KLEE: A Framework for Distributed Top-k Query Algorithms

Η εργασία αναφέρεται στο πρόβλημα των top-k queries που αφορούν δεδομένα τα οποία είναι κατανεμημένα σε πολλούς κόμβους. Για τον υπολογισμό του κόστους του αλγορίθμου συνυπολογίζονται ο λανθάνων χρόνος του δικτύου, η κατανάλωση εύρους ζώνης και ο φόρτος κάθε κόμβου. Παρουσιάζεται ο KLEE, ένας νέος αλγόριθμος για κατανεμημένες top-k επερωτήσεις, σχεδιασμένος για υψηλή απόδοση και ευελιξία.

Όσον αφορά την απόδοση, οι πιο επιτυχημένες προσεγγίσεις του top-k προβλήματος βασίζονται σε αλγορίθμους κατωφλίου(TA). Η περιοχή αυτή έχει κατανοηθεί σε μεγάλο βαθμό όσον αφορά δεδομένα σε ένα κόμβο αλλά είναι σε πρώιμο στάδιο για κατανεμημένα συστήματα.

Θεωρούμε ένα κατανεμημένο σύστημα με N κόμβους P_i , i=1...N, που είναι συνδεδεμένοι π.χ. με ένα πίνακα κατακερματισμού. Data items είναι έγγραφα όπως web pages ή δομημένα data items όπως περιγραφείς ταινιών. Κάθε data item είναι συνδεδεμένο με ένα set από περιγραφείς, text terms or attribute values. Η inverted index list ενός περιγραφέα (descriptor) είναι η λίστα των data items στα οποία ο descriptor εμφανίζεται ταξινομημένη σε φθίνουσα σειρά των *scores*. Θεωρούμε παραλλαγές TA-sorted που δεν επιτρέπουν την τυχαία προσπέλαση στις index lists ώστε να έχουμε λιγότερο φόρτο στους κόμβος και καλύτερη επίδοση.

Κάθε κόμβος P_j έχει αποθηκευμένη μια index list $I_j(t)$ για έναν όρο t. Αποτελείται από ζευγάρια (*docID,score*) όπου το *score* ε (0,1] και δείχνει το πόσο σημαντικό είναι το έγγραφο *docID* για τον όρο *t*. Ένα ερώτημα *q(T, k)* που αρχικοποιείται σε έναν κόμβο P_{init} αποτελείται από το μη κενό σύνολο $T=\{t_1, t_2, \ldots t_t\}$ και έναν ακέραιο *k*. Έστω m οι κόμβοι που περιέχουν τις πιο σχετικές index lists με $m \leq t$. Στόχος μας είναι να επινοήσουμε αποδοτικές μεθόδους ώστε η P_{init} να προσπελαύνει τις m λίστες και να δημιουργεί το αποτέλεσμα για τα top-k έγγραφα. Το αποτέλεσμα είναι η διατεταγμένη λίστα *(docID,TotalScore)* όπου το *TotalScore* είναι μια μονοτονική συνάρτηση των επιμέρους *score*. Χρησιμοποιούμε την άθροιση. Σημειώνουμε πως το P_{init} συντονίζει μια συγκεκριμένη ερώτηση και συνήθως άλλος κόμβος συντονίζει άλλη ερώτηση.

Μια πρώτη λύση θα ήταν να συγκεντρώνονται οι m λίστες στον αρχικό κόμβο και εκεί να εκτελεσθεί μια κεντρική TA-style μέθοδος. Για ένα peer-to-peer σύστημα είναι μη αποδεκτή γιατί γίνεται σπατάλη του εύρους ζώνης κατά την μεταφορά των λιστών στο P_{init}. Εναλλακτική προσέγγιση να εκτελέσουμε τον TA στον P_{init} αλλά κάθε φορά να προσπελαύνουμε ένα entry. Και αυτό είναι μη αποδεκτό καθώς έχουμε πολλά μικρά μηνύματα στο δίκτυο και χρειαζόμαστε τόσους γύρους μηνυμάτων όσο και το μέγιστο μέγεθος μεταξύ των λιστών. Όπως έδειξαν στο [CW04] ένας αποδοτικός κατανεμημένος top-k αλγόριθμος σε σχέση με το δίκτυο θα πρέπει να τερματίζει σε ένα μικρό αριθμό βημάτων. Σε κάθε βήμα ο P_{init} μαζεύει πληροφορίες από τις index lists των κόμβων και προσπαθεί με έξυπνο τρόπο να καταλάβει εάν έχει την απαραίτητη πληροφορία ώστε να εκτιμήσει το αποτέλεσμα με υψηλή ποιότητα και να τερματίσει την διαδικασία. Ουσιαστικά πρέπει να εκτιμήσει το *TotalScore*.

Έχουμε δυο σημαντικά ζητήματα κατά την εκτίμηση του *TotalScore*:
α) missing scores όταν τα id των document περιλαμβάνονται στις αναφορές κάποιων κόμβων και όχι σε άλλων. Αυτό κάνει πολύπλοκο τον υπολογισμό του *TotalScore* για αυτά τα *document* με αποτέλεσμα να μην μπορούμε να απαλείψουμε νωρίς υποψήφια top-k ή να παράγουμε μια καλή εκτίμηση του top-k.
β) missing documents: είναι ένα πρόβλημα για τον συντονιστή όταν δεν έχει πλήρη εικόνα των *document.* Είναι δύσκολο για τον συντονιστή να μάθει για έγγραφα το οποία είναι χαμηλά στις λίστες για κάθε όρο αλλά συνολικά το άθροισμα να τα κάνει καλά υποψήφια για το αποτέλεσμα.

Ο αλγόριθμος βασίζεται στην παραλλαγή *TA-sorted* η οποία επεξεργάζεται τις *entries* (*docID/score)* στις σχετικές λίστες με φθίνουσα σειρά για τα *score*, χρησιμοποιεί *round robin* και κάνει μόνο σειριακή αναζήτηση (sequential access) στις λίστες. Ο *TA-sorted* κρατάει μια ουρά από υποψήφια για top-k και τα μέχρι στιγμής αποτελέσματα στη μνήμη. Ο αλγόριθμος κρατάει ανά πάσα στιγμή μαζί με το έγγραφο *d* το *worstScore* και το *bestScore.* Το πρώτο είναι το άθροισμα των *score* για κάθε λίστα που έχει εμφανιστεί το *d*, ενώ το *bestScore* είναι το *worstScore* προσαυξημένο με τις τιμές (*high(i)*) που έχουν τα τελευταία έγγραφα στις λίστες στις οποίες δεν υπάρχει το *d*. Το *current top-k* είναι το σύνολο με τα *k documents* που έχουν το καλύτερο *worstScore*. Το *topKscore* είναι το ελάχιστο μεταξύ των *k documents*. Εάν για ένα υποψήφιο ισχύει *bestScore<topKscore* τότε το *d* παύει να είναι υποψήφιο. Ο αλγόριθμος τερματίζει όταν η λίστα υποψηφίων είναι άδεια

Η προτεινόμενη προσέγγιση βασίζεται στο να έχουμε έναν συντονιστή για κάθε επερώτηση και μια ομάδα από κόμβους. Ο συντονιστής είναι ο P_{init} ενώ η ομάδα είναι οι κόμβοι που έχουν τις απαραίτητες λίστες. Ο αλγόριθμος λειτουργεί με φάσεις επικοινωνίας. Σε κάθε φάση ο συντονιστής ζητά και λαμβάνει ένα τμήμα τις λίστας που έχει κάθε κόμβος ώστε να τρέξει έναν αλγόριθμο για top-k βασισμένο στα δεδομένα που έχει λάβει.

Κάθε κόμβος διατηρεί ένα σύνολο από στατιστικά μετά-δεδομένα για να περιγράψει την λίστα του. Συγκεκριμένα κρατείτε πληροφορία σε μορφή ιστογράμματος για να περιγραφεί η κατανομή του score στην λίστα. Το score είναι (0,1]. Για απλότητα θεωρούμε ότι τα ιστογράμματα έχουν ίσο πλάτος και είναι χωρισμένα σε n κελιά. Κάθε κελί είναι υπεύθυνο για το 1/n του εύρους του score. Για κάθε κελί ο κόμβος διατηρεί την εξής πληροφορία:

1. Την άνω και κάτω τιμή , *ub[i]* και *lb[i]*, που καλύπτει το κελί
2. Το *freq[i]* που είναι ο αριθμός των *document* που πέφτουν στο κελί
3. Το *avg[i],* που είναι το μέσο score των *document* στο κελί
4. Το *filter[i]* που αναπαριστά τα *document* που βρίσκονται στο κελί. Για την υλοποίησή του χρησιμοποιούμε *Bloom filters*.

Στην πρώτη φάση κάθε κόμβος δίνει στον συντονιστή την τοπική top-k λίστα και ένα μέρος του *Bloom filter* του. Στην συνέχεια ο συντονιστής επιλύει το πρόβλημα των *missing scores* βρίσκοντας μέσω του *Bloom filter* σε ποιο κελί ανήκει το *docID*, έστω c, και το αντικαθιστά με το *avg[c]*.

Για το πρόβλημα των *missing document*, αφού έχει λύσει το προηγούμενο, ο συντονιστής υπολογίζει μια πρώτη εκτίμηση του *topKscore*. Ο κάθε κόμβος στέλνει στον συντονιστή μία λίστα με τα υποψήφια του τα οποία έχουν *score>topKscore/m*. Τώρα ο συντονιστής μπορεί να υπολογίσει μια υψηλότερου επιπέδου *top-k* λύση. Διαισθητικά, με αυτόν τον τρόπο χρησιμοποιώντας τα *Bloom filter* ο συντονιστής συγκεντρώνει αρκετή πληροφορία σε βάθος λιστών ώστε να μην χρειάζεται να σπαταλήσουμε μεγάλο εύρος ζώνης.

Επιστρέφοντας στο πρόβλημα των *missing document*, παίρνοντας τις μέσες τιμές προκύπτουν δύο προβλήματα:

1. Εάν λείπουν πολλά *document* τότε θα έχουμε μια προσέγγιση χαμηλής ποιότητας.
2. Εάν το m είναι μεγάλο τότε θα έχουμε μικρό *topKscore/m* και θα επιστραφούν πολλά υποψήφια και θα σπαταλήσουμε το εύρος ζώνης.

Για αυτό μερικές φορές εκτελούμε ένα ακόμα βήμα στο οποίο κάνουμε σμίκρυνση της λίστας των υποψηφίων. Η γενική ιδέα είναι να βρούμε τα *documents* που βρίσκονται σε πιο πολλές από τις λίστες και να στείλουμε μόνο αυτά. Αυτά έχουν μεγαλύτερη πιθανότητα να έχουν *totalScore>topKscore*. Σε αυτή τη φάση κάθε κόμβος βρίσκει τα περιεχόμενα της λίστας των υποψηφίων και στη συνέχεια φτιάχνει ένα διάνυσμα *bitmap* τέτοιο ώστε σε κάθε bit μέσω μιας πλειάδας *hash function* να έχει αντιστοιχηθεί το κατάλληλο *documen*t. Αυτό λέγεται *Candidate List Filter ή CLF*. Ο συντονιστής από την πρώτη φάση γνωρίζει για κάθε κόμβο πόσα *document* έχουν *score>topKscore/m* και στέλνει σε κάθε κόμβο τον μέγιστο αυτόν των αριθμών, έστω b ώστε όλα τα *CLF* να δημιουργηθούν με το ίδιο μέγεθος b. Τελικά ο συντονιστής λαμβάνει από κάθε κόμβο τα CLF και δημιουργεί ένα συγκεντρωτικό πίνακα μεγέθους m×b. Η γραμμή i αντιστοιχεί στον κόμβο i. Ο *CLF Matrix* από κατασκευής του δείχνει σε κάθε στήλη πόσα *document* είναι κοινά υποψήφια σε κάθε λίστα. Έτσι επιλέγουμε ένα κατώφλι για την εμφάνιση των *document*, το οποίο να είναι στην πλειοψηφία των document, και ζητάμε μόνο αυτά.

Στην όλη διαδικασία πρέπει να εξετάσουμε τα εξής:

1. Να συγκεντρώσουμε την απαραίτητη πληροφορία με χαμηλό κόστος σε εύρος ζώνης
2. Να αποφύγουμε μεγάλο φιλτράρισμα των υποψηφίων για να έχουμε μια καλή ποιότητα δεδομένων
3. Και να μπορούμε να προβλέψουμε εάν χρειάζεται το *CLF* προτού το φτιάξουμε και το χρησιμοποιήσουμε

Ο αλγόριθμος εκτελείται σε τέσσερα βήματα: .

Στο πρώτο βήμα ο συντονιστής στέλνει ένα request με βάση την αρχική επερώτηση. Οι κόμβοι απαντούν στέλνοντας την τοπική top-k λίστα και ένα μέρος του ιστογράμματος τους. Για κάθε κελί από αυτά που ανήκουν στα “high-end”, δηλαδή αυτά που έχουν τα documents με τα πιο μεγάλα score στέλνουν το ιστόγραμμα(ub[i], lb[i], freq[i], avg[i] and filter[i]) i=1…c ενώ για τα υπόλοιπα n-c το avg[i] και το freg[i]. Ο συντονιστής υπολογίζει την top-k λίστα χρησιμοποιώντας την πληροφορία που έχει από τα ιστογράμματα. Αυτομάτα έχουμε και τις λίστες των υποψηφίων που είναι όσα έχουν

$$score > \frac{topKscore}{m}$$

Το δεύτερο βήμα, αυτό της βελτιστοποίησης, εκτελείται τοπικά στον συντονιστή. Χρησιμοποιώντας πληροφορία από το πρώτο βήμα βρίσκει την πιθανότητα μια στήλη να έχει R άσσους στον CLF Matrix. Με βάση αυτήν την πιθανότητα και αναλόγως πιο κόστος θέλουμε να ελαχιστοποιήσουμε αποφασίζουμε εάν θα εκτελέσουμε ένα ακόμη βήμα. Ουσιαστικά γίνεται trade-off μεταξύ του εύρους ζώνης και του αριθμού των φάσεων επικοινωνίας.

Το τρίτο βήμα εκτελείται αναλόγως της απόφασης στο προηγούμενο βήμα. Τώρα υπολογίζουμε τα νέα υποψήφια και στέλνονται οι λίστες στους κατάλληλους κόμβους.

Το τελευταίο είναι το βήμα τις ανάκτησης στην οποία ο συντονιστής ανακτά τα υποψήφια από τους κόμβους και υπολογίζει το τελικό αποτέλεσμα.

Οι κύριοι παράμετροι που χαρακτηρίζουν την λειτουργικότητα του KLEE είναι ο αριθμός c, των κελιών για τα οποία στέλνονται φίλτρα κατά την πρώτη φάση και ο αριθμός *R* των *bits* που χρειάζονται ώστε να θεωρηθεί ενδιαφέρουσα η στήλη του *CLF* στο τρίτο βήμα. Άλλες παράμετροι είναι ο *load factor* για τα *Bloom filters* και ο αριθμός των *hash function* που χρησιμοποιούνται. Αυτές επηρεάζουν την πιθανότητα να έχουμε *false positive* και την κρατούν κάτω από ένα κατώφλι δεδομένου ενός πλήθους από *entries*

Για τον έλεγχο της ορθότητας και της απόδοσης του αλγορίθμου υλοποιήθηκαν οι DTA, TPUT, X-TPUT, KLEE-3, KLEE-4. Οι δύο τελευταίοι είναι οι δυο παραλλαγές του KLEE με δύο και τρία βήματα επικοινωνίας. Οι αλγόριθμοι υλοποιήθηκαν σε Java. Τα δεδομένα ήταν αποθηκευμένα τοπικά σε κάθε κόμβο. Τα πειράματα έγιναν σε υπολογιστές με 3GHz Pentium επεξεργαστές. Τα πειράματα έγιναν τόσο σε πραγματικά όσο και σε συνθετικά δεδομένα. Τα πραγματικά δεδομένα αφορούσαν τις βάσεις δεδομένων GOV και IMDB. Τα συνθετικά δεδομένα αφορούν δεδομένα που δημιουργήθηκαν ακολουθώντας τον νόμο του Zipf αλλάζοντας το θ. Επίσης πειράχτηκε» η ΒΔ Gov αλλάζοντας τα *score*. Σαν αποτέλεσμα είδαμε ότι ο KLEE υπερτερεί των υπολοίπων αλγορίθμων.

Συνοψίζοντας ο KLEE σε σχέση με άλλους αλγόριθμους της κατηγορίας του όχι μόνο υιοθετεί λίγα βήματα εκτέλεσης αλλά πηγαίνει ένα βήμα παραπέρα εισάγοντας καινούρια χαρακτηριστικά:

1. Ο KLEE έχει δυο εκδοχές μια με δύο φάσεις και μια με τρεις φάσεις επικοινωνίας. Αναγνωρίζει ότι ο αριθμός των φάσεων είναι μόνο ένας τρόπος για να μειώσουμε το χρόνο απόκρισης και άρα να αυξήσουμε την αποδοτικότητα. Συγκεκριμένα ο KLEE εξασφαλίζει ότι δεν έχουμε τυχαία I/O στο δίκτυο των συνεργαζόμενων κόμβων και έχουμε μικρότερα μηνύματα μιας και το δίκτυο και τα I/O είναι σημαντικοί παράγοντες.
2. Ο KLEE είναι ελαστικός καθώς δίνει την δυνατότητα για trade-off μεταξύ αποδοτικότητας και ποιότητας των αποτελεσμάτων και φάσεων επικοινωνίας και εύρους ζώνης.
3. Είναι εξοπλισμένος με διάφορες ρυθμιστικές παραμέτρους και γίνεται συζήτηση για το πώς αυτές προσαρμόζονται αυτόματα στα δεδομένα.