 ο αριθμός αυτός θα ανέλθει στα 175 Zettabytes το 2025. Αυτός ο ρυθμός αύξησης των δεδομένων, σχεδόν κατά 2x ανά δυο έτη, τροφοδοτείται από μια σειρά από τεχνολογίες και κοινωνικά φαινόμενα. Κάποιες από τις κυριότερες πηγές αύξησης του όγκου των δεδομένων είναι το user generated content, το οποίο οδηγείται κατά κύριο λόγο από τη δυνατότητα εγγραφής εικόνας και βίντεο των σύγχρονων smartphones και τη δυνατότητα άμεσης αποστολής του περιεχομένου στα κοινωνικά δίκτυα, όπως για παράδειγμα το Facebook, το Instagram, το Twitter και το YouTube. Το 2019 κάθε λεπτό προβάλλονται 4,5 εκατομμύρια βίντεο στο YouTube, ενώ προβάλλονται 694.444 ώρες βίντεο στο Netflix [3]. Η δυνατότητα άμεσου σχολιασμού αυτών των δεδομένων στις πλατφόρμες αυτές δημιουργεί με τη σειρά της νέα δεδομένα. Οι ανάγκες επικοινωνίας των ανθρώπων με κάθε μορφή, όπως tweets, γραπτά μηνύματα, μήνυμα email, ηχητική ή κλήση βίντεο, σύντομα βίντεο ή gifs αποτελούν σημαντικό κομμάτι της δημιουργίας νέων δεδομένων σε καθημερινή βάση. Στην ετήσια έκθεση της DOMO «Data Never Sleeps 7.0» [3] αναφέρεται ότι κάθε λεπτό ανεβαίνουν στο Twitter 511.200 tweets, αποστέλλονται 188.000.000 emails, ενώ πραγματοποιούνται 231.840 κλήσεις μέσω Skype. Είναι σημαντικό επίσης να αναφέρουμε ότι κάθε συναλλαγή που κάνουμε με υπηρεσίες που δεν έχουν να κάνουν άμεσα με ανταλλαγή δεδομένων, όπως πλατφόρμες ενοικίασης κατοικιών (Airbnb), υπηρεσίες ταξί (Uber, Beat) δημιουργούμε νέα δεδομένα τόσο όταν ανεβάζουμε μια αγγελία όσο και με την κάθε επαφή μας με την υπηρεσία όπως πχ οι διαδρομές στο ταξί.

Επιπλέον, με την αύξηση του εύρους ζώνης που είναι διαθέσιμο στους χρήστες, χάρη στα δίκτυα νέας γενιάς (5G, VDSL, Fiber to The Home) ο ρυθμός παραγωγής νέων δεδομένων αναμένεται να συνεχίσει να αυξάνεται με γοργούς ρυθμούς.

Οι πλατφόρμες ροών δεδομένων (streaming platforms) όπως το Netflix και το Spotify όχι μόνο δημιουργούν μεγάλη κίνηση στο δίκτυο μέσω της αποστολής βίντεο και μουσικής αντίστοιχα, αλλά δημιουργούν με τη σειρά τους νέα δεδομένα όπως στατιστικά χρήσης της υπηρεσίας ανά χρήστη, χώρα, είδος ταινίας ή μουσικής κλπ. Ένας άλλος παράγοντας που συμβάλλει στην αύξηση του ρυθμού παραγωγής δεδομένων είναι οι κάθε λογής έξυπνες συσκευές. Η παραγωγή οικονομικών αισθητήρων κάθε είδους (θερμοκρασίας, φωτεινότητας, ατμοσφαιρικής πίεσης, καρδιακών παλμών κλπ.) σε συνδυασμό με τους υπολογιστές ολοκληρωμένους σε μια ψηφίδα (System on Chip ή αλλιώς SoC) οι οποίοι ενσωματώνουν σε ένα ολοκληρωμένο Chip όλες τις βασικές λειτουργίες ενός υπολογιστικού συστήματος (CPU, RAM, WiFi, Bluetooth κλπ.) έχει δημιουργήσει μια σειρά από έξυπνες συσκευές (πχ., smartwatches, έξυπνες λάμπες, θερμοστάτες) οι οποίες λειτουργούν με εξαιρετικά χαμηλή κατανάλωση ενέργειας και παίρνουν διαρκώς μετρήσεις τις οποίες αποστέλλουν στο διαδίκτυο.

Εκτός του ανθρώπου, πληροφορίες πλέον ανταλλάσσουν και οι ίδιες οι μηχανές μεταξύ τους. Τα τελευταία χρόνια όλο και περισσότερες συσκευές, όπως ψυγεία, τηλεοράσεις και άλλες οικιακές αλλά και βιομηχανικές συσκευές αποκτούν δυνατότητα σύνδεσης στο διαδίκτυο, συνθέτοντας αυτό που αποκαλούμε Internet of Things (IoT). Αυτό τους δίνει τη δυνατότητα της μεταξύ τους συνεργασίας καθώς και της καλύτερης επαφής με τον άνθρωπο.

Η παραγωγή τόσο μεγάλου όγκου δεδομένων θέτει νέες προκλήσεις στην αποτελεσματική αποθήκευση και επεξεργασία τους. Η ανάγκη αποθήκευσης των δεδομένων είναι κάτι κατανοητό. Οι χρήστες περιμένουν τα δεδομένα που ανεβάζουν στο διαδίκτυο να είναι διαθέσιμα κάθε στιγμή. Επίσης τα δεδομένα καταγραφής των έξυπνων μετρητών πρέπει να είναι διαθέσιμα για αναφορά, για παράδειγμα οι τιμές της θερμοκρασίας κάθε μέρα, τα τελευταία 40 έτη σε διάφορες περιοχές του πλανήτη.

1.2. Η ανάγκη επεξεργασίας τεράστιου όγκου δεδομένων

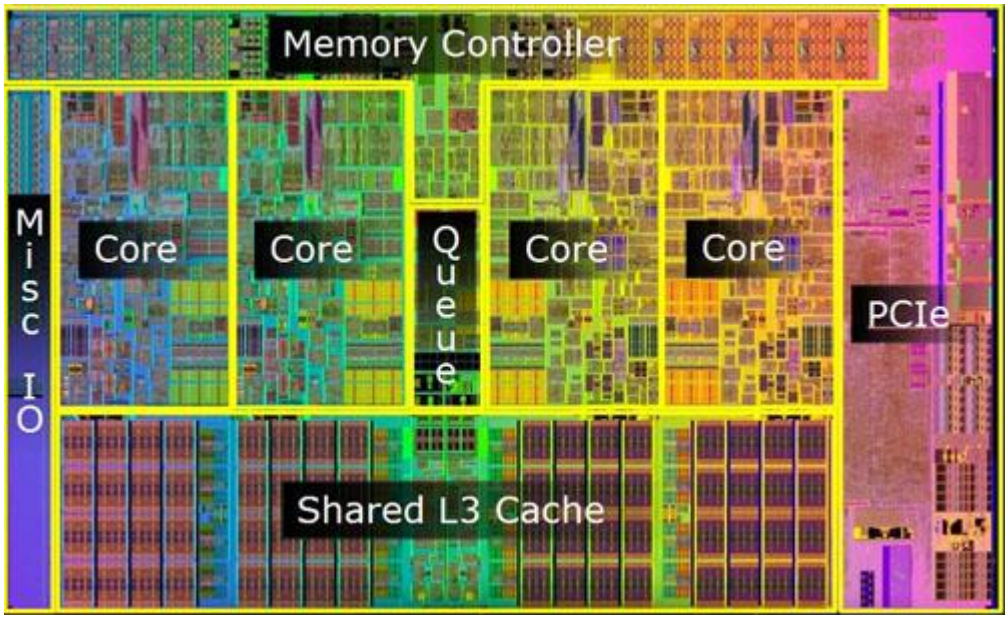
Σήμερα, ο τεράστιος όγκος των δεδομένων που παράγονται καθημερινά μπορεί να επηρεάσει δραστικά κάθε κλάδο και ανθρώπινη δραστηριότητα. Για παράδειγμα αναλύοντας ιατρικά δεδομένα μπορούμε να βρούμε συσχετίσεις μεταξύ αύξησης μιας σειράς χαρακτηριστικών, όπως συσσώρευση ουσιών στο αίμα και συγκέντρωση πρωτεϊνών, με την πιθανότητα εκδήλωσης διάφορων ασθενειών. Σύμφωνα με έρευνα της McKinsey [4] η χρήση δεδομένων μεγάλης κλίμακας στην Ιατρική έχει τη δυνατότητα να συσχετίσει διάφορες μεταβολές στον οργανισμό και να βοηθήσει στην αποτελεσματικότερη πρόληψη. Αντίστοιχα, κοιτώντας τις θερμοκρασίες περιοχών του πλανήτη σε βάθος χρόνου μπορούμε να βρούμε αυτές που παρουσιάζουν την μεγαλύτερη αύξηση στη θερμοκρασία αλλά και να συνδέσουμε την αύξηση αυτή με πιθανά αίτια όπως αύξηση διοξειδίου του άνθρακα ή μείωση του πρασίνου στις περιοχές αυτές. Αλλά και επιχειρήσεις μπορούν να καταλήξουν σε χρήσιμα συμπεράσματα για τους πελάτες τους όπως για παράδειγμα ποιες ιστοσελίδες επισκέπτονται συχνά, τι άλλα προϊόντα αγοράζουν παράλληλα με το δικό τους, σε ποιες περιοχές ή ηλικιακές ομάδες έχουν μεγαλύτερη απήχηση κλπ. Η γνωστότερη ίσως υπηρεσία streaming ταινιών, το Netflix χρησιμοποιεί εκτενώς στατιστικά από τα interactions των χρηστών με την εφαρμογή, όπως ποιες ταινίες τους άρεσαν ή πόσο χρόνο παρακολούθησαν μια σειρά. Εφαρμόζοντας αλγορίθμους μηχανικής μάθησης πάνω στα δεδομένα των χρηστών του μπορεί να ανακαλύπτει συσχετίσεις μεταξύ των ταινιών που άρεσαν στο χρήστη και ταινιών που είναι πιθανό να του αρέσουν στο μέλλον, δημιουργώντας έτσι αποτελεσματικότερα recommender systems [5]. Σύμφωνα με το Netflix, το 75% του περιεχομένου που βλέπουν οι χρήστες έχει προταθεί από ένα recommender system [6]. Τα συστήματα αυτά υπολογίζεται πως συνεισφέρουν κάθε χρόνο 1 δις. δολάρια στα έσοδα της υπηρεσίας. Όπως γίνεται αντιληπτό, από την ανάλυση δεδομένων μεγάλης κλίμακας μπορεί να προκύψει νέα γνώση για την πρόληψη και καταπολέμηση ασθενειών, την επιβράδυνση ή και αναστολή της υπερθέρμανσης του πλανήτη καθώς και για την αύξηση της κερδοφορίας των επιχειρήσεων ή ακόμα και δημιουργία καλύτερων προϊόντων.

Αφού λοιπόν καταλάβαμε γιατί θα πρέπει να επεξεργαστούμε όλα αυτά τα δεδομένα, μένει να βρούμε τον κατάλληλο τρόπο για να το πραγματοποιήσουμε.

1.3. Ρυθμός αύξησης υπολογιστικής ισχύος

Είπαμε παραπάνω ότι ο ρυθμός παραγωγής δεδομένων αυξάνεται συνεχώς, οδηγώντας σε διπλασιασμό των διαθέσιμων δεδομένων κάθε 2 χρόνια. Στο διάσημο πλέον ρητό του, το 1977 ο Gordon Moore, συνιδρυτής της Intel ισχυρίστηκε πως ο αριθμός των ολοκληρωμένων κυκλωμάτων που μπορούν να ενσωματωθούν σε ένα chip θα διπλασιάζεται κάθε δυο χρόνια [7]. Παραδόξως ο ισχυρισμός αυτός προσεγγιστικά ισχύει ακόμα και σήμερα, ενώ είναι ευρύτερα γνωστός ως ο Νόμος του Moore (Moore's Law). Ο διπλάσιος όμως αριθμός transistor δεν ισοδυναμεί σε διπλασιασμό της απόδοσης, καθώς μεγάλο μέρος των transistor δεν χρησιμοποιείται για τη μονάδα επεξεργασίας, αλλά σε άλλα βοηθητικά υποσυστήματα όπως κρυφές μνήμες, ελεγκτές μνήμης (integrated memory controllers ή αλλιώς IMC) και για τη διασύνδεση των υποσυστημάτων αυτών μεταξύ τους.

Αυτό φαίνεται και στην παρακάτω εικόνα [8] όπου παρουσιάζεται το die ενός intel 4-core επεξεργαστή:



Εικόνα 1: Intel Lynnfield die shot.

Επιπλέον υπάρχει ένα όριο στο πόσους υπολογιστικούς πυρήνες μπορούμε να προσθέσουμε σε ένα chip, καθώς μετά θα πρέπει να αυξήσουμε αντίστοιχα τις κρυφές μνήμες και τους memory controllers προκειμένου να μην έχουμε starvation λόγω έλλειψης bandwidth. Εκτός από αυτό, υπάρχουν θερμικά όρια σχετικά με το πόση ποσότητα θερμότητας μπορεί να απαχθεί από μια τόσο μικρή επιφάνεια. Χαρακτηριστικό είναι ότι ένας κορυφαίος server CPU το 2010 είχε 8 πυρήνες [9], ενώ το 2019 ένας αντίστοιχος έχει 64 πυρήνες [10]. Αυτή η αύξηση του αριθμού των υπολογιστικών πυρήνων κατά 8X είναι πράγματι εντυπωσιακή. Αν όμως αναλογιστούμε ότι το 2010 είχαμε 1.2 Zettabytes δεδομένων, ενώ το 2019 έχουμε αντίστοιχα 40 Zettabytes, μια άνω του 32X αύξηση μεγέθους, η αύξηση αυτή φαίνεται αισθητά μικρότερη. Εκτός από αυτό, οι επεξεργαστές αυτοί είναι μια τάξη μεγέθους πιο ακριβοί από τους απλούς επεξεργαστές που χρησιμοποιούν οι οικιακοί υπολογιστές.

1.4. Κατανεμημένα συστήματα

Οι μεγάλες εταιρίες του διαδικτύου όπως η Google και η Yahoo! ήταν οι πρώτοι που αντιμετώπισαν τέτοιο όγκο δεδομένων, ήδη από τις αρχές της χιλιετίας. Βλέποντας λοιπόν ότι υπάρχει όριο στην υπολογιστική ισχύ που μπορεί να χωρέσει σε ένα σύστημα (vertical scaling) στράφηκαν στη λύση της χρήσης πολλών απλών υπολογιστών, οι οποίοι συνδεδεμένοι μεταξύ τους με δίκτυο λειτουργούν σαν ένα σύστημα αυξημένης υπολογιστικής ισχύος [11].

Εικόνα 2: Google data center.

Ο κάθε υπολογιστής διαθέτει το δικό του λειτουργικό σύστημα, ενώ ένα λογισμικό, το οποίο ονομάζεται middleware, αναλαμβάνει να διαχειριστεί τους διαθέσιμους πόρους και να τους παρουσιάσει στον χρήστη σαν ένα ενιαίο σύστημα. Το ενδιάμεσο αυτό λογισμικό είναι η βάση αυτού που αποκαλούμε κατανεμημένο σύστημα (distributed system).

Χαρακτηριστικό σύστημα είναι το GFS της Google [12], το οποίο διαχειρίζεται τον αποθηκευτικό χώρο σε κάθε υπολογιστή και τον κάνει να εμφανίζεται σαν ενιαίος. Παράλληλα ασχολείται με θέματα που αφορούν το failover, δηλαδή την αστοχία κόμβων μέσω της αντιγραφής ενός block δεδομένων σε πολλαπλούς κόμβους. Πάνω στο σύστημα αυτό, σχεδιάστηκε το σύστημα κατανεμημένης επεξεργασίας της Google, το Map Reduce [13].

1.5. Συστήματα βασισμένα στην κύρια μνήμη

Μια κατηγορία συστημάτων που γνωρίζει μεγάλη αύξηση του ενδιαφέροντος τα τελευταία χρόνια είναι οι βάσεις δεδομένων βασισμένες στην κύρια μνήμη (In-memory databases ή IMDB). Σε αντίθεση με τα συστήματα βάσεων δεδομένων που αποθηκεύουν τα δεδομένα σε σκληρούς δίσκους, τα συστήματα αυτά χρησιμοποιούν την κύρια μνήμη για την αποθήκευση των δεδομένων αλλά και για την εκτέλεση των ερωτημάτων πάνω σε αυτά. Καθώς η κύρια μνήμη είναι τάξεις μεγέθους ταχύτερη από τους σκληρούς δίσκους, τα συστήματα αυτά μπορούν να επιτύχουν επιδόσεις κατά πολύ καλύτερες από τα συμβατικά συστήματα που αποθηκεύουν τα δεδομένα στο δίσκο. Τέτοια συστήματα χρησιμοποιούνται τόσο για επιτάχυνση των δοσοληψιών, όπως για παράδειγμα caching, όσο και για ανάλυση μεγάλου όγκου δεδομένων. Παράδειγμα συστήματος της πρώτης κατηγορίας είναι το Redis [30] και το Memcached [29], ενώ παράδειγμα συστήματος προσανατολισμένο στην επεξεργασία δεδομένων είναι το Spark [15].

Όπως τα μεγαλύτερα πλεονεκτήματα των συστημάτων αυτών, αυτό της υψηλής ταχύτητας και του χαμηλού χρόνου απόκρισης, προέρχονται από τη χρήση της κύριας μνήμης, αντίστοιχα συμβαίνει και για τα μεγαλύτερα μειονεκτήματα, όπως αυτό της απώλειας δεδομένων και της υποβάθμισης των επιδόσεων όταν ο όγκος δεδομένων ξεπερνά το μέγεθος της κύριας μνήμης. Για το πρώτο πρόβλημα, το οποίο οφείλεται στην πτητικότητα της κύριας μνήμης, χρησιμοποιούνται λύσεις όπως καταγραφή ημερολογίου (journaling) και περιοδική αποθήκευση (flushing) στο δίσκο. Το δεύτερο πρόβλημα, αυτό της υποβάθμισης επιδόσεων, οφείλεται στο μέγεθος της κύριας μνήμης, το οποίο είναι τάξεις μικρότερο από αυτό του δίσκου. Καθώς η μνήμη γεμίζει, τμήματα δεδομένων μεταφέρονται από τη μνήμη στον (πολύ πιο αργό) σκληρό δίσκο, το οποίο έχει ως αποτέλεσμα την σημαντική επιβράδυνση της επεξεργασίας των δεδομένων. Για το πρόβλημα αυτό έχουν προταθεί λύσεις όπως η δειγματοληψία και η συμπίεση δεδομένων. Στη δειγματοληψία, επιλέγεται προσεκτικά ένα υποσύνολο των δεδομένων τέτοιο ώστε αφενός ο όγκος των δεδομένων να μειωθεί αρκετά ώστε να χωράει στην κύρια μνήμη και αφετέρου να έχουμε την ελάχιστη δυνατή απώλεια πληροφορίας. Πάνω σε αυτή την ιδέα έχουν σχεδιαστεί συστήματα, όπως το BlinkDB [28], τα οποία προσφέρουν σημαντική επιτάχυνση, με μικρή απώλεια ακρίβειας. Σε κάθε περίπτωση η μέθοδος της δειγματοληψίας είναι απωλεστική (lossy), καθώς κρατάμε μόνο ένα υποσύνολο των αρχικών δεδομένων. Στην συμπίεση δεδομένων αντίθετα, κρατάμε το πλήρες σύνολο των δεδομένων και επιδιώκουμε τη μείωση του μεγέθους τους μέσω αποδοτικών τεχνικών αναπαράστασης αλλά και μέσω εξάλειψης της επανάληψης όμοιων μεταξύ τους τιμών. Η δυνατότητα μη απωλεστικής συμπίεσης δεδομένων είναι πολύ σημαντική σε συστήματα που εκτελούν συναλλαγές, όπως για παράδειγμα τραπεζικά συστήματα αλλά και σε συστήματα ανάλυσης δεδομένων, για παράδειγμα όταν θα πρέπει να παραχθούν λεπτομερείς αναφορές (reports). Η αποδοτική συμπίεση δεδομένων παραμένει μια πρόκληση, καθώς θα πρέπει το μέγεθος των δεδομένων να μικρύνει αρκετά ώστε να χωρέσει στην κύρια μνήμη, αλλά και η διαδικασία αυτή να μην καθυστερεί σημαντικά την εκτέλεση ερωτημάτων.

1.6. Σκοπός της διπλωματικής

Σκοπός της παρούσας διπλωματικής είναι να εξετάσουμε κατά πόσο η χρήση συμπίεσης δεδομένων μπορεί να οδηγήσει σε αποδοτικότερη εκτέλεση ερωτημάτων ανάλυσης δεδομένων. Το ζητούμενο είναι να καταφέρουμε να αποθηκεύσουμε όσο το δυνατόν περισσότερα δεδομένα στην κύρια μνήμη, ενώ ταυτόχρονα το υπολογιστικό κόστος της συμπίεσης να μην επιβαρύνει δυσανάλογα την ταχύτητα εκτέλεσης των ερωτημάτων. Για το σκοπό αυτό δημιουργήσαμε το hybrid columnar, ένα σύστημα το οποίο συμπιέζει τα δεδομένα στη μνήμη και στη συνέχεια εκτελεί ερωτήματα (queries) απευθείας πάνω σε αυτά, χωρίς να έχει προηγηθεί αποσυμπίεση τους.

1.7. Δομή της εργασίας

Αρχικά κάνουμε μια εισαγωγή στην ιεραρχία μνήμης προκειμένου να εξηγήσουμε για ποιο λόγο θέλουμε να έχουμε όσο το δυνατόν περισσότερα δεδομένα στην κύρια μνήμη, αλλά και για ποιο λόγο αυτό δεν είναι πάντα εφικτό. Στη συνέχεια κάνουμε μια εισαγωγή στη συμπίεση δεδομένων ως λύση στο πρόβλημα αυτό και περιγράφουμε τις τεχνικές συμπίεσης που εξετάσαμε. Σε επόμενο κεφάλαιο περιγράφουμε τις αρχιτεκτονικές δημοφιλών προτύπων αποθήκευσης και επεξεργασίας μεγάλου όγκου δεδομένων, όπως το parquet και κάνουμε μια εισαγωγή στο σύστημα που δημιουργήσαμε, το hybrid columnar. Στη συνέχεια ακολουθεί περιγραφή της αρχιτεκτονικής του συστήματος, καθώς και του τρόπου εισαγωγής δεδομένων και εκτέλεσης ερωτημάτων. Στη συνέχεια ακολουθεί πειραματική αξιολόγηση του συστήματος για ένα πλήθος συνθετικών datasets, αλλά και πάνω σε πραγματικά δεδομένα. Σκοπός ήταν η μελέτη της συμπεριφοράς του συστήματος που δημιουργήσαμε, αλλά και των επιμέρους τεχνικών συμπίεσης που αυτό υλοποιεί, τόσο σε χώρο όσο και σε χρόνο, ανάλογα με τα χαρακτηριστικά του dataset. Επίσης συγκρίναμε το σύστημα που υλοποιήσαμε, με ένα από τα κυριότερα και ευρέως χρησιμοποιούμενα συστήματα στο χώρο της ανάλυσης δεδομένων, το parquet.

2. Κατανεμημένα συστήματα

Στο κεφάλαιο αυτό περιγράφεται πιο αναλυτικά η αρχιτεκτονική των κατανεμημένων συστημάτων.

2.1. Map Reduce

Το σύστημα αυτό αναλαμβάνει να δρομολογήσει τις εργασίες στους κόμβους του συστήματος, με τρόπο αποδοτικό σύμφωνα με τα δεδομένα που έχει ο κάθε κόμβος, την επικοινωνία των κόμβων και την ανάκαμψη από αστοχία υλικού (failover).

Η μεγαλύτερη όμως επανάσταση που έφερε το Map Reduce αφορά το προγραμματιστικό μοντέλο, δηλαδή τον τρόπο που οι προγραμματιστές γράφουν κώδικα για την επίλυση ενός προβλήματος. Ενώ ο συμβατικός τρόπος είναι οι προγραμματιστές να δημιουργούν νήματα (threads) στα οποία αναθέτουν ένα κομμάτι του προβλήματος και στη συνέχεια είναι υπεύθυνοι για το συγχρονισμό τους, στο Map Reduce η είσοδος ενός προβλήματος είναι μια σειρά από γραμμές. Κάθε γραμμή μπορεί να είναι ένας αριθμός, μια σειρά από αριθμούς ή ένα κείμενο. Στο μοντέλο αυτό υπάρχουν μόνο δυο συναρτήσεις, η Map και η Reduce. Η Map παίρνει μια γραμμή ως key-value pair, για παράδειγμα (αριθμός γραμμής, περιεχόμενα γραμμής) και με τη σειρά της αφού κάνει κάποιους μετασχηματισμούς στα δεδομένα αυτά, παράγει στην έξοδο επίσης ένα key-value pair. Στη συνέχεια μια συνάρτηση Reduce συγκεντρώνει όλα τα key-value pairs που έχουν το ίδιο key και τα ενώνει σε ένα.

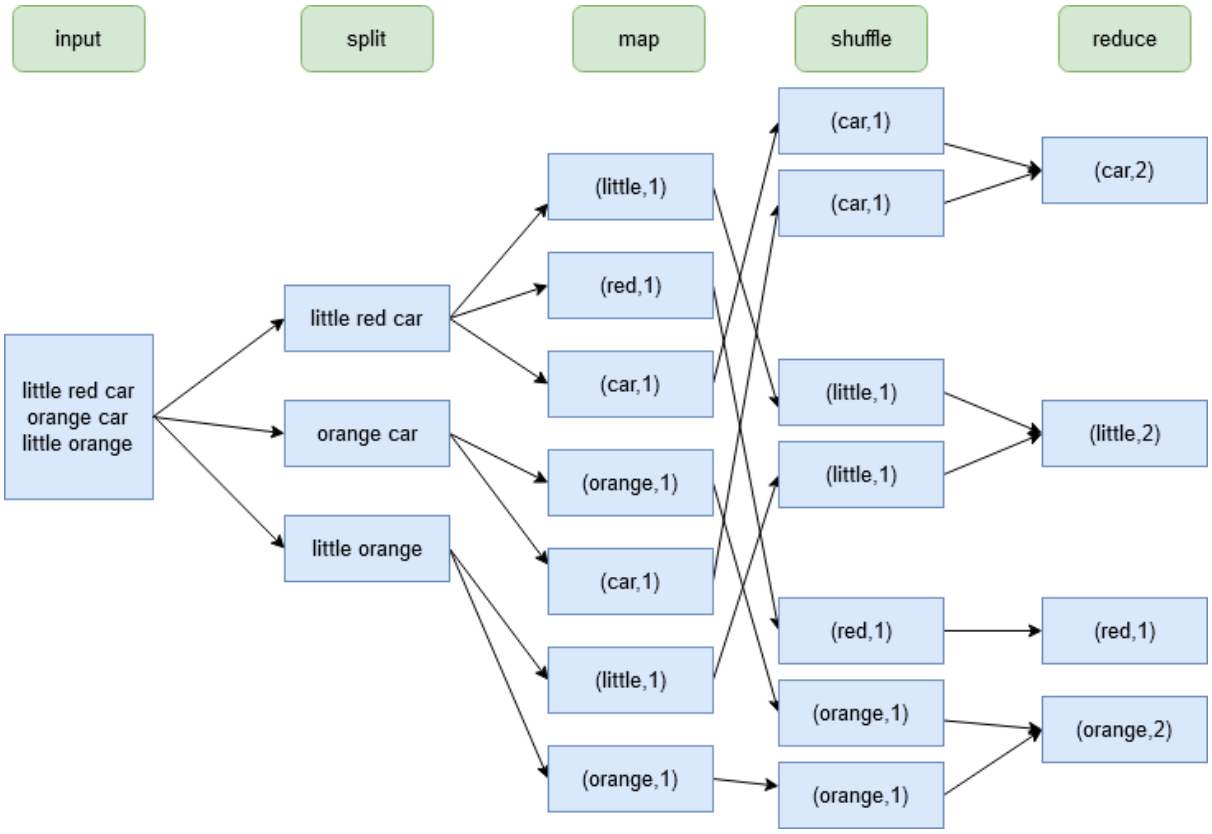
Για παράδειγμα μπορούμε να πολλαπλασιάσουμε όλα τα στοιχεία μιας λίστας με το 2 και να τα αθροίσουμε ως εξής:

```
list.map(a => a * 2).reduce((a,b) => a + b)
```

Σε ένα πρόγραμμα Map Reduce μπορεί να υπάρχουν πολλά στάδια όπου η έξοδος του ενός είναι η είσοδος του επόμενου. Για παράδειγμα μπορούμε εύκολα να υπολογίσουμε πόσες φορές εμφανίζεται μια λέξη σε ένα κείμενο ως εξής:

```
text.flatMap(line => line.split(" ")).map(word => (word,
1)).reduceByKey((a,b) => a + b)
```

Τα στάδια εκτέλεσης του προγράμματος φαίνονται στο παρακάτω σχήμα:



Το μοντέλο αυτό ήταν τόσο επαναστατικό που επηρέασε τον τρόπο που γράφονται παράλληλα προγράμματα και οδήγησε στη δημιουργία πολλών projects ανοιχτού λογισμικού που υιοθετούν την αρχιτεκτονική του, με κυριότερο το Hadoop [14].

2.2. Hadoop

Η δημιουργία του Hadoop επηρεάστηκε άμεσα από το αντίστοιχο paper της Google για το Map Reduce. Η ιδέα ήταν να φτιαχτεί ένα τέτοιο σύστημα ευρείας χρήσης και ανοιχτού κώδικα, ώστε να μπορεί να χρησιμοποιηθεί από οποιονδήποτε. Το Hadoop παρέχει ένα σύστημα κατανεμημένης αποθήκευσης αρχείων και κατανεμημένης επεξεργασίας αντίστοιχο με το Map Reduce.

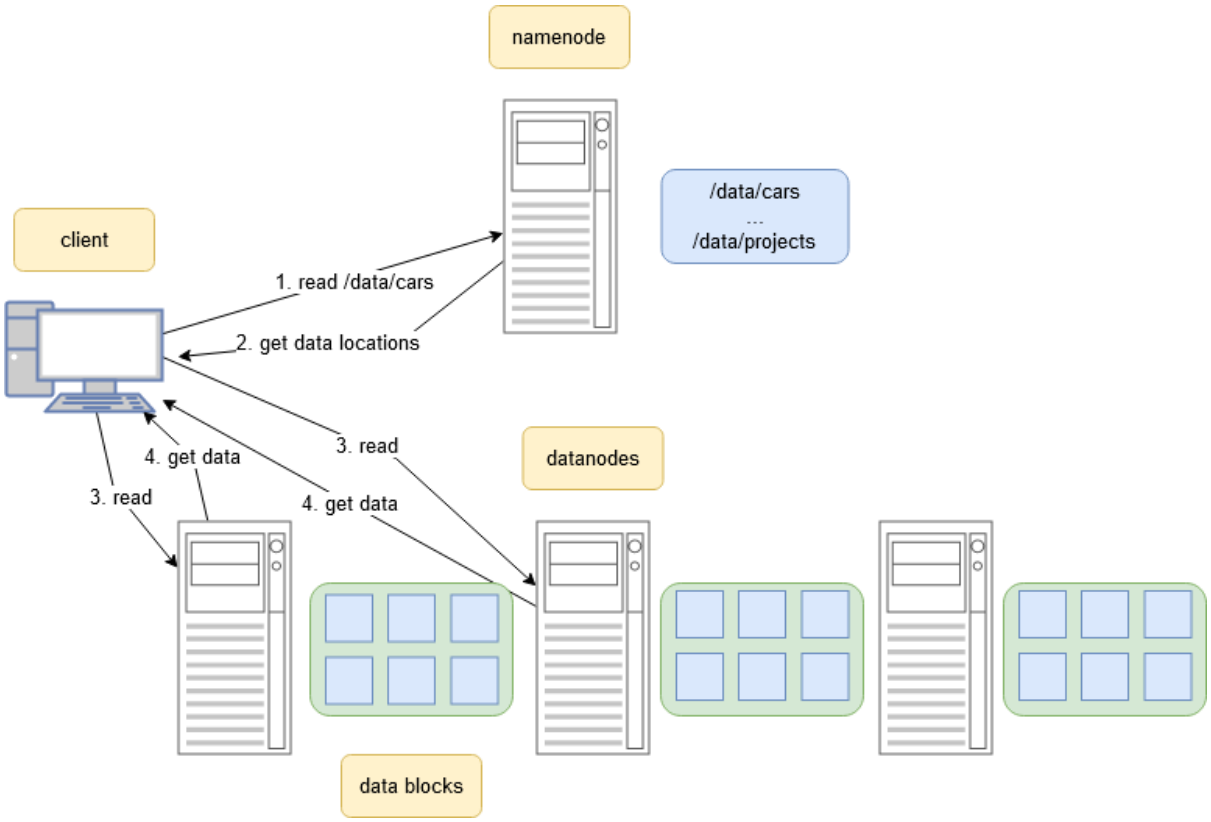
Συγκεκριμένα αποτελείται από τα παρακάτω μέρη:

- Hadoop Distributed File System (HDFS)
- Hadoop
- Map Reduce

Hadoop Distributed File System (HDFS)

Το HDFS είναι το κατανεμημένο σύστημα που διαχειρίζεται το σύστημα αρχείων του κάθε υπολογιστή και το εμφανίζει στο χρήστη σαν ένα ενιαίο αποθηκευτικό χώρο, επιτρέποντας την αποθήκευση μεγάλου όγκου δεδομένων. Το κάθε αρχείο στο HDFS χωρίζεται σε blocks συνήθως των 128 MB και τα blocks αυτά μοιράζονται στους κόμβους του συστήματος. Ένα block μπορεί να αποσταλεί σε περισσότερους του ενός κόμβους (replication) τόσο για λόγους απόδοσης όσο και προστασίας από την απώλεια δεδομένων. Οι υπολογιστικοί κόμβοι που περιέχουν block δεδομένων ονομάζονται datanodes. Στην αρχιτεκτονική του HDFS υπάρχει ένας κεντρικός κόμβος, ο namenode, που περιέχει πληροφορίες για τα blocks που αποτελούν το κάθε αρχείο, καθώς και τους κόμβους που αυτά βρίσκονται. Όταν ένας χρήστης θέλει κάποιο αρχείο, αρχικά επικοινωνεί με το namenode, ο οποίος του δίνει τις διευθύνσεις των κόμβων που έχουν τα blocks του αρχείου. Στη συνέχεια ο χρήστης ζητά απευθείας τα blocks από τους κόμβους αυτούς, χωρίς τη διαμεσολάβηση του namenode.

Το σχήμα αυτό φαίνεται παρακάτω:



Εικόνα 3: Διάβασμα δεδομένων από τους datanodes.

Hadoop YARN

Το σύστημα αυτό ασχολείται με τη δρομολόγηση των εργασιών στους κόμβους του συστήματος. Όταν ένας χρήστης υποβάλλει μια εργασία αυτή φτάνει στο master node. Αυτός ο κόμβος είναι υπεύθυνος για την δρομολόγηση των εργασιών στους υπολογιστικούς κόμβους, αλλά και για την παρακολούθηση της λειτουργία τους ώστε να σε περίπτωση αστοχίας ενός κόμβου να δρομολογήσει την εργασία σε άλλον κόμβο.

Map Reduce

Το προγραμματιστικό μοντέλο το οποίο περιγράψαμε παραπάνω, με βάση το οποίο γράφονται οι εφαρμογές των χρηστών.

Ο κάθε κόμβος του Hadoop cluster διαβάζει τα δεδομένα από το δίσκο του, τα επεξεργάζεται και στη συνέχεια γράφει τα αποτελέσματα πάλι εκεί. Επίσης οποιαδήποτε ενδιάμεσα αποτελέσματα στις εργασίες Map Reduce γράφονται επίσης στο δίσκο. Καθώς ο δίσκος είναι πολύ πιο αργός από την κύρια μνήμη, αυτό μπορεί να αποτελέσει bottleneck. Αυτό οδήγησε μια ομάδα ερευνητών από το πανεπιστήμιο Berkeley της California στη δημιουργία του Spark [15].

2.3. Spark

Το Spark, όπως και το Hadoop επιτρέπει την επεξεργασία δεδομένων μεγάλης κλίμακας. Αντίθετα με το Hadoop, έχει τη δυνατότητα να αποθηκεύει τα δεδομένα στη μνήμη, επιτρέποντας έως και 100 φορές πιο γρήγορη επεξεργασία δεδομένων για εφαρμογές που κάνουν έντονη χρήση IO [16]. Τα δεδομένα αποθηκεύονται στη μνήμη ως RDDs (Resilient Distributed Datasets) και μπορούν να χρησιμοποιηθούν άμεσα για επεξεργασία. Το αποτέλεσμα της επεξεργασίας μπορεί επίσης να είναι ένα RDD και να χρησιμοποιηθεί για επόμενες εργασίες. Το Spark δεν είναι αντικαταστάτης του Hadoop, αλλά αντίθετα χρησιμοποιεί το κατανεμημένο σύστημα αρχείων του (HDFS).

Το σύστημα αποτελείται από ένα σύνολο κόμβων, τους workers, οι οποίοι αναλαμβάνουν την επεξεργασία δεδομένων, καθώς και ένα κόμβο, τον master, ο οποίος αναλαμβάνει την ανάθεση εργασιών στους workers και το συντονισμό τους.

Αρχιτεκτονική

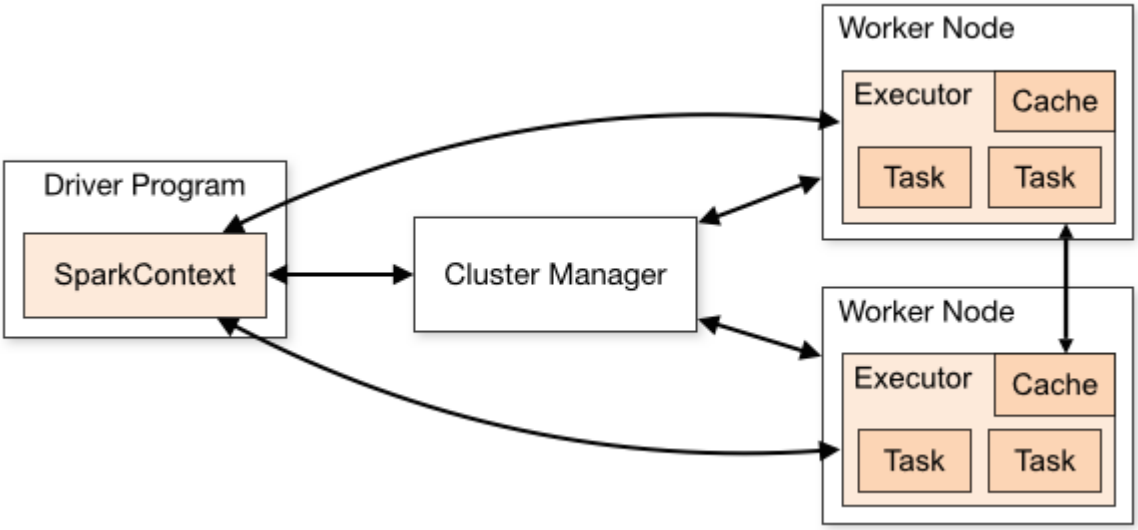
Μια εγκατάσταση spark αποτελείται από τα παρακάτω μέρη:

Worker: Ο κάθε worker είναι ένας αυτόνομος υπολογιστικός κόμβος, ο οποίος δεν έχει κάποια επικοινωνία με τους υπόλοιπους workers. Αντίθετα, περιμένει ανάθεση εργασιών από τον master, τις οποίες εκτελεί μέσω των executors που διαθέτει. Οι executors είναι διεργασίες που τρέχουν στον worker προκειμένου να εκτελεστούν οι εργασίες που έχουν ανατεθεί σε αυτόν. Οι υπολογισμοί εκτελούνται τοπικά, πάνω στα δεδομένα που διαθέτει ο worker. Συγκεκριμένα, ο κάθε executor παίρνει ένα partition του RDD και το επεξεργάζεται, με την διαδικασία αυτή να επαναλαμβάνεται μέχρι οι executors να επεξεργαστούν όλα τα partitions του worker που αντιστοιχούν στο υπό επεξεργασία RDD. Με τον τρόπο αυτό αποφεύγεται η μεταφορά δεδομένων στο δίκτυο. Μόλις οι διεργασίες εκτελεστούν, τα αποτελέσματα μεταφέρονται πίσω στον master.

Cluster Manager: Είναι υπεύθυνος για τη δέσμευση των πόρων που χρειάζεται η κάθε εφαρμογή.

Master: Ο master είναι ο μόνος που γνωρίζει την ύπαρξη όλων των workers του cluster. Σε αυτόν τρέχει ο driver, η οποία είναι η διεργασία που τρέχει τον κώδικα του χρήστη. Στη διεργασία αυτή δημιουργείται το SparkContext, το αντικείμενο μέσω του οποίου πραγματοποιείται η επικοινωνία με τους workers. Μέσω heartbeats που λαμβάνει περιοδικά από τους workers, το SparkContext γνωρίζει την κατάστασή τους καθώς και την πορεία των εργασιών. Μόλις ένας executor ολοκληρώσει τη διαδικασία που του έχει ανατεθεί, στέλνει τα αποτελέσματα πίσω στο SparkContext.

Η αρχιτεκτονική αυτή φαίνεται στο παρακάτω σχήμα [17]:



Resilient Distributed Datasets (RDD)

Κεντρικό κομμάτι της αρχιτεκτονικής του Spark είναι τα RDDs. Τα RDDs είναι συλλογές δεδομένων, μόνο για ανάγνωση, κατανεμημένες σε ένα σύνολο υπολογιστών. Σε αντίθεση με άλλες τεχνικές αποθήκευσης δεδομένων στη μνήμη μιας συστοιχίας υπολογιστών, όπως αρχιτεκτονικές κατανεμημένης μνήμης, Key-Value Stores και βάσεις δεδομένων, ένα RDD δεν περιέχει δεδομένα, αλλά το μετασχηματισμό μέσω του οποίου θα προκύψουν τα δεδομένα. Με τον τρόπο αυτό αποφεύγεται η μεταφορά δεδομένων μεταξύ των κόμβων της συστοιχίας, κάτι ιδιαίτερα αργό καθώς οι ταχύτητες του δικτύου είναι κατά πολύ μικρότερες από αυτές των μνημών και των επεξεργαστών. Οι μετασχηματισμοί αυτοί εφαρμόζονται σε ένα RDD και παράγουν ως αποτέλεσμα ένα νέο RDD [18].

Ορισμένοι γνωστοί μετασχηματισμοί είναι οι:

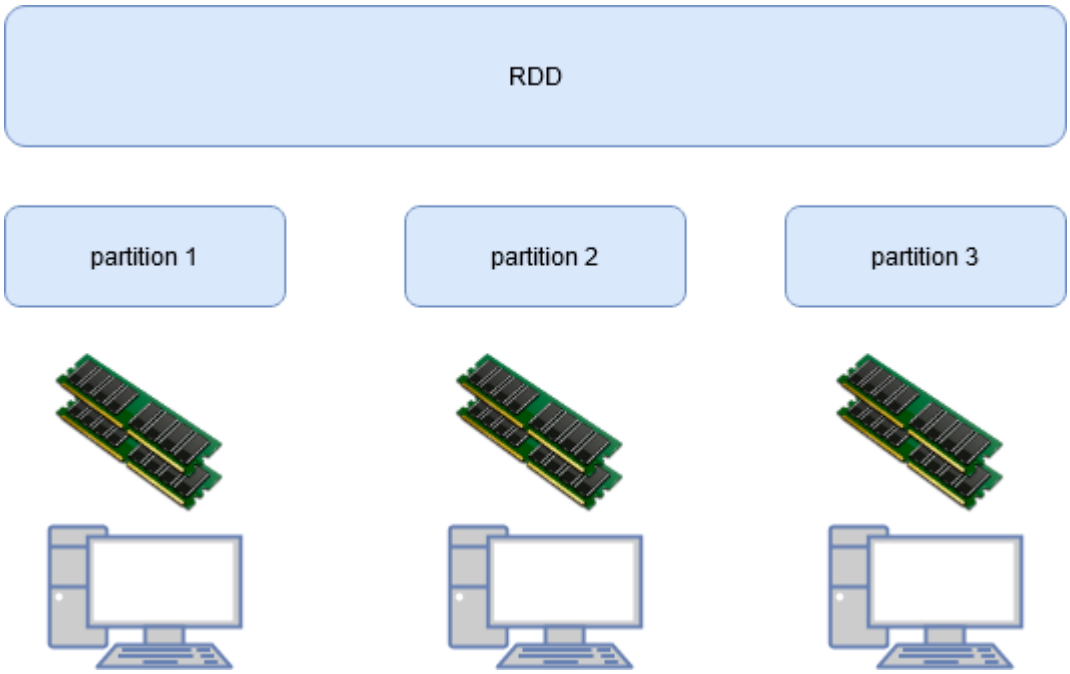
- **Map**: Εφαρμόζει μια μέθοδο σε κάθε στοιχείο του αρχικού RDD και δημιουργεί ένα νέο RDD με τα στοιχεία αυτά. Για παράδειγμα μπορούμε να αυξήσουμε τα στοιχεία ενός RDD κατά 1 με την παρακάτω εντολή: list.map(a => a + 1)
- **Filter**: Δημιουργεί ένα νέο RDD το οποίο περιέχει μόνο τα στοιχεία του αρχικού RDD, για τα οποία ισχύει ο περιορισμός που έχουμε δώσει στο filter. Για παράδειγμα για να κρατήσουμε μόνο τους αριθμούς μικρότερους του 10 γράφουμε τον παρακάτω εντολή: list.filter(a => a < 10)
- **ReduceByKey**: Παίρνει ως είσοδο ένα RDD από Key-Value pairs και επιστρέφει ένα RDD από Key-Value pairs, όπου για κάθε key έχουν συναθροιστεί όλα τα values που αντιστοιχούν σε αυτό, σύμφωνα με τη συνάρτηση συνάθροισης που έχουμε δώσει στο ReduceByKey. Για παράδειγμα αν έχουμε ένα RDD με ζευγάρια (κατάστημα, ποσό) και θέλουμε να υπολογίσουμε το συνολικό ποσό ανά κατάστημα γράφουμε: list.reduceByKey((a,b) => a + b)

Στο σημείο αυτό, τα RDD έχουν μόνο την πληροφορία για το πώς παράχθηκαν. Ακολουθώντας το παράδειγμα της οκνηρής αποτίμησης (lazy evaluation), ένα RDD θα υπολογίσει τα δεδομένα του μόνο αν χρειαστεί. Αυτό γίνεται με μια σειρά λειτουργιών που ονομάζονται πράξεις (actions). Οι πράξεις εκτελούνται πάνω στο RDD και επιστρέφουν μια τιμή, ως αποτέλεσμα των υπολογισμών πάνω στο RDD.

Ορισμένες γνωστές πράξεις είναι οι:

- **Count**: Επιστρέφει τον αριθμό των στοιχείων του RDD
- **Collect**: Επιστρέφει όλα τα στοιχεία του RDD, σε μορφή λίστας
- **Reduce**: Εφαρμόζει σε όλα τα στοιχεία του RDD τη συνάρτηση συνάθροισης που έχουμε ορίσει και επιστρέφει το αποτέλεσμα.

Μπορούμε αν θέλουμε να αποθηκεύσουμε τα αποτελέσματα ενός RDD στη μνήμη, έτσι ώστε κάθε φορά που κάποια πράξη απαιτεί τον υπολογισμό του, να επιστρέφουν άμεσα τα αποτελέσματα από τη μνήμη.



Εικόνα 4: Κατανομή ενός RDD στη μνήμη του cluster.

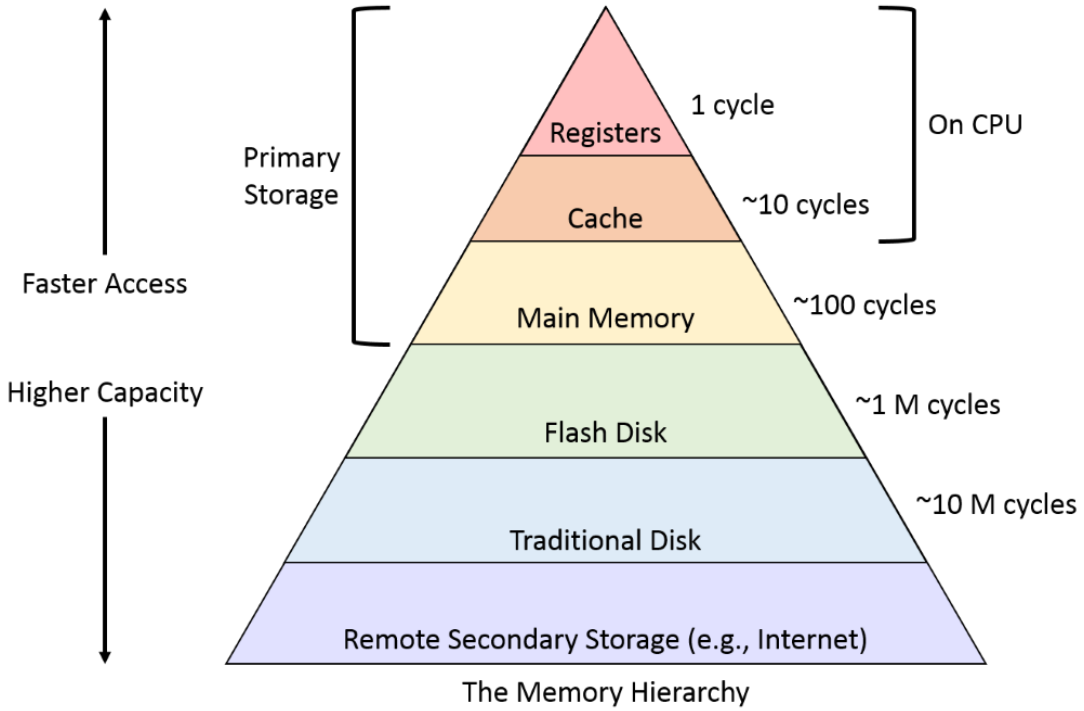
Όμως για ποιο λόγο το Spark είναι τόσο ταχύτερο από το Hadoop;

3. Ιεραρχία μνήμης

Τα σύγχρονα υπολογιστικά συστήματα βασίζονται σε μια ιεραρχία μνήμης πολλών επιπέδων, προκειμένου να συνδυάσουν τις επιδόσεις του ανώτερου επιπέδου με τη χωρητικότητα του κατώτερου [19]. Κάθε φορά που ο επεξεργαστής χρειάζεται δεδομένα και δεν τα βρίσκει σε κάποιο επίπεδο, τα αναζητά στο αμέσως επόμενο (και πιο αργό) επίπεδο.

Στη βάση της ιεραρχίας μνήμης βρίσκονται οι σκληροί δίσκοι (μαγνητικοί και SSD) οι οποίοι έχουν μεγάλη χωρητικότητα αλλά μικρή ταχύτητα. Στο πιο πάνω επίπεδο βρίσκεται η κύρια μνήμη (RAM) η οποία έχει πολύ μικρότερη χωρητικότητα αλλά και πολύ υψηλότερη ταχύτητα από το σκληρό δίσκο. Στη συνέχεια βρίσκεται μια μικρή περιοχή μνήμης (cache memory) στο εσωτερικό του επεξεργαστή η οποία κρατά δεδομένα που έχει χρησιμοποιήσει πρόσφατα ο επεξεργαστής. Στην κορυφή της ιεραρχίας βρίσκονται οι καταχωρητές του επεξεργαστή οι οποίοι παρέχουν τα δεδομένα μέσα σε ένα κύκλο λειτουργίας, αλλά η χωρητικότητά τους περιορίζεται στα ορίσματα των πράξεων που εκτελεί εκείνη τη στιγμή ο επεξεργαστής.

Παρακάτω παρουσιάζεται μια απεικόνιση της ιεραρχίας αυτής καθώς και ενδεικτικά μεγέθη για την ταχύτητα και το μέγεθος του κάθε επιπέδου [20].



The Memory Hierarchy

Η απόδοση της ιεραρχίας επηρεάζεται από την ταχύτητα όλων των επιπέδων. Για παράδειγμα αν έχουμε ένα μεγάλο dataset αποθηκευμένο στο δίσκο αυτό θα πρέπει πρώτα να διαβαστεί από εκεί πριν μετακινηθεί στην κύρια μνήμη. Αντίθετα, αν το dataset αυτό χωρούσε στην κύρια μνήμη του συστήματος ο επεξεργαστής θα το διάβαζε απευθείας από εκεί, σπαταλώντας τάξεις μεγέθους λιγότερους κύκλους περιμένοντας τα δεδομένα. Αυτό σημαίνει πως αν μπορούσαμε να μειώσουμε το μέγεθος του αρχείου ώστε να χωρέσει στη μνήμη τότε θα πετυχαίναμε τάξεις μεγέθους καλύτερες επιδόσεις. Στο σημείο έρχεται να δώσει τη λύση η συμπίεση δεδομένων.

3.1. Συμπίεση δεδομένων

Σκοπός της συμπίεσης δεδομένων είναι η μείωση του χώρου που απαιτείται για την αναπαράσταση της πληροφορίας. Η συμπίεση δεδομένων διακρίνεται σε απωλεστική (lossy) και μη απωλεστική (lossless). Με τη χρήση απωλεστικής συμπίεσης πετυχαίνουμε μεγαλύτερη μείωση του μεγέθους των δεδομένων, χάνουμε όμως ένα μέρος αυτών, λόγω της διαδικασίας της δειγματοληψίας και της αποκοπής πληροφορίας. Αντίθετα στην μη απωλεστική συμπίεση κρατάμε το σύνολο των δεδομένων, με αντίκτυπο τη μικρότερη μείωση του μεγέθους των δεδομένων. Τεχνικές συμπίεσης δεδομένων χρησιμοποιούνται εδώ και δεκαετίες για διάφορες χρήσεις όπως συμπίεση εικόνας (JPEG, BMP, GIF), ήχου (MP3, Vorbis, FLAC), βίντεο (Xvid, H.264, H.265), άλλα και δεδομένων (ZIP, RAR , 7Z, TAR). Στις εφαρμογές multimedia χρησιμοποιούμε κυρίως απωλεστική συμπίεση, με τα πιο δημοφιλή πρότυπα το MP3 και το h.264, για ήχο και κινούμενη εικόνα αντίστοιχα. Στις εφαρμογές αυτές, εκμεταλλευόμενοι την ανθρώπινη αντίληψη για την εικόνα και τον ήχο κρατάμε μόνο ένα κομμάτι της πληροφορίας και αφαιρούμε το υπόλοιπο [21]. Για παράδειγμα κάποιες συχνότητες δεν γίνονται ιδιαίτερα αντιληπτές από τον άνθρωπο, οπότε η πληροφορία που τις αναπαριστά μπορεί να αφαιρεθεί. Αντίθετα, στις εφαρμογές δεδομένων, όπως συμπίεση αρχείων ή αποστολή δεδομένων στο δίκτυο, χρησιμοποιούνται συνήθως μη απωλεστικές μέθοδοι συμπίεσης καθώς εκεί θέλουμε απόλυτη ακρίβεια στα δεδομένα. Για παράδειγμα σε μια βάση δεδομένων δεν είναι αποδεκτό να σβηστούν ή να αλλοιωθούν πληροφορίες που αφορούν ονόματα κωδικούς ή ηλικίες χρηστών. Η συμπίεση σε αυτές τις μεθόδους επιτυγχάνεται με την απαλοιφή της πλεονάζουσας πληροφορίας. Για παράδειγμα αν έχουμε το όνομα Γιώργος 280 φορές, αντί να το αποθηκεύσουμε 280 φορές μπορούμε να αποθηκεύσουμε (Γιώργος, 280). Η τεχνική αυτή η οποία λέγεται Run Length Encoding αποτελεί τη βάση για πολλές μεθόδους μη απωλεστικής συμπίεσης.

Για όλες αυτές τις χρήσεις το ζητούμενο είναι η μείωση του κόστους αποθήκευσης και μεταφοράς δεδομένων, ώστε να έχουμε λιγότερα bytes να αποθηκεύσουμε και αντίστοιχα να μεταφέρουμε πάνω από το δίκτυο.

Όσον αφορά την ανάλυση δεδομένων, μείωση του απαιτούμενου χώρου αποθήκευσης σημαίνει πως μεγαλύτερο μέρος του dataset μπορεί να χωρέσει στην κύρια μνήμη. Έχοντας τα δεδομένα στην κύρια μνήμη, έχουμε εξαλείψει ένα σημαντικό bottleneck, αυτό της μεταφοράς δεδομένων από το δίσκο.

Στο σημείο αυτό όμως, θα πρέπει να λάβουμε υπόψη πως τα δεδομένα στην κύρια μνήμη είναι συμπιεσμένα και άρα θα πρέπει να τα αποσυμπιέσουμε πριν τα επεξεργαστούμε. Συνεπώς, οι τεχνικές συμπίεσης που θα χρησιμοποιήσουμε, εκτός από τη μείωση του μεγέθους, θα πρέπει να λαμβάνουν υπόψη και το υπολογιστικό κόστος της αποσυμπίεσης, προκειμένου να μη σπαταλάμε υπολογιστική ισχύ, η οποία θα μπορούσε να χρησιμοποιηθεί για την επεξεργασία των δεδομένων.

Αν χρησιμοποιήσουμε μια τεχνική η οποία πετυχαίνει μεγάλο ποσοστό συμπίεσης αλλά είναι υπολογιστικά απαιτητική, ένα μέρος από το κέρδος που θα έχουμε από τη μείωση χώρου θα το χάσουμε λόγω του χρόνου που θα σπαταλά ο επεξεργαστής για την αποσυμπίεση των δεδομένων. Αντίθετα, αν χρησιμοποιήσουμε μια υπολογιστικά ελαφριά μέθοδο συμπίεσης, η οποία όμως δεν πετυχαίνει μεγάλο ποσοστό συμπίεσης, μεγάλο τμήμα των δεδομένων θα βρίσκεται στο δίσκο, συνεπώς ο επεξεργαστής τον περισσότερο χρόνο θα είναι αδρανής, περιμένοντας δεδομένα από το δίσκο.

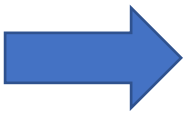
Προκειμένου λοιπόν να πετύχουμε τις μέγιστες επιδόσεις θα πρέπει να ισορροπήσουμε ανάμεσα στην ικανότητα συμπίεσης και στο υπολογιστικό κόστος της αποσυμπίεσης. Οι τεχνικές που επιλέξαμε είναι οι RLE (Run Length Encoding), Bitmap, Roaring, Delta, Bit packing και Dictionary. Παρακάτω ακολουθεί μια σύντομη περιγραφή της κάθε μιας:

3.1.1. RLE (Run Length Encoding)

Η τεχνική αυτή εκμεταλλεύεται την ύπαρξη συνεχόμενων επαναλήψεων μιας πληροφορίας σε ένα dataset. Για παράδειγμα αν σε ένα dataset έχουμε τις επαναλήψεις ονομάτων (nick, …, nick) N φορές και αμέσως μετά (george, …, george) M φορές αυτό μπορεί να αναπαρασταθεί ως (nick, N), (george, M). Προκειμένου να κρατήσουμε και την πληροφορία για τη θέση του κάθε ονόματος στο dataset μπορούμε αντί για την δυάδα να αναπαραστήσουμε τα δεδομένα με την τριάδα (nick, 0, N), (george, N, M), όπου 0 η θέση της πρώτης εμφάνισης της λέξης nick και αντίστοιχα N η θέση της πρώτης εμφάνισης της λέξης george.

Παρακάτω φαίνεται αυτή η τεχνική:

index	name
0	george
1	george
...	...
999	george
1000	nick
1001	nick
…	...
2086	nick
...	…



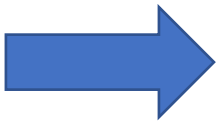
Index	name
0	(george, 0, 1000)
1	(nick, 1000, 1087)
...	…

3.1.2. Bitmap encoding

Στην τεχνική αυτή για κάθε μοναδικό στοιχείο ενός dataset αντιστοιχεί μια σειρά από bit (bit-string) με μέγεθος ίσο με το πλήθος των γραμμών του dataset, όπου στην θέση i αντιστοιχεί το 1 αν το στοιχείο αυτό υπάρχει στη γραμμή i και το 0 αν δεν υπάρχει. Για παράδειγμα αν η λέξη Γιώργος εμφανίζεται στις γραμμές 1, 2, 7, 8 αντί να γράψουμε τη λέξη Γιώργος στις γραμμές αυτές, αναπαριστούμε την πληροφορία ως Γιώργος: [0 1 1 0 0 0 0 1 1]. Έτσι για κάθε γραμμή αντί να αποθηκεύονται τα ίδια τα δεδομένα τα οποία μπορούν να καταλαμβάνουν συνήθως 4 bytes αν είναι αριθμητικά ή αρκετά bytes αναλόγως του μήκους τους αν είναι συμβολοσειρές, αποθηκεύεται μόνο ένα bit. Βεβαίως αυτό γίνεται για το κάθε μοναδικό στοιχείο του dataset, άρα χρειαζόμαστε L x C bit για την αποθήκευση του dataset, όπου L το μήκος και C το cardinality του dataset. Οπότε αν έχουμε ένα dataset με μεγάλο cardinality (αριθμός διαφορετικών στοιχείων) τότε η μέθοδος αυτή δεν είναι αποδοτική.

Παρακάτω φαίνεται αυτή η τεχνική:

index	data
0	1
1	3
2	1
3	2
4	2
5	1



index	1	2	3
0	1	0	0
1	0	0	1
2	1	0	0
3	0	1	0
4	0	1	0
5	1	0	0

Στο παράδειγμα αυτό ο αριθμός των διαφορετικών στοιχείων είναι 3 ({1,2,3}), οπότε για κάθε γραμμή αποθηκεύουμε 3 bit αντί για 32 bit (4 bytes) που απαιτεί κάθε ακέραιος. Η μέθοδος αυτή έχει το πλεονέκτημα ότι δεν επηρεάζεται από την κατανομή των δεδομένων ή το αν είναι ταξινομημένα. Αντίθετα επηρεάζεται άμεσα από το cardinality του dataset.

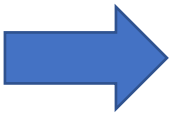
3.1.3. Roaring

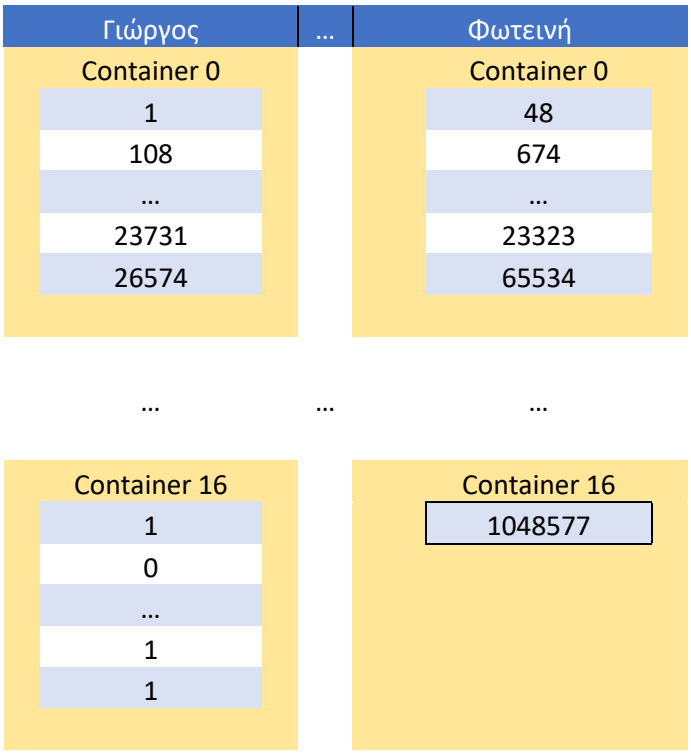
Η τεχνική του bitmap encoding αποδίδει πολύ καλά όταν έχουμε μικρό αριθμό ξεχωριστών δεδομένων (cardinality), όπως για παράδειγμα ένα dataset με ονόματα πολιτειών της Αμερικής. Αντίθετα, αν έχουμε μεγάλο cardinality η τεχνική αυτή δεν αποδίδει καλά καθώς οδηγεί σε σπατάλη χώρου. Αυτό συμβαίνει επειδή αν έχουμε cardinality πχ 1200, τότε σε κάθε γραμμή i αποθηκεύουμε 1200 bit (ένα για κάθε διαφορετικό στοιχείο) προκειμένου να δούμε αν το στοιχείο αυτό υπάρχει στη γραμμή i. Στο σημείο αυτό μπορούμε να κάνουμε την εξής παρατήρηση: αν έχουμε 1200 διαφορετικά στοιχεία αυτό σημαίνει ότι για κάθε 1 (εμφάνιση ενός στοιχείου σε μια δεδομένη γραμμή) ακολουθούν κατά μέσο όρο 1200 μηδενικά (τα υπόλοιπα στοιχεία). Φυσικά αυτό εξαρτάται από την κατανομή και σε κάποια σημεία μπορεί να υπάρχουν περισσότερα από 1200 μηδενικά, ενώ σε κάποια άλλα λιγότερα, αλλά και πάλι το συμπέρασμα παραμένει αληθές. Για να εξαλειφθεί αυτός ο πλεονασμός έχουν προταθεί πολλές τεχνικές συμπίεσης ενός bitmap, όπως για παράδειγμα η κωδικοποίηση με RLE. Η τεχνική αυτή όπως περιγράψαμε παραπάνω εξαλείφει την πλεονάζουσα πληροφορία, αποθηκεύοντας μαζί με τα δεδομένα και ένα μετρητή που δείχνει πόσες φορές αυτά εμφανίζονται στη σειρά. Για παράδειγμα τα 1200 μηδενικά θα αποθηκευτούν ως (0, 1200). Μια άλλη ενδιαφέρουσα μέθοδος συμπίεσης bitmaps, την οποία επιλέξαμε στην εργασία αυτή, είναι η κωδικοποίηση roaring [22]. Στην τεχνική αυτή το dataset χωρίζεται σε containers των 65536 στοιχείων (2^{16}). Υπάρχουν δυο τύποι containers, ένας που αποθηκεύει σε bitmaps και ένας που αποθηκεύει σε πίνακα ακεραίων των 16 bit. Η λογική είναι ότι αν ένα στοιχείο εμφανίζεται λιγότερο από 1 στις 16 φορές στην περιοχή που ορίζει το container, αντί να αποθηκευτεί 0 ή 1 στην αντίστοιχη θέση ενός bitmap, είναι πιο αποδοτικό αν απλά αποθηκευτεί σε ένα πίνακα ακεραίων η θέση στην οποία εμφανίζεται. Έτσι αν πχ η λέξη Γιώργος εμφανίζεται μόνο 10 φορές στις πρώτες 65536 θέσεις του dataset, αποθηκεύουμε σε ένα container τύπου πίνακα ακεραίων τις 10 θέσεις στις οποίες εμφανίζεται η λέξη αυτή. Με τον τρόπο αυτό δεσμεύουμε 10 x 16 = 160 bit, αντί για 1200 αν χρησιμοποιούσαμε bitmap. Αντίθετα αν σε κάποιες άλλες θέσεις πιο

κάτω εμφανίζεται πολύ συχνά (τουλάχιστον μια φορά ανά 16 γραμμές) τότε αποθηκεύεται σε bitmap container το οποίο προσφέρει πιο συμπαγή αναπαράσταση σε δεδομένα που εμφανίζονται συχνά.

Η τεχνική αυτή παρουσιάζεται παρακάτω:

index	data
0	Γιάννης
1	Γιώργος
2	Ευγενία
…	…
…	…
65534	Φωτεινή
65535	Καλλιθέα
…	…
…	…
…	…
1048576	Γιώργος
1048577	Φωτεινή
1048578	Γιώργος
…	…
…	…
1114110	Γιώργος
1114111	Γιώργος





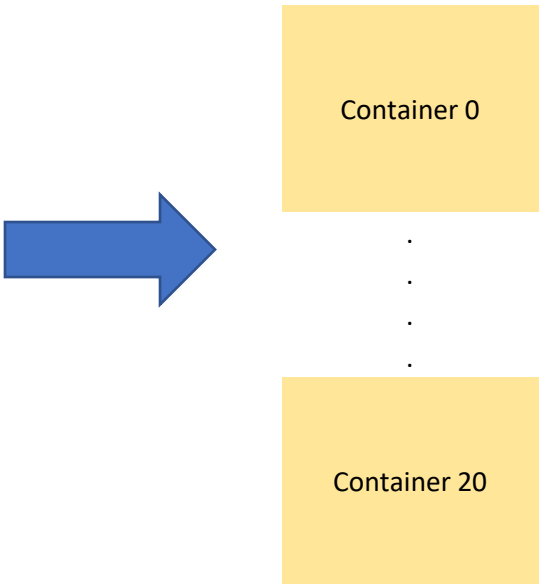
Στο παραπάνω παράδειγμα, οι πρώτες 65536 θέσεις του dataset, οι οποίες αντιστοιχούν στο πρώτο container, περιέχουν μεγάλο πλήθος διαφορετικών ονομάτων. Καθώς το κάθε όνομα εμφανίζεται λίγες φορές (λιγότερες από το 1 / 16 του μεγέθους του container) επιλέγουμε να αποθηκεύσουμε τα δεδομένα σε array container. Αντίθετα οι θέσεις του dataset που αντιστοιχούν στο container 16 περιλαμβάνουν πολύ συχνά το όνομα Γιώργος. Στην περίπτωση αυτή, για το όνομα Γιώργος δεσμεύεται ένα bitmap container, ενώ τα ονόματα που εμφανίζονται σπάνια αποθηκεύονται σε ένα array container.

3.1.4. Bit packing

Οι ακέραιοι αριθμοί σε ένα υπολογιστικό σύστημα αναπαρίστανται από τύπους δεδομένων σταθερού μήκους, ανεξάρτητα από το μέγεθος του αριθμού. Στα περισσότερα συστήματα οι ακέραιοι έχουν μήκος 32 bit και ο τύπος αυτός μπορεί να αποθηκεύσει αριθμούς από −2,147,483,647 μέχρι 2,147,483,647. Μεγάλο πλήθος datasets περιλαμβάνουν στοιχεία μικρού μεγέθους όπως ηλικία, θερμοκρασία και άλλα μεγέθη μικρού εύρους. Για παράδειγμα σε ένα dataset που περιλαμβάνει αριθμούς εύρους [0 - 100] είναι περιττό να χρησιμοποιούμε μεταβλητές των 32 bit, όταν αυτοί οι αριθμοί μπορούν να αποθηκευτούν σε $\log_2(100) = 7$ bit. Προκειμένου να αποθηκεύσουμε αυτά τα στοιχεία αποδοτικά έχουν προταθεί πολλές μέθοδοι. Η μέθοδος που σχεδιάσαμε και υλοποιήσαμε λειτουργεί ως εξής: Τα δεδομένα ενός dataset αποθηκεύονται σε containers των 1000 στοιχείων. Αρχικά τα δεδομένα αποθηκεύονται σε μια προσωρινή περιοχή αποθήκευσης (buffer), η οποία υλοποιείται με τη μορφή μιας λίστας ακεραίων. Ταυτόχρονα με την εισαγωγή του κάθε στοιχείου ενημερώνουμε και μια μεταβλητή που κρατά το μικρότερο στοιχείο και αντίστοιχα μια άλλη που κρατά το μεγαλύτερο στοιχείο που έχουμε βάλει. Αυτές οι μεταβλητές χρησιμεύουν για να υπολογίσουμε το εύρος των δεδομένων στο container, το οποίο υπολογίζεται ως [maxValue - minValue]. Μόλις το container γεμίσει δημιουργούμε ένα πίνακα ακεραίων στον οποίο μεταφέρουμε τα δεδομένα του buffer. Εφόσον η κάθε θέση του πίνακα είναι 32 bits θα πρέπει τα το μέγεθος του κάθε στοιχείου να διαιρείται ακριβώς με το 32. Για παράδειγμα αν για το εύρος των δεδομένων χρειαζόμαστε 7 bit, θα χρησιμοποιήσουμε 8 bit για την αποθήκευση του κάθε στοιχείου ώστε όλα τα στοιχεία να είναι aligned μέσα σε 32άδες. Έτσι, αν είναι να αποθηκεύσουμε τα δεδομένα 4, 5, 7, 2, 88, 6, 1, 3, χρησιμοποιώντας 8 bit για το καθένα θα τοποθετήσουμε τα {4, 5, 7, 2} στην πρώτη θέση του πίνακα και τα {88, 6, 1, 3} στη δεύτερη θέση του πίνακα. Συνολικά θα έχουμε δεσμεύσει δυο θέσεις των 32 bit η κάθε μια, άρα συνολικά 64 bit, ενώ αν αποθηκεύαμε τις τιμές αυτές απλά σαν ακεραίους θα χρειαζόμασταν 8 * 32 = 256 bit.

Η τεχνική αυτή παρουσιάζεται παρακάτω:

index	data
0	4
1	5
2	7
…	…
999	2
…	…
…	…
…	…
…	…
20000	88
…	…
20997	6
20998	1
20999	3



buffer

index	data
0	4
1	5
2	7
3	2
4	88
5	6
6	1
7	3

bit packing

cell 0				cell 1			
4	5	7	2	88	6	1	3
bit[0-7]	bit[8-15]	bit[16-23]	bit[24-31]	bit[0-7]	bit[8-15]	bit[16-23]	bit[24-31]

3.1.5. Delta encoding

Πολλές φορές είναι αποδοτικό αντί να αποθηκεύουμε τα στοιχεία όπως είναι, να αποθηκεύουμε τη διαφορά του κάθε στοιχείου από το προηγούμενο. Για παράδειγμα το dataset [102, 108, 111, 120, 128, 129] μπορούμε να το αποθηκεύσουμε ως [0, 6, 3, 9, 8, 1], χρησιμοποιώντας 4 bit για την αποθήκευση κάθε στοιχείου αντί για 8 αν χρησιμοποιούσαμε bit packing. Η τεχνική αυτή έχει το πλεονέκτημα ότι τα οφέλη της δεν περιορίζονται σε μικρούς αριθμούς, όπως το bit packing αλλά μπορεί να είναι αποδοτική σε αριθμούς οποιοδήποτε μεγέθους. Για παράδειγμα το [208474673, 208474673, 208474674] θα αποθηκευτεί ως [0, 0, 1] οδηγώντας σε δραματική μείωση του μεγέθους. Η τεχνική αυτή αποδίδει καλύτερα όταν τα γειτονικά στοιχεία έχουν κοντινές τιμές μεταξύ τους, για παράδειγμα όταν το dataset είναι ταξινομημένο.

Η τεχνική αυτή παρουσιάζεται παρακάτω:

initial data:	102

index	data
0	102
1	108
2	111
3	120
4	128
5	129

index	delta
0	0
1	6
2	3
3	9
4	8
5	1

Καθώς αυτή η τεχνική είναι ιδανική για τη μείωση του μεγέθους σε bit που χρειάζονται για την αναπαράσταση των δεδομένων, οι διαφορές (deltas) αποθηκεύονται με την τεχνική bit packing.

3.1.6. Dictionary encoding

Πολλές φορές τα dataset αποτελούνται από αλφαριθμητικά δεδομένα, όπως ονόματα, διευθύνσεις ή ακόμα και περιγραφές ενός αντικειμένου. Στην περίπτωση αυτή αντί να αποθηκεύουμε τα αλφαριθμητικά αυτά, των οποίων το μέγεθος της αναπαράστασης εξαρτάται από το μήκος τους, μπορούμε να αντιστοιχίσουμε το κάθε διαφορετικό αλφαριθμητικό με έναν αριθμό και να αποθηκεύουμε αυτόν στη θέση του. Ο αριθμός αυτός είναι ένας ακέραιος σταθερού μήκους 32 bit. Η μέθοδος αυτή, η οποία λέγεται dictionary encoding λειτουργεί ως εξής: Για κάθε νέο στοιχείο που διαβάζουμε κοιτάμε αν το έχουμε τοποθετήσει στο dictionary. Αν το έχουμε τοποθετήσει ήδη, παίρνουμε τον αριθμό που αντιστοιχεί σε αυτό και αποθηκεύουμε αυτόν αντί για το αλφαριθμητικό. Αν δεν το έχουμε στο λεξικό, το τοποθετούμε αναθέτοντας του τον αμέσως επόμενο αριθμό. Για παράδειγμα αν είναι το πρώτο στοιχείο που μπαίνει στο λεξικό παίρνει τον αριθμό 0, το δεύτερο τον αριθμό 1 κ.ο.κ. Στη συνέχεια αποθηκεύουμε τον αριθμό αυτό στη θέση του αλφαριθμητικού.

Παρακάτω παρουσιάζεται η τεχνική του dictionary encoding:

index	data
0	Γιώργος
1	Γιώργος
2	Κώστας
3	Γιώργος
4	Αντώνης
5	Μαρία
6	Αντώνης

Dictionary

key	value
Γιώργος	0
Κώστας	1
Αντώνης	2
Μαρία	3

Index	data
0	0
1	0
2	1
3	0
4	2
5	3
6	2

4. Συστήματα ανάλυσης δεδομένων

Τα συστήματα που χρησιμοποιούνται προκειμένου να επεξεργαστούμε μεγάλο όγκο δεδομένων είναι κατά βάση κατανεμημένα συστήματα, όπως το Hadoop και το Spark. Για τη βελτιστοποίηση της διαδικασίας ανάλυσης δεδομένων, έχουν δημιουργηθεί συστήματα που βασίζονται στην υπάρχουσα υπολογιστική υποδομή (συνήθως Hadoop ή Spark) και πάνω σε αυτή παρέχουν αποτελεσματικότερο τρόπο αναπαράστασης δεδομένων, όπως το Apache Parquet [23] ή φυσικότερο τρόπο δημιουργίας ερωτημάτων, όπως το Apache Impala [24]. Στην εργασία αυτή θα επικεντρωθούμε στο κομμάτι της αποδοτικότερης αναπαράστασης των δεδομένων. Για το σκοπό αυτό θα εξηγήσουμε τον τρόπο οργάνωσης των δεδομένων στη μνήμη, καθώς και τον τρόπο λειτουργίας του Parquet. Το parquet χρησιμοποιείται από ένα μεγάλο πλήθος εταιριών στον τομέα της ανάλυσης δεδομένων, όπως το Twitter και την Cloudera. Επιπλέον, ο τρόπος που αποθηκεύει και συμπιέζει δεδομένα το καθιστούν μια ενδιαφέρουσα αναφορά, καθώς και ένα μέτρο σύγκρισης για το σύστημα που σχεδιάσαμε για αποδοτικότερη αναπαράσταση και επεξεργασία δεδομένων, το hybrid columnar.

4.1. Columnar λογική

Στα περισσότερα συστήματα βάσεων δεδομένων τα δεδομένα αποθηκεύονται σε γραμμές. Για παράδειγμα σε μια βάση δεδομένων για πελάτες μιας επιχείρησης, μια εγγραφή για ένα νέο πελάτη μπορεί να είναι η:

5	Γιώργος Κ.	31	Ιλισίων 20

Η εγγραφή αυτή μπαίνει στον πίνακα με τους υπάρχοντες πελάτες:

ID	Name	Age	Address
0	Μάριος Ι.	29	Περισσού 18
1	Νίκος Ρ.	32	Καισαρείας 5
2	Γιώργος Ν.	35	Ερμιόνης 12
3	Γιάγκος Μ.	30	Καλλιρόης 8
4	Ορφέας Α.	22	Ηρακλείτου 10
5	Γιώργος Κ.	31	Ιλισίων 20

Ο πίνακας στη μνήμη θα έχει την παρακάτω μορφή:

memory layout
0
Μάριος Ι.
29
Περισσού 18
…
5
Γιώργος Κ.
31
Ιλισίων 20

Όπως βλέπουμε, τα δεδομένα αποθηκεύονται κατά γραμμή. Για το λόγο αυτό τα συστήματα αυτά λέγονται Row Stores. Τα συστήματα αυτά είναι πολύ αποδοτικά όταν ψάχνουμε μια εγγραφή για να τη διαβάσουμε ή να την επεξεργαστούμε, καθώς τα δεδομένα της εγγραφής είναι αποθηκευμένα σειριακά. Έτσι κάνουμε χρήση της χωρικής τοπικότητας της μνήμης cache, καθώς τα δεδομένα που χρειαζόμαστε είναι αποθηκευμένα κοντά μεταξύ τους και μειώνουμε τις προσβάσεις στην κύρια μνήμη. Αυτός είναι και ο λόγος που όλα τα συστήματα βάσεων δεδομένων που χρησιμοποιούνται στην παραγωγή για συναλλαγές (Online Transactional Processing ή OLTP) και περιλαμβάνουν συχνές εισαγωγές, προσπέλαση, ενημέρωση και διαγραφή εγγραφών (Create, Read, Update, Delete ή CRUD) είναι Row Stores.

Ας δούμε όμως και μια άλλη κατηγορία ερωτημάτων. Έστω ότι μια εταιρία θέλει να μάθει το μέσο όρο ηλικίας των πελατών της. Για να εκτελέσει το ερώτημα αυτό σε ένα σύστημα Row Store φέρνει την κάθε εγγραφή στη μνήμη, διαβάζει το πεδίο που αντιστοιχεί στην ηλικία του πελάτη και αθροίζει την ηλικία με όλες τις υπόλοιπες. Για να διαβάσει την ηλικία της κάθε εγγραφής, ένα σύστημα Row Store φέρνει όλη την εγγραφή στη μνήμη ανεξάρτητα αν εμείς χρειαζόμαστε μόνο ένα πεδίο της. Αυτό δημιουργεί μεγάλη πίεση στο υποσύστημα μνήμης, καθώς φέρνει πολύ περισσότερα δεδομένα από αυτά που χρειάζεται το ερώτημα στη βάση. Επίσης τα δεδομένα αυτά πιάνουν γραμμές της μνήμης cache, γεμίζοντάς τη με δεδομένα που δεν χρειαζόμαστε, ρίχνοντας έτσι τις επιδόσεις της.

Ας δούμε μια άλλη προσέγγιση στην αποθήκευση μιας βάσης στη μνήμη. Αντί να αποθηκεύουμε τα στοιχεία ανά εγγραφή, να τα αποθηκεύουμε ανά στήλη. Έτσι, τα δεδομένα του πεδίου **ID** θα αποθηκεύονται στη σειρά, στη συνέχεια τα δεδομένα του πεδίου **Name**, μετά τα δεδομένα του πεδίου **Age** και τέλος, τα δεδομένα του πεδίου **Address**.

Ο τρόπος αυτός αποθήκευσης φαίνεται παρακάτω:

memory layout
0
…
5
Μάριος Ι.
…
Γιώργος Κ.
29
…
31
Περισσού 18
…
Ιλισίων 20

Στην περίπτωση αυτή, για να βρούμε το μέσο όρο των ηλικιών φέρνουμε από τη μνήμη μόνο τα πεδία που αφορούν τις ηλικίες, εξοικονομώντας πολύτιμο bandwidth από τη μνήμη. Επιπλέον, επειδή τα δεδομένα της κάθε στήλης αποθηκεύονται σειριακά στη μνήμη κάνουμε πολύ καλύτερη χρήση της χωρικής τοπικότητας της κρυφής μνήμης, καθώς όταν φέρνουμε μια τιμή από τη μνήμη το σύστημα φέρνει ολόκληρο block με κοντινές τιμές, τις οποίες στη συνέχεια θα διαβάσουμε από την κρυφή μνήμη. Τα συστήματα αυτά που

αποθηκεύουν δεδομένα ανά στήλη ονομάζονται Column Stores [25]. Τα περισσότερα εμπορικά συστήματα που προορίζονται για ανάλυση δεδομένων (data analytics) ανήκουν στην οικογένεια των Column Stores. Ο τρόπος αυτός αποθήκευσης, εκτός από το ότι κάνει αποδοτικότερη χρήση του bandwidth της κύριας μνήμης και αξιοποιεί καλύτερα την χωρική τοπικότητα της κρυφής μνήμης, ευνοεί και τη δυνατότητα συμπίεσης των δεδομένων [26]. Αυτό συμβαίνει επειδή σε συνεχόμενες θέσεις μνήμης είναι αποθηκευμένα δεδομένα ίδιου τύπου πχ ακέραιοι που αναπαριστούν τις ηλικίες. Επιπλέον, ενώ μια εγγραφή έχει έναν (συνήθως) μικρό αριθμό πεδίων ο οποίος παραμένει σταθερός καθώς αυξάνονται τα δεδομένα, μια στήλη έχει έναν μεγάλο αριθμό δεδομένων ίδιου τύπου, τα οποία αυξάνονται όσο εισάγονται νέα δεδομένα. Το πλήθος και η ομοιογένεια των δεδομένων είναι ιδιαίτερα ευεργετικοί παράγοντες για τη συμπίεση δεδομένων καθώς μέσα στα δεδομένα αυτά είναι πολύ πιθανό να υπάρχουν διπλότυπα (duplicates) πχ πολλά ίδια ονόματα (συνήθως χρησιμοποιούμε Dictionary ή Bitmap τεχνικές) ή γειτονικά δεδομένα να έχουν ίδια (όπου ευνοούνται Run Length Encoding τεχνικές) ή κοντινή (Delta τεχνικές) τιμή. Για τους λόγους αυτούς, το σύστημα που σχεδιάσαμε χρησιμοποιεί Columnar λογική για την αποθήκευση των δεδομένων.

4.2. Apache Parquet

Η ανάγκη ανάλυσης μεγάλου όγκου πληροφοριών στο Twitter, κυρίως logs, οδήγησε στη δημιουργία του Parquet. Το parquet είναι ένα file format, δηλαδή ένας τρόπος οργάνωσης και αποθήκευσης δεδομένων, το οποίο μπορεί να χρησιμοποιηθεί σε υπάρχοντα κατανεμημένα συστήματα επεξεργασίας δεδομένων, όπως το Spark. Δεν είναι δηλαδή ένα σύστημα ανάλυσης δεδομένων, αλλά αντίθετα μπορεί να χρησιμοποιηθεί στη θέση απλών αρχείων κειμένου (.txt, .csv) για την ανάγνωση και αποθήκευση δεδομένων που χρησιμοποιούν αυτά τα συστήματα. Χρησιμοποιώντας τεχνικές από το χώρο της ανάλυσης δεδομένων, το parquet αποθηκεύει τα δεδομένα κατά στήλες (column stores). Επιπλέον χρησιμοποιεί συμπίεση στα δεδομένα των στηλών, ώστε να επιτευχθεί μικρότερο μέγεθος αρχείων.

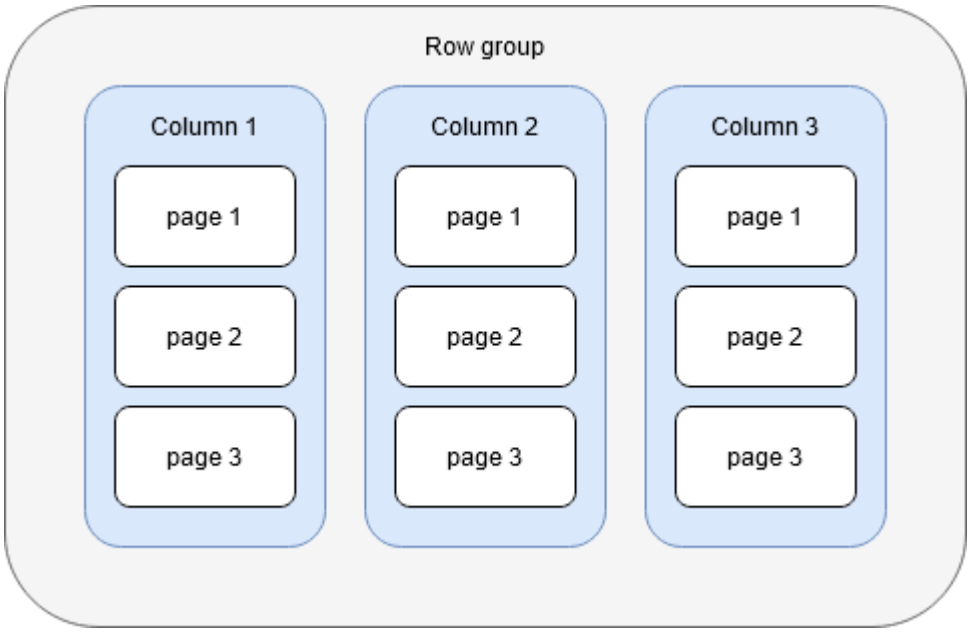
Αρχιτεκτονική

Κάθε αρχείο στο parquet χωρίζεται σε row groups, δηλαδή συνεχόμενα τμήματα ενός αρχείου. Για παράδειγμα ένα αρχείο με 1000 εγγραφές μπορεί να αποτελείται από 5 row groups, με το πρώτο να περιλαμβάνει τις εγγραφές από 0 μέχρι 199, το δεύτερο από 200 έως 399 και αντίστοιχα τα υπόλοιπα.

Το κάθε row group αποτελείται από ένα σύνολο από columns, ανάλογα με τον αριθμό των πεδίων του αρχείου. Για παράδειγμα αν έχουμε 3 πεδία, το κάθε row group θα έχει 3 columns. Η κάθε column του parquet μπορεί να διαβάζεται παράλληλα με τις υπόλοιπες, επιταχύνοντας έτσι την όλη διαδικασία.

Τα columns αυτά αποτελούνται από ένα σύνολο από δομές, οι οποίες ονομάζονται pages. Το κάθε page κρατά ένα υποσύνολο των δεδομένων μιας στήλης και μπορεί να χρησιμοποιεί διαφορετική μέθοδο συμπίεσης από τα υπόλοιπα pages της ίδιας στήλης.

Η αρχιτεκτονική του parquet φαίνεται στο παρακάτω σχήμα:



Το parquet υποστηρίζει τους παρακάτω τύπους δεδομένων:

- BOOLEAN
- INT32
- INT64
- INT96
- FLOAT
- DOUBLE
- BYTE_ARRAY
- FIXED_LENGTH_BYTE_ARRAY

Ενώ μπορεί να κωδικοποιήσει τα δεδομένα με τις ακόλουθες μεθόδους:

- PLAIN
- GROUP_VAR_INT
- PLAIN_DICTIONARY
- RLE
- BIT_PACKED

Επίσης τα κωδικοποιημένα δεδομένα μπορεί να τα συμπιέσει με μια από τις μεθόδους:

- UNCOMPRESSED
- SNAPPY
- GZIP
- LZO

5. Hybrid Columnar

Σχεδιάζοντας ένα in memory σύστημα ανάλυσης δεδομένων (data analytics) μεγάλης κλίμακας χρησιμοποιήσαμε την columnar λογική. Ο λόγος ήταν τόσο οι καλύτερες επιδόσεις σε ερωτήματα που χρησιμοποιούν συναθροίσεις (aggregations) σε υποσύνολο των πεδίων του dataset, όσο και η μεγαλύτερη δυνατότητα συμπίεσης που δίνει ο τρόπος αυτός αποθήκευσης. Η συμπίεση δεδομένων παίζει καθοριστικό ρόλο στο να μπορούμε να εκτελούμε τα ερωτήματα κατευθείαν από την κύρια μνήμη, χωρίς να κάνουμε χρήση του δίσκου.

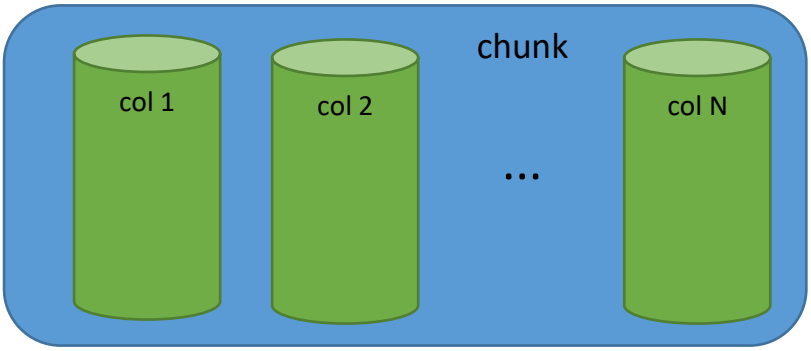
Το κάθε πεδίο του dataset αποθηκεύεται σε μια κολόνα (column). Η κάθε κολόνα ανάλογα με τον τύπο και την κατανομή των δεδομένων του dataset που αποθηκεύει χρησιμοποιεί την καταλληλότερη μέθοδο συμπίεσης, ανεξάρτητα από τις υπόλοιπες κολόνες. Η μέθοδος συμπίεσης μπορεί να είναι μια από τις:

- PLAIN
- RLE
- Bitmap
- Roaring
- Bit packing
- Delta
- Dictionary

Επιπλέον η κάθε κολόνα μπορεί να χρησιμοποιεί διαφορετική μέθοδο συμπίεσης σε κάθε της τμήμα (partition), ανάλογα με την κατανομή των δεδομένων στο κάθε partition.

Οι κολόνες αυτές μπαίνουν σε μια δομή, την οποία ονομάσαμε chunk, η οποία δεν είναι παρά μια αφαίρεση ενός table, αποτελούμενου από τις κολόνες αυτές.

Το σχήμα αυτό παρουσιάζεται παρακάτω:



Η κάθε κολόνα είναι ανεξάρτητη από τις υπόλοιπες και μπορούμε να εκτελέσουμε επάνω της απευθείας ερωτήματα (queries), χωρίς να έχει προηγηθεί αποσυμπίεση. Στο επόμενο κεφάλαιο ακολουθεί μια αναλυτικότερη περιγραφή της αρχιτεκτονικής και του τρόπου λειτουργίας του hybrid columnar.

5.1. Εισαγωγή δεδομένων

Το σύστημα που σχεδιάσαμε μπορεί να λειτουργήσει είτε ανεξάρτητα, σαν standalone εφαρμογή διαβάζοντας δεδομένα από τον τοπικό δίσκο, είτε πάνω από το Spark χρησιμοποιώντας την κατανεμημένη δυνατότητα επεξεργασίας που αυτό παρέχει.

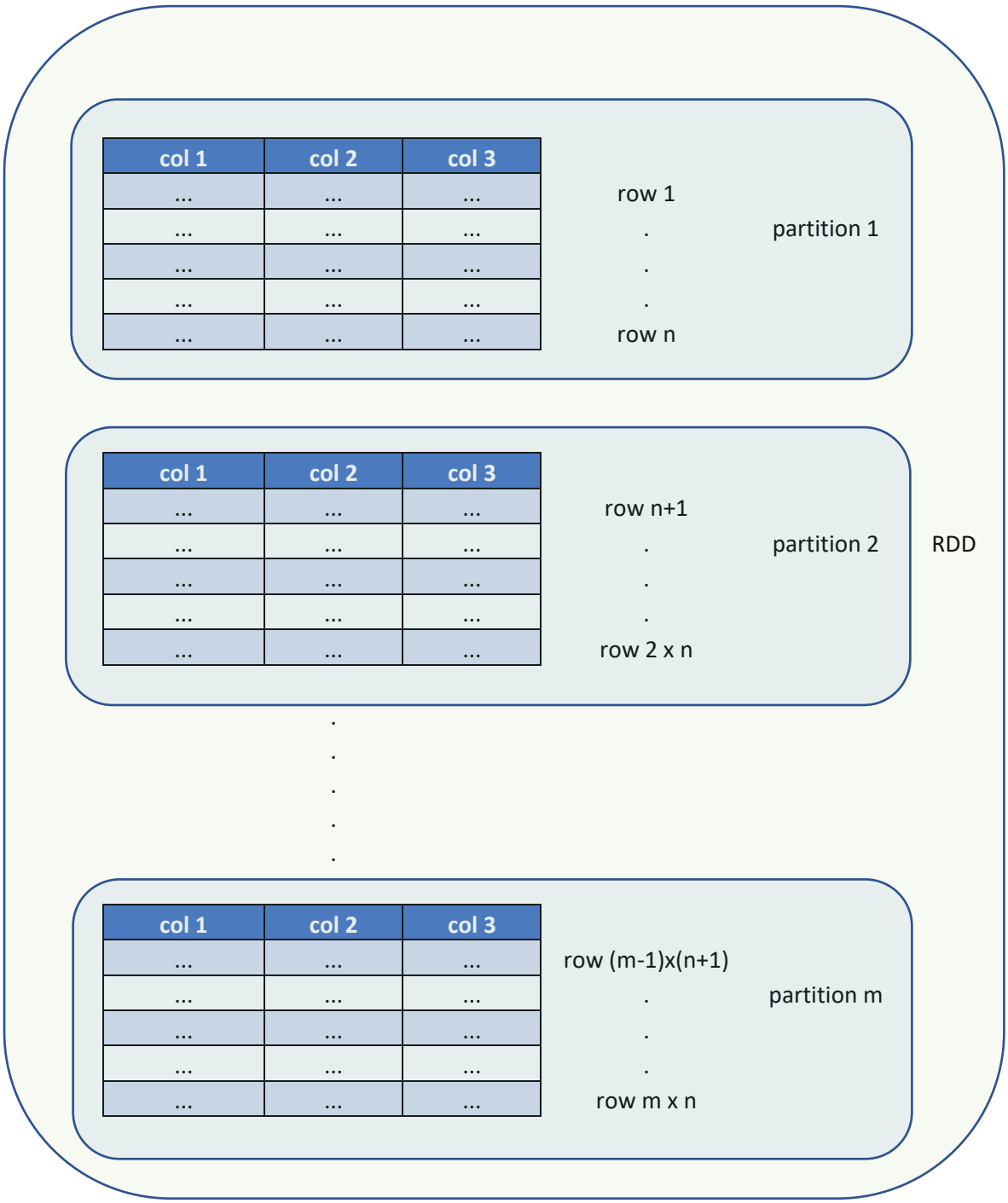
Σε πρώτο βήμα, εισάγουμε το dataset ή κάποιο υποσύνολο του dataset. Για παράδειγμα αν θέλουμε να εισάγουμε τα πεδία name και age του παρακάτω dataset:

ID	Name	Age
1	Γιώργος	20
2	Νίκος	21
3	Αντώνης	21
4	Ορφέας	19
5	Φωτεινή	20
6	Γιάγκος	21
7	Γιώργος	19
8	Νίκος	20

Name	Age
Γιώργος	20
Νίκος	21
Αντώνης	21
Ορφέας	19
Φωτεινή	20
Γιάγκος	21
Γιώργος	19
Νίκος	20

Στο σημείο αυτό έχουμε ένα RDD από Rows. Το RDD (Resilient Distributed Dataset) χωρίζεται σε partitions τα οποία βρίσκονται κατανεμημένα στους κόμβους του cluster.

Το σχήμα αυτό παρουσιάζεται παρακάτω:

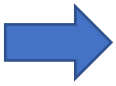


Το κάθε partition έχει ένα κομμάτι του dataset.

Στη συνέχεια τα δεδομένα ταξινομούνται λεξικογραφικά προκειμένου να επιτύχουμε καλύτερη συμπίεση. Όπως στη λεξικογραφική ταξινόμηση οι συμβολοσειρές (strings) ταξινομούνται με βάση το χαρακτήρα στην πρώτη θέση, στη συνέχεια στην επόμενη θέση, μέχρι το τέλος της συμβολοσειράς, έτσι και το dataset ταξινομείται με βάση το πρώτο πεδίο της κάθε εγγραφής και στη συνέχεια το δεύτερο μέχρι και το τελευταίο πεδίο. Χρησιμοποιήσαμε μια παραλλαγή της λεξικογραφικής ταξινόμησης, στην οποία οι εγγραφές ταξινομούνται όχι με βάση τη σειρά των πεδίων, αλλά με βάση το cardinality των πεδίων.

Για παράδειγμα το παρακάτω dataset θα ταξινομηθεί ως:

ID	Name	Age
1	Γιώργος	20
2	Νίκος	21
3	Αντώνης	21
4	Ορφέας	19
5	Φωτεινή	20
6	Γιάγκος	21
7	Γιώργος	19
8	Νίκος	20



ID	Name	Age
7	Γιώργος	19
4	Ορφέας	19
1	Γιώργος	20
8	Νίκος	20
5	Φωτεινή	20
3	Αντώνης	21
6	Γιάγκος	21
2	Νίκος	21

Η ταξινόμηση έγινε πρώτα με βάση το τρίτο πεδίο (cardinality 3), μετά το δεύτερο (cardinality 6) και τέλος το πρώτο (cardinality 8). Αυτό γίνεται για να δημιουργούνται μεγαλύτερα run lengths και άρα να αυξάνεται η δυνατότητα συμπίεσης του dataset.

Η ταξινόμηση γίνεται εσωτερικά σε κάθε partition και όχι σε ολόκληρο το dataset. Η επιλογή αυτή έγινε ώστε να μην έχουμε μετακίνηση (shuffle) δεδομένων ανάμεσα στους κόμβους του cluster, η οποία θα κόστιζε πολύ σε χρόνο. Το μειονέκτημα σε σχέση με το global sort του dataset είναι πως αν κάποια δεδομένα υπάρχουν σε περισσότερα του ενός partition θα αποθηκευτούν σε κάθε ένα από αυτά τα partitions, ενώ στην περίπτωση του global sort θα κατέληγαν στο ίδιο partition και άρα θα πετυχαίναμε καλύτερη συμπίεση. Η απόφαση αυτή ήταν ένα κλασικό time vs space tradeoff, όπου θυσιάσαμε κάποιο χώρο στη μνήμη προκειμένου να πετύχουμε καλύτερους χρόνους κατά την εισαγωγή των δεδομένων.

Στο σημείο αυτό έχουμε ένα ταξινομημένο dataset αποτελούμενο από rows. Στο επόμενο στάδιο θα χωρίσουμε το dataset σε στήλες (columns) και για κάθε μια από αυτές το σύστημα θα επιλέξει την κατάλληλη μέθοδο συμπίεσης.

5.2. Επιλογή της κατάλληλης μεθόδου συμπίεσης

Για τον αλγόριθμο επιλογής της κατάλληλης μεθόδου συμπίεσης εξετάσαμε ένα σύνολο από επιλογές.

5.2.1. Επιλογή με βάση τα χαρακτηριστικά του dataset

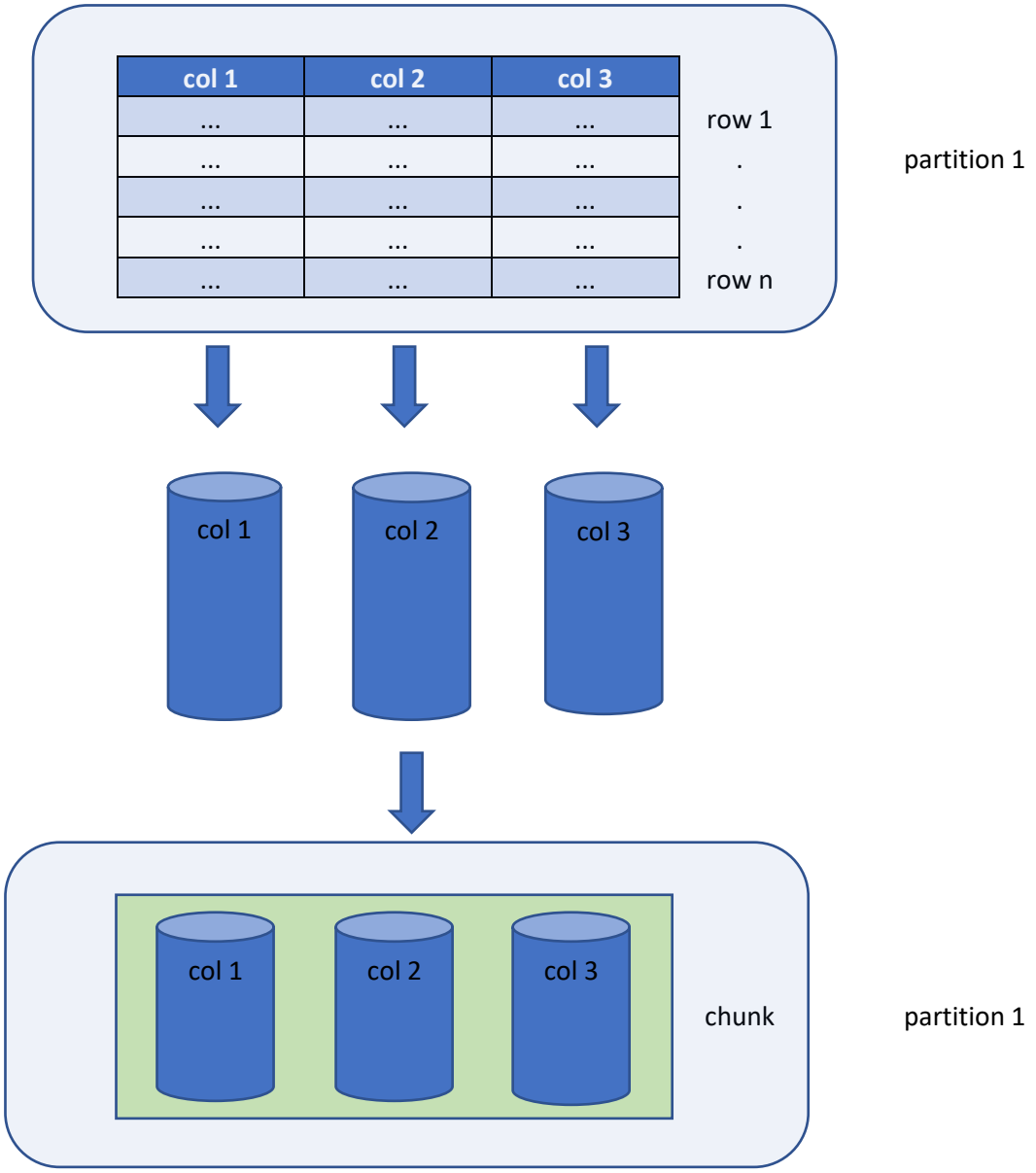
Αρχικά σχεδιάσαμε ένα σύστημα βασισμένο σε κανόνες, όπου ανάλογα με τα χαρακτηριστικά των δεδομένων της κάθε στήλης, όπως για παράδειγμα το πλήθος των runs, το average run length και το cardinality θα επιλέγει την κατάλληλη μέθοδο συμπίεσης. Για τη δημιουργία των κανόνων βασιστήκαμε στη σχετική βιβλιογραφία [26] αλλά και σε μετρήσεις που κάναμε με διάφορες τεχνικές συμπίεσης.

5.2.2. Δοκιμή όλων των μεθόδων συμπίεσης

Στη σημείο αυτό σκεφτήκαμε πως από τη στιγμή που φέρνουμε όλο το dataset από το δίσκο στην κύρια μνήμη του συστήματος προκειμένου να κάνουμε το prediction, θα μπορούσαμε πολύ απλά να δοκιμάσουμε να συμπιέσουμε την κάθε στήλη με όλες τις πιθανές μεθόδους και να διαλέξουμε την καλύτερη. Από τη στιγμή που τα δεδομένα έχουν ήδη μεταφερθεί στην κύρια μνήμη το overhead είναι αμελητέο, οπότε αυτός είναι και ο αλγόριθμος που τελικά επιλέξαμε να χρησιμοποιήσουμε στο σύστημά μας.

Στο σημείο αυτό το RDD από sorted Rows που είχαμε προηγουμένως έχει μετατραπεί σε ένα RDD από chunks, όπου το κάθε chunk περιέχει τις κολόνες του dataset.

Το σχήμα αυτό παρουσιάζεται παρακάτω:



Στο παραπάνω σχήμα φαίνεται πως ένα partition από ταξινομημένες γραμμές μετατρέπεται σε ένα partition που περιέχει ένα chunk. Το chunk περιέχει τις στήλες του dataset στο συγκεκριμένο partition, όπου η κάθε μια χρησιμοποιεί τον καταλληλότερο αλγόριθμο συμπίεσης για τη συγκεκριμένη στήλη του dataset.

Έχοντας στο dataset συμπιεσμένο στη μνήμη, μπορούμε να εκτελούμε ερωτήματα πάνω σε αυτό, χωρίς να έχει προηγηθεί αποσυμπίεση των δεδομένων. Ο τρόπος που πραγματοποιούμε τα ερωτήματα ακολουθεί το προγραμματιστικό μοντέλο MapReduce.

Για παράδειγμα για να επιλέξουμε όλες τις εγγραφές των πελατών που η ηλικία τους είναι μικρότερη των 29 ετών μπορούμε να γράψουμε:

```
compressedDataset.map(chunk => chunk.getColumn("age").selectLessThan(29
)).collect
```

Ενώ για να βρούμε το πλήθος των πελατών που η ηλικία τους είναι ανάμεσα σε 18 και 32 έτη, γράφουμε:

```
compressedDataset.map(chunk => chunk.getColumn("age").selectBetween(18,
 32).cardinality).reduce((a,b) => a + b)
```

Επίσης στο hybrid columnar υποστηρίζονται και group by queries τα οποία γράφονται με την παρακάτω μορφή:

```
compressedDataset.flatMap(chunk => chunk).map(row => (row.get("age").to
String.toInt, row.getRunLength)).reduceByKey((a,b) => a + b).collect
```

Στο παραπάνω παράδειγμα υπολογίσαμε το πλήθος των πελατών ανά ηλικία.

5.3. Εκτέλεση ερωτημάτων

Το hybrid columnar υποστηρίζει select queries, aggregates και group by queries. Στην ενότητα αυτή περιγράφουμε τον τρόπο εκτέλεσης των ερωτημάτων αυτών, ανάλογα με την τεχνική συμπίεσης που χρησιμοποιείται στην κάθε στήλη.

5.3.1. Select queries

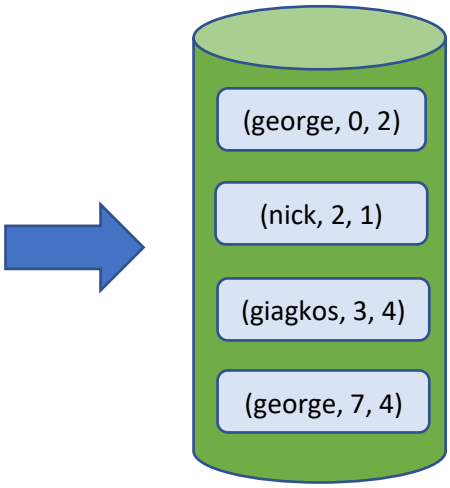
Κατά την εκτέλεση των ερωτημάτων αυτών, το hybrid columnar διαβάζει σειριακά όλα τα chunks του dataset. Για κάθε chunk εξάγει την κολόνα πάνω στην οποία θα γίνει η επιλογή και εκτελεί το predicate του query στην κολόνα αυτή. Το αποτέλεσμα αυτής της διαδικασίας είναι ένα bitmap το οποίο έχει την τιμή 1 στη θέση κάθε γραμμής για την οποία ισχύει το predicate και 0 αλλιώς. Αν το query περιλαμβάνει περισσότερες από μια κολόνες, η διαδικασία αυτή επαναλαμβάνεται για κάθε μια από αυτές. Για παράδειγμα σε ένα query της μορφής ***select [...] from clients where clients.age > 30 and clients.city = Athens*** η διαδικασία αυτή γίνεται πάνω στις στήλες age και city, όπου παράγονται και τα αντίστοιχα bitmaps για την κάθε στήλη. Το τελικό bitmap υπολογίζεται εφαρμόζοντας την πράξη AND ανάμεσα στο Bitmap που αντιστοιχεί στην στήλη age και στην στήλη city. Στη συνέχεια αυτό το bitmap μπορούμε να το χρησιμοποιήσουμε για την εκτέλεση aggregate queries. Ο τρόπος εκτέλεσης του query ποικίλει, ανάλογα με τη μέθοδο συμπίεσης της κολόνας στην οποία εκτελείται.

RLE

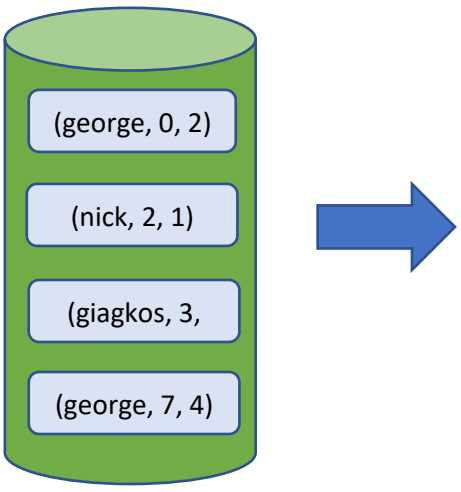
Στην τεχνική συμπίεσης RLE τα δεδομένα κωδικοποιούνται σε τριάδες της μορφής (value, index, length), όπου value η τιμή που αποθηκεύουμε, index η θέση της στο αρχικό σύνολο δεδομένων και length ο αριθμός των συνεχόμενων επαναλήψεων της συγκεκριμένης τιμής.

Το σχήμα αυτό φαίνεται παρακάτω:

index	Names
0	george
1	george
2	nick
3	giagkos
4	giagkos
5	giagkos
6	giagkos
7	george
8	george
9	george
10	george



Το σύστημα διαβάζει κάθε τριάδα της στήλης και συγκρίνει την τιμή του πεδίου value της, με την τιμή της συνθήκης του ερωτήματος που τρέχουμε πάνω στην κολόνα. Για παράδειγμα αν θέλουμε να βρούμε όλες τις γραμμές στις οποίες εμφανίζεται το όνομα george, για κάθε τριάδα ελέγχει αν στο πεδίο *value* υπάρχει η τιμή george. Αν υπάρχει, τοποθετεί την τιμή 1 στις θέσεις του bitmap που βρίσκονται από τη θέση index μέχρι τη θέση index + length της συγκεκριμένης τριάδας. Το αποτέλεσμα είναι ένα bitmap το οποίο έχει την τιμή 1 στις γραμμές που υπάρχει το όνομα george και 0 στις υπόλοιπες, όπως φαίνεται παρακάτω:



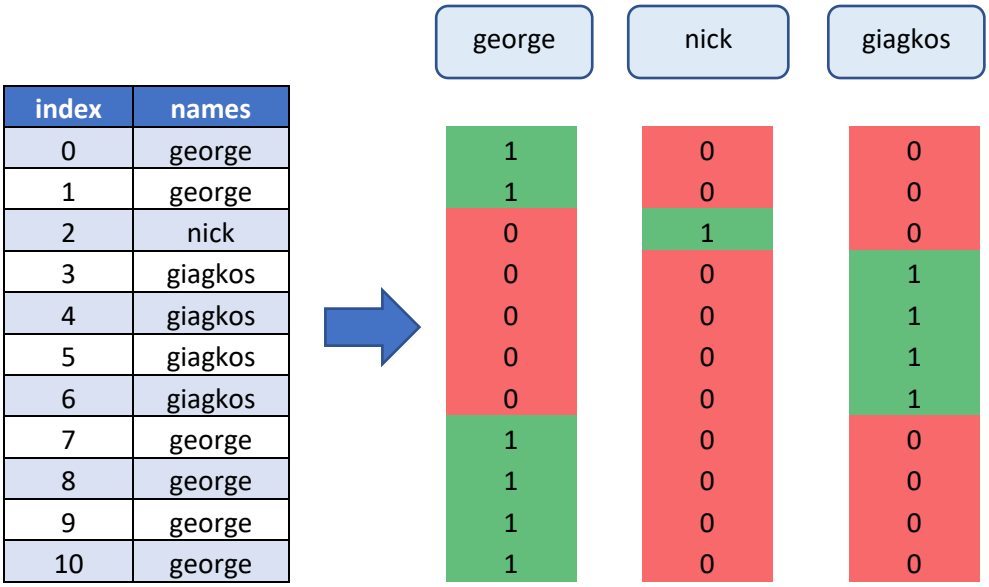
index	Value
0	1
1	1
2	0
3	0
4	0
5	0
6	0
7	1
8	1
9	1
10	1

Το bitmap αυτό μπορεί να συνδυαστεί με bitmaps που έχουν παραχθεί από άλλα select queries, ώστε να παραχθεί το αποτέλεσμα μια σύνθετης συνθήκης ή να χρησιμοποιηθεί στη συνέχεια ως είσοδος ενός aggregate query.

ROARING

Σε αυτή την τεχνική συμπίεσης τα δεδομένα αποθηκεύονται σε ένα HashMap, το οποίο έχει για κλειδιά όλες τις μοναδικές τιμές του συνόλου των δεδομένων. Στο κάθε κλειδί αντιστοιχεί ένα bitmap το οποίο έχει την τιμή 1 στις θέσεις που αντιστοιχούν στις γραμμές του συνόλου δεδομένων που εμφανίζεται η συγκεκριμένη τιμή και 0 στις υπόλοιπες.

Το σχήμα αυτό φαίνεται παρακάτω:



index	names
0	george
1	george
2	nick
3	giagkos
4	giagkos
5	giagkos
6	giagkos
7	george
8	george
9	george
10	george

george	nick	giagkos
1	0	0
1	0	0
0	1	0
0	0	1
0	0	1
0	0	1
0	0	1
1	0	0
1	0	0
1	0	0
1	0	0

Το σύστημα διαβάζει κάθε κλειδί του hashmap και το συγκρίνει με την τιμή της συνθήκης του ερωτήματος που τρέχουμε πάνω στην κολόνα. Αν αυτό ταιριάζει τότε απλά επιστρέφει το bitmap που αντιστοιχεί στο κλειδί αυτό. Για παράδειγμα αν θέλουμε να βρούμε τις γραμμές που έχουν το όνομα george το σύστημα επιστρέφει το bitmap που αντιστοιχεί κλειδί george. Αν ταιριάζουν περισσότερα από ένα κλειδιά, για παράδειγμα αν ψάχνουμε για τις γραμμές που έχουν το όνομα george ή το όνομα giagkos, πραγματοποιείται η πράξη *OR* μεταξύ των bitmaps που αντιστοιχούν σε αυτά τα ονόματα και παράγεται το τελικό bitmap, όπως φαίνεται παρακάτω:

Index	value
0	1
1	1
2	0
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1

Bit Packing

Στην τεχνική Bit Packing τα δεδομένα χωρίζονται σε containers, το καθένα με μέγεθος το πολύ 1000 ακεραίων. Για το λόγο αυτό, τύποι δεδομένων διαφορετικοί των ακεραίων υποστηρίζονται από την τεχνική αυτή συμπίεσης σε συνδυασμό με ένα dictionary. Μέσα σε κάθε container τα δεδομένα αποθηκεύονται σαν μια σειρά από bit, με το κάθε στοιχείο να καταλαμβάνει ίδιο αριθμό bit με τα υπόλοιπα.

Η τεχνική αυτή παρουσιάζεται παρακάτω:

index	names
0	george
1	george
2	nick
3	giagkos
4	giagkos
5	giagkos
6	giagkos
7	george
8	george
9	george
10	george

bit value
00
00
01
10
10
10
10
00
00
00
00

key	value
george	0
nick	1
giagkos	2

Έστω ότι θέλουμε να βρούμε όλες τις γραμμές που εμφανίζεται το όνομα george. Το σύστημα διαβάζει κάθε στοιχείο του container και ελέγχει αν έχει την τιμή που αντιστοιχεί στο κλειδί george. Αν το συγκεκριμένο στοιχείο έχει την τιμή αυτή τότε το σύστημα τοποθετεί την τιμή 1 στην θέση του bitmap που αντιστοιχεί στο στοιχείο αυτό και 0 σε αντίθετη περίπτωση.

bit value
00
00
01
10
10
10
10
00
00
00
00

index	value
0	1
1	1
2	0
3	0
4	0
5	0
6	0
7	1
8	1
9	1
10	1

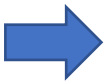
Key	value
George	0
Nick	1
Giagkos	2

Delta

Η τεχνική αυτή είναι παρόμοια με το Bit Packing, καθώς η κολόνα χωρίζεται πάλι σε containers των 1000 ακεραίων, όπου μέσα στο κάθε container το κάθε στοιχείο αποθηκεύεται ως μια σειρά από bit. Η διαφορά είναι ότι στην τεχνική Delta, αντί για το κάθε στοιχείο, αποθηκεύουμε τη διαφορά του από το προηγούμενό του στοιχείο.

Η τεχνική αυτή παρουσιάζεται παρακάτω:

index	names
0	george
1	george
2	nick
3	giagkos
4	giagkos
5	giagkos
6	giagkos
7	george
8	george
9	george
10	george



bit value
10
10
11
11
10
10
10
00
10
10
10

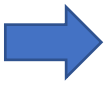
key	value
george	0
nick	1
giagkos	2

start value
0

offset
2

Κατά την αναζήτηση διαβάζονται σειριακά όλα τα container. Στο κάθε container διαβάζεται η αρχική τιμή και στη συνέχεια προστίθεται στο πρώτο στοιχείο. Για προστίθεται η τιμή του με αυτή του προηγούμενού του στοιχείου και έτσι προκύπτει η τιμή που είναι αποθηκευμένη στη συγκεκριμένη θέση. Στην περίπτωση που ψάχνουμε τις γραμμές με το όνομα george, αν η τιμή της θέσης αυτής είναι 0, αποθηκεύουμε την τιμή 1 στο bitmap εξόδου, όπως φαίνεται παρακάτω:

bit value
10
10
11
11
10
10
10
00
10
10
10



index	value
0	1
1	1
2	0
3	0
4	0
5	0
6	0
7	1
8	1
9	1
10	1

start value	offset
0	2

key	value
george	0
nick	1
giagkos	2

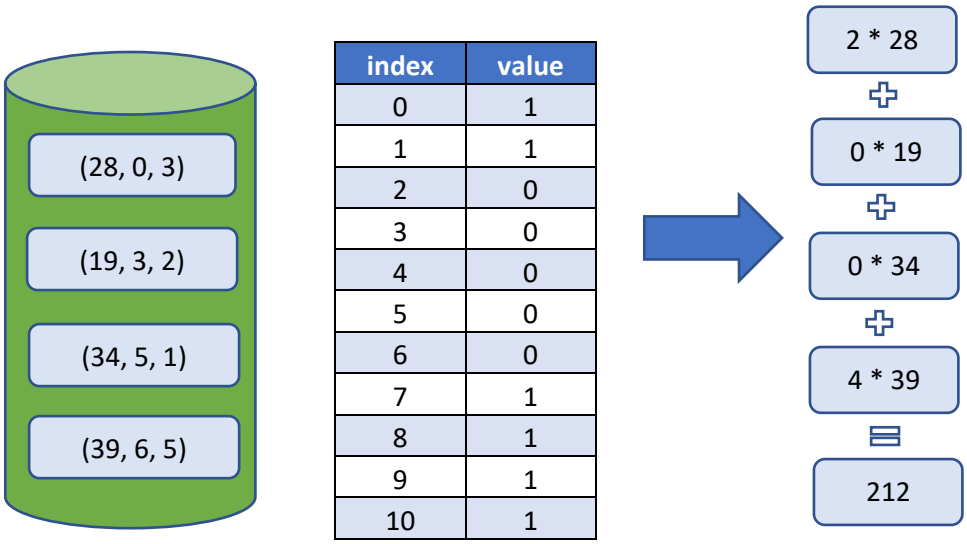
5.3.2. Aggregate queries

Έχοντας το Bitmap από το προηγούμενο βήμα, το hybrid columnar διαβάζει σειριακά τα chunks και εξάγει τις κολόνες πάνω στις οποίες θέλουμε να εκτελέσουμε aggregates. Για κάθε μια από τις κολόνες αυτές εκτελεί το αντίστοιχο aggregate function με το Bitmap ως παράμετρο. Αν έχουμε για παράδειγμα ένα query της μορφής ***select avg(price) from sales where sales.state = CA*** το hybrid columnar εξάγει από το κάθε chunk την στήλη price και εκτελεί επάνω της τη μέθοδο avg, με παράμετρο το Bitmap που έχει 1 στις γραμμές που το πεδίο state έχει την τιμή CA. Αντίστοιχα, ο τρόπος εκτέλεσης διαφέρει ανάλογα με την χρησιμοποιούμενη τεχνική συμπίεσης.

RLE

Σε κολόνες που χρησιμοποιούν αυτή την τεχνική συμπίεσης, το σύστημα διαβάζει την κάθε τριάδα και ελέγχει τα πεδία *index* και *length*. Στη συνέχεια μετράει στο bitmap που παίρνει ως είσοδο, τον αριθμό των 1 που βρίσκονται από τον αριθμό της γραμμής που δείχνει το *index* μέχρι τον αριθμό της γραμμής που δείχνει το *index* + *length*. Με βάση τον αριθμό αυτό και την τιμή του πεδίου *value* της τριάδας που διάβασε κάνει τις ανάλογες πράξεις, ανάλογα τον τύπο του aggregate. Για παράδειγμα, αν θέλουμε να βρούμε το άθροισμα των ηλικιών που αντιστοιχούν στο όνομα george, πολλαπλασιάζει το πεδίο *value* με τον αριθμό αυτό και προσθέτει το αποτέλεσμα στο συνολικό άθροισμα, το οποίο και επιστρέφει.

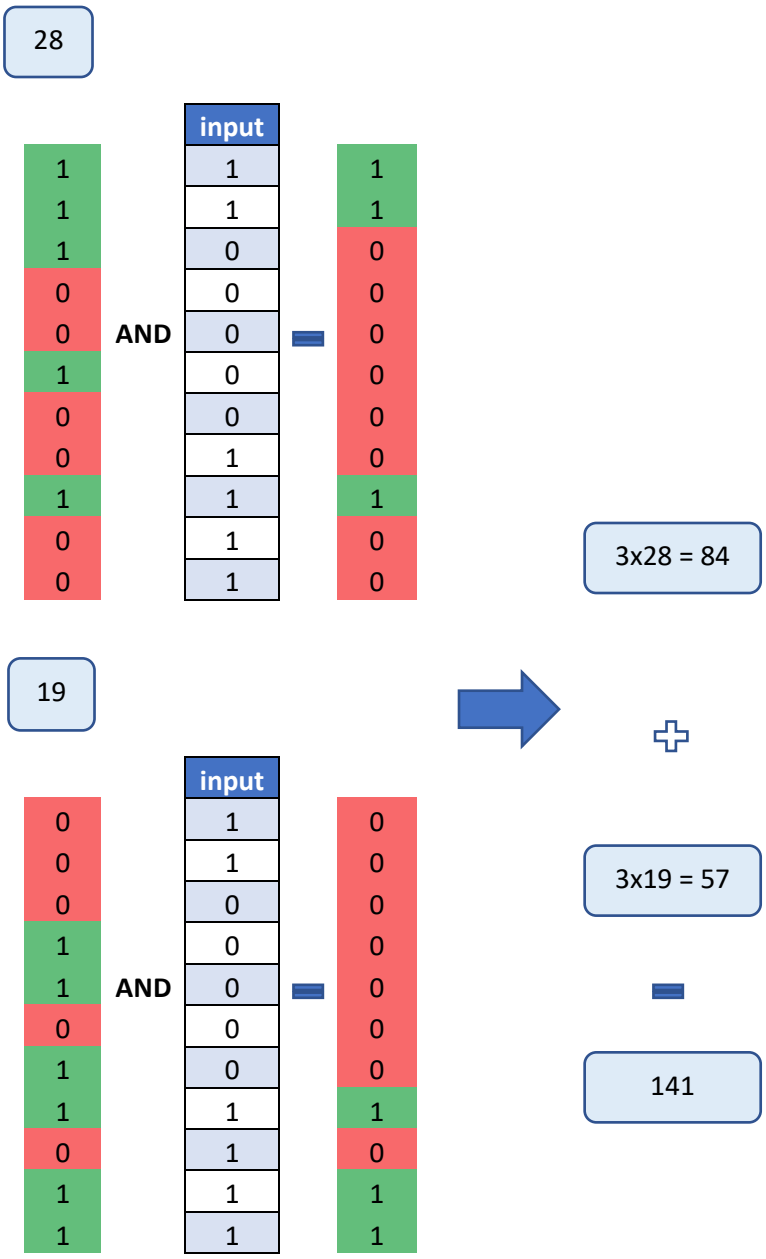
Η τεχνική αυτή φαίνεται παρακάτω:



ROARING

Τα ερωτήματα συνάθροισης σε αυτή την τεχνική συμπίεσης εκτελούνται ως εξής: έχοντας ως είσοδο το bitmap που δείχνει τις γραμμές που χρειαζόμαστε, διαβάζουμε το κάθε κλειδί της κολόνας. Για το κάθε κλειδί παίρνουμε το bitmap που αντιστοιχεί σε αυτό και εκτελούμε την πράξη AND ανάμεσα σε αυτό και το bitmap της εισόδου. Το αποτέλεσμα είναι ένα bitmap που περιέχει την τιμή 1 στις γραμμές που βρίσκεται το κλειδί και ταυτόχρονα χρειάζεται να τις συμπεριλάβουμε. Στο bitmap αυτό μετράμε τον αριθμό των 1 και ανάλογα με τον τύπο της συνάθροισης που θα χρησιμοποιήσουμε κάνουμε τις ανάλογες πράξεις. Για παράδειγμα αν θέλουμε να βρούμε το άθροισμα των ηλικιών που αντιστοιχούν στο όνομα george, πολλαπλασιάζουμε τον αριθμό των 1 με την το κλειδί και στη συνέχεια προσθέτουμε το αποτέλεσμα στο συνολικό άθροισμα.

Η διαδικασία αυτή παρουσιάζεται παρακάτω:



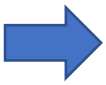
Bit Packing

Για την εκτέλεση των aggregate queries σε κολόνες που χρησιμοποιούν την τεχνική bit packing, το hybrid columnar λειτουργεί ως εξής: διαβάζει σειριακά όλα τα container της κολόνας και ακόλουθα για το κάθε container διαβάζει όλα τα στοιχεία που περιέχει. Για το κάθε στοιχείο ελέγχει αν το index του έχει την τιμή 1 στην αντίστοιχη θέση του bitmap εισόδου. Αν το bitmap έχει την τιμή 1, τότε η τιμή του στοιχείου προστίθεται στο συνολικό άθροισμα, ενώ σε διαφορετική περίπτωση την αγνοεί.

Η διαδικασία αυτή εμφανίζεται παρακάτω:

value	bit value
28	00011100
28	00011100
28	00011100
19	00010011
19	00010011
28	00011100
19	00010011
19	00010011
28	00011100
19	00010011
19	00010011

input
1
1
0
0
0
0
0
1
1
1
1



output
28
28
0
0
0
0
0
19
28
19
19

=

141

Delta

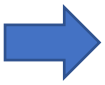
Η εκτέλεση aggregate queries στην τεχνική αυτή είναι παρόμοια με αυτή του bit packing, με τη μόνη διαφοροποίηση να βρίσκεται στο γεγονός ότι για να βρούμε την τιμή του κάθε στοιχείου, υπολογίζουμε τις διαφορές του από τα προηγούμενά του στοιχεία στο container.

Ο τρόπος που εκτελούνται τα aggregate queries στην τεχνική αυτή φαίνεται παρακάτω:

value	bit value
28	00010011
28	00010011
28	00010011
19	00000000
19	00010011
28	00011100
19	00000000
19	00010011
28	00011100
19	00000000
19	00010011

start value	offset
28	19

input
1
1
0
0
0
0
0
1
1
1
1

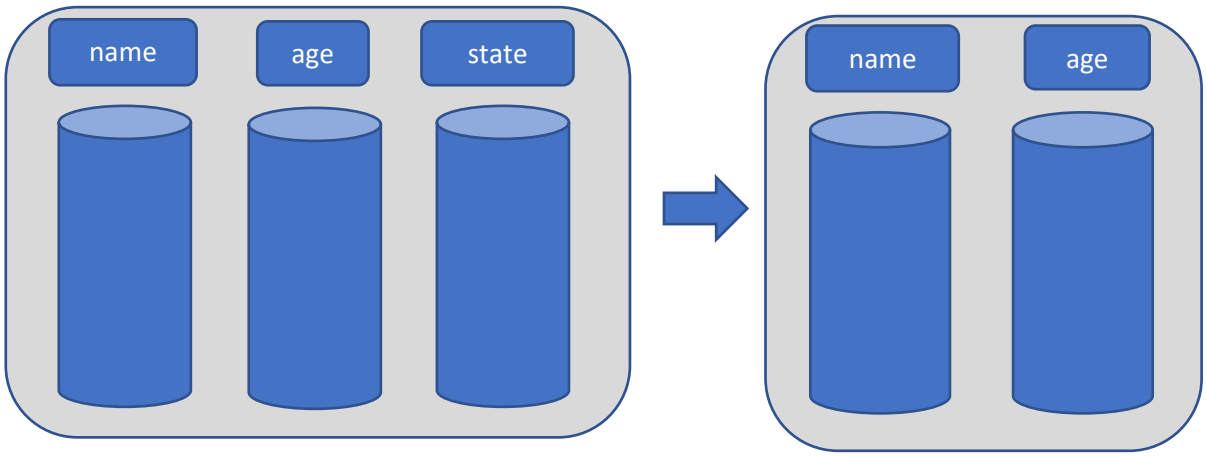


output
28
28
0
0
0
0
0
19
28
19
19

141

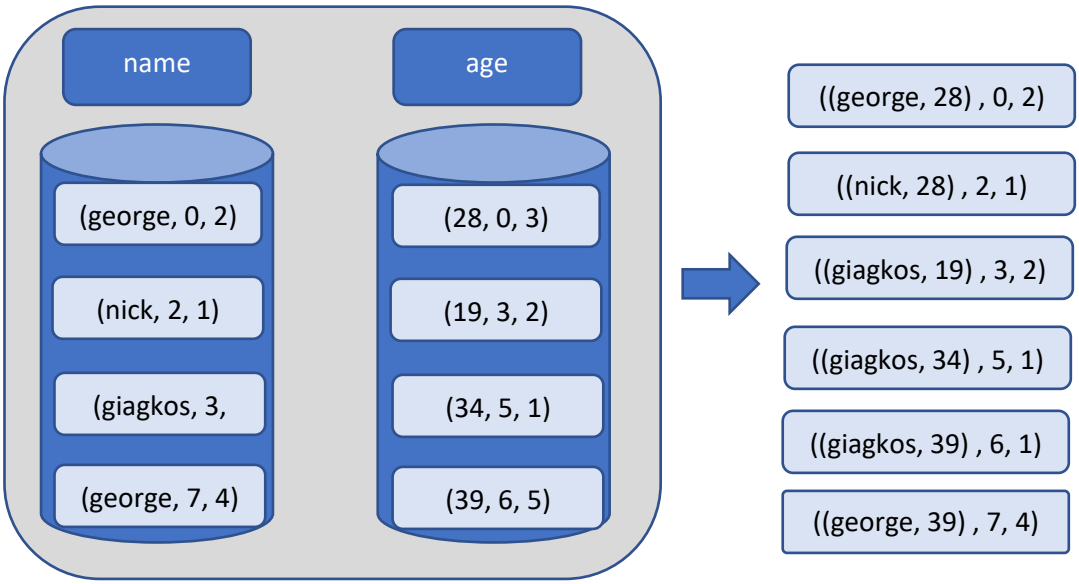
5.3.3. Group by queries

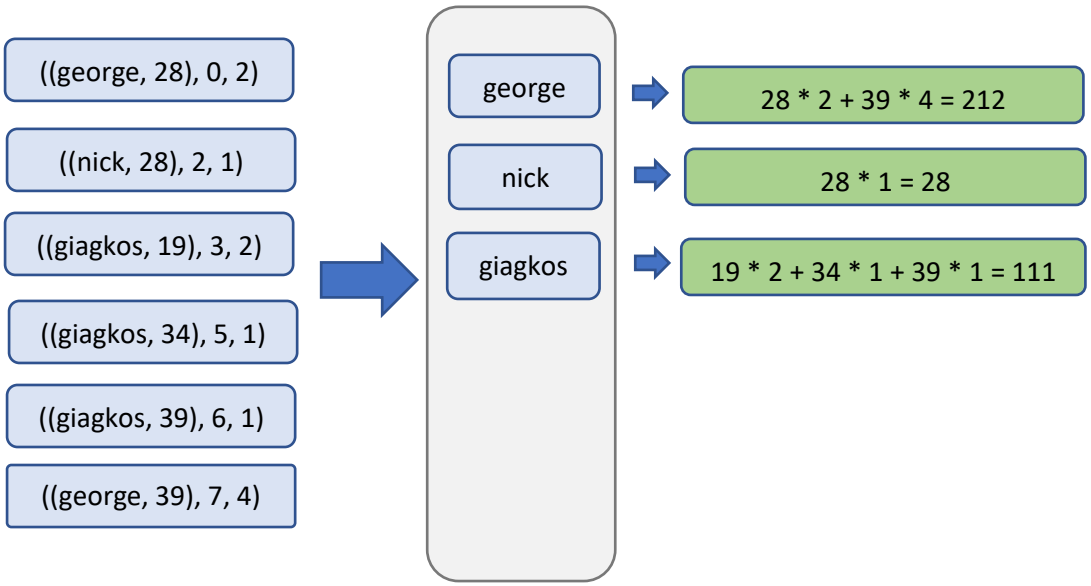
Για την εκτέλεση των group by queries, το hybrid columnar δημιουργεί αρχικά ένα νέο chunk το οποίο περιέχει μόνο τις στήλες πάνω στις οποίες θέλουμε να εκτελέσουμε το query. Για παράδειγμα σε ένα query της μορφής ***select sum(age) from users group by name*** δημιουργείται αρχικά ένα chunk με τις στήλες name και age, όπως φαίνεται παρακάτω:



Στη συνέχεια διατρέχουμε όλες τις γραμμές του νέου chunk και για κάθε ζεύγος (name, age) ελέγχουμε αν υπάρχει το name ως κλειδί σε ένα hashmap, όπου αποθηκεύονται όλοι οι συνδυασμοί. Αν υπάρχει, προσθέτουμε στο αντίστοιχο value του hashmap την τιμή του πεδίου age, ενώ αν δεν υπάρχει προσθέτουμε το ζεύγος (name, age) στο hashmap, έχοντας ως value την τιμή του πεδίου age. Το τελικό αποτέλεσμα είναι ένα hashmap, όπου ως κλειδιά έχει όλα τα ονόματα και ως values το άθροισμα των ηλικιών τους.

Η διαδικασία αυτή φαίνεται παρακάτω:





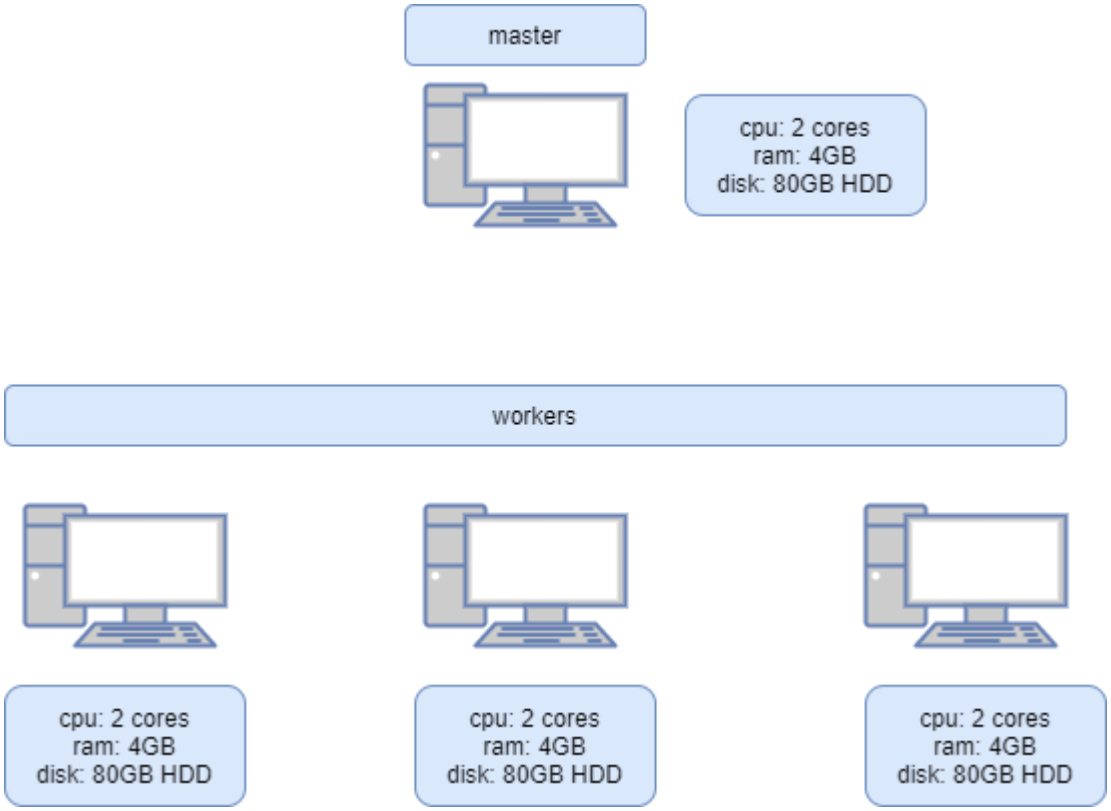
6. Πειραματική αξιολόγηση του συστήματος

Για να αξιολογήσουμε την απόδοση του συστήματος και να το συγκρίνουμε με το map reduce πάνω στα ασυμπίεστα δεδομένα και το parquet, εκτελέσαμε queries σε ένα εύρος datasets διαφορετικών κατανομών και cardinalities.

6.1. Test Setup

Για την εκτέλεση των πειραμάτων δημιουργήσαμε μια συστοιχία υπολογιστών (cluster), η οποία αποτελούταν από 4 εικονικές μηχανές (VMs). Η κάθε εικονική μηχανή είχε διαθέσιμους 2 πυρήνες, 4GB RAM και 80GB αποθηκευτικού χώρου. Το λειτουργικό σύστημα που χρησιμοποιήσαμε σε κάθε εικονική μηχανή ήταν το Ubuntu Linux 16.04 LTS. Πάνω σε αυτή τη συστοιχία εγκαταστήσαμε το Apache Spark, δεσμεύοντας 1 εικονική μηχανή για το ρόλο του master και του name node και τις υπόλοιπες 3 για το ρόλο των workers και των data nodes.

Το σχήμα αυτό φαίνεται παρακάτω:



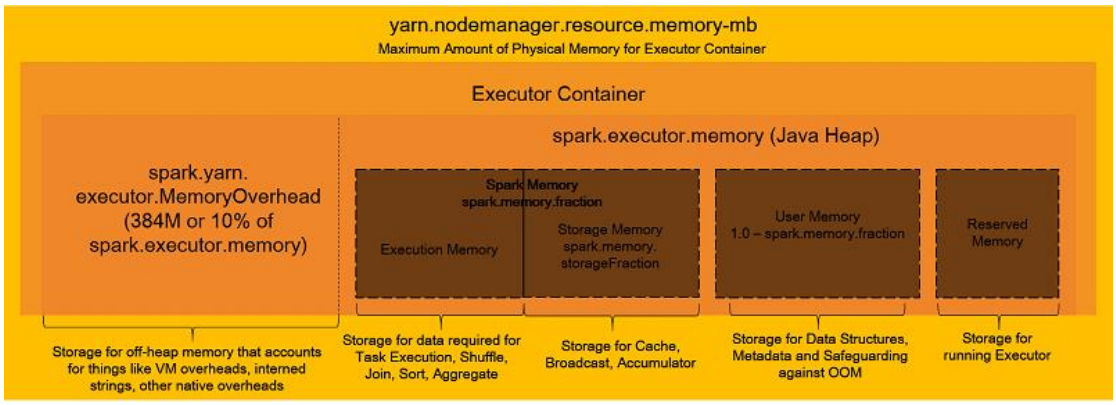
Εικόνα 5: Το σύστημα πάνω στο οποίο εκτελέσαμε τα πειράματα.

Από το παραπάνω setup προκύπτει πως η συνολική υπολογιστική ισχύς των 3 workers ήταν 6 υπολογιστικοί πυρήνες και 12GB RAM. Αυτό όμως δεν σημαίνει πως είχαμε διαθέσιμη όλη την υπολογιστική ισχύ για την εκτέλεση των πειραμάτων. Σε κάθε εικονική μηχανή που τρέχει ο worker δεσμεύσαμε 1GB για το λειτουργικό σύστημα. Αυτό μας αφήνει 3GB για τους workers. Στη συνέχεια, σε κάθε worker ορίζουμε 2 executors, οπότε κάθε executor έχει θεωρητικά 3/2 = 1.5GB. Όμως το memory overhead για την εκτέλεση του κάθε executor είναι max(384, executor_memory * 0.07), άρα στην περίπτωσή μας 384 MB overhead ανά executor. Αυτό μας αφήνει με ~1100MB ανά executor. Μετά, έχουμε ένα ποσοστό που το

δίνουμε εμείς, στην περίπτωσή μας 10% που πάει για overhead του VM. Το υπόλοιπο 90%, δηλαδή περίπου 990 MB πάει για την εκτέλεση και την cache memory του executor.

Εκεί μοιράζουμε το μέγεθος σε 90%, δηλαδή 891 MB για RDD caching per executor. Οπότε σε ένα σύστημα με 6 executors έχουμε 6 * 891 ≈ 5.2GB για RDD caching.

Στο παρακάτω σχήμα [27] φαίνεται η κατανομή της μνήμης στους executors:



Εικόνα 6: Κατανομή μνήμης στους executors.

6.2. Τεχνικές υπό αξιολόγηση

Εκτελέσαμε ένα σύνολο πειραμάτων τόσο σε συνθετικά όσο και σε πραγματικά δεδομένα προκειμένου να μετρήσουμε την αποδοτικότητα σε χρόνο και σε μνήμη των παρακάτω τεχνικών συμπίεσης:

- RLE: Run Length Encoding, αποθήκευση συνεχόμενων τιμών ως (value, start, runLength).
- DELTA: Αποθήκευση των διαφορών των διαδοχικών δεδομένων αντί για την τιμή τους.
- Bit Packing: Αποθήκευση των δεδομένων χρησιμοποιώντας όσα bit χρειάζεται για την αναπαράστασή τους.
- Roaring: Συμπίεση bitmaps μέσω λιστών.
- Hybrid Columnar: Το σύστημα που σχεδιάσαμε, το οποίο επιλέγει την καλύτερη μέθοδο συμπίεσης ανά στήλη και ανά τμήμα στήλης.
- Parquet: Ένα από τα δημοφιλέστερα πρότυπα συμπιεσμένων αρχείων στο Spark.
- Map Reduce: Απευθείας εκτέλεση ερωτημάτων πάνω σε ασυμπίεστα δεδομένα.

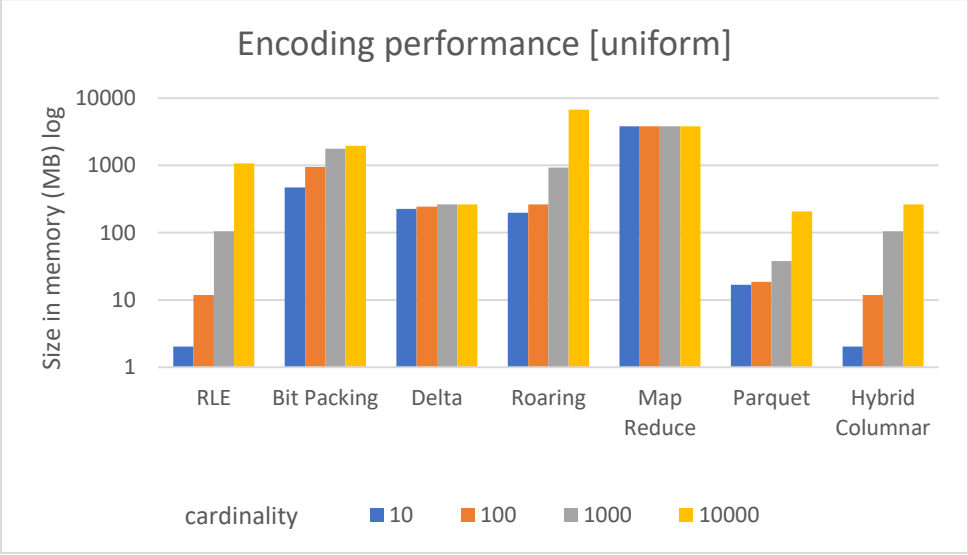
6.3. Συνθετικά δεδομένα

Συγκεκριμένα δημιουργήσαμε datasets που ακολουθούν ομοιόμορφη (uniform), κανονική (normal), καθώς και εκθετική (exponential) κατανομή. Στην εκθετική κατανομή πειραματιστήκαμε με δυο datasets κάθε φορά, ένα με μεγάλο λ (μεγάλο skew) και ένα με μικρότερο λ (μικρότερο skew). Για κάθε κατανομή δημιουργήσαμε datasets των 10, 100, 1.000 και 10.000 διαφορετικών στοιχείων. Σε κάθε περίπτωση τα datasets αποτελούνται από 10^9 γραμμές και μια στήλη. Ο λόγος που επιλέξαμε να πειραματιστούμε με διαφορετικές κατανομές είναι πως στον πραγματικό κόσμο τα datasets μπορεί να ακολουθούν πολλές διαφορετικές κατανομές. Για παράδειγμα οι ηλικίες των μαθητών σε ένα σχολείο είναι σχετικά ομοιόμορφα κατανεμημένες, από 6 έως 12 ετών σε ένα δημοτικό, ενώ οι ηλικίες των εργαζομένων σε μια εταιρία πολλές φορές συγκεντρώνονται γύρω από μια τιμή, για παράδειγμα αρκετοί είναι μεταξύ 30 και 40 ετών ακολουθώντας μια κατανομή κοντά στην κανονική. Αντίθετα, η συγκέντρωση των ανθρώπων σε μεγάλα αστικά κέντρα όπως και το κατά κεφαλήν εισόδημα ακολουθούν συνήθως μια αρκετά skewed κατανομή, όπου μεγάλο μέρος των ανθρώπων ή του πλούτου συσσωρεύεται σε μεγάλες πόλεις και λίγους ανθρώπους αντίστοιχα. Επίσης κάποια datasets αναπαριστούν δεδομένα με μικρό πλήθος διαφορετικών τιμών, όπως ηλικίες, θερμοκρασίες, ονόματα πολιτειών ή νομών, ενώ κάποια άλλα datasets αναπαριστούν δεδομένα με μεγάλο πλήθος διαφορετικών τιμών, όπως τιμές μετοχών ή εμπορικές επωνυμίες.

Σκοπός είναι να αξιολογήσουμε την αναμενόμενη συμπεριφορά του συστήματος σε ένα εύρος πραγματικών datasets, αλλά και να μελετήσουμε τη συμπεριφορά της κάθε τεχνικής συμπίεσης που υλοποιεί το σύστημά μας ανάλογα με την κατανομή και το πλήθος των διαφορετικών τιμών του dataset.

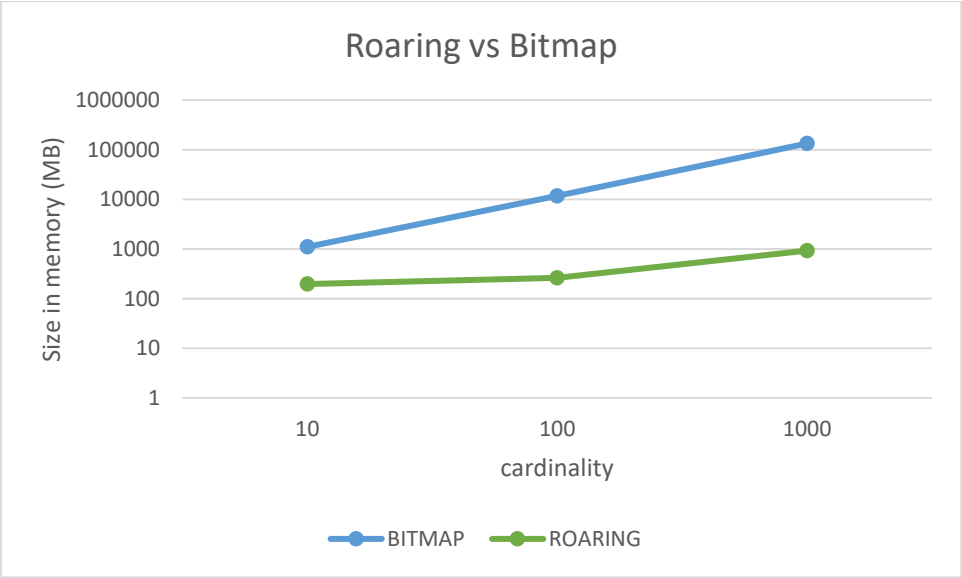
6.3.1. Μέγεθος αποτυπώματος στη μνήμη

Αρχικά συγκρίνουμε τα μεγέθη που καταλαμβάνουν οι διάφορες τεχνικές στη μνήμη.



Εικόνα 7: Εξέλιξη του αποτυπώματος στη μνήμη των τεχνικών συμπίεσης σε σχέση με τον αριθμό των διαφορετικών στοιχείων, ομοιόμορφη κατανομή.

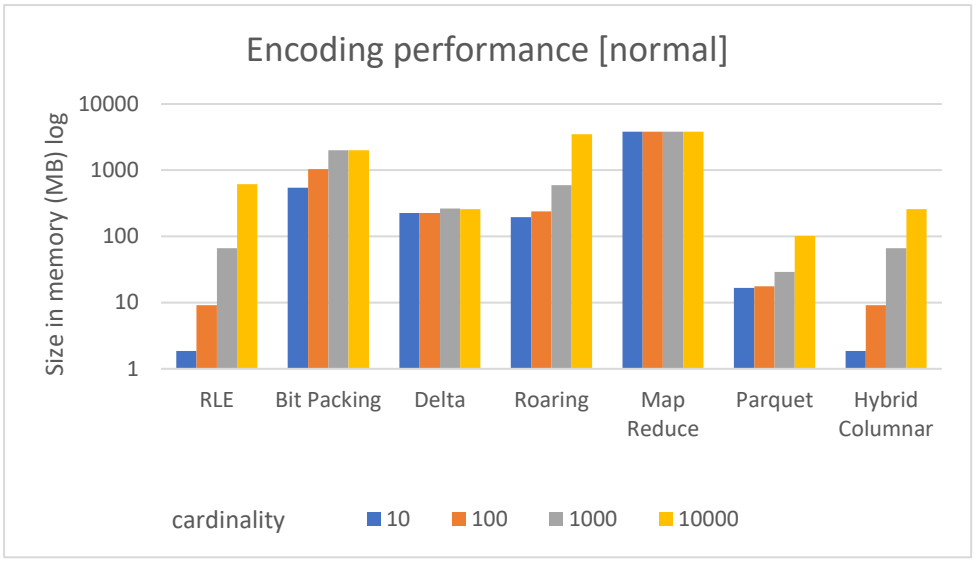
Στο διάγραμμα αυτό βλέπουμε πως τόσο το σύστημά μας όσο και το parquet καταλαμβάνουν τάξεις μεγέθους λιγότερο χώρο στη μνήμη για την αποθήκευση του dataset σε σχέση με το ασυμπίεστο. Παρατηρούμε επίσης πως κάποιες τεχνικές πετυχαίνουν πολύ υψηλό ποσοστό συμπίεσης, αλλά είναι πολύ ευαίσθητες ως προς το πλήθος των διαφορετικών τιμών (RLE). Αντίθετα, κάποιες άλλες ενώ δεν πετυχαίνουν τόσο υψηλό ποσοστό συμπίεσης, καταφέρνουν να κρατήσουν το μέγεθός τους στη μνήμη σε λογικό επίπεδο καθώς αυξάνεται το πλήθος των διαφορετικών τιμών (BIT PACKING, DELTA). Ειδικότερα, η DELTA δείχνει να παραμένει αμετάβλητη από την αύξηση του cardinality. Αυτή η συμπεριφορά καθιστά τις τεχνικές αυτές (BIT PACKING, DELTA) κατάλληλες για datasets με πολλές διαφορετικές τιμές, όπου δεν υπάρχουν μεγάλα run lengths, τα οποία είναι απαραίτητα για να αποδώσει καλά η RLE. Στο διάγραμμα αυτό φαίνεται επίσης και η μετάβαση που κάνει το hybrid columnar, από RLE σε DELTA, για cardinality 10.000, χρησιμοποιώντας έτσι την καλύτερη τεχνική συμπίεσης σε κάθε περίπτωση. Επίσης, παρατηρώντας την τεχνική ROARING αποκαλύπτεται ένα χαρακτηριστικό γνώρισμα των bitmaps, ότι τα πηγαίνουν καλά για μικρά cardinalities. Για μεγαλύτερα cardinalities η ROARING τεχνική συμπίεσης κρατά το μέγεθος σε επίπεδα πολύ μικρότερα από το απλό bitmap. Η διαφορά αυτή σε ρυθμό αύξησης του μεγέθους των δυο τεχνικών σε σχέση με το cardinality εμφανίζεται παρακάτω:



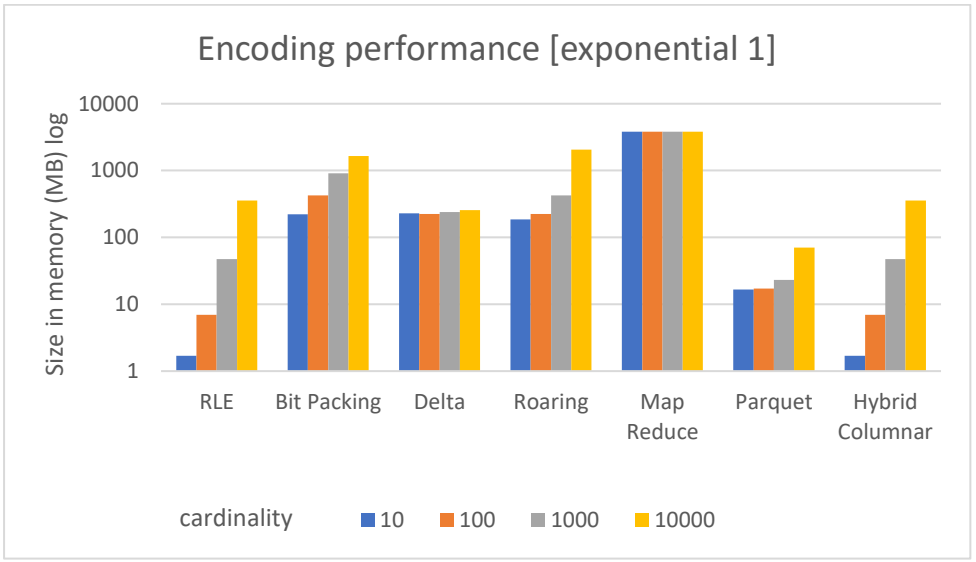
Εικόνα 8: Ρυθμός αύξησης του μεγέθους στη μνήμη για τις τεχνικές bitmap και roaring.

Στο διάγραμμα αυτό παρατηρούμε ότι το μέγεθος του bitmap ακολουθεί το cardinality του dataset, έχοντας 10 φορές μεγαλύτερο μέγεθος για cardinality 100 σε σχέση με 10 και αντίστοιχα 10 φορές μεγαλύτερο μέγεθος για cardinality 1000 σε σχέση με 10. Παρατηρούμε δηλαδή μια απόλυτη συσχέτιση μεταξύ cardinality και αποτυπώματος στη μνήμη. Αντίθετα, το roaring εμφανίζει και αυτό αύξηση του μεγέθους στη μνήμη, αλλά με πολύ μικρότερο ρυθμό. Μια ακόμα διαφορά είναι πως για μεγάλα cardinalities το roaring αποθηκεύει τα δεδομένα ως ακεραίους των 16 bit, βάζοντας έτσι ένα πάνω όριο στα bit που χρειάζονται για την αποθήκευση κάθε στοιχείου. Σε αντίθεση με το roaring, το bitmap για κάθε στοιχείο χρειάζεται αριθμό bit ανάλογο του cardinality καθιστώντας τη χρήση του απαγορευτική για datasets που περιλαμβάνουν μεγάλο πλήθος διαφορετικών στοιχείων. Για το λόγο αυτό, στο hybrid columnar το roaring έχει αντικαταστήσει πλήρως το bitmap, καθώς αποδίδει καλύτερα σε κάθε περίπτωση.

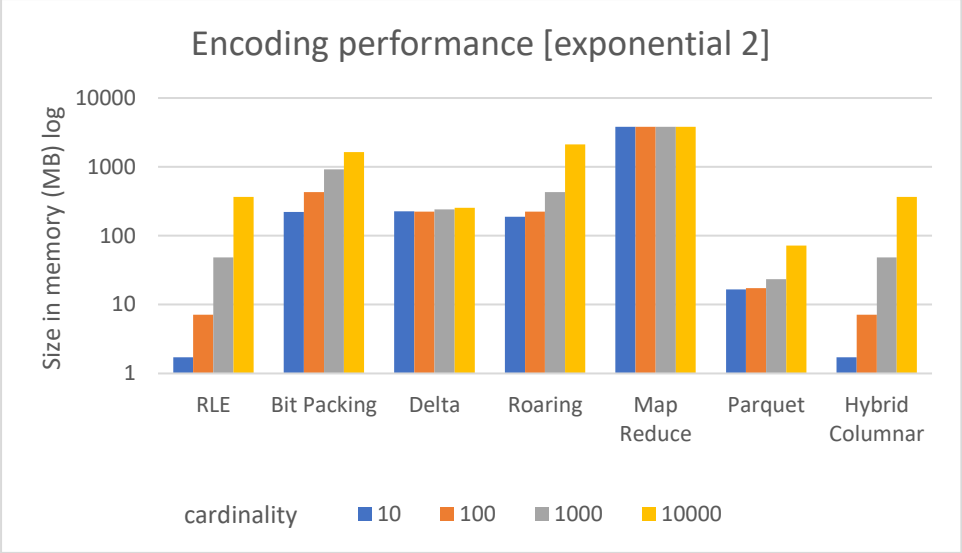
Στη συνέχεια βλέπουμε το μέγεθος που καταλαμβάνουν οι διάφορες τεχνικές συμπίεσης για τις υπόλοιπες κατανομές:



Εικόνα 9: Εξέλιξη του αποτυπώματος στη μνήμη των τεχνικών συμπίεσης σε σχέση με τον αριθμό των διαφορετικών στοιχείων, κανονική κατανομή.



Εικόνα 10: Εξέλιξη του αποτυπώματος στη μνήμη των τεχνικών συμπίεσης σε σχέση με τον αριθμό των διαφορετικών στοιχείων, εκθετική κατανομή.



Εικόνα 11: Εξέλιξη του αποτυπώματος στη μνήμη των τεχνικών συμπίεσης σε σχέση με τον αριθμό των διαφορετικών στοιχείων, εκθετική κατανομή.

Αυτό που παρατηρούμε είναι ότι στις κατανομές αυτές (normal, exponential) οι τεχνικές RLE και ROARING φαίνεται να τα πηγαίνουν καλύτερα σε σχέση με τη uniform κατανομή. Μάλιστα, όσο πιο skewed είναι η κατανομή, όσο δηλαδή υπάρχει μεγαλύτερη ανομοιομορφία στη συχνότητα εμφάνισης των τιμών του dataset τόσο καλύτερα τα πηγαίνουν οι τεχνικές αυτές, με το hybrid columnar να χρησιμοποιεί το RLE ακόμα και για το υψηλότερο cardinality στις exponential κατανομές. Αντίθετα, οι τεχνικές Bit Packing και Delta επηρεάζονται ελάχιστα από τις διαφορετικές κατανομές. Καλύτερα στις skewed κατανομές τα πηγαίνει και το parquet, πετυχαίνοντας τις καλύτερες επιδόσεις για μεγάλα cardinalities.

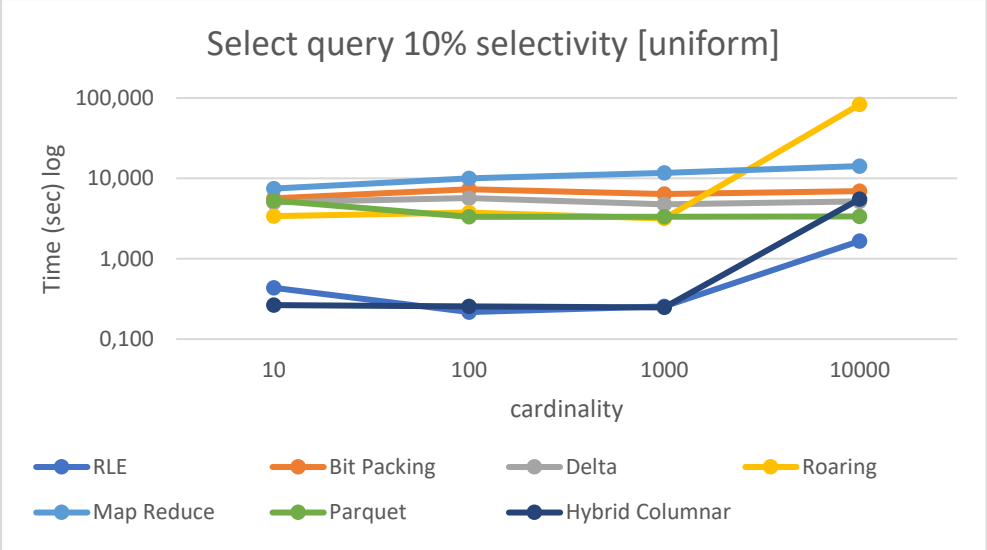
6.3.2. Ταχύτητα εκτέλεσης

Στη συνέχεια εκτελούμε queries πάνω στα datasets προκειμένου να συγκρίνουμε την ταχύτητα των διάφορων τεχνικών συμπίεσης. Υπενθυμίζουμε πως το ζητούμενο σε ένα σύστημα ανάλυσης δεδομένων δεν είναι το απόλυτα μεγαλύτερο ποσοστό συμπίεσης, αλλά ένας συνδυασμός αποδεκτού μεγέθους στη μνήμη και υψηλής ταχύτητας εκτέλεσης. Για κάθε dataset εκτελούμε 3 queries, ένα select sum με μικρό selectivity (10%), ένα select sum με μεγάλο selectivity (75%) και ένα group by query.

6.3.2.1. Select queries

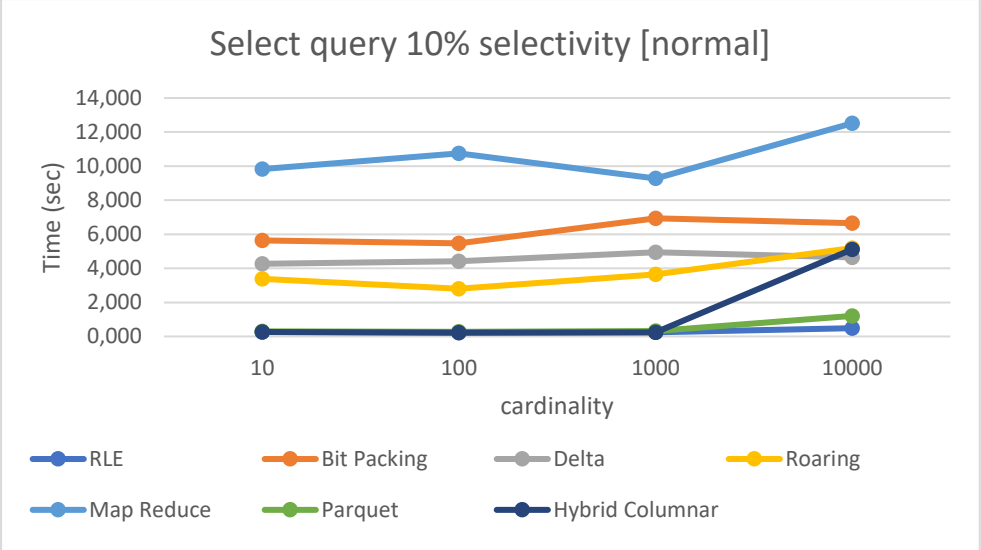
Ως selectivity ορίζουμε το ποσοστό του dataset που επεξεργάζεται το query. Για παράδειγμα ένα *select sum(price) where price < 10* query που τρέχει σε ένα dataset με τιμές στο εύρος [0-100] έχει 10% selectivity. Η λογική πίσω από τα διαφορετικά selectivities είναι να δούμε κατά πόσο επηρεάζεται το κάθε σύστημα από την επεξεργασία του dataset.

Ξεκινάμε με τις μετρήσεις για το select sum query με μικρό selectivity:



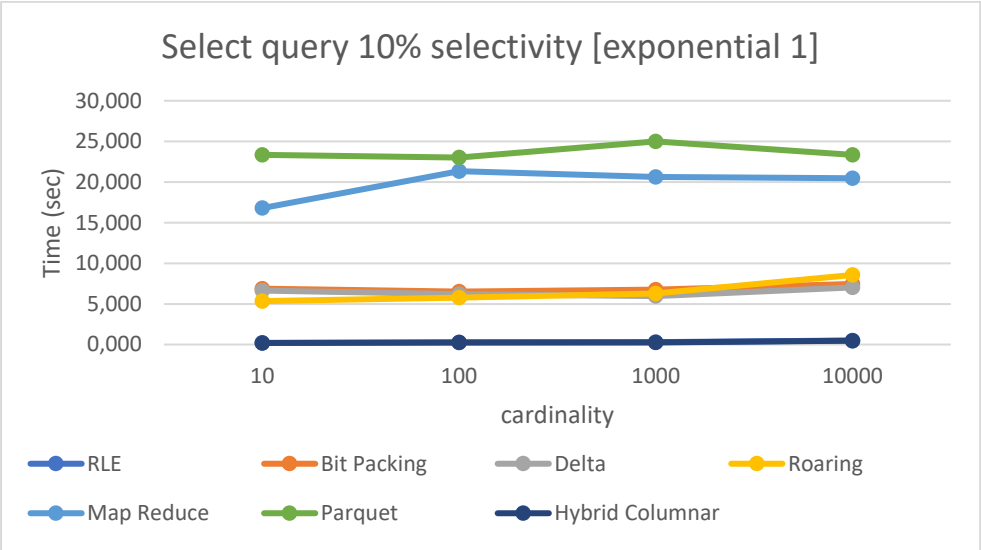
Εικόνα 12: Χρόνος εκτέλεσης select query με μικρό selectivity, σε δεδομένα που ακολουθούν ομοιόμορφη κατανομή.

Στην ομοιόμορφη κατανομή βλέπουμε πως οι επιδόσεις των περισσότερων τεχνικών συμπίεσης κυμαίνονται στα ίδια επίπεδα, με εξαίρεση αυτών του RLE το οποίο είναι κατά πολύ ταχύτερο. Ενδιαφέρον παρουσιάζει και το γεγονός πως ακόμα και το ασυμπίεστο δεν είναι κατά πολύ πιο αργό από το parquet, παρότι το parquet καταλαμβάνει τάξεις μεγέθους λιγότερο χώρο στη μνήμη. Οι επιδόσεις του hybrid columnar ταυτίζονται με αυτές του RLE και είναι κατά πολύ καλύτερες τόσο σε σχέση με το ασυμπίεστο όσο και σε σχέση με το parquet για cardinalities μέχρι και 1000, ενώ για cardinality 10000 είναι ίδιες με τη DELTA, μαρτυρώντας τις επιλογές που κάνει το hybrid columnar για κάθε cardinality.

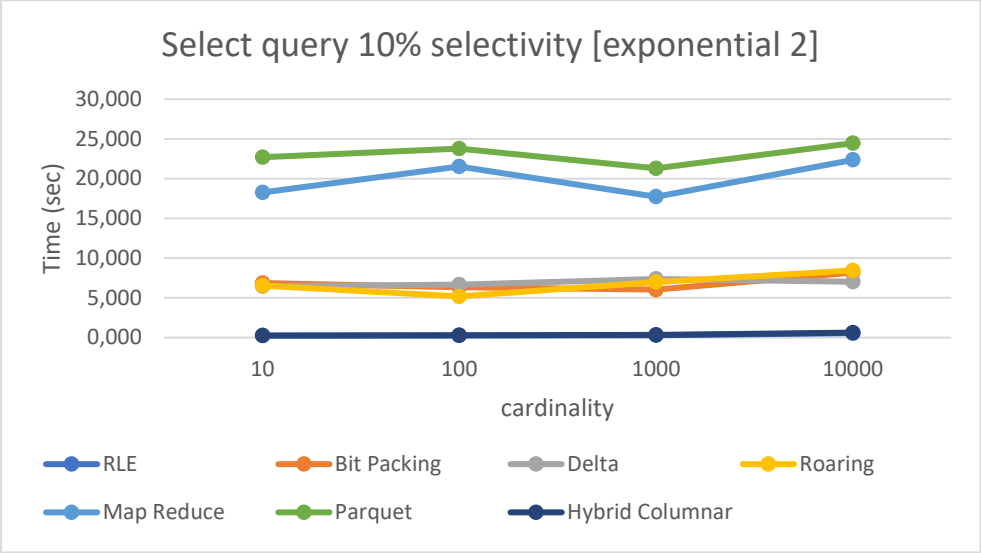


Εικόνα 13: Χρόνος εκτέλεσης select query με μικρό selectivity, σε δεδομένα που ακολουθούν ομοιόμορφη κατανομή.

Το ενδιαφέρον στην κανονική κατανομή είναι πως όλες οι τεχνικές συμπίεσης πετυχαίνουν καλύτερους χρόνους από το ασυμπίεστο. Ειδικά το parquet στην κανονική κατανομή πετυχαίνει μια τάξη μεγέθους καλύτερους χρόνους σε σχέση με τις επιδόσεις του στην ομοιόμορφη κατανομή. Γιατί όμως συμβαίνει αυτό; Στην κανονική κατανομή το μεγαλύτερο μέρος του δείγματος συγκεντρώνεται γύρω από μια μέση τιμή, με τις τιμές να εμφανίζονται όλο και πιο σπάνια όσο απομακρύνονται από αυτή. Καθώς το query επεξεργάζεται τις 10% μικρότερες τιμές του dataset, δεν επεξεργάζεται το 10% του dataset όπως θα έκανε στην ομοιόμορφη κατανομή, αλλά πολύ μικρότερο ποσοστό του. Στο σημείο αυτό αποκαλύπτεται ένα optimization του parquet. Για κάθε container που παίρνει από τη μνήμη ελέγχει τη μέγιστη και την ελάχιστη τιμή του και αν το query δεν περιλαμβάνει τιμές σε αυτό το εύρος αφήνει το container και προχωρά στο επόμενο, πετυχαίνοντας έτσι καλύτερες επιδόσεις.



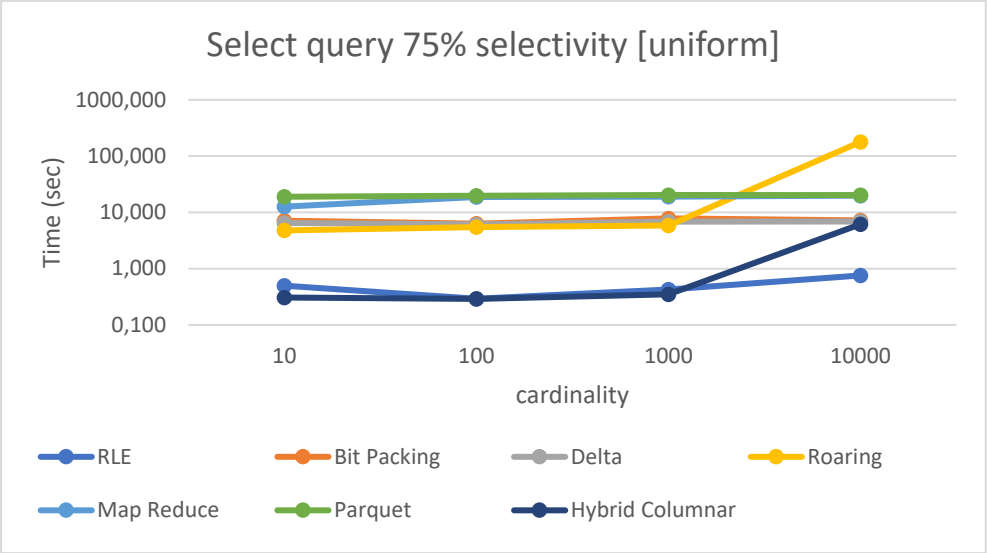
Εικόνα 14: Χρόνος εκτέλεσης select query με μικρό selectivity, σε δεδομένα που ακολουθούν εκθετική κατανομή.



Εικόνα 15: Χρόνος εκτέλεσης select query με μικρό selectivity, σε δεδομένα που ακολουθούν εκθετική κατανομή.

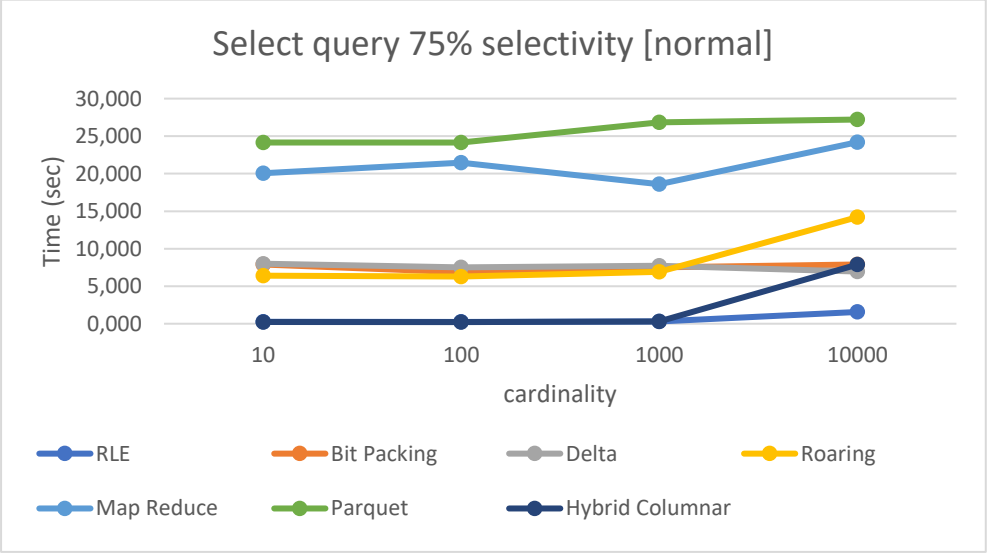
Καθώς προχωράμε στα dataset που ακολουθούν εκθετική κατανομή βλέπουμε το parquet να απαιτεί τάξεις μεγέθους περισσότερο χρόνο σε σχέση με την κανονική κατανομή και να είναι πιο αργό και από το ασυμπίεστο. Αυτό συμβαίνει για τον ίδιο λόγο που πριν ήταν το πιο γρήγορο. Συγκεκριμένα, στην εκθετική κατανομή ο κυριότερος όγκος δεδομένων συγκεντρώνεται στις πρώτες τιμές του dataset, οπότε όταν επεξεργαζόμαστε το μικρότερο 10% των τιμών, στην πραγματικότητα επεξεργαζόμαστε ένα πολύ μεγαλύτερο κομμάτι του dataset. Αντίθετα, το hybrid columnar πετυχαίνει με διαφορά τις καλύτερες επιδόσεις, με τους χρόνους του να ταυτίζονται με αυτούς του RLE για κάθε cardinality. Ο λόγος που το RLE πετυχαίνει καλύτερες επιδόσεις στην εκθετική κατανομή είναι πως αυτή είναι πιο skewed, έχει δηλαδή λίγες τιμές που εμφανίζονται πολύ συχνά δημιουργώντας μεγάλα run lengths, τα οποία και εκμεταλλεύεται το RLE.

Για να διερευνήσουμε περισσότερο την επίδραση που έχει το ποσοστό των δεδομένων του dataset που ακουμπά το query στις επιδόσεις του κάθε συστήματος τρέχουμε το ίδιο query, αυτή τη φορά με selectivity 75%.



Εικόνα 16: Χρόνος εκτέλεσης select query με μεγάλο selectivity, σε δεδομένα που ακολουθούν ομοιόμορφη κατανομή.

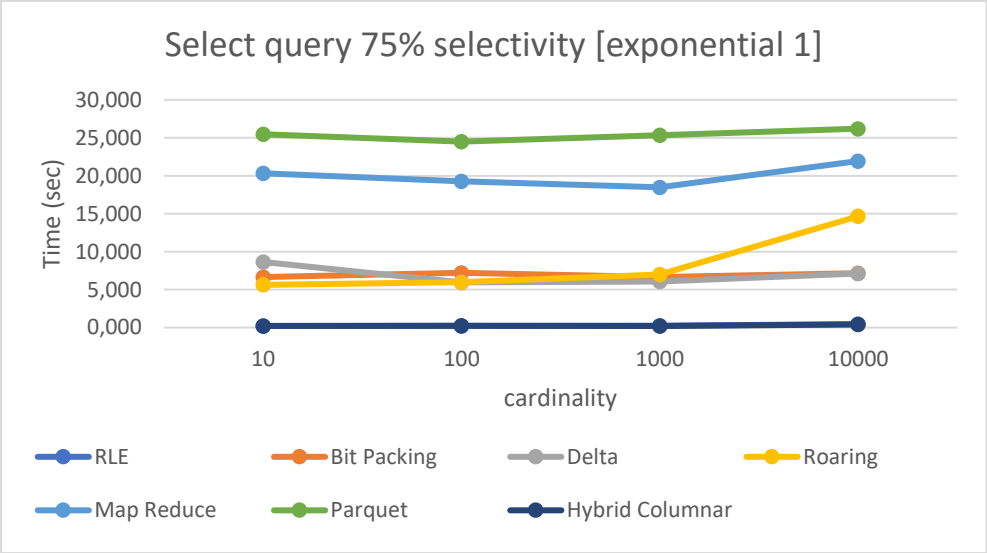
Στην ομοιόμορφη κατανομή βλέπουμε πως το parquet έχει ακριβώς την ίδια συμπεριφορά με το ασυμπίεστο. Το hybrid columnar δεν φαίνεται να επηρεάζεται από το selectivity, πετυχαίνοντας τις καλύτερες επιδόσεις. Επίσης φαίνεται πάλι η μετάβαση που κάνει από RLE σε DELTA για cardinality 10.000. Η επιλογή αυτή γλυτώνει στο hybrid columnar αρκετό χώρο στη μνήμη, το κάνει όμως πιο αργό καθώς φαίνεται ότι η DELTA είναι αρκετά πιο αργή από την RLE ανεξαρτήτως selectivity και cardinality.



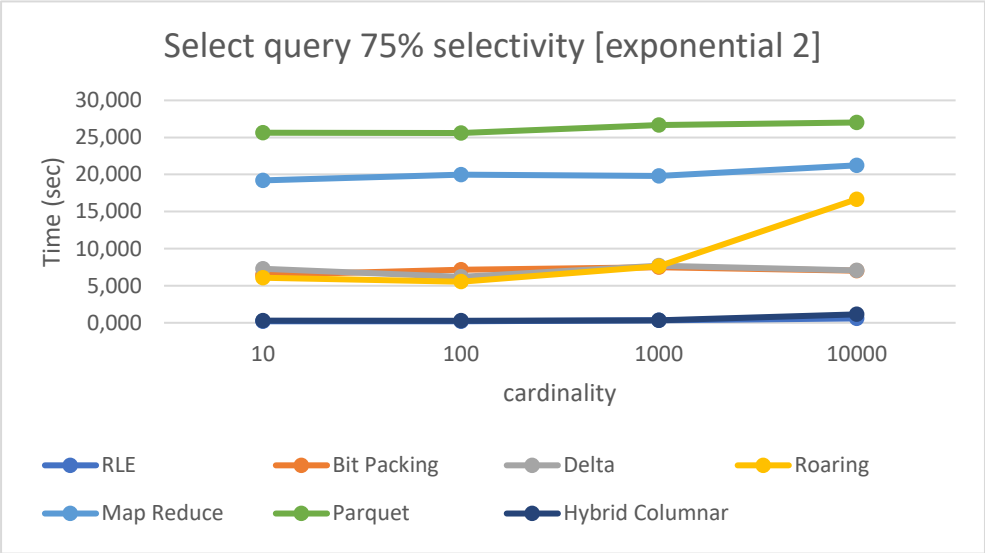
Εικόνα 17: Χρόνος εκτέλεσης select query με μεγάλο selectivity, σε δεδομένα που ακολουθούν κανονική κατανομή.

Κοιτάζοντας τα αποτελέσματα της κανονικής κατανομής επιβεβαιώνεται πως το parquet είναι όντως dependent από το ποσοστό του dataset που ακουμπά το κάθε query. Ενώ για

την ίδια κατανομή είχε επιδόσεις αντίστοιχες του hybrid columnar, για το ίδιο query με μεγαλύτερο selectivity οι επιδόσεις του parquet πέφτουν σημαντικά και είναι χειρότερες ακόμα και σε σχέση με το ασυμπίεστο.



Εικόνα 18: Χρόνος εκτέλεσης select query με μεγάλο selectivity, σε δεδομένα που ακολουθούν εκθετική κατανομή.



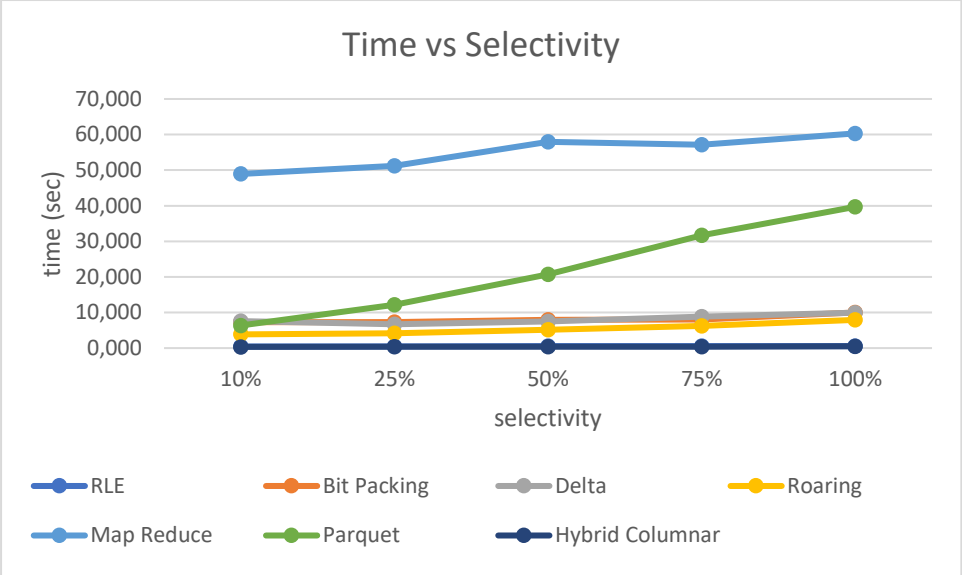
Εικόνα 19: Χρόνος εκτέλεσης select query με μεγάλο selectivity, σε δεδομένα που ακολουθούν εκθετική κατανομή.

Πηγαίνοντας στις εκθετικές κατανομές βλέπουμε το hybrid columnar να πετυχαίνει τάξεις μεγέθους καλύτερες επιδόσεις, τόσο σε σχέση με το ασυμπίεστο όσο και σε σχέση με το parquet. Όπως και με το μικρό selectivity έτσι και εδώ δεν παρατηρούμε το σκαλοπάτι που κάνει το hybrid columnar στις άλλες κατανομές, καθώς στις εκθετικές κατανομές επιλέγει το RLE σε κάθε περίπτωση.

6.3.2.2 Επίδραση του selectivity στα select queries

Στα παραπάνω queries εξετάσαμε την απόδοση των select queries για διάφορες κατανομές και cardinalities. Τρέξαμε κάθε select query για μικρό (10%) και μεγάλο (75%) selectivity προκειμένου να δούμε το βαθμό που αυτό επηρεάζει την απόδοση του συστήματος. Προκειμένου να αξιολογήσουμε πιο αναλυτικά την επίδραση του selectivity για τα διάφορα συστήματα, τρέξαμε ένα select query για κατανομή uniform και cardinality 100, με selectivity 10%, 25%, 50%, 75% και 100%.

Τα αποτελέσματα παρουσιάζονται παρακάτω:

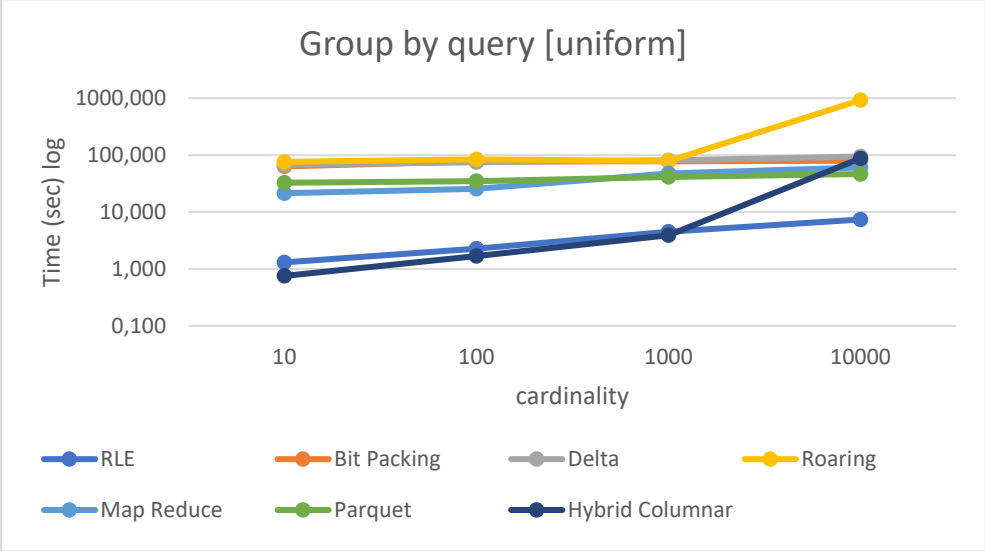


Εικόνα 20: Επίδραση του selectivity στην απόδοση του συστήματος

Όπως παρατηρούμε το hybrid columnar, όπως και οι επιμέρους τεχνικές συμπίεσης που υποστηρίζει δεν επηρεάζονται καθόλου από την αύξηση του selectivity. Το ίδιο ισχύει και στο ασυμπίεστο dataset. Αντίθετα, οι επιδόσεις του parquet επηρεάζονται γραμμικά από την αύξηση του selectivity. Αυτό είναι αναμενόμενο, καθώς το parquet κρατά στοιχεία για κάθε page, όπως τις min και max τιμές που αυτό περιέχει, οπότε μπορεί να προσπεράσει τα pages αυτά που δεν έχουν τιμές στο εύρος που αναζητούμε.

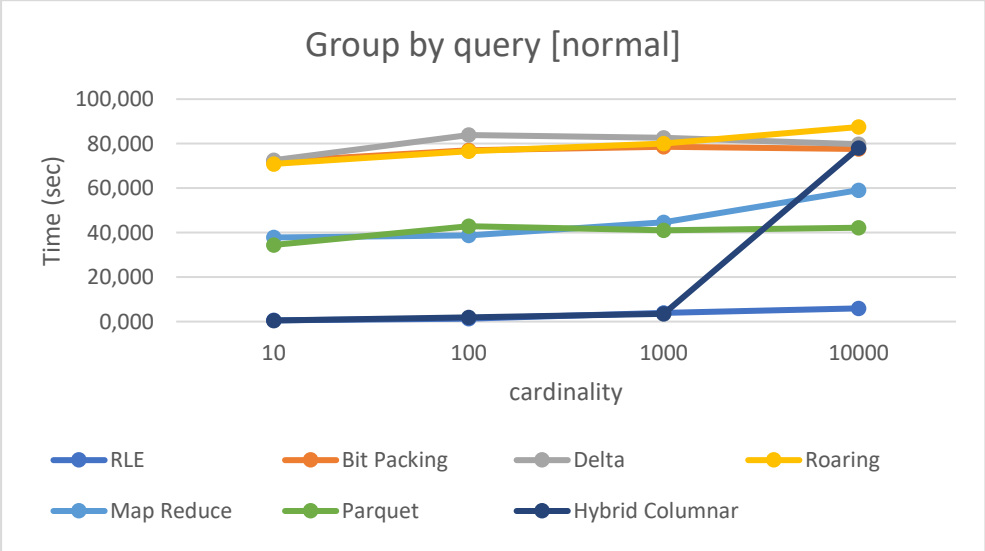
6.3.2.3. Group by Queries

Έχοντας δει την επίδραση του selectivity πάνω στις διάφορες τεχνικές συμπίεσης, θα αξιολογήσουμε την απόδοση των συστημάτων σε group by queries. Τα queries αυτά είναι πολύ συνήθη σε εφαρμογές ανάλυσης δεδομένων και αρκετά απαιτητικά από πλευράς υπολογιστικού χρόνου, οπότε μια τέτοια σύγκριση αποκτά ενδιαφέρον.



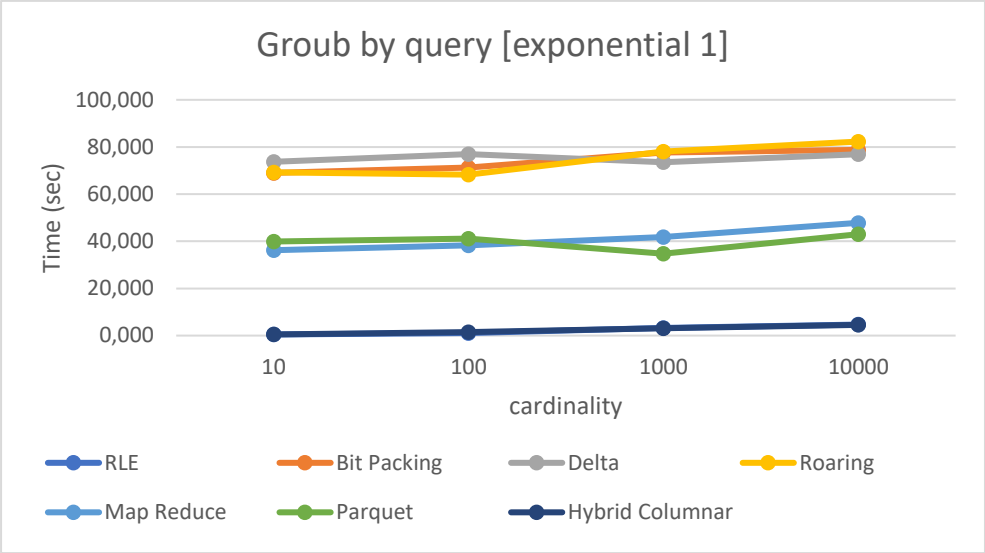
Εικόνα 21: Επιδόσεις των διάφορων τεχνικών σε group by queries, σε δεδομένα που ακολουθούν ομοιόμορφη κατανομή.

Στην ομοιόμορφη κατανομή βλέπουμε το ασυμπίεστο να έχει παρόμοιες επιδόσεις με το parquet και το hybrid columnar να πετυχαίνει τάξεις μεγέθους καλύτερους χρόνους για όλα τα cardinalities εκτός του τελευταίου. Στο τελευταίο cardinality η επίδοση του hybrid columnar πέφτει κατά πολύ, ως αποτέλεσμα της μετάβασης σε DELTA κωδικοποίηση για το cardinality αυτό.

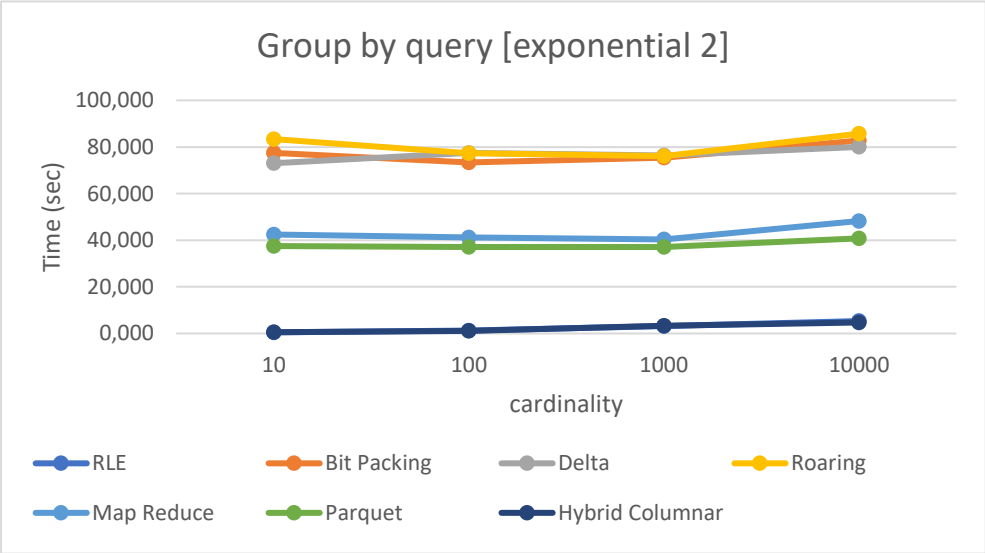


Εικόνα 22: Επιδόσεις των διάφορων τεχνικών σε group by queries, σε δεδομένα που ακολουθούν κανονική κατανομή.

Τα αποτελέσματα στην κανονική κατανομή είναι αντίστοιχα με αυτά της ομοιόμορφης, με το hybrid columnar να είναι ταχύτερο παντού, εκτός από το τελευταίο cardinality όπου εμφανίζεται το χαρακτηριστικό γόνατο λόγω της επιλογής της DELTA.



Εικόνα 23: Επιδόσεις των διάφορων τεχνικών σε group by queries, σε δεδομένα που ακολουθούν εκθετική κατανομή.



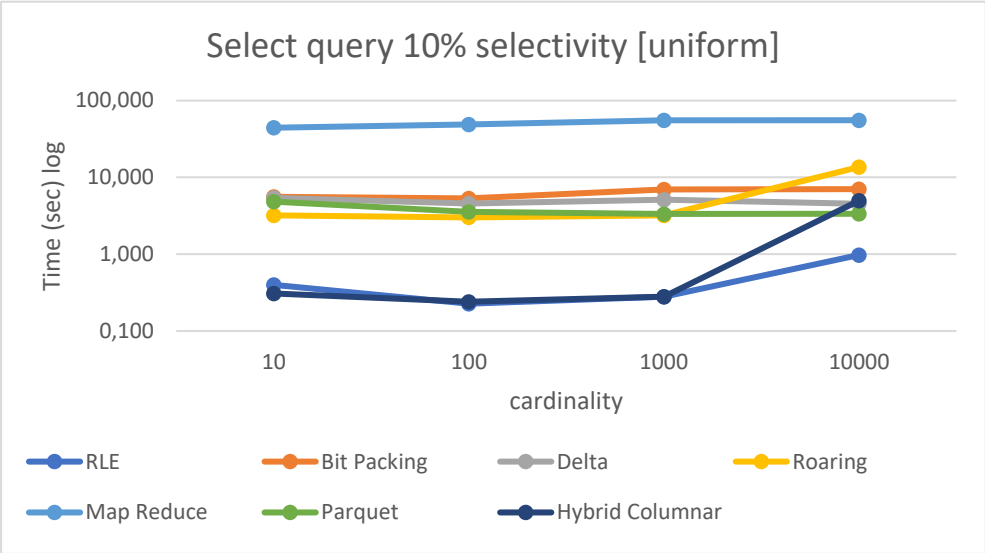
Εικόνα 24: Επιδόσεις των διάφορων τεχνικών σε group by queries, σε δεδομένα που ακολουθούν εκθετική κατανομή.

Στις εκθετικές κατανομές το hybrid columnar είναι καλύτερο παντού, καθώς σε όλα τα cardinalities επιλέγει την RLE, η οποία πετυχαίνει και τις καλύτερες επιδόσεις. Είναι επίσης ενδιαφέρον πως οι επιδόσεις του parquet ταυτίζονται με αυτές του ασυμπίεστου, παρότι το τελευταίο καταλαμβάνει τάξεις μεγέθους περισσότερο χώρο στη μνήμη. Αντίθετα, το hybrid columnar εκτελεί τα queries σε τάξεις μεγέθους λιγότερο χρόνο, δείχνοντας πόσο σημαντικό είναι οι τεχνικές συμπίεσης να είναι lightweight, ώστε εκτός από space efficiency να πετυχαίνουν και ικανοποιητικό time efficiency.

6.3.1. Ταχύτητα εκτέλεσης για μικρό μέγεθος μνήμης

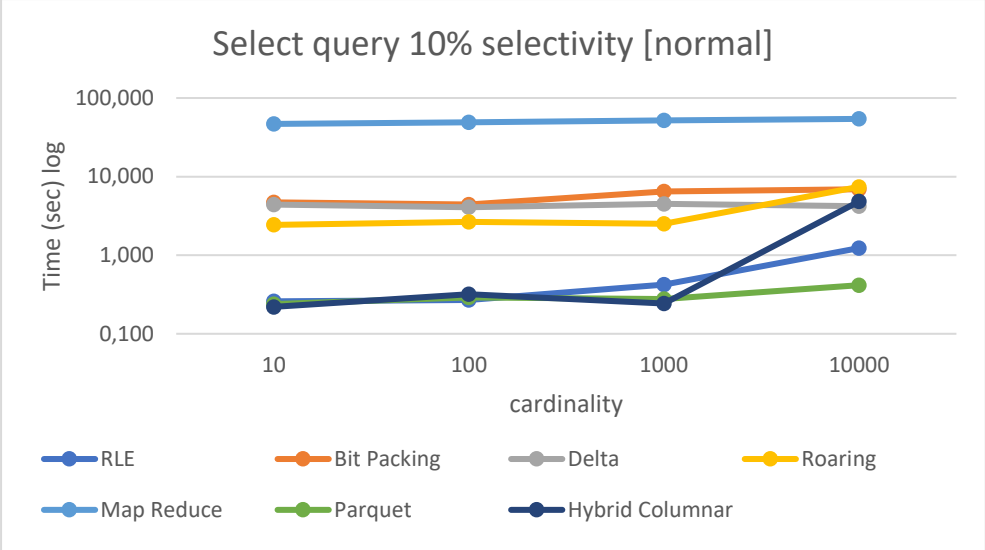
Μέχρι τώρα είχαμε επικεντρωθεί στην ταχύτητα των διάφορων τεχνικών συμπίεσης σε σχέση με το queries πάνω στο ασυμπίεστο dataset. Στο setup που τρέξαμε τα πειράματα είχαμε διαθέσιμα 4.2GB για την αποθήκευση του dataset στη μνήμη, κάτι που σημαίνει πως ακόμα και το ασυμπίεστο dataset, το οποίο απαιτούσε 3.8GB, χωρούσε ολόκληρο στη RAM. Ο σημαντικότερος όμως λόγος χρήσης ενός συστήματος που συμπιέζει τα δεδομένα, όπως το parquet ή το hybrid columnar είναι η δυνατότητα εκτέλεσης queries κατευθείαν από τη RAM, ακόμα και όταν αυτά δεν χωράνε στην ασυμπίεστη μορφή τους. Επομένως, για να δούμε τις διαφορές αυτών των συστημάτων σε ένα σενάριο όπου το dataset δεν θα χωρά ασυμπίεστο στην κύρια μνήμη τρέξαμε τα ίδια πειράματα, αυτή τη φορά με 800MB διαθέσιμα για αποθήκευση του dataset.

6.3.1.1 Select queries



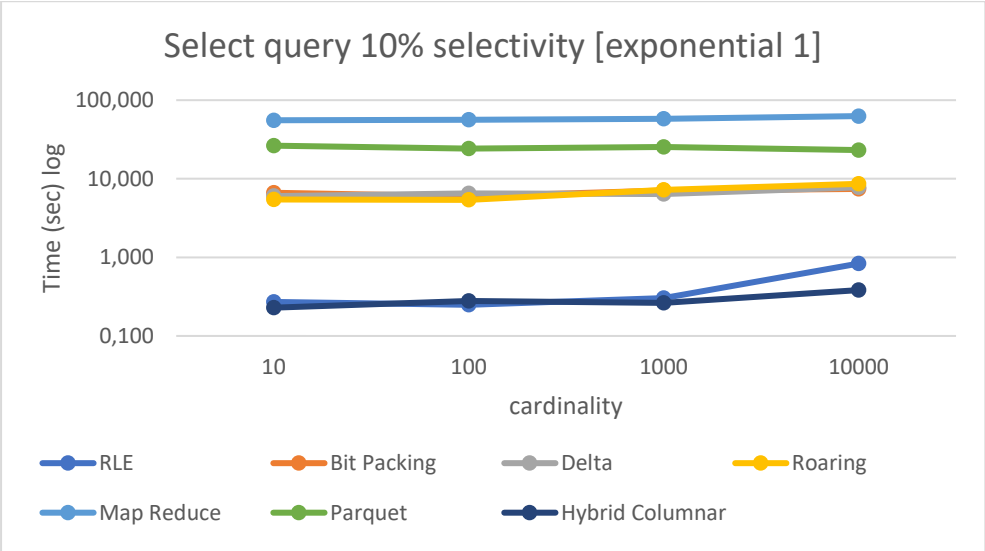
Εικόνα 25: Επιδόσεις select query με μικρό selectivity για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν ομοιόμορφη κατανομή.

Έχοντας λιγότερη μνήμη από αυτή που χρειάζεται για την αποθήκευση ολόκληρου του dataset στη μνήμη, το ασυμπίεστο είναι πλέον εμφανώς πιο αργό από το parquet, το οποίο χωρά ολόκληρο το dataset στη μνήμη. Το hybrid columnar είναι πάλι πολύ ταχύτερο τόσο από το ασυμπίεστο, όσο και από το parquet για όλα τα cardinalities εκτός του τελευταίου.

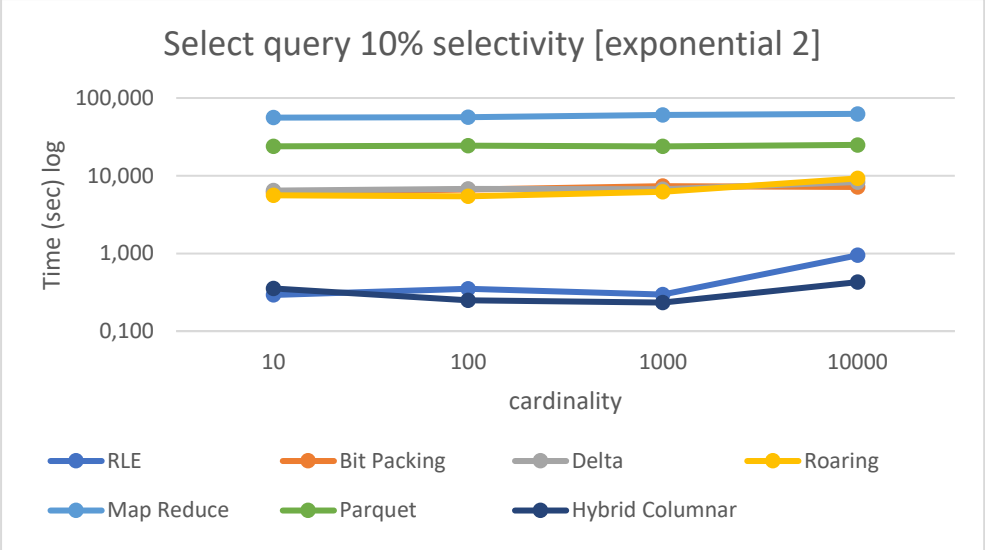


Εικόνα 26: Επιδόσεις select query με μικρό selectivity για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν κανονική κατανομή.

Την ίδια εικόνα για το ασυμπίεστο παρατηρούμε και εδώ, καθώς είναι εμφανώς πιο αργό σε σχέση με το setup με την περισσότερη διαθέσιμη μνήμη. Το parquet, όπως και στο προηγούμενο setup πετυχαίνει εξαιρετικές επιδόσεις καθώς λόγω selectivity και κατανομής ακουμπά μικρό ποσοστό του dataset.



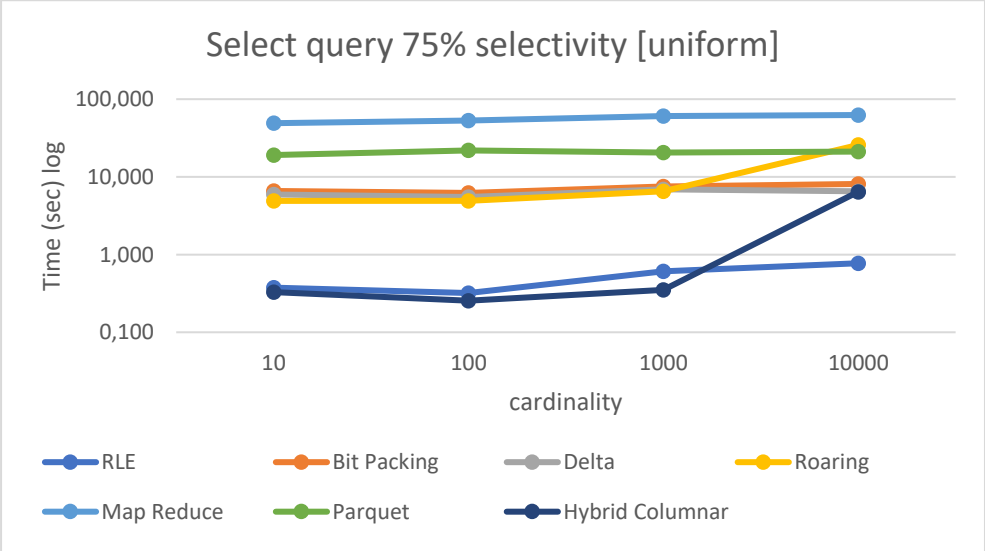
Εικόνα 27: Επιδόσεις select query με μικρό selectivity για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν εκθετική κατανομή.



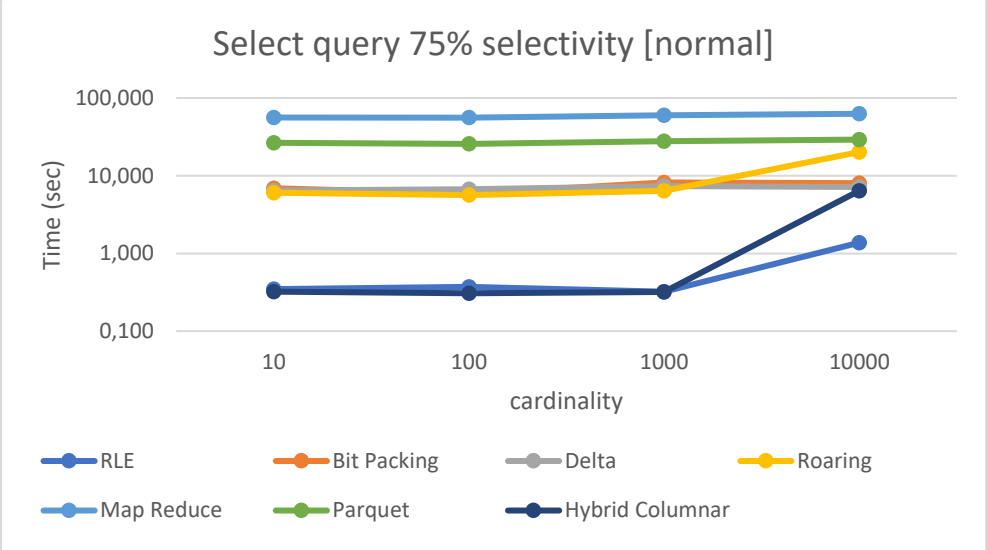
Εικόνα 28: Επιδόσεις select query με μικρό selectivity για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν εκθετική κατανομή.

Η εικόνα δεν αλλάζει ιδιαίτερα σε σχέση με το πρώτο setup, με το hybrid columnar να έχει τάξεις μεγέθους καλύτερες επιδόσεις από το parquet. Η διαφορά σε σχέση με πριν είναι πως το parquet έχει σαφώς καλύτερες επιδόσεις σε σχέση με το ασυμπίεστο, ενώ στο setup με την περισσότερη μνήμη οι χρόνοι τους κυμαινόντουσαν στα ίδια επίπεδα.

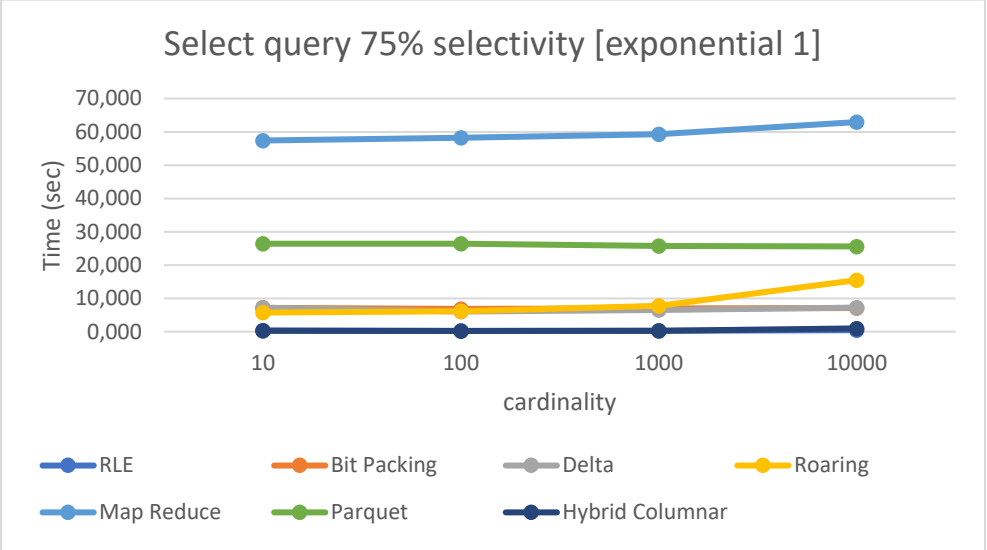
Στη συνέχεια, στο ίδιο setup των 800MB διαθέσιμης μνήμης, τρέχουμε τα queries με selectivity 75%.



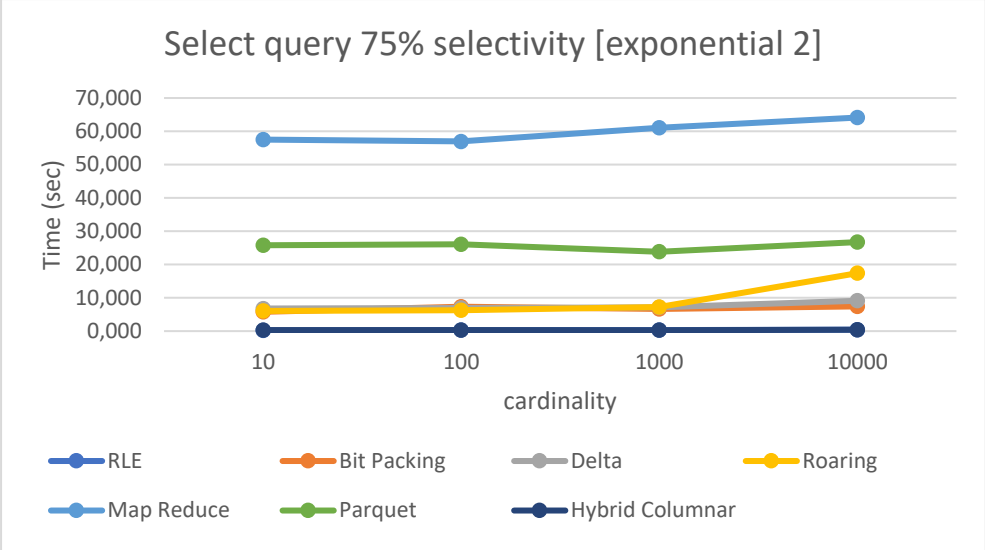
Εικόνα 29: Επιδόσεις select query με μεγάλο selectivity για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν ομοιόμορφη κατανομή.



Εικόνα 30: Επιδόσεις select query με μεγάλο selectivity για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν κανονική κατανομή.



Εικόνα 31: Επιδόσεις select query με μεγάλο selectivity για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν εκθετική κατανομή.

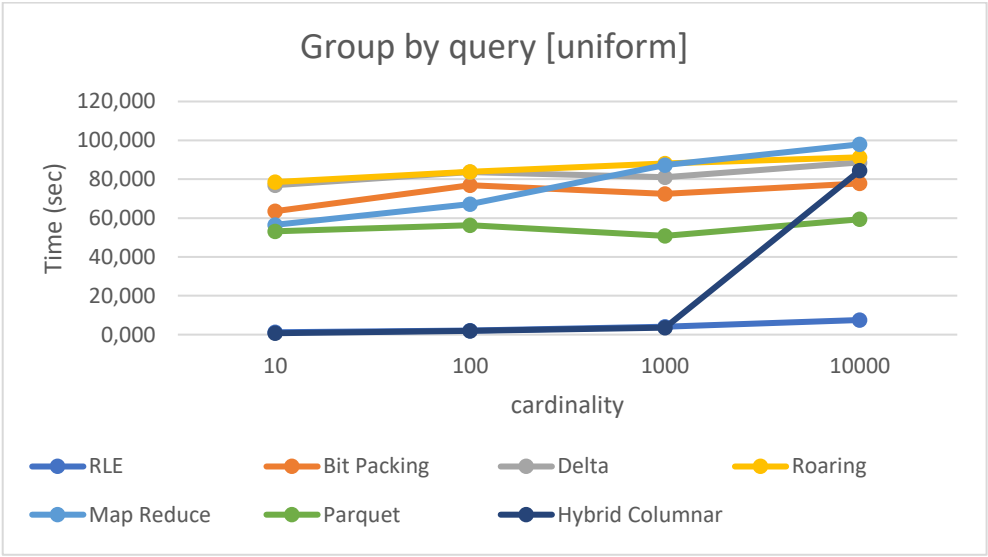


Εικόνα 32: Επιδόσεις select query με μεγάλο selectivity για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν εκθετική κατανομή.

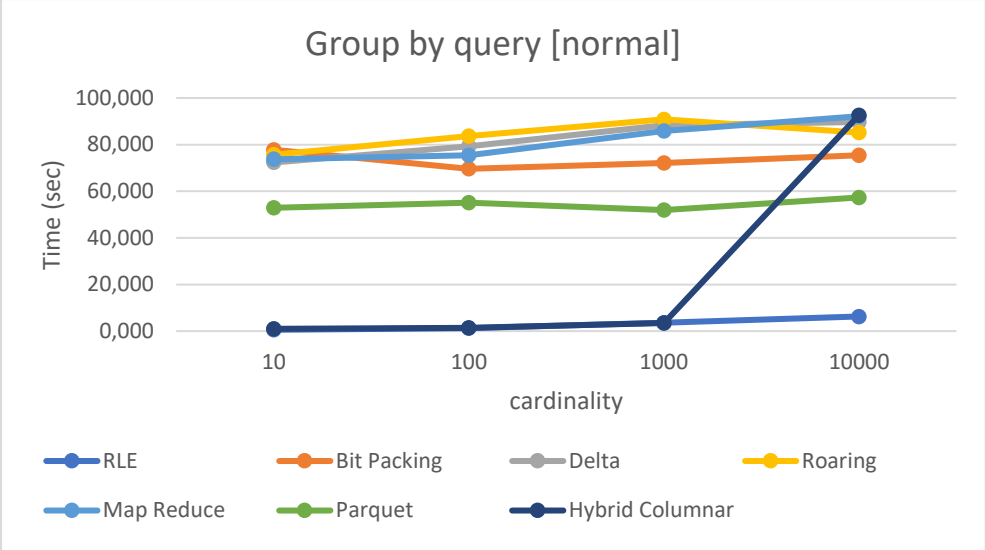
Στα queries με μεγάλο selectivity φαίνεται καθαρά τα οφέλη της συμπίεσης δεδομένων, καθώς τα συστήματα που χρησιμοποιούν συμπίεση (parquet, hybrid columnar) είναι σαφώς πιο γρήγορα. Το hybrid columnar εξακολουθεί να είναι τάξεις μεγέθους πιο γρήγορο από το parquet, με το τελευταίο να πετυχαίνει καλύτερες επιδόσεις σε σχέση με το ασυμπίεστο dataset, το οποίο στο setup αυτό δεν χωρά στη μνήμη του cluster.

6.3.1.2. Group by queries

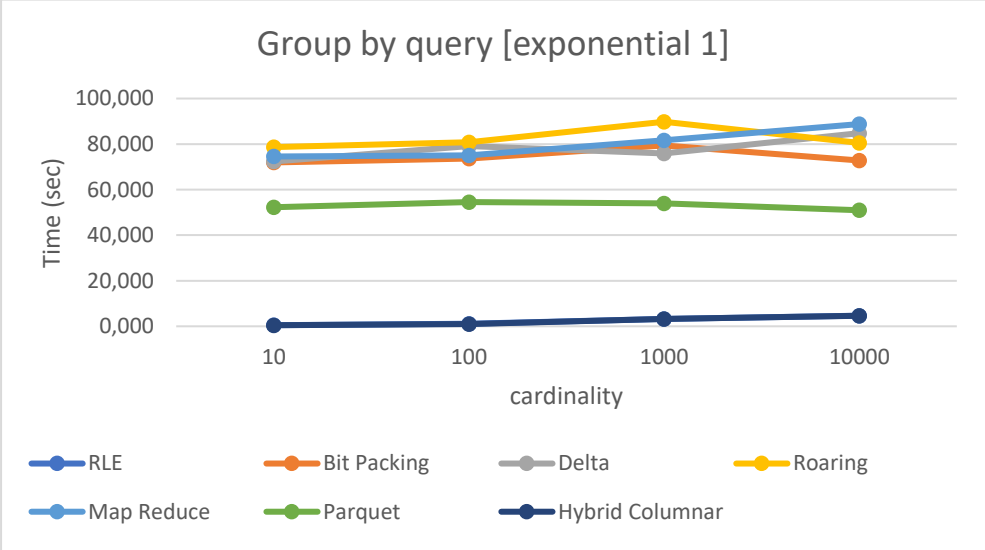
Τέλος, για το setup αυτό τρέχουμε και τα group by queries προκειμένου να εντοπίσουμε τυχόν διαφορές σε σχέση με το setup των 4.2GB μνήμης.



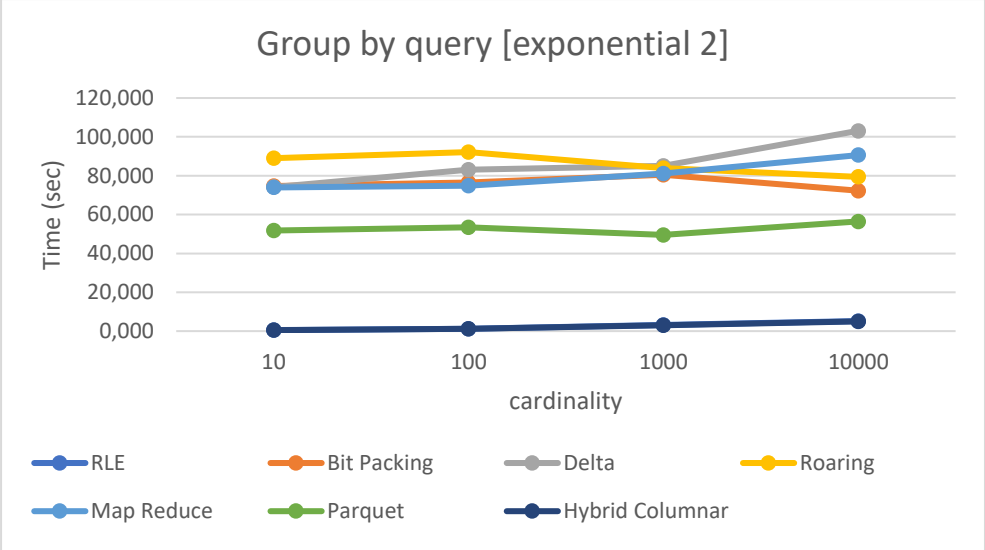
Εικόνα 33: Επιδόσεις group by query για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν ομοιόμορφη κατανομή.



Εικόνα 34: Επιδόσεις group by query για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν κανονική κατανομή.



Εικόνα 35: Επιδόσεις group by query για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν εκθετική κατανομή.

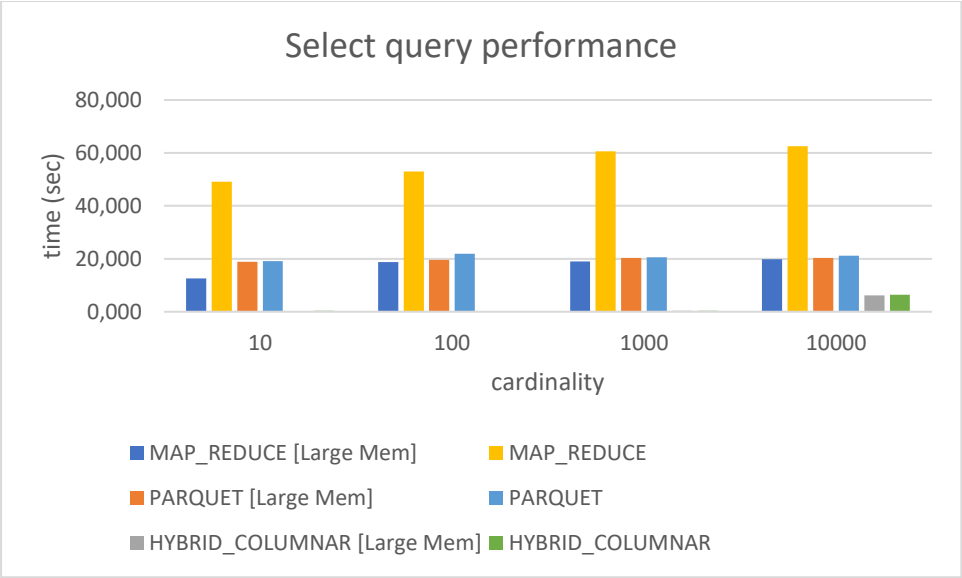


Εικόνα 36: Επιδόσεις group by query για μικρό μέγεθος διαθέσιμης μνήμης, σε δεδομένα που ακολουθούν εκθετική κατανομή.

Στο setup αυτό το hybrid columnar εξακολουθεί να είναι ταχύτερο του parquet. Αυτό που άλλαξε σε σχέση με το προηγούμενο setup είναι πως το parquet πλέον ξεπερνά σε επιδόσεις το ασυμπίεστο map reduce.

Σαν συμπέρασμα προκύπτει πως η χρήση συμπίεσης δεδομένων επιταχύνει σημαντικά την εκτέλεση των queries ακόμα και αν το dataset χωράει στην ασυμπίεστη μορφή του ολόκληρο στην κύρια μνήμη. Στις περιπτώσεις που το dataset δεν χωράει ολόκληρο στην μνήμη, στην ασυμπίεστη μορφή του, η διαφορά στο χρόνο εκτέλεσης σε σχέση με τα συστήματα που χρησιμοποιούν συμπίεση μεγαλώνει περεταίρω.

Αυτό φαίνεται χαρακτηριστικά στο παρακάτω διάγραμμα:



Στο διάγραμμα αυτό συγκρίνουμε τις επιδόσεις ενός select query με selectivity 75% πάνω σε dataset με uniform κατανομή, για 2 διαφορετικά setup, ένα με 4GB και ένα με 800MB διαθέσιμη μνήμη. Παρατηρούμε ότι αν το dataset χωρά στη μνήμη, οι επιδόσεις των

διαφορετικών συστημάτων είναι πανομοιότυπες. Αντίθετα στο ασυμπίεστο dataset υπάρχουν εμφανείς διαφορές ανάμεσα στο setup των 4GB σε σχέση με αυτό των 800MB, όπου το dataset δεν χωρά στη μνήμη.

6.4. Επιδόσεις για πολλαπλές στήλες

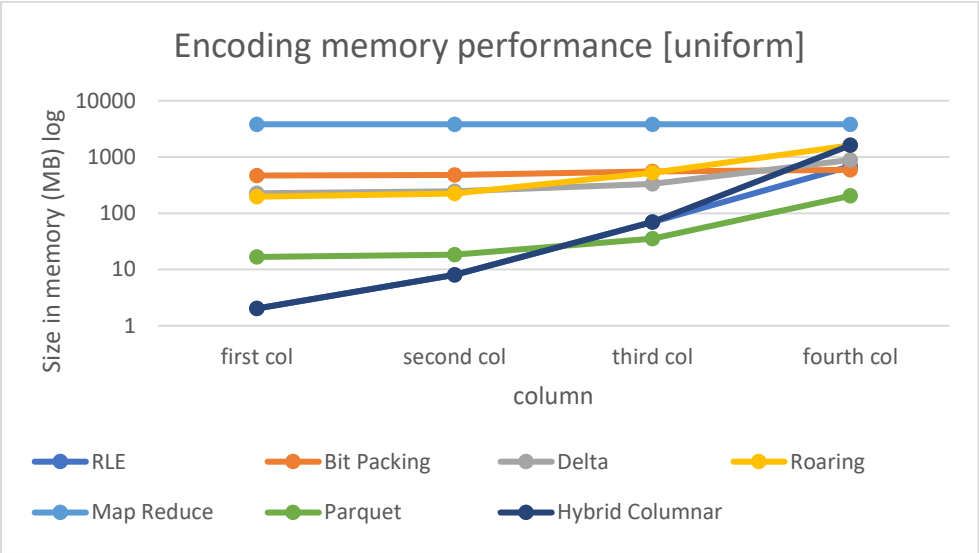
Έχοντας πλέον δει την επίδραση του cardinality και της κατανομής του dataset στις επιδόσεις των συστημάτων, μένει άλλη μια παράμετρος που θα εξετάσουμε. Τα datasets που τρέξαμε μέχρι στιγμής αποτελούνται από μια στήλη. Όμως ένα πραγματικό dataset συνήθως αποτελείται από περισσότερες από μια στήλες. Αυτό αποτελεί μια επιπλέον πρόκληση, καθώς με εξαίρεση την πρώτη στήλη, όλες οι υπόλοιπες στήλες θα εμφανίζουν μια περιοδικότητα στην κατανομή των δεδομένων τους. Για παράδειγμα σε ένα dataset με δυο στήλες, όπου η κάθε μια θα έχει τιμές στο εύρος [0-10] και στο οποίο θα γίνει λεξικογραφική ταξινόμηση, στην πρώτη στήλη θα εμφανίζονται πρώτα όλα τα 0, μετά όλα τα 1 κ.ο.κ. Αντίθετα, στη δεύτερη στήλη, για κάθε ακολουθία μια τιμής στην πρώτη στήλη, θα εμφανίζονται πρώτα τα 0 μετά τα 1 κ.ο.κ. Αυτό θα γίνεται για κάθε ακολουθία τιμών της πρώτης στήλης, δημιουργώντας επαναλήψεις στη δεύτερη στήλη, όπως φαίνεται παρακάτω:

1	1
1	2
…	…
1	10
…	
10	1
10	2
…	…
10	10

Για να εξετάσουμε τη συμπεριφορά των διάφορων τεχνικών συμπίεσης στο φαινόμενο αυτό, για κάθε κατανομή δημιουργήσαμε dataset με 4 στήλες, με όλες τις στήλες να έχουν μεταξύ τους τα ίδια δεδομένα. Σε κάθε περίπτωση το cardinality που επιλέχθηκε ήταν 10. Αυτό δημιουργεί 10*10 = 100 επαναλήψεις στη δεύτερη στήλη, 10*10*10 = 1000 επαναλήψεις στην τρίτη στήλη και 10*10*10*10 = 10.000 επαναλήψεις στην τέταρτη.

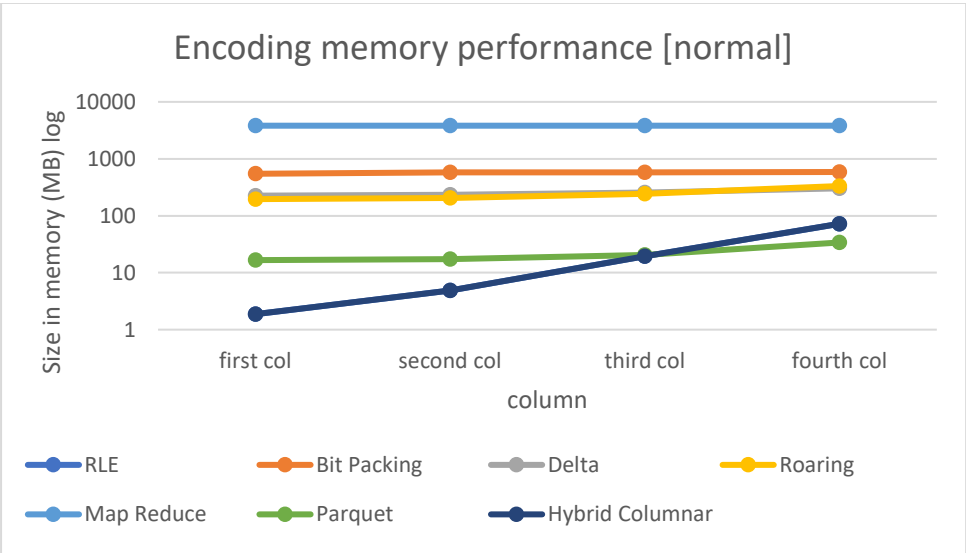
6.4.1. Μέγεθος αποτυπώματος στη μνήμη

Ξεκινάμε με το αποτύπωμα μνήμης για κάθε κατανομή.



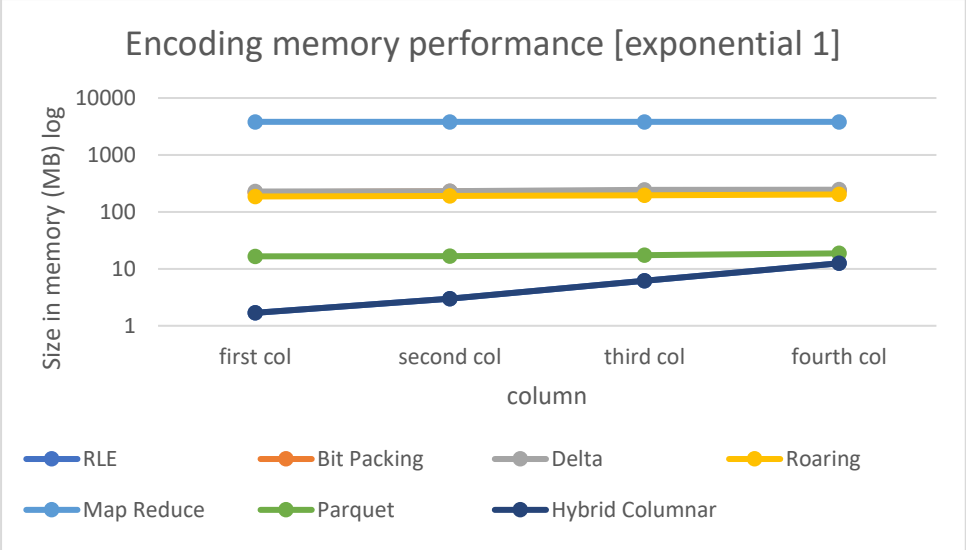
Εικόνα 37: Μέγεθος αποτυπώματος στη μνήμη, ανά στήλη σε δεδομένα που ακολουθούν ομοιόμορφη κατανομή.

Στην ομοιόμορφη κατανομή, όσο προχωράμε στις στήλες, βλέπουμε παρόμοιο pattern με την αύξηση του cardinality, με όλες τις τεχνικές συμπίεσης να επηρεάζονται από τον αριθμό των επαναλήψεων, αλλά με πιο αργό ρυθμό σε σχέση με την αύξηση του cardinality που είδαμε στις προηγούμενες μετρήσεις.

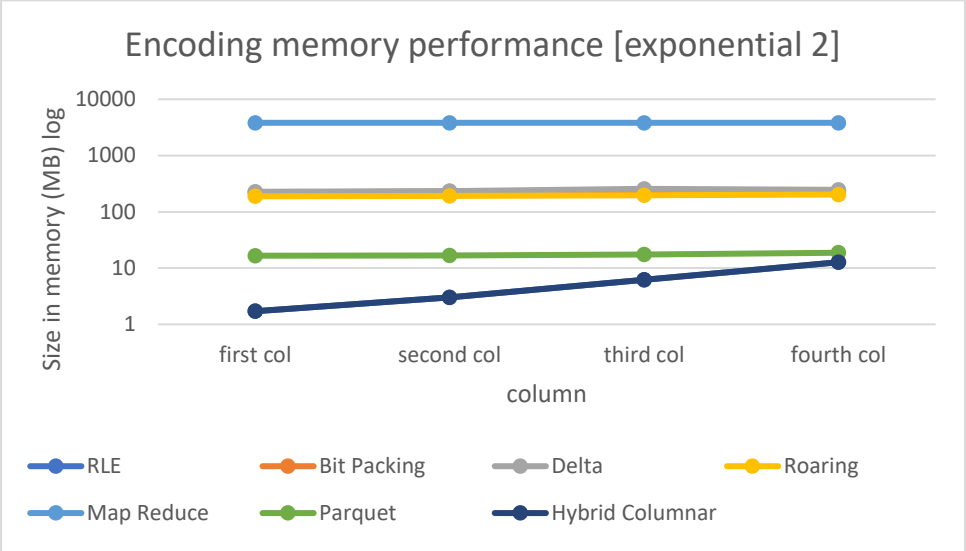


Εικόνα 38: Μέγεθος αποτυπώματος στη μνήμη, ανά στήλη σε δεδομένα που ακολουθούν κανονική κατανομή.

Στην κανονική κατανομή βλέπουμε πως πλέον οι τεχνικές συμπίεσης δεν επηρεάζονται από την αύξηση του αριθμού των επαναλήψεων, με εξαίρεση την RLE που επηρεάζεται, αλλά σε μικρότερο βαθμό σε σχέση με την αύξηση του cardinality.



Εικόνα 39: Μέγεθος αποτυπώματος στη μνήμη, ανά στήλη σε δεδομένα που ακολουθούν εκθετική κατανομή.



Εικόνα 40: Μέγεθος αποτυπώματος στη μνήμη, ανά στήλη σε δεδομένα που ακολουθούν εκθετική κατανομή.

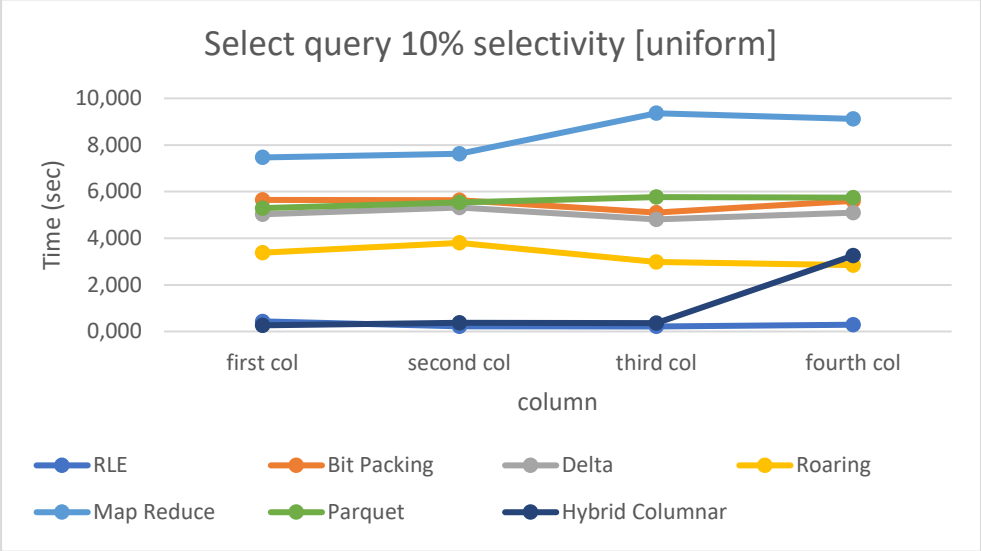
Στις εκθετικές κατανομές βλέπουμε μια ακόμα πιο αργή αύξηση του μεγέθους της μνήμης του RLE σε σχέση με τον αριθμό των επαναλήψεων, κάτι που καθιστά το hybrid columnar αποδοτικότερο από το parquet σε κάθε στήλη. Ενδιαφέρον παρουσιάζει και η συμπεριφορά του parquet, το οποίο καταλαμβάνει το ίδιο μέγεθος στη μνήμη, ανεξαρτήτως στήλης.

6.4.2. Ταχύτητα εκτέλεσης

Ας δούμε τώρα και τους χρόνους που χρειάζεται η κάθε τεχνική.

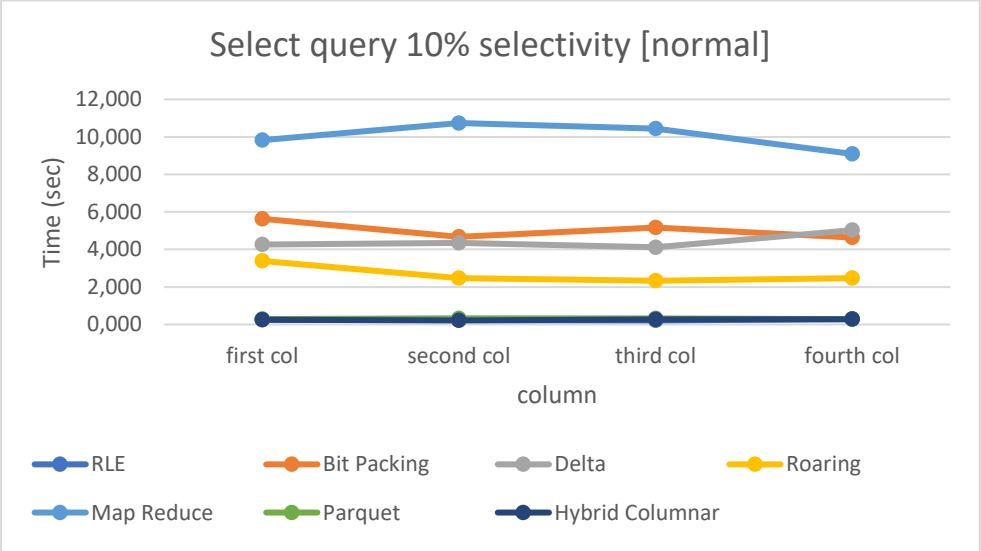
6.4.2.1. Select queries

Αρχικά, εξετάζουμε τη συμπεριφορά των τεχνικών συμπίεσης για μικρό selectivity.



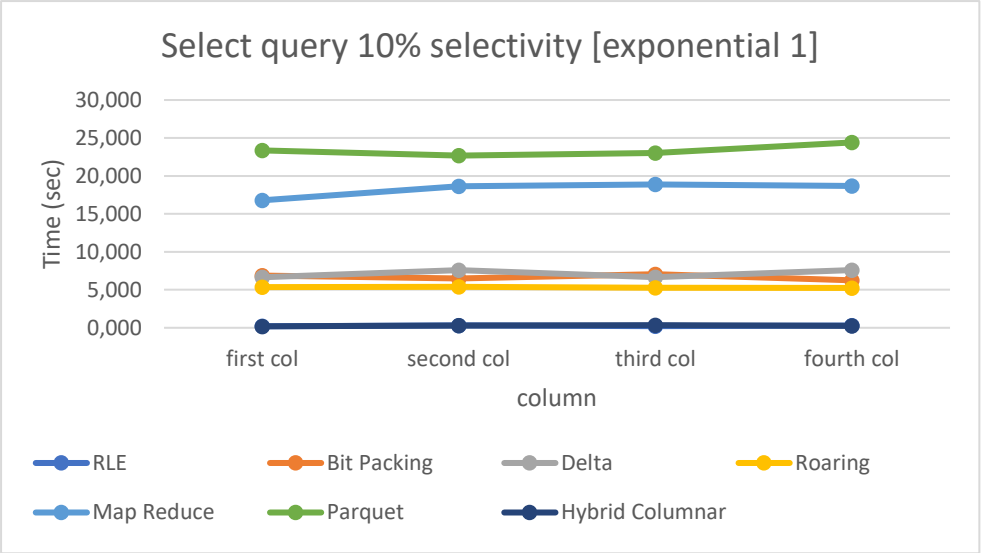
Εικόνα 41: Απόδοση select query για μικρό selectivity, σε σχέση με τον αριθμό της στήλης για ομοιόμορφη κατανομή.

Στην ομοιόμορφη κατανομή μια εμφανής διαφορά σε σχέση με τα queries που μελετούσαν την επίδραση του cardinality, είναι οι επιδόσεις του ROARING, το οποίο συμπεριφέρεται πολύ καλύτερα στον αυξημένο αριθμό επαναλήψεων, ξεπερνώντας όλες τις υπόλοιπες τεχνικές, εκτός του RLE. Σε κάθε περίπτωση το hybrid columnar είναι κατά πολύ ταχύτερο του parquet.

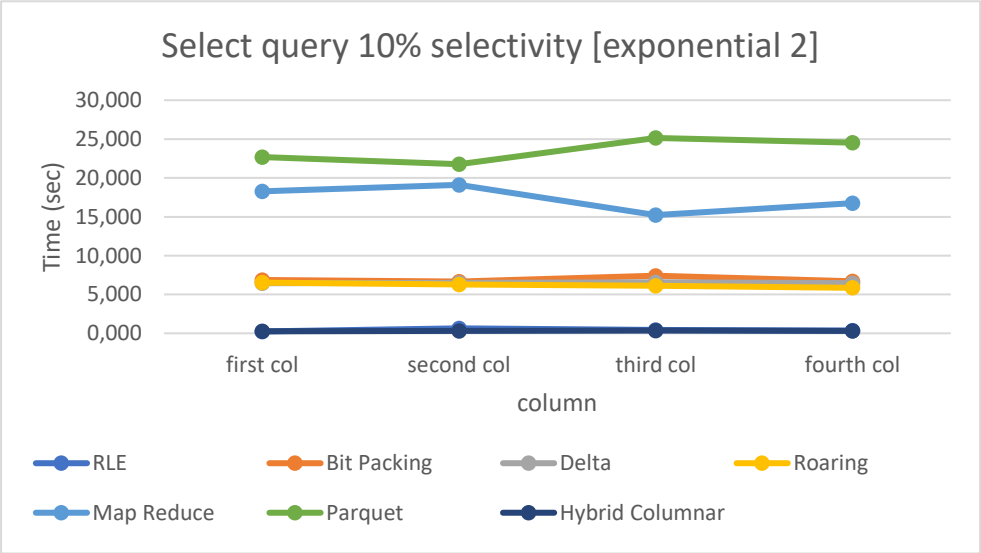


Εικόνα 42: Απόδοση select query για μικρό selectivity, σε σχέση με τον αριθμό της στήλης για κανονική κατανομή.

Η μόνη διαφορά εδώ, σε σύγκριση με την αύξηση του cardinality είναι πως το hybrid columnar και το parquet εμφανίζουν παρόμοια συμπεριφορά σε κάθε περίπτωση.



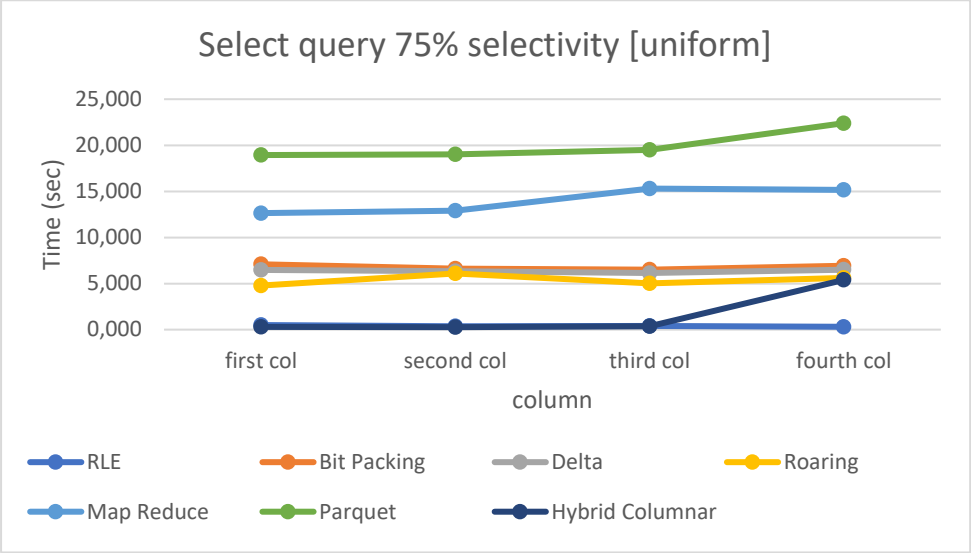
Εικόνα 43: Απόδοση select query για μικρό selectivity, σε σχέση με τον αριθμό της στήλης για εκθετική κατανομή.



Εικόνα 44: Απόδοση select query για μικρό selectivity, σε σχέση με τον αριθμό της στήλης για εκθετική κατανομή.

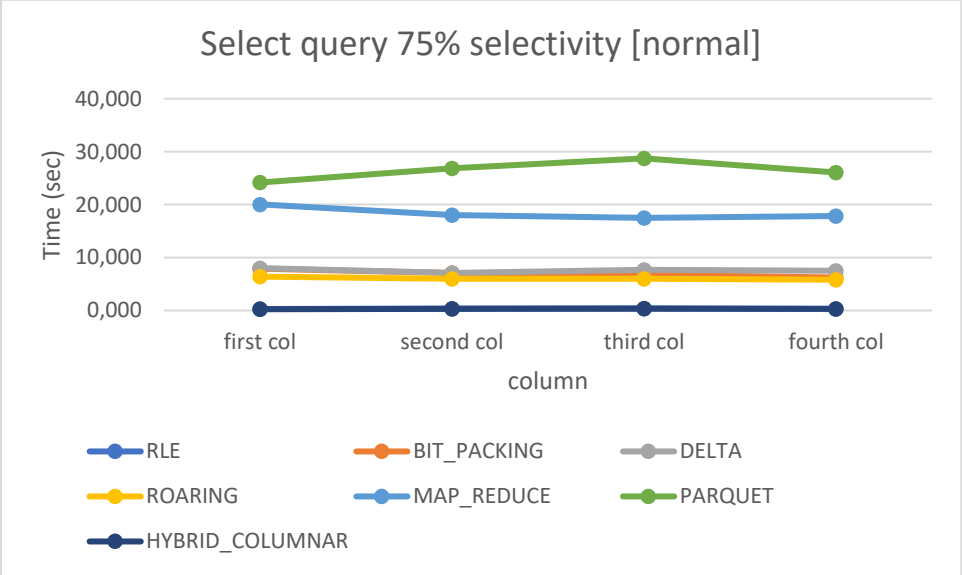
Στις εκθετικές κατανομές η συμπεριφορά των τεχνικών συμπίεσης είναι ακριβώς η ίδια με αυτή των αντίστοιχων queries που μελετούν το cardinality. Συγκεκριμένα, το parquet εμφανίζει χειρότερες επιδόσεις και από το ασυμπίεστο map reduce, ενώ το hybrid columnar είναι τάξεις μεγέθους ταχύτερο και από τα δυο.

Ας δούμε τώρα τη συμπεριφορά των τεχνικών και για μεγαλύτερο selectivity:



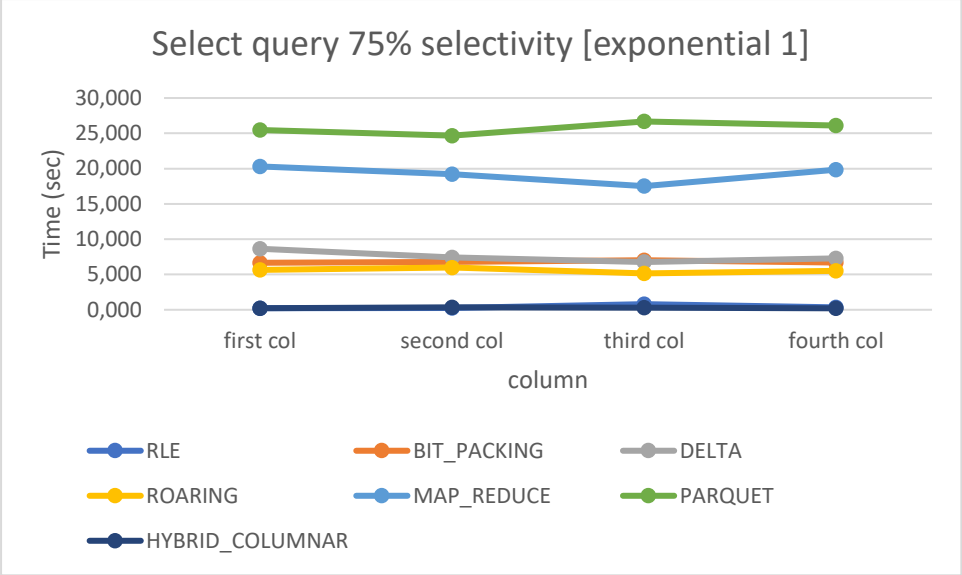
Εικόνα 45: Απόδοση select query για μεγάλο selectivity, σε σχέση με τον αριθμό της στήλης για ομοιόμορφη κατανομή.

Τα αποτελέσματα είναι τα ίδια με τα προηγούμενα queries για το cardinality, με τη μόνη διαφορά τις καλύτερες επιδόσεις του ROARING.

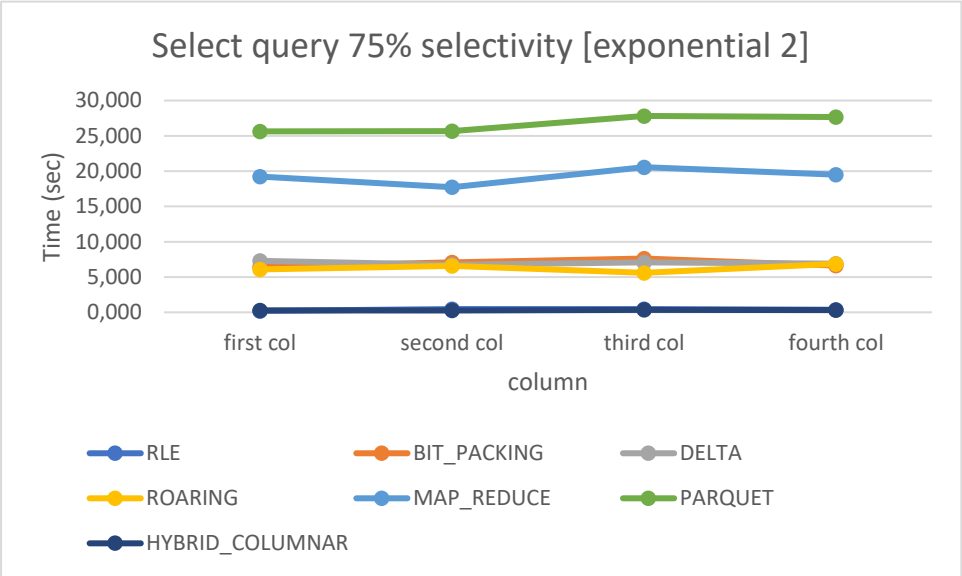


Εικόνα 46: Απόδοση select query για μεγάλο selectivity, σε σχέση με τον αριθμό της στήλης για κανονική κατανομή.

Στην κανονική κατανομή το hybrid columnar επιλέγει πάντα την RLE κωδικοποίηση και έτσι δεν κάνει το χαρακτηριστικό σκαλοπάτι που εμφανίζεται στο μεγαλύτερο cardinality.



Εικόνα 47: Απόδοση select query για μεγάλο selectivity, σε σχέση με τον αριθμό της στήλης για εκθετική κατανομή.

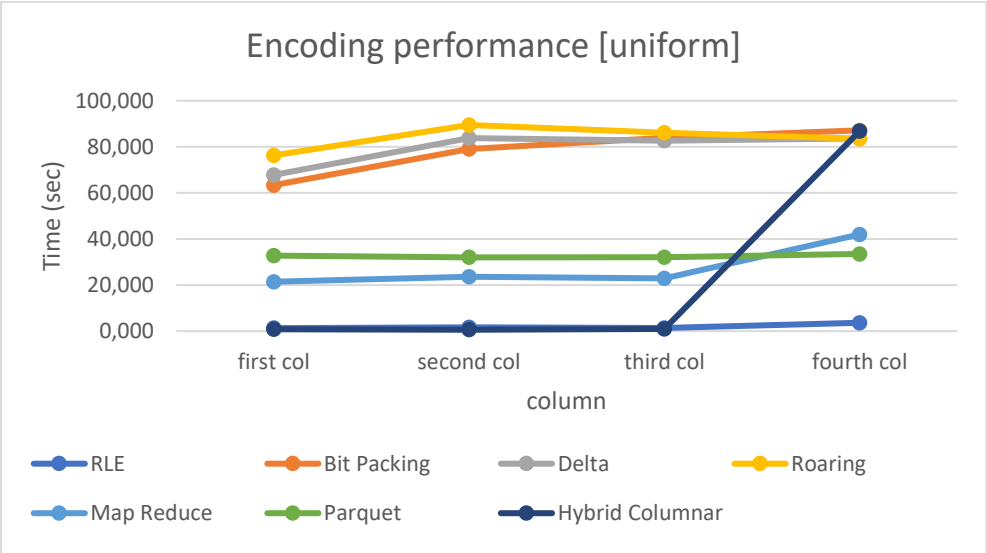


Εικόνα 48: Απόδοση select query για μεγάλο selectivity, σε σχέση με τον αριθμό της στήλης για εκθετική κατανομή.

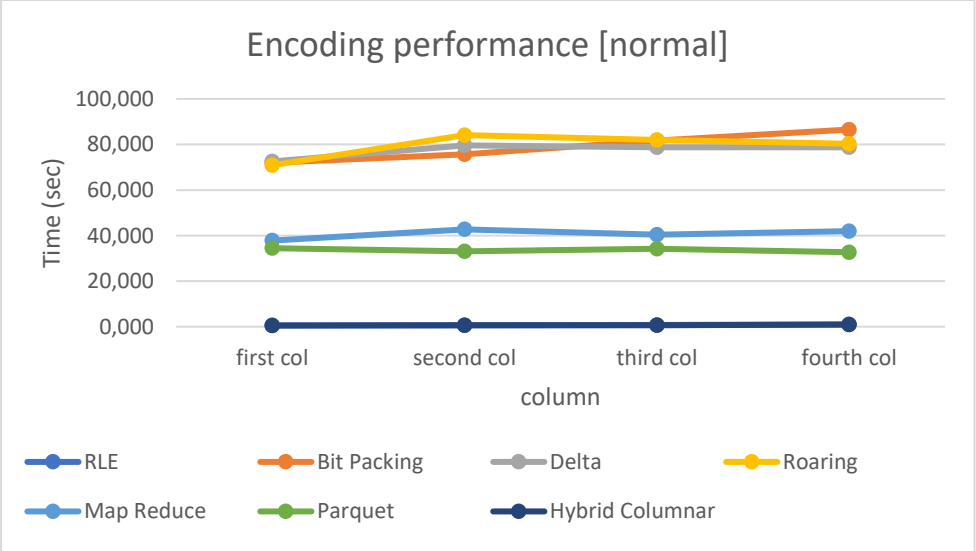
Στις εκθετικές κατανομές οι επιδόσεις είναι ίδιες με τα queries τα σχετικά με το cardinality.

6.4.2.2. Group by queries

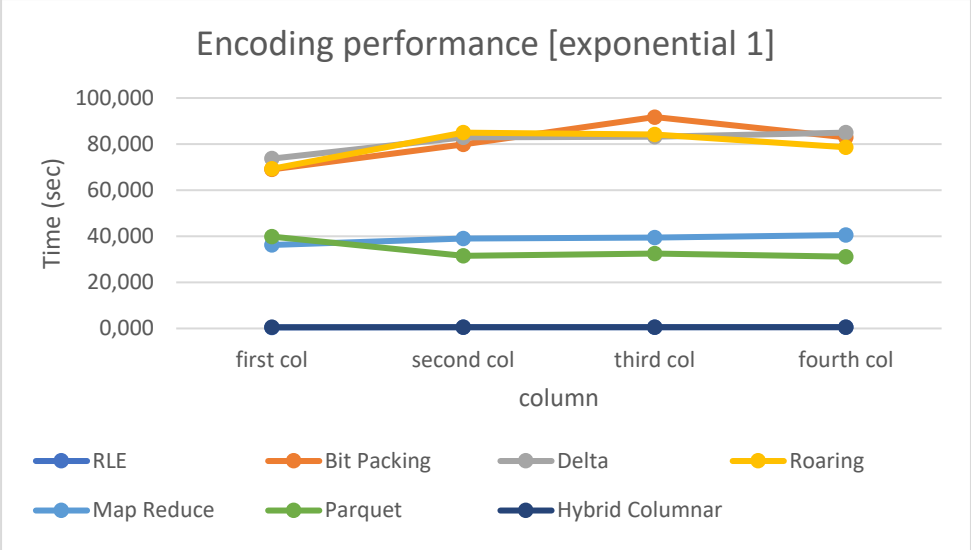
Τέλος, ας δούμε πως συμπεριφέρονται οι τεχνικές στα group by queries.



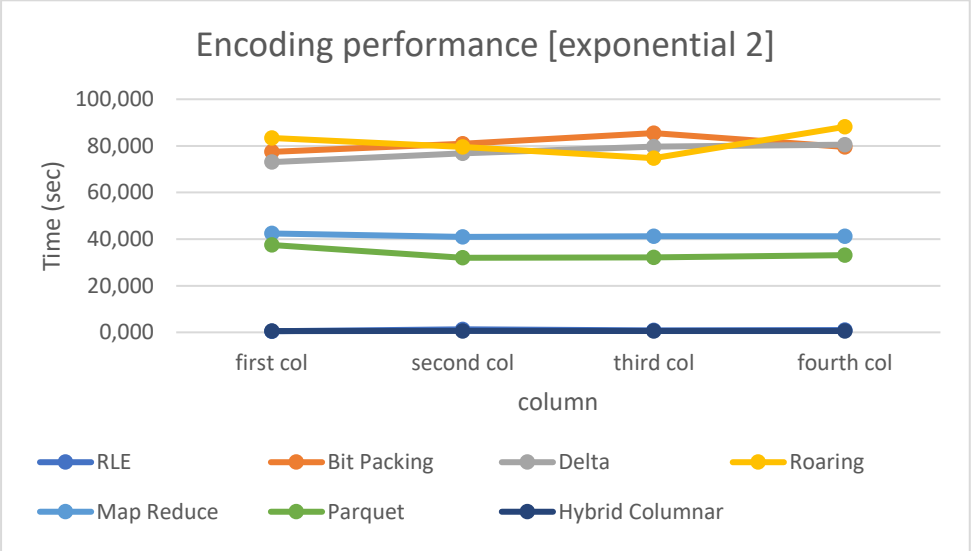
Εικόνα 49: Απόδοση group by query, σε σχέση με τον αριθμό της στήλης για ομοιόμορφη κατανομή.



Εικόνα 50: Απόδοση group by query, σε σχέση με τον αριθμό της στήλης για κανονική κατανομή.



Εικόνα 51: Απόδοση group by query, σε σχέση με τον αριθμό της στήλης για εκθετική κατανομή.



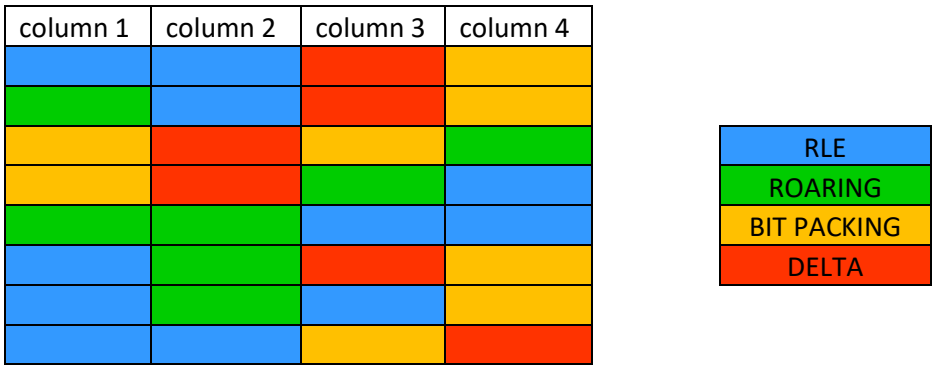
Εικόνα 52: Απόδοση group by query, σε σχέση με τον αριθμό της στήλης για εκθετική κατανομή.

Η μόνη διαφορά σε σχέση με τα queries σχετικά με το cardinality είναι η έλλειψη σκαλοπατιού στο hybrid columnar στην κανονική κατανομή.

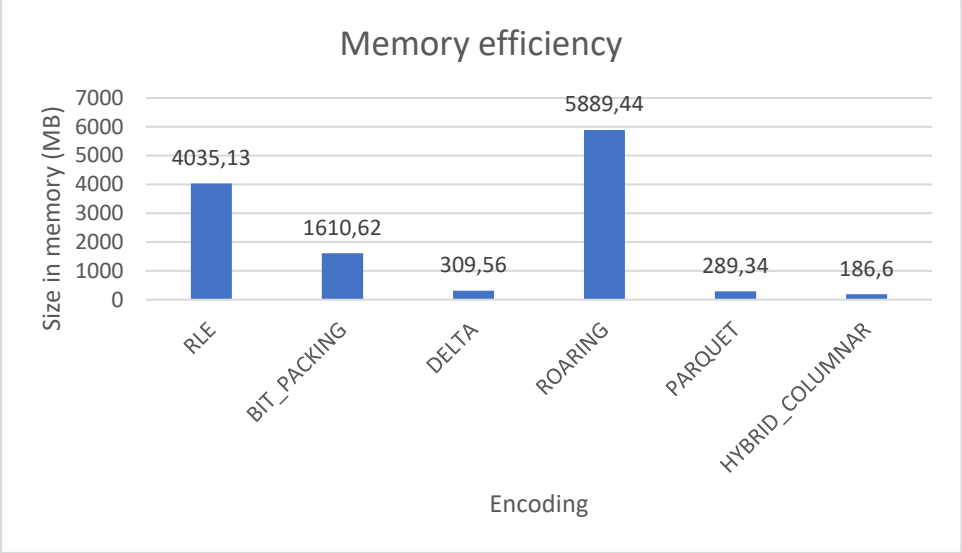
6.5. Επιδόσεις για πολλαπλά cardinalities

Γνωρίζουμε ότι το hybrid columnar, όπως και το parquet έχει την ικανότητα να επιλέγει διαφορετική μέθοδο συμπίεσης ανά στήλη. Αυτό είναι ιδιαίτερα χρήσιμο, καθώς κάθε στήλη του dataset μπορεί να ακολουθεί διαφορετική κατανομή και γενικά να έχει διαφορετικά χαρακτηριστικά από τις υπόλοιπες. Επιπλέον και τα δυο αυτά συστήματα έχουν τη δυνατότητα να συμπιέζουν διαφορετικά κομμάτια της ίδιας στήλης με διαφορετική μέθοδο. Το χαρακτηριστικό αυτό είναι ιδιαίτερα χρήσιμο όταν τα στοιχεία μιας στήλης δεν είναι ομοιόμορφα κατανεμημένα, για παράδειγμα όταν ένα μεγάλο πλήθος διαφορετικών στοιχείων είναι συγκεντρωμένο σε ένα σημείο της στήλης, ενώ σε ένα άλλα σημείο να είναι συγκεντρωμένο μεγάλο πλήθος ίδιων στοιχείων.

Το χαρακτηριστικό αυτό φαίνεται παρακάτω:



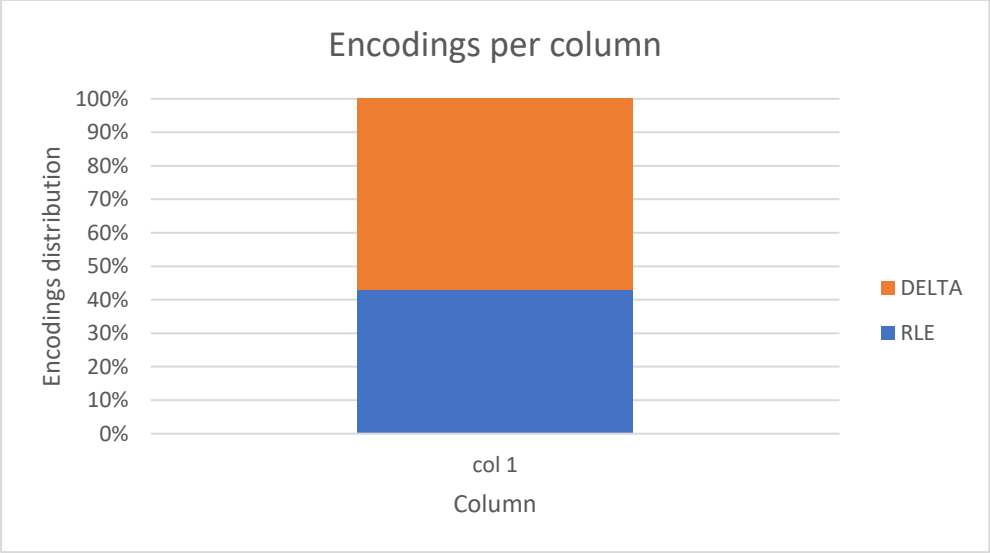
Για να εξετάσουμε την συμπεριφορά των συστημάτων δημιουργήσαμε ένα dataset που ακολουθεί κανονική κατανομή, χρησιμοποιώντας διαφορετικά cardinalities για τα διάφορα τμήματα. Συγκεκριμένα για το πρώτο 20% έχουμε 10 διαφορετικά στοιχεία και για τα υπόλοιπα 100, 1000 και 10000 αντίστοιχα. Στη συνέχεια θελήσαμε να μετρήσουμε το χώρο που καταλαμβάνουν οι διάφορες τεχνικές συμπίεσης στη μνήμη, ώστε να αξιολογήσουμε αν όντως με τη χρήση διαφορετικών τεχνικών ανά κομμάτι στη στήλης πετυχαίνουμε καλύτερη συμπίεση δεδομένων.



Εικόνα 53: Μέγεθος αποτυπώματος στη μνήμη, ανά τεχνική συμπίεσης.

Στο διάγραμμα αυτό φαίνεται ο χώρος που χρειάζεται αν συμπιέσουμε ολόκληρη τη στήλη με μια συγκεκριμένη τεχνική όπως DELTA ή RLE και αντίστοιχα ο χώρος που χρειάζεται το parquet και το hybrid columnar τα οποία κάνουν χρήση της τεχνικής αυτής. Παρατηρούμε πως πράγματι με την τεχνική αυτή βελτιώνουμε σε μεγάλο βαθμό τη δυνατότητα συμπίεσης του hybrid columnar.

Παρακάτω φαίνεται η κατανομή των τεχνικών συμπίεσης που χρησιμοποίησε το hybrid columnar στο συγκεκριμένο dataset.



Εικόνα 54: Κατανομή των τεχνικών συμπίεσης σε μια στήλη.

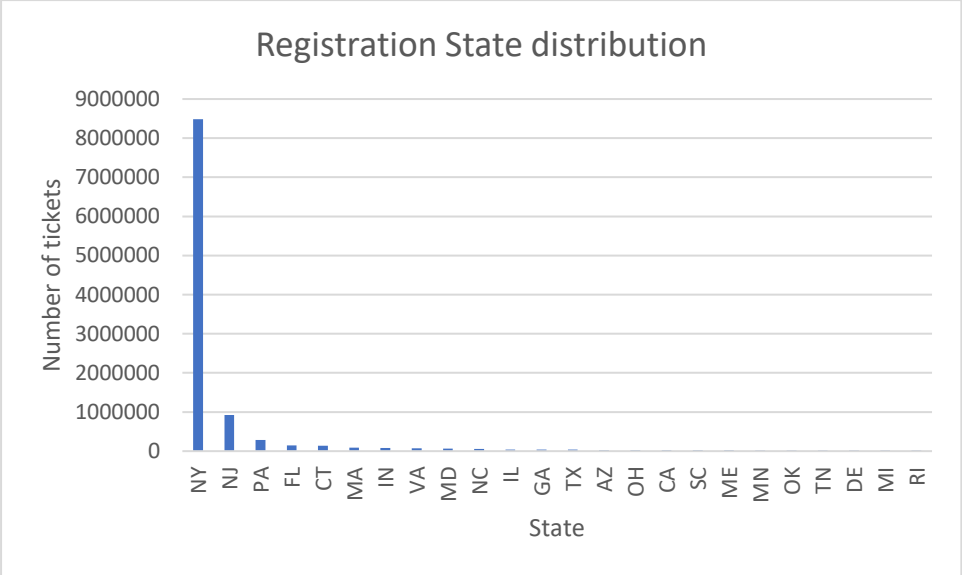
6.6. Αξιολόγηση σε πραγματικό dataset

Έχοντας δει τη συμπεριφορά των συστημάτων και των επιμέρους τεχνικών συμπίεσης σε διαφορετικές κατανομές, cardinalities και αριθμό επαναλήψεων, θα κλείσουμε το πειραματικό κομμάτι με μετρήσεις πάνω σε ένα πραγματικό dataset. Το dataset αυτό αφορά κλήσεις της τροχαίας κατά το έτος 2017, στη Νέα Υόρκη. Το dataset αυτό παρουσιάζει ενδιαφέρον, καθώς περιλαμβάνει πολλές στήλες, όπου η κάθε μια έχει διαφορετικό πλήθος μοναδικών τιμών, κατανομή και τύπο δεδομένων. Το dataset αυτό περιλαμβάνει 43 στήλες, από τις οποίες θα επεξεργαστούμε 5 με βάση το ενδιαφέρον που παρουσιάζουν.

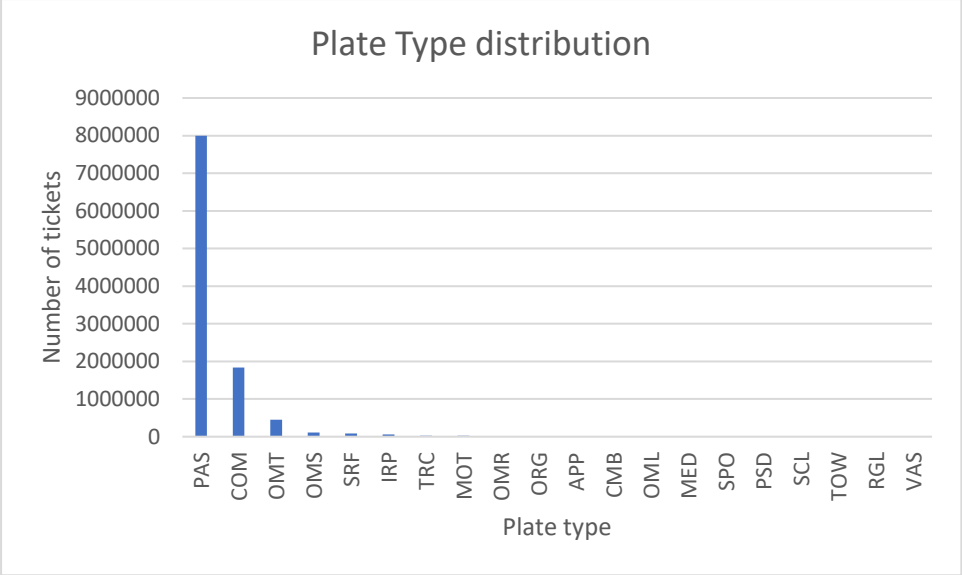
Οι στήλες που επιλέξαμε είναι οι Registration State, Plate Type, Issue Date, Vehicle Body Type και Vehicle Make. Παρακάτω φαίνονται τα χαρακτηριστικά τους:

Column	Unique values	Representative values
Registration State	67	NY 79%, NJ 9%, PA 3%, FL 1%
Plate Type	86	PAS 74%, COM 17%, OMT 4%, OMS 1%
Issue Date	2063	17/12/2016 4%, 7/10/2016 0.62%
Vehicle Body Type	1849	SUBN 34%, 4DSD 29%, VAN 13%, DELV 6%
Vehicle Make	5704	FORD 12%, TOYOT 11%, HONDA 10%

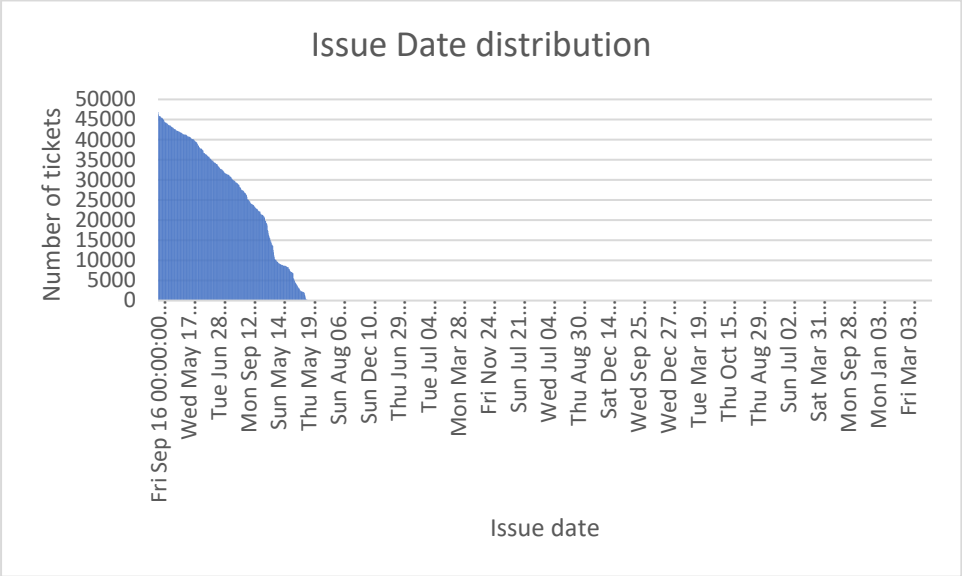
Από το ποσοστό που εκπροσωπούν οι αντιπροσωπευτικές τιμές της κάθε στήλης, συμπεραίνουμε πως κάποιες από τις στήλες θα ακολουθούν εκθετική κατανομή. Παρακάτω παρουσιάζονται οι κατανομές τους:



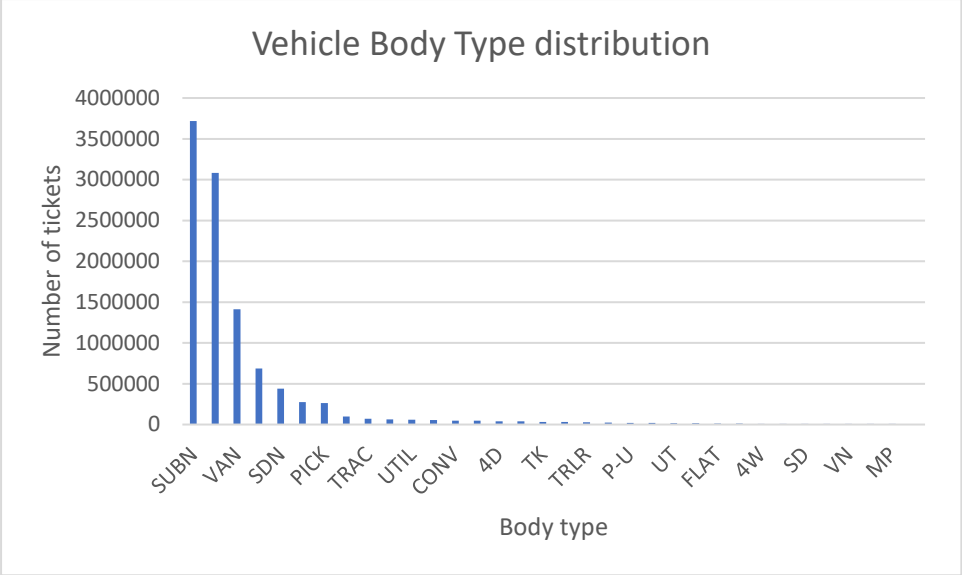
Εικόνα 55: Αριθμός κλήσεων ανά πολιτεία.



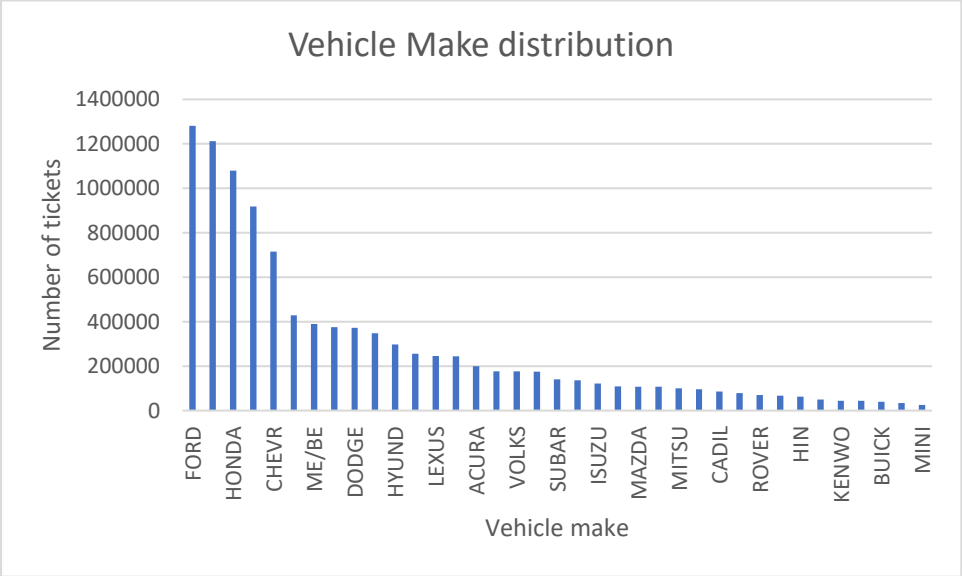
Εικόνα 56: Αριθμός κλήσεων ανά τύπο άδειας αυτοκινήτου.



Εικόνα 57: Αριθμός κλήσεων ανά ημερομηνία.



Εικόνα 58: Αριθμός κλήσεων ανά τύπο αμαξώματος.

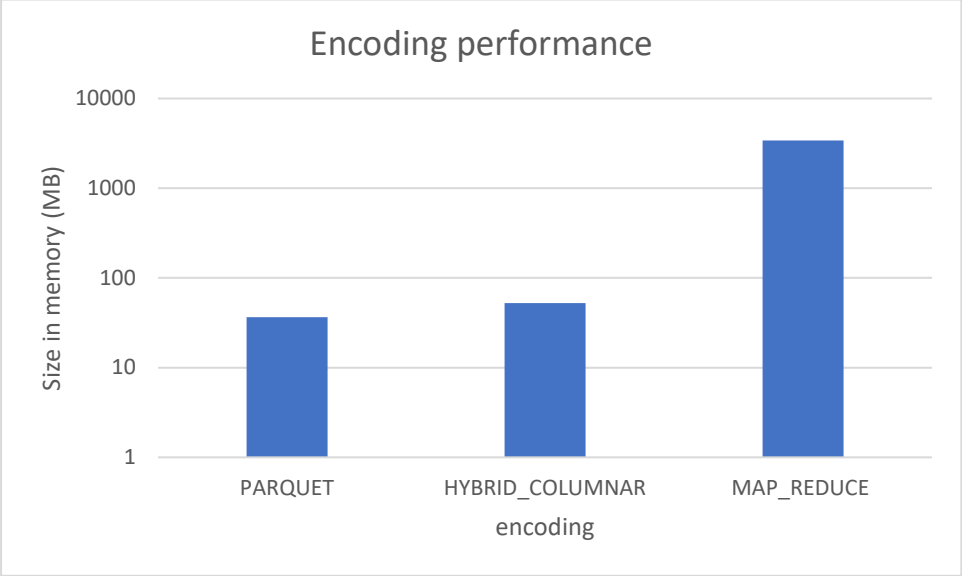


Εικόνα 59: Αριθμός κλήσεων ανά μάρκα αυτοκινήτου.

Βλέπουμε ότι οι κατανομές των στηλών είναι ιδιαίτερα skewed, σε παρόμοιο βαθμό με την εκθετική κατανομή.

6.6.1. Μέγεθος αποτυπώματος στη μνήμη

Έχοντας δει τα χαρακτηριστικά του dataset, περνάμε στα πειράματα. Αρχικά μετράμε το αποτύπωμα του κάθε συστήματος στη μνήμη:



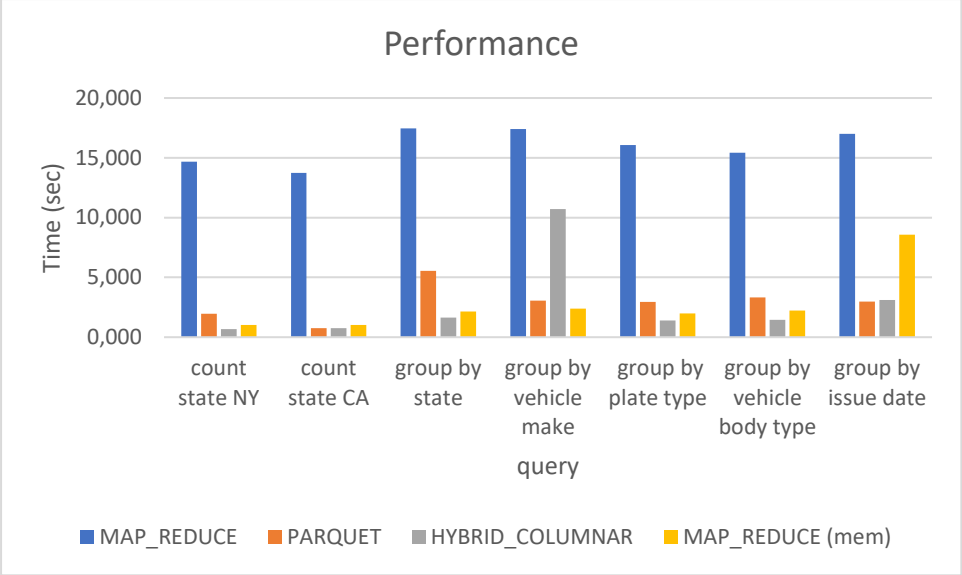
Εικόνα 60: Μέγεθος δεδομένων στη μνήμη ανά τεχνική.

Βλέπουμε το hybrid columnar και το parquet να έχουν συγκρίσιμες επιδόσεις, καταλαμβάνοντας και τα δυο τάξεις μεγέθους λιγότερη μνήμη από το ασυμπίεστο map reduce.

6.6.2. Ταχύτητα εκτέλεσης

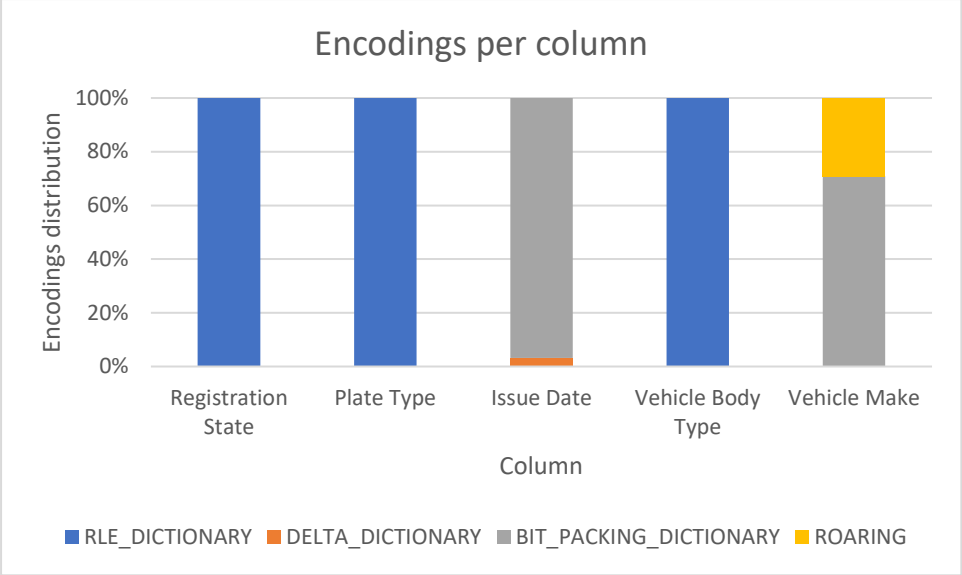
Περνώντας στους χρόνους, τρέξαμε 7 queries εκ των οποίων ένα select με χαμηλό selectivity, ένα select με υψηλό selectivity και 5 group by queries, ένα για κάθε στήλη. Τρέξαμε τα queries σε hybrid columnar, parquet και ασυμπίεστο dataset. Για το ασυμπίεστο dataset τρέξαμε queries τόσο με το dataset αποθηκευμένο στη μνήμη όσο και κατευθείαν από το δίσκο. Στόχος ήταν να δούμε κατά πόσο αξίζει η χρήση συμπίεσης δεδομένων, όταν το ασυμπίεστο dataset χωρά στην κύρια μνήμη.

Τα αποτελέσματα παρουσιάζονται παρακάτω:



Εικόνα 61: Επιδόσεις των τεχνικών για ένα σύνολο ερωτημάτων.

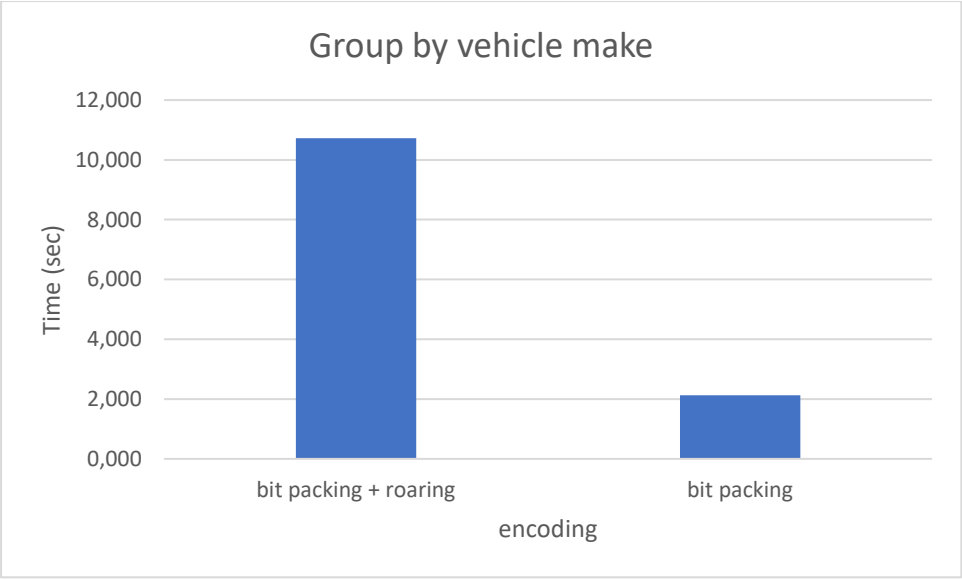
Από τα αποτελέσματα των μετρήσεων παρατηρούμε ότι εφόσον χωράει ολόκληρο στην κύρια μνήμη, το ασυμπίεστο dataset πετυχαίνει επιδόσεις συγκρίσιμες με το parquet και το hybrid columnar, σε όλα τα queries εκτός του group by date. Αν το dataset δεν χωρά στην κύρια μνήμη και αναγκαζόμαστε να το τρέξουμε από το δίσκο, οι χρόνοι του είναι μια τάξη μεγέθους μεγαλύτεροι σε κάθε query. Αυτό είναι και το μεγαλύτερο πλεονέκτημα των συστημάτων που χρησιμοποιούν συμπίεση δεδομένων, καθώς ακόμα και για πολύ μεγάλα dataset αποκρίνονται εντός λίγων δευτερολέπτων, επιτρέποντας έτσι στον χρήστη να εκτελεί διαδραστικά queries χωρίς να χάνει τον ρυθμό της εργασίας του, περιμένοντας τα αποτελέσματα. Συγκρίνοντας τα δυο συστήματα συμπίεσης μεταξύ τους, βλέπουμε πως στα περισσότερα queries οι επιδόσεις του hybrid columnar είναι στα ίδια επίπεδα ή και αρκετά καλύτερες σε κάποιες περιπτώσεις, από αυτές του parquet. Παρατηρούμε όμως ότι στο group by vehicle make query, οι επιδόσεις του hybrid columnar υστερούν κατά πολύ από αυτές του parquet αλλά και του ασυμπίεστου. Για να δούμε γιατί συμβαίνει αυτό, ας δούμε τι επιλογές κάνει το hybrid columnar όσον αφορά τις τεχνικές συμπίεσης για την κάθε στήλη.



Εικόνα 62: Κατανομή χρησιμοποιούμενων τεχνικών συμπίεσης ανά στήλη.

Βλέπουμε ότι στη στήλη vehicle make, οι τεχνικές που επιλέγονται είναι οι bit packing και Roaring. Δεδομένου ότι bit packing χρησιμοποιεί και η στήλη issue date, υποπτευόμαστε ότι η αιτία της καθυστέρησης είναι η επιλογή του roaring. Για να το επιβεβαιώσουμε τρέχουμε το ίδιο query, έχοντας όμως επιλέξει συμπίεση bit packing για τη στήλη vehicle make.

Τα αποτελέσματα φαίνονται παρακάτω:



Εικόνα 63: Επιδόσεις hybrid columnar με αυτόματη επιλογή συμπίεσης και με επιλογή αποκλειστικά bit packing για τη στήλη vehicle make.

Βλέπουμε ότι οι χαμηλές επιδόσεις του hybrid columnar στο συγκεκριμένο query πράγματι οφείλονται στην επιλογή του roaring σε ορισμένα chunks της στήλης vehicle make. Το συμπέρασμα είναι πως η συγκεκριμένη τεχνική συμπίεσης δεν ενδείκνυται για μεγάλα cardinalities, όπως άλλωστε και τα bitmaps γενικότερα. Αυτό συμβαίνει επειδή για κάθε γραμμή που διαβάζουμε πρέπει να διατρέχουμε όλες τις bitmap στήλες και να ελέγχουμε αν η τιμή στη συγκεκριμένη γραμμή είναι 1, οπότε είναι και το στοιχείο που ψάχνουμε.

Λόγω αυτής της ιδιαιτερότητας των bitmaps, σε περίπτωση που έχουμε μεγάλα cardinalities, τεχνικές συμπίεσης όπως το bit packing είναι προτιμότερες.

7. Συμπεράσματα

Σκοπός της παρούσας διπλωματικής ήταν να διερευνήσουμε κατά πόσο είναι εφικτό, κάνοντας χρήση τεχνικών συμπίεσης, να πετύχουμε υψηλότερες επιδόσεις στην ανάλυση δεδομένων, σε σχέση με τη χρήση map reduce πάνω σε ασυμπίεστα δεδομένα. Πράγματι, το σύστημα που σχεδιάσαμε προκειμένου να υλοποιήσουμε το σύνολο των τεχνικών συμπίεσης που μελετήσαμε πέτυχε επιδόσεις τάξεις μεγέθους καλύτερες από το ασυμπίεστο map reduce. Ενδεικτικά σε ερωτήματα group by με μικρό cardinality οι χρόνοι εκτέλεσης για map reduce και hybrid columnar ήταν **42,407** sec και **0,540** sec αντίστοιχα, μια βελτίωση κοντά στο **80X**. Αυτό όμως που δεν περιμέναμε είναι πως σε πολλές περιπτώσεις το σύστημα που σχεδιάσαμε ήταν πολύ πιο γρήγορο και από το Parquet, το οποίο κάνει επίσης χρήση συμπίεσης. Σε queries με μεγάλο selectivity και cardinality 10000 οι χρόνοι για parquet και hybrid columnar ήταν **26,200** sec και **0,416** sec αντίστοιχα.

Από τα αποτελέσματα των πειραμάτων προκύπτει το συμπέρασμα πως με τη χρήση συμπίεσης δεδομένων μπορούμε να σχεδιάσουμε ένα σύστημα, το οποίο αποκρίνεται σε τάξεις μεγέθους λιγότερο χρόνο από τα συστήματα που επεξεργάζονται ασυμπίεστα δεδομένα. Επιπλέον, σε πολλές περιπτώσεις το σύστημα που σχεδιάσαμε πέτυχε πολύ καλύτερους χρόνους και από το parquet, ίσως το χαρακτηριστικότερο και ευρέως χρησιμοποιούμενο σύστημα συμπίεσης δεδομένων στο Spark. Αυτό μας έδειξε πως υπάρχουν αρκετές βελτιώσεις που μπορεί να γίνουν σε εμπορικά συστήματα και να οδηγήσουν σε ακόμα ταχύτερα και αποδοτικότερα συστήματα ανάλυσης δεδομένων. Αρκετές βελτιώσεις όμως μπορούν να γίνουν και στο σύστημα που σχεδιάσαμε.

8. Μελλοντικές επεκτάσεις – βελτιώσεις

Μια αλλαγή που θα μπορούσαμε να κάνουμε αφορά τη βελτίωση της συμπεριφοράς του hybrid columnar σε datasets με μεγάλο πλήθος διαφορετικών τιμών. Αυτό θα μπορούσε να επιτευχθεί με βελτίωση του RLE, ώστε να ανταποκρίνεται καλύτερα σε μικρά run lengths, ίσως κάνοντας χρήση μικρότερων ακεραίων στο αντίστοιχο πεδίο του RLE triple, το οποίο αποθηκεύει τα δεδομένα στη μορφή (index, run length, value). Επιπροσθέτως θα μπορούσαν να συνδυαστούν μεταξύ τους διαφορετικές τεχνικές, όπως το RLE με το bit packing, ώστε στο RLE triple να χρησιμοποιούμε ακριβώς όσα bit χρειαζόμαστε. Μια πρόσθετη αλλαγή θα μπορούσε να είναι ο περιορισμός του Roaring σε μικρά σχετικά cardinalities, καθώς από ένα σημείο και μετά, παρότι παραμένει memory efficient, δεν είναι αρκετά time efficient. Θα μπορούσαμε επίσης αντί να επιλέγουμε μέθοδο συμπίεσης αποκλειστικά με κριτήριο το μικρότερο αποτύπωμα στη μνήμη, να λαμβάνουμε υπόψη και το χρόνο εκτέλεσης της εκάστοτε τεχνικής. Μια άλλη βελτίωση, η οποία υπάρχει και στο parquet, είναι να κρατάμε το εύρος τιμών που περιλαμβάνονται το κάθε chunk και αν αυτό δεν περιλαμβάνεται στο query που τρέχουμε να αγνοούμε το συγκεκριμένο chunk, αποφεύγοντας έτσι να κάνουμε άσκοπους υπολογισμούς. Επιπλέον, παρότι οι πειραματισμοί μας για την εύρεση της βέλτιστης τεχνικής συμπίεσης με χρήση μηχανικής μάθησης (ML) δεν έδωσαν τα αναμενόμενα αποτελέσματα, σε σχέση με τη brute force προσέγγιση της δοκιμής όλων των τεχνικών συμπίεσης, θα μπορούσε να γίνει επιπλέον μελέτη πάνω σε αυτό.

9. Βιβλιογραφία

[1] John Gantz and David Reinsel. THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. December 2012

[2] David Reinsel – John Gantz – John Rydning. The Digitization of the World From Edge to Core. November 2018

[3] Data never sleeps. https://www.domo.com/learn/data-never-sleeps-7

[4] James Manyika Michael Chui Brad Brown Jacques Bughin Richard Dobbs Charles Roxburgh Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. McKinsey Global Institute

[5] Carlos A. Gomez-Uribe and Neil Hunt. 2015. The Netflix recommender system: Algorithms, business value, and innovation. ACM Trans. Manage. Inf. Syst. 6, 4, Article 13 (December 2015)

[6] Netflix Recommendations: Beyond the 5 stars. https://netflixtechblog.com/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429

[7] GORDON E. MOORE. Cramming more components onto integrated circuits. Electronics, Volume 38, Number 8, April 19, 1965

[8] Intel's Core i7 870 & i5 750. https://www.anandtech.com/show/2832

[9] Intel® Xeon® Processor X7560 https://ark.intel.com/content/www/us/en/ark/products/46499/intel-xeon-processor-x7560-24m-cache-2-26-ghz-6-40-gt-s-intel-qpi.html

[10] AMD EPYC™ 7H12. https://www.amd.com/en/products/cpu/amd-epyc-7h12

[11] Google Iowa data center. https://www.google.com/about/datacenters/gallery/

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004), pp. 137-150

[14] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler. MSST '10: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)May 2010 Pages 1–10

[15] MATEI ZAHARIA, REYNOLD S. XIN, PATRICK WENDELL, TATHAGATA DAS, MICHAEL ARMBRUST, ANKUR DAVE, XIANGRUI MENG, JOSH ROSEN, SHIVARAM VENKATARAMAN, MICHAEL J. FRANKLIN, ALI GHODSI, JOSEPH GONZALEZ, SCOTT SHENKER, AND ION STOICA. Apache Spark: A Unified Engine for Big Data Processing. Communications of the ACM October 2016

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

[17] Spark Architecture and Deployment Environment. https://medium.com/@goyalsaurabh66/spark-architecture-and-deployment-f713ac031a88

[18] RDD Programming Guide. https://spark.apache.org/docs/latest/rdd-programming-guide.html

[19] John L. Hennessy, David Patterson. Computer Architecture: A Quantitative Approach

[20] Memory hierarchy. https://www.cs.swarthmore.edu/~kwebb/cs31/f18/memhierarchy/mem_hierarchy.html

[21] Karlheinz Brandenburg. MP3 AND AAC EXPLAINED. Fraunhofer Institute for Integrated Circuits FhG-IIS A, Erlangen, Germany. The *Audio* Engineering Society *17th International Conference*: *High-Quality Audio Coding*, 2-5 September 1999

[22] D. Lemire1 , G. Ssi-Yan-Kai2 , O. Kaser3. Consistently faster and smaller compressed bitmaps with Roaring. Software: Practice and Experience Volume 46, Issue 11, pages 1547-1569, November 2016

[23] Apache Parquet. https://parquet.apache.org/

[24] Apache Impala. https://impala.apache.org/

[25] Daniel J. Abadi, Samuel R. Madden, Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really?. SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada

[26] Chunbin Lin, Jianguo Wang, Yannis Papakonstantinou. Data Compression for Analytics over Large-scale In-memory Column Databases. June 2016

[27] Best practices for successfully managing memory for Apache Spark applications on Amazon EMR. https://aws.amazon.com/blogs/big-data/best-practices-for-successfully-managing-memory-for-apache-spark-applications-on-amazon-emr/

[28] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. Eurosys '13, 15-17 April 2013, Praque, Czech Republic

[29] Memcached. https://memcached.org/

[30] Redis. https://redis.io/