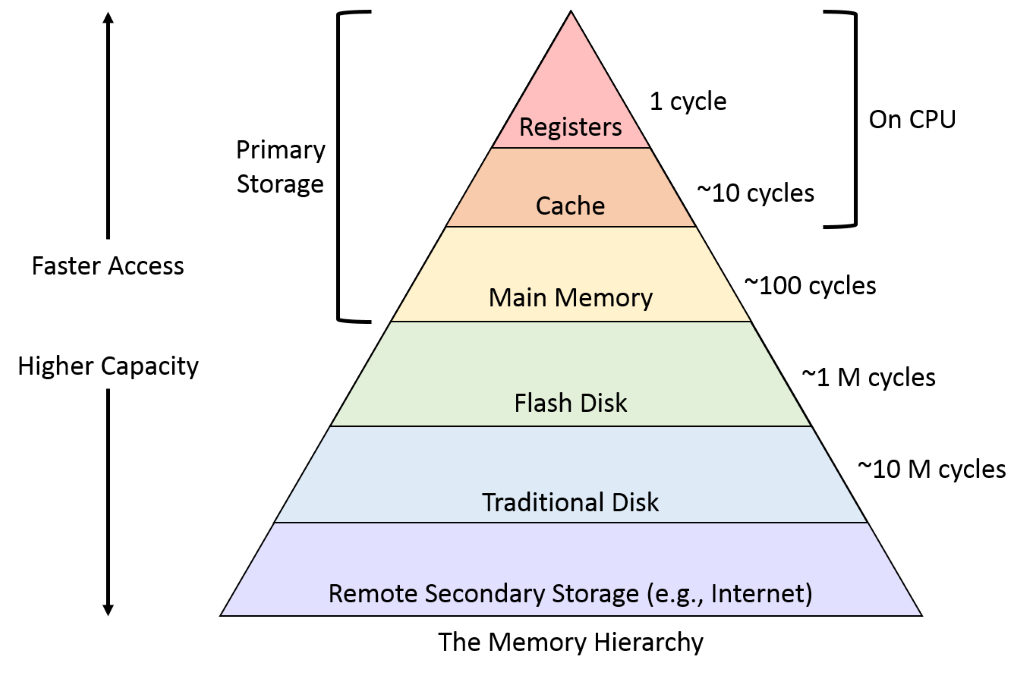
Ιεραρχία μνήμης:

Τα σύγχρονα υπολογιστικά συστήματα βασίζονται σε μια ιεραρχία μνήμης πολλών επιπέδων, προκειμένου να συνδυάσουν τις επιδόσεις του ανώτερου επιπέδου με τη χωρητικότητα του κατώτερου. 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.



Όσο αποδοτική και αν είναι η ιεραρχία αυτή, η απόδοσή της 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 και στη συνέχεια στις μνήμες που βρίσκονται στο εσωτερικό του επεξεργαστή. 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.

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. 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, Γιώργος ... 280 φορές αντί να αποθηκεύσουμε το όνομα Γιώργος 280 φορές μπορούμε να αποθηκεύσουμε (Γιώργος, 280), όπου 280 οι φορές που εμφανίζεται η λέξη Γιώργος. Η τεχνική αυτή η οποία λέγεται Run Length Encoding αποτελεί τη βάση για πολλές μεθόδους μη απωλεστικής συμπίεσης.

Για όλες τις αυτές τις χρήσεις το ζητούμενο είναι η εξοικονόμηση χώρου, για παράδειγμα χώρου στο δίσκο ή στους δίσκους DVD ή η αποδοτικότερη χρήση του δίαυλου, όπως για παράδειγμα οι ταινίες που βλέπουμε μέσω internet. Τα τελευταία χρόνια όμως, με την έκρηξη της παραγωγής αλλά και ανάλυσης των δεδομένων μεγάλης κλίμακας για την εξαγωγή γνώσης έχει αρχίσει να δημιουργείται ενδιαφέρον για τη συμπίεση των δεδομένων, όχι τόσο για τη μείωση του απαιτούμενου χώρου, αλλά κυρίως του απαιτούμενου χρόνου. Στο προηγούμενο κεφάλαιο περιγράψαμε την ιεραρχία μνήμης των σύγχρονων υπολογιστικών συστημάτων. Εκεί είδαμε πως όσο ανεβαίνουμε επίπεδο, από τον δίσκο προς τον επεξεργαστή, το διαθέσιμο bandwidth αυξάνει κατά τάξεις μεγέθους, αλλά το ίδιο μειώνεται και ο διαθέσιμος χώρος αποθήκευσης. Στόχος μας στην ανάλυση δεδομένων είναι να τοποθετήσουμε τα δεδομένα όσο πιο κοντά γίνεται στον επεξεργαστή. Αν για παράδειγμα καταφέρουμε να μειώσουμε το μέγεθος ενός αρχείου αρκετά ώστε να χωρέσει στην κύρια μνήμη του συστήματος, θα έχουμε τη δυνατότητα να πετύχουμε τάξεις μεγέθους καλύτερες επιδόσεις. Για να πετύχουμε την επιθυμητή αύξηση των επιδόσεων θα πρέπει η μέθοδος συμπίεσης που θα επιλέξουμε να μην επιβαρύνει ιδιαίτερα τον επεξεργαστή, καθώς τότε 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. Στις μέχρι τώρα χρήσεις της συμπίεσης, όπως για παράδειγμα στο streaming μέσω διαδικτύου ή στην αποθήκευση αρχείων μια τέτοια υπολογιστική επιβάρυνση είναι συχνά αποδεκτή καθώς γίνεται σπάνια η διαδικασία της συμπίεσης και αποσυμπίεσης και επιπλέον το κέρδος σε χώρο και σε απαιτούμενο bandwidth του δικτύου είναι το κυρίως ζητούμενο. Αντίθετα στην ανάλυση μεγάλου όγκου δεδομένων το ζητούμενο είναι το throughput του επεξεργαστή. Για το λόγο αυτό θα χρησιμοποιήσουμε lightweight τεχνικές συμπίεσης.

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:

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 Νίκος, …, Νίκος, Ν φορές και αμέσως μετά Γιώργος, …, Γιώργος Μ φορές αυτό μπορεί να αναπαρασταθεί ως (Νίκος, Ν), (Γιώργος, Μ). Προκειμένου να κρατήσουμε και την πληροφορία για τη θέση του κάθε ονόματος στο dataset μπορούμε αντί για την δυάδα να αναπαραστήσουμε να δεδομένα με την τριάδα (Νίκος, 0, Ν), (Γιώργος, Ν, Μ), όπου 0 η θέση της πρώτης εμφάνισης της λέξης Νίκος και αντίστοιχα Ν η θέση της πρώτης εμφάνισης της λέξης Γιώργος.

Παρακάτω φαίνεται αυτή η τεχνική:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **index** | **name** |  |  |  |  |  |
| 0 | george |  |  |  |  |  |
| 1 | george |  |  |  |  |  |
| ... | ... |  | | | **index** | **name** |
| 999 | george | 0 | (george, 0, 1000) |
| 1000 | nick | 1 | (nick, 1000, 1087) |
| 1001 | nick | ... | … |
| … | ... |  |  |  |  |  |
| 2086 | nick |  |  |  |  |  |
| ... | … |  |  |  |  |  |

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** |  |  |  | **index** | **1** | **2** | **3** |
| 0 | 1 |  | | | 0 | 1 | 0 | 0 |
| 1 | 3 | 1 | 0 | 0 | 1 |
| 2 | 1 | 2 | 1 | 0 | 0 |
| 3 | 2 | 3 | 0 | 1 | 0 |
| 4 | 2 |  |  |  | 4 | 0 | 1 | 0 |
| 5 | 1 |  |  |  | 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.

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, 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). Στην εργασία αυτή επιλέξαμε την κωδικοποίηση roaring για συμπίεση bitmaps. Στην τεχνική αυτή το dataset χωρίζεται σε containers των 65536 στοιχείων (216). 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** |  |  | Γιώργος | | | … | Φωτεινή | | |
| 0 | Γιάννης |  |  |  | Container 0 |  |  |  | Container 0 |  |
| 1 | Γιώργος |  |  |  | 1 |  |  |  | 48 |  |
| 2 | Ευγενία |  |  |  | 108 |  |  |  | 674 |  |
| … | … |  |  |  | … |  |  |  | … |  |
| … | … |  |  |  | 23731 |  |  |  | 23323 |  |
| 65534 | Φωτεινή |  |  |  | 26574 |  |  |  | 65534 |  |
| 65535 | Καλλιθέα |  | |  |  |  |  |  |  |  |
| … | … |  |  |  |  |  |  |  |
| … | … |  | … |  | … |  | … |  |
| … | … |  |  |  |  |  |  |  |  |  |
| 1048576 | Γιώργος |  |  |  | Container 16 |  |  |  | Container 16 |  |
| 1048577 | Φωτεινή |  |  |  | 1 |  |  |  | 1048577 |  |
| 1048578 | Γιώργος |  |  |  | 0 |  |  |  |  |  |
| … | … |  |  |  | … |  |  |  |  |  |
| … | … |  |  |  | 1 |  |  |  |  |  |
| 1114110 | Γιώργος |  |  |  | 1 |  |  |  |  |  |
| 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.

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 log2(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 |  |  |  | Container 0 | |
| 1 | 5 |  |  |  |
| 2 | 7 |  |  |  |
| … | … |  |  |  |
| 999 | 2 |  | | |
| … | … | . | |
| … | … | . | |
| … | … |  |  |  | . | |
| ... | … |  |  |  | . | |
| 20000 | 88 |  |  |  | Container 20 | |
| … | … |  |  |  |
| 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] |

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** |  |  |  | **index** | **delta** |
| 0 | 102 |  | | | 0 | 0 |
| 1 | 108 | 1 | 6 |
| 2 | 111 | 2 | 3 |
| 3 | 120 | 3 | 9 |
| 4 | 128 |  |  |  | 4 | 8 |
| 5 | 129 |  |  |  | 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.

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** |  |  | Dictionary | |  | **index** | **data** |
| 0 | Γιώργος |  |  | **key** | **value** |  | 0 | 0 |
| 1 | Γιώργος |  | | Γιώργος | 0 |  | 1 | 0 |
| 2 | Κώστας | Κώστας | 1 |  | 2 | 1 |
| 3 | Γιώργος | Αντώνης | 2 |  | 3 | 0 |
| 4 | Αντώνης | Μαρία | 3 |  | 4 | 2 |
| 5 | Μαρία |  |  |  |  |  | 5 | 3 |
| 6 | Αντώνης |  |  |  |  |  | 6 | 2 |

Spark

Το Spark είναι ένα σύστημα κατανεμημένης επεξεργασίας δεδομένων, το οποίο χειρίζεται δεδομένα που βρίσκονται σε μια δομή που λέγεται Resilient Distributed Dataset (RDD). Τα RDDs είναι σαν λίστες από δεδομένα, με τη διαφορά ότι σε αντίθεση με τις τοπικές δομές δεδομένων, όπως λίστες ή πίνακες, τα RDDs είναι κατανεμημένα σε όλους τους κόμβους του cluster. Στην πραγματικότητα μπορούμε πολύ εύκολα από μια τοπική δομή δεδομένων να δημιουργήσουμε ένα RDD:



Στο σημείο αυτό έχουμε φτιάξει μια λίστα η οποία υπάρχει μόνο στον κόμβο που εργαζόμαστε. Στη συνέχεια μπορούμε να δημιουργήσουμε το RDD ως εξής:



Βλέπουμε ότι από τη λίστα ακεραίων έχουμε δημιουργήσει ένα RDD ακεραίων (RDD[Int]). Σε αντίθεση με τη λίστα, το RDD βρίσκεται κατανεμημένο σε partitions τα οποία μπορεί να βρίσκονται σε διαφορετικά συστήματα το καθένα. Μπορούμε να δούμε τον αριθμό των partitions που δημιουργήθηκαν με την παρακάτω εντολή:



Όπου βλέπουμε ότι το RDD που δημιουργήσαμε είναι κατανεμημένο σε 4 partitions. Στο σημείο αυτό πρέπει να αναφέρουμε άλλη μια χαρακτηριστική διαφορά του RDD από μια τοπική συλλογή (collection) δεδομένων. Όταν δημιουργήσαμε τη λίστα το σύστημα μας επέστρεψε τον τύπο της (List[Int]) και τα δεδομένα της (List(1, 2, 3, 4, 5, 6)). Όταν δημιουργήσαμε το RDD μας επέστρεψε τον τύπο του (RDD[Int]), αλλά αντί για τα δεδομένα του μας επέστρεψε ένα ParallelCollectionRDD. Τι είναι αυτό; Τα RDDs προκύπτουν από transformations πάνω σε υπάρχοντα δεδομένα ή άλλα RDDs. Παράδειγμα τέτοιου transformation είναι η εντολή parallelize η οποία παίρνει ένα collection και δημιουργεί ένα RDD. Στο RDD αποθηκεύεται το transformation, δηλαδή η ενέργεια που χρειάζεται για να δημιουργηθεί και όχι τα ίδια τα δεδομένα. Αυτό έχει δυο οφέλη: πρώτον είναι πιο memory efficient να αποθηκεύονται οι ενέργειες και όχι τα δεδομένα και δεύτερον, σε περίπτωση απώλειας ενός partition, τα δεδομένα δεν χάνονται καθώς έχουμε γνώση για το πως δημιουργήθηκαν. Οπότε το μόνο που μένει είναι να εκτελεστούν οι ενέργειες ξανά και να παραχθούν τα ίδια δεδομένα. Ένα άλλο transformation είναι το map, το οποίο για κάθε στοιχείο του RDD κάνει κάποιες ενέργειες και με τη σειρά του παράγει ένα νέο RDD. Για παράδειγμα, αν θέλουμε να αυξήσουμε όλα τα στοιχεία κατά 1 μπορούμε να γράψουμε το εξής:



Οπότε πώς παίρνουμε τα δεδομένα ενός RDD; Μπορούμε να εκτελέσουμε ένα action, το οποίο εκτελεί τις ενέργειες πάνω στο RDD και επιστρέφει τα αποτελέσματα. Ένα τέτοιο action είναι το collect.

Για να πάρουμε τα στοιχεία της κατανεμημένης μας λίστας:



Και τα στοιχεία της λίστας που προκύπτει από την προσαύξηση του κάθε στοιχείου της αρχικής λίστας κατά 1:



Ένα άλλο πολύ χρήσιμο action είναι το reduce, το οποίο κάνει συνάθροιση (aggregation) σε όλα τα στοιχεία του RDD και παράγει το τελικό αποτέλεσμα. Για παράδειγμα, για να αθροίσουμε τα στοιχεία μιας λίστας γράφουμε:



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 γραμμή – 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 Ids θα αποθηκεύονται στη σειρά, στη συνέχεια τα Ονόματα, μετά οι Ηλικίες και τέλος, οι Διευθύνσεις. 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. 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. 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.

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: Αποθήκευση των δεδομένων ως έχει, χωρίς κάποια μορφή συμπίεσης
* RLE: Συνεχόμενες τιμές αποθηκεύονται ως τριάδες (value, index, runLength), όπου value η τιμή, index η γραμμή στην οποία ανήκει και runLength το πόσες φορές εμφανίζεται συνεχόμενα η συγκεκριμένη τιμή.
* Bitmap: Αντί να αποθηκεύονται οι τιμές, για την κάθε τιμή υπάρχει μια σειρά από bits, όσα είναι και ο αριθμός των γραμμών, όπου ένα bit της γραμμής n είναι αναμμένο (1) αν στη γραμμή n υπάρχει το στοιχείο αυτό και σβηστό (0) αν δεν υπάρχει.
* Roaring: Είναι μια τεχνική συμπίεσης bitmaps, όπου το dataset χωρίζεται σε containers των 65536 στοιχείων. Αν μέσα σε ένα container βρίσκονται 16 ή λιγότερα στοιχεία το container αυτό λειτουργεί ως απλό bitmap. Αν βρίσκονται σε αυτό περισσότερα από 16 στοιχεία τότε η θέση του καθενός αντί για bitmap, αποθηκεύεται σε λίστα ως ακέραιος των 16 bit.
* Bit packing: Τα δεδομένα αποθηκεύονται σε containers των 1000 στοιχείων, χρησιμοποιώντας όσα bits είναι απαραίτητο. Για παράδειγμα αν το μικρότερο στοιχείο του dataset είναι το 0 και το μεγαλύτερο το 232, τα στοιχεία θα χρησιμοποιούν 8 bit για την αναπαράστασή τους.
* Delta: Όμοια με το bit packing, αλλά αντί να αποθηκεύουμε τα στοιχεία, αποθηκεύουμε τη διαφορά του κάθε στοιχείου με το προηγούμενό του.
* Dictionary: Όταν αποθηκεύουμε μη αριθμητικά δεδομένα, για παράδειγμα συμβολοσειρές (strings), αντιστοιχίζουμε σε κάθε μοναδικό στοιχείο έναν αριθμό και στη συνέχεια αποθηκεύουμε τον αριθμό αυτό στη θέση του στοιχείου.

Οι κολόνες αυτές μπαίνουν σε μια δομή, την οποία ονομάσαμε chunk, η οποία δεν είναι παρά μια θήκη για τις κολόνες αυτές, προσφέροντας την αφαίρεση ενός ενιαίου dataset.

Το σχήμα αυτό παρουσιάζεται παρακάτω:

col 2

col 1

col N

...

chunk

Η κάθε κολόνα είναι ανεξάρτητη από τις υπόλοιπες και μπορούμε να εκτελέσουμε επάνω της απευθείας ερωτήματα (queries), χωρίς να έχει προηγηθεί αποσυμπίεση.

Περιγραφή συστήματος

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.

Στην περίπτωση που το Hybrid Columnar χρησιμοποιηθεί πάνω από το Spark, το μόνο που χρειάζεται είναι να συμπεριλάβουμε το .jar της εφαρμογής κατά την εκκίνηση του Spark:



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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | |  | |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| **col 1** | **col 2** | **col 3** |  |  |  |
| ... | ... | ... | row 1 |  |  |
| ... | ... | ... | . | partition 1 |  |
| ... | ... | ... | . |  |  |
| ... | ... | ... | . |  |  |
| ... | ... | ... | row n |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| **col 1** | **col 2** | **col 3** |  |  |  |
| ... | ... | ... | row n+1 |  |  |
| ... | ... | ... | . | partition 2 | RDD |
| ... | ... | ... | . |  |  |
| ... | ... | ... | . |  |  |
| ... | ... | ... | row 2 x n |  |  |
|  |  |  |  |  |  |
|  | . |  |  |  |  |
|  | . |  |  |  |  |
|  | . |  |  |  |  |
|  | . |  |  |  |  |
|  | . |  |  |  |  |
|  |  |  |  |  |  |
| **col 1** | **col 2** | **col 3** |  |  |  |
| ... | ... | ... | row (m-1)x(n+1) |  |  |
| ... | ... | ... | . | partition m |  |
| ... | ... | ... | . |  |  |
| ... | ... | ... | . |  |  |
| ... | ... | ... | row m x n |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

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:

|  |  |  |
| --- | --- | --- |
| **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 |

Θα ταξινομηθεί ως:

|  |  |  |
| --- | --- | --- |
| **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 του dataset γίνεται εσωτερικά σε κάθε partition και όχι globally. 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.

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

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 αλλά και σε μετρήσεις που κάναμε με διάφορες τεχνικές συμπίεσης.

Νευρωνικά δίκτυα

Στη συνέχεια εξετάσαμε τη χρήση Machine Learning τεχνικών και συγκεκριμένα LSTM Recurrent Neural Networks. Στα δίκτυα αυτά το σύστημα διαβάζει το dataset και με βάση το μοντέλο που έχει δημιουργηθεί από το κατάλληλο training κατηγοριοποιεί το dataset σε ένα από τα διαθέσιμα encodings. Στην ουσία λύνουμε ένα classification πρόβλημα, όπου η είσοδος είναι ολόκληρο το dataset και οι κλάσεις όλα τα πιθανά encodings. Στην προσέγγιση αυτή παίρναμε καλό accuracy και επιπλέον είχαμε ένα δυναμικό σύστημα, το οποίο μπορούσε να παίρνει αποφάσεις και για dataset τα οποία δεν έχει προηγουμένως συναντήσει.

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:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| |  | | --- | |  | |  |  |  |  |  |  |
|  | **col 1** | **col 2** | **col 3** |  |  |  |
|  | ... | ... | ... | row 1 |  |  |
|  | ... | ... | ... | . |  | partition 1 |
|  | ... | ... | ... | . |  |  |
|  | ... | ... | ... | . |  |  |
|  | ... | ... | ... | row n |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  | col 1 | col 2 | col 3 |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| col 1  col 2  col 3   |  | | --- | |  | |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  | chunk |  | partition 1 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

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:



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



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



Στο παραπάνω παράδειγμα υπολογίσαμε το πλήθος των πελατών ανά ηλικία.

Για να αξιολογήσουμε την απόδοση του συστήματος και να το συγκρίνουμε με το ασυμπίεστο και το parquet, εκτελέσαμε queries σε ένα εύρος από συνθετικά datasets διαφορετικών κατανομών και cardinalities. 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 109 rows 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, πολιτείες ή νομούς, 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.

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

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 το πέρασμα 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:

Στο διάγραμμα αυτό παρατηρούμε ότι το μέγεθος του bitmap ακολουθεί το cardinality του dataset, έχοντας 10Χ μέγεθος για cardinality 100 σε σχέση με 10 και αντίστοιχα 100Χ για cardinality 1000 σε σχέση με 10. 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:

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 περνά σε 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.

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 διαφορετικά queries, ένα select sum με μικρό selectivity (10%), ένα select sum με μεγάλο selectivity (75%) και ένα group by query. 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 selectivity 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:

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.

Το ενδιαφέρον στην κανονική κατανομή είναι πως όλες οι τεχνικές συμπίεσης πετυχαίνουν καλύτερους χρόνους από το ασυμπίεστο. Ειδικά το parquet πετυχαίνει μια τάξη μεγέθους καλύτερους χρόνους σε σχέση με την 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.

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%.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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%.

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.

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.

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.