HELLENIC REPUBLIC

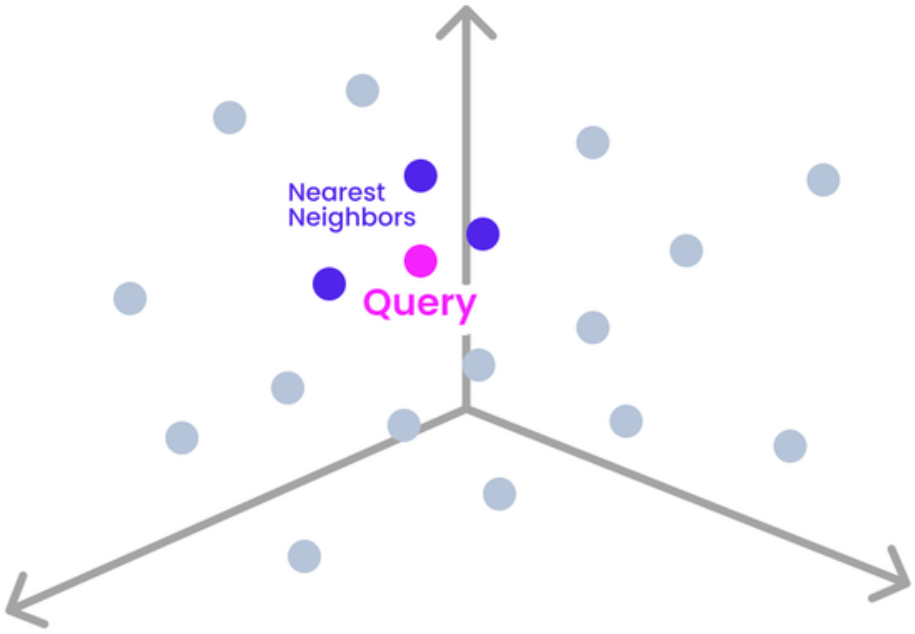
National and Kapodistrian University of Athens

EST. 1837

Ανάπτυξη Λογισμικού Για Πληροφοριακά Συστήματα Κ23α

ΤΕΛΙΚΗ ΑΝΑΦΟΡΑ Project 2024-2025

Approximate Nearest Neighbor Search - ANNS



Διδάσκοντες:
Πασκαλής Σαράντης
Ιωαννίδης Γιάννης

Εργασία των:
Δημητρακάκης Ηλίας, **1115202000046**
Λαμπρόπουλος Βίκτορας, **1115201900096**

Σύντομη Εισαγωγή

Το παρόν αποτελεί την τελική υποβολή μας, συνοδευόμενο από επεξηγήσεις και ανάλυση επιδόσεων πριν και μετά την βελτιστοποίηση αναζήτησης των προσεγγιστικά K εγγύτερων γειτόνων σε γράφο (**k-ANNS**) χρησιμοποιώντας τον αλγόριθμο Vamana. Η μέθοδος αναζήτησης αυτή εφαρμόζεται σε διάφορους τομείς, όπως στα συστήματα συστάσεων, στη μηχανική μάθηση και στη βιοπληροφορική, καθώς και σε οποιαδήποτε εφαρμογή απαιτεί την αναζήτηση για παρόμοια δεδομένα με ταχύτητα και αποτελεσματικότητα.

Η ανάπτυξη της εργασίας έγινε με χρήση:

- `C++` (`-std=c++20`)
- Meson Build Tool
- ninja
- CPU που υποστηρίζει AVX/AVX2 για **SIMD instructions**.

Benchmarking

Όλες οι μετρήσεις που παρουσιάζονται παρακάτω έγιναν με βάση τα εξής specs:

CPU:

- **Model Name:** AMD Ryzen 7 5800X 8-Core Processor
- **Number of CPUs:** 16
- **CPU Family:** 25
- **CPU Model:** 33
- **Threads per Core:** 2
- **Cores per Socket:** 8
- **Sockets:** 1
- **Max CPU MHz:** 3800.0000
- **Min CPU MHz:** 2200.0000

Memory:

- **DDR Type:** DDR4
- **Size:** 16 GB (2 x 8 GB modules)
- **Clock Speed:** 3600 MHz

Κριτήρια Ανάλυσης

Όπως φαίνεται παρακάτω, συγκρίνουμε τις διάφορες εκδόσεις της εργασίας μας με βάση δύο μετρικές: ***Recall@k*** και ***Queries Per Second (QPS)***.

- **Recall**: ποσοστό όμοιων στοιχείων σε δύο σύνολα.

Vamana Indexing - Πρώτη Υλοποίηση

Στην πρώτη έκδοση, αρχικοποιούμε τον γράφο G να είναι R-regular με κάθε κόμβο να έχει τυχαίους γείτονες. Έπειτα, έχοντας βρει το medoid του dataset , ξεκινάμε από αυτό για κάθε κόμβο του γράφου G - χρησιμοποιώντας άπληστο αλγόριθμο αναζήτησης (greedy_search) και κρατώντας πάντα το πολύ k (robust_prune) - να βρούμε τους προσεγγιστικά k κοντινότερους γείτονες για κάθε query node.
Η ανάγνωση datasets ως αρχεία εισόδου γίνονται με πρότυπο τα αρχεία που παρέχονται στη σελίδα http://corpus-texmex.irisa.fr/.

Filtered/Stitched Vamana Indexing - Δεύτερη Υλοποίηση

Εφόσον είδαμε πως παίρναμε πολυ καλά αποτελέσματα (Recall@k $\geq$ 0,95) στο πρώτο σκέλος, στο δεύτερο κληθήκαμε να υλοποιήσουμε προσεγγιστική αναζήτηση, αλλά αυτή τη φορά κάθε σημείο μπορεί να έχει κάποιο φίλτρο. Έτσι, σε αυτή την έκδοση, η αναζήτηση πρέπει να βασίζεται και στην απόσταση αλλά και στο τυχόν φίλτρο του query node. Παρουσιάζουμε δύο αλγορίθμους Vamana Indexing:

- **Filtered**
- **Stitched**

Πριν την εκτέλεση οποιουδήποτε από των δύο αλγόριθμων indexing, υπολογίζουμε τα εξής:

- Το σύνολο F, το οποίο περιέχει όλα τα φίλτρα του dataset.
- Το σύνολο f, το οποίο περιέχει το φίλτρο f_x του σημείου x του dataset, $\forall x$.
- Το medoid (starting point αναζήτησης) για κάθε φίλτρο $f_x \in$ F ($\mathbf{st}(f)$) .

Filtered:
Ο αλγόριθμος Filtered δημιουργεί έναν μεγάλο γράφο που περιλαμβάνει όλα τα σημεία του dataset. Έτσι, η διαδικασία άπληστης αναζήτησης (filtered_greedy_search) και περικοπής (filtered_robust_prune) είναι όπως και πριν, με την λεπτομέρεια πως πλέον λαμβάνουμε υπόψη και το φίλτρο του query node.

Stitched:
Ο αλγόριθμος Stitched δημιουργεί για κάθε φίλτρο του dataset και έναν υπογράφο G_f,όπου τα στοιχεία του είναι μόνο τα σημεία με το φίλτρο αυτό. Με αυτή τη διαφοροποίηση, ο κάθε υπογράφος μπορεί να θεωρηθεί ως "άφιλτρος", οπότε μπορούμε να αγνοήσουμε τα φίλτρα και να εφαρμόσουμε σε καθέναν από αυτούς κατευθείαν Vamana Indexing (αλγόριθμος πρώτου σκέλους).

Η ανάγνωση datasets ως αρχεία εισόδου γίνονται με πρότυπο τα αρχεία που παρέχονται στις σελίδες:

- https://github.com/transactionalblog/sigmod-contest-2024 (dummy-*.bin).
- https://zenodo.org/records/13998879.

Στα πλαίσια του μαθήματος, **ασχολούμαστε μόνο με το categorical attribute (C) των σημείων των datasets και query type 0 (unfiltered query node) ή 1 (filtered query node). Επίσης, κάθε σημείο μπορεί να έχει το πολύ 1 φίλτρο.**

Αρχική Βελτίωση: Υπολογισμός Απόστασης

Η πρώτη σκέψη μας ήταν να τροποποιήσουμε την συνάρτηση υπολογισμού Ευκλείδιας Απόστασης, καταργώντας τον υπολογισμό της ρίζας, καθώς προσθέτει περιττό overhead. Αν και ήταν μια καλή αρχή, χρονομετρώντας την κλήση της συνάρτησης, παίρναμε πίσω χρόνο κατά Μ.Ο. **221 ns**. Έπειτα, αντικαθιστώντας τη συνάρτηση με μια αντίστοιχη που χρησιμοποιεί SIMD instructions, καταφέραμε να ρίξουμε τον χρόνο υπολογισμού στα **115 ns**, πετυχαίνοντας σχεδόν διπλάσια απόδοση σε αυτό το κομμάτι.
Δοκιμάσαμε και distance caching στις συναρτήσεις που υπολογίζουν απόσταση, αλλά δεν παρατηρήσαμε ουσιαστική διαφορά, οπότε κρατήσαμε την αρχική προσέγγιση.

Πρώτο πρόβλημα

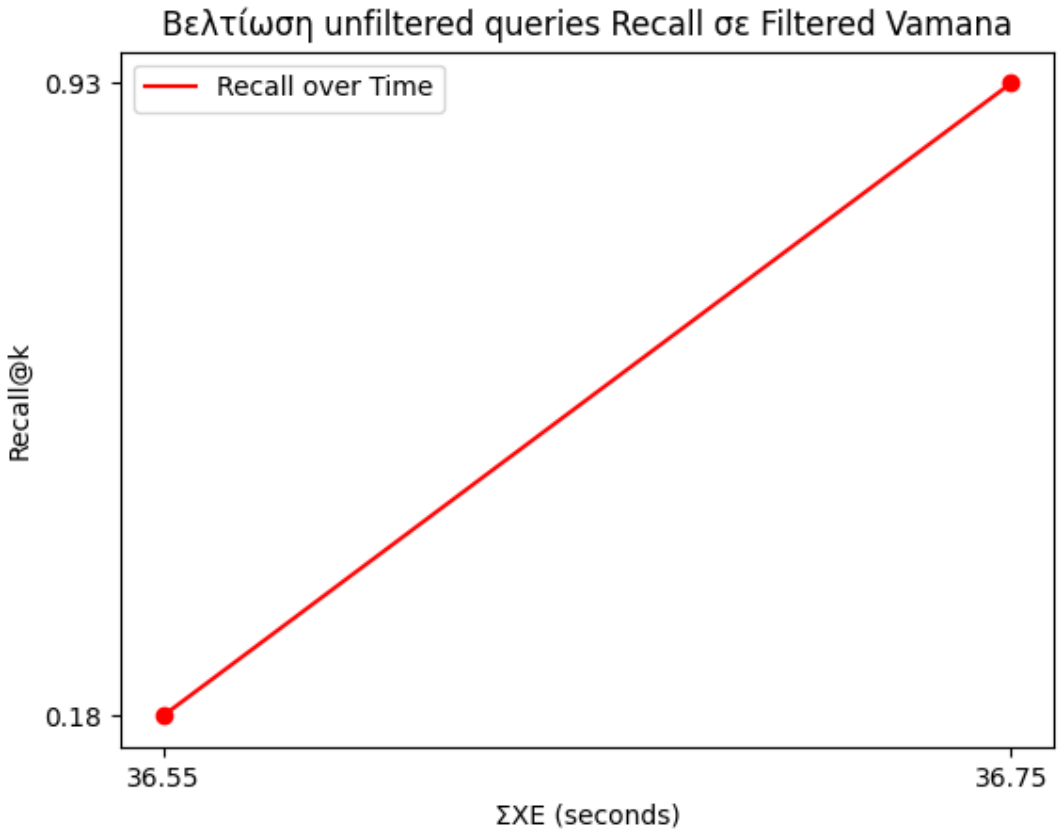
Ενώ η προσθήκη αυτής της λειτουργικότητας επιστρέφει εξαιρετικά αποτελέσματα για query nodes που έχουν φίλτρο (recall@k $\geq 0,95$), το ίδιο δεν μπορεί να ειπωθεί για unfiltered query nodes, των οποίων το μέσο recall@k κυμαίνεται μεταξύ του 0,08 και 0,20 . Αυτό οφείλεται στο γεγονός ότι βάσει του αρχικού αλγορίθμου, αναθέταμε σε unfiltered queries όλα τα φίλτρα του dataset, με αποτέλεσμα να μην περιορίζουμε σωστά την αναζήτηση για τα κοντινότερα σημεία.

Βελτίωση 1: Νέα Προσέγγιση για unfiltered queries

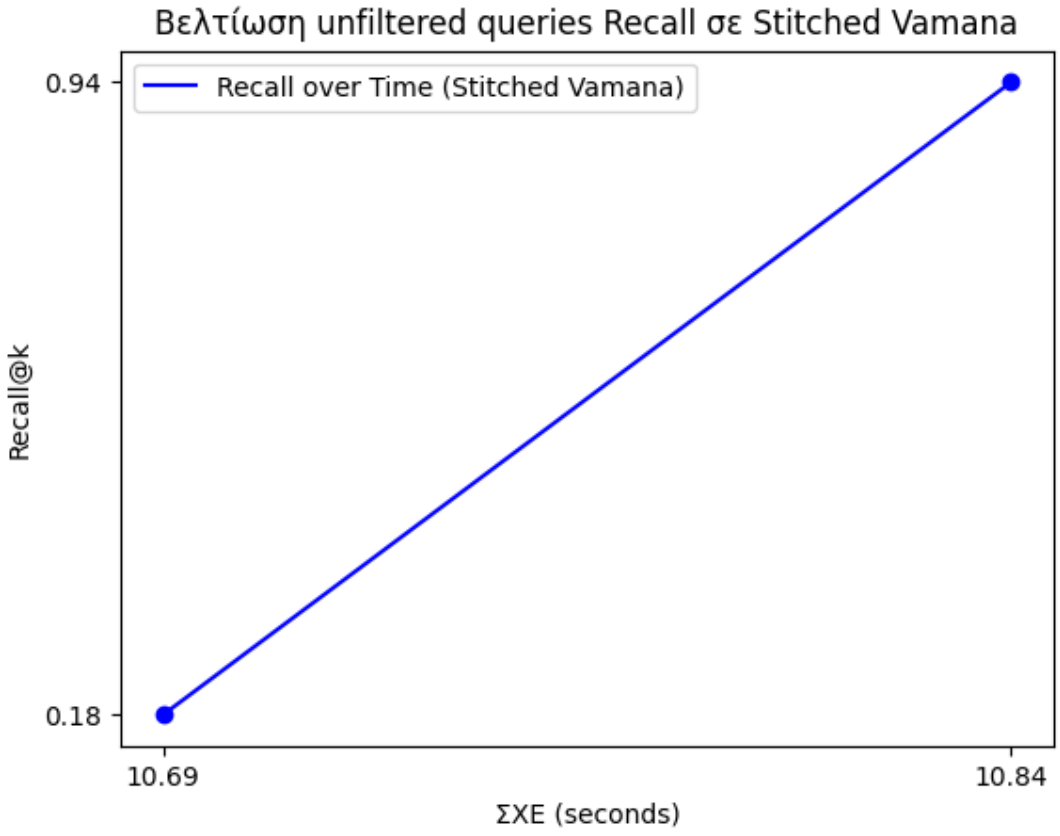
Ακολουθώντας την παρακάτω προσέγγιση για κάθε **unfiltered query node**:

1. Για κάθε φίλτρο που του έχει ανατεθεί, βρίσκουμε τα k'=u (βλ. Παρατηρήσεις-Συμπεράσματα Πειραμάτων) κοντινότερα nodes με αυτό το φίλτρο και τα προσθέτουμε στο σύνολο των starting points.
2. Εκτέλεση αλγορίθμου Filtered Greedy Search με το νέο σύνολο.

Με αυτή την αλλαγή, **παρατηρούμε μεγάλη αύξηση του recall@k, ξεπερνώντας πλέον το 0,9 για unfiltered queries**.



Dataset 10.000 σημείων, 2516 unfiltered queries, k:100, a:1.2, L:120, R:100

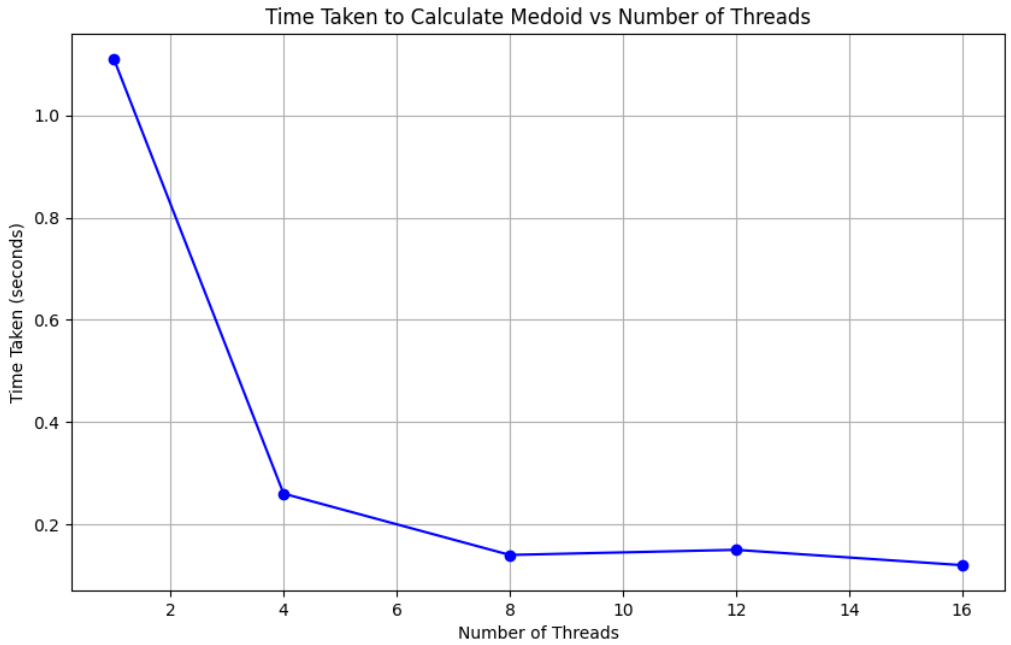


Dataset 10.000 σημείων, 2516 unfiltered queries, k:100, a:1.2, L_{small}:120, R_{small}:100

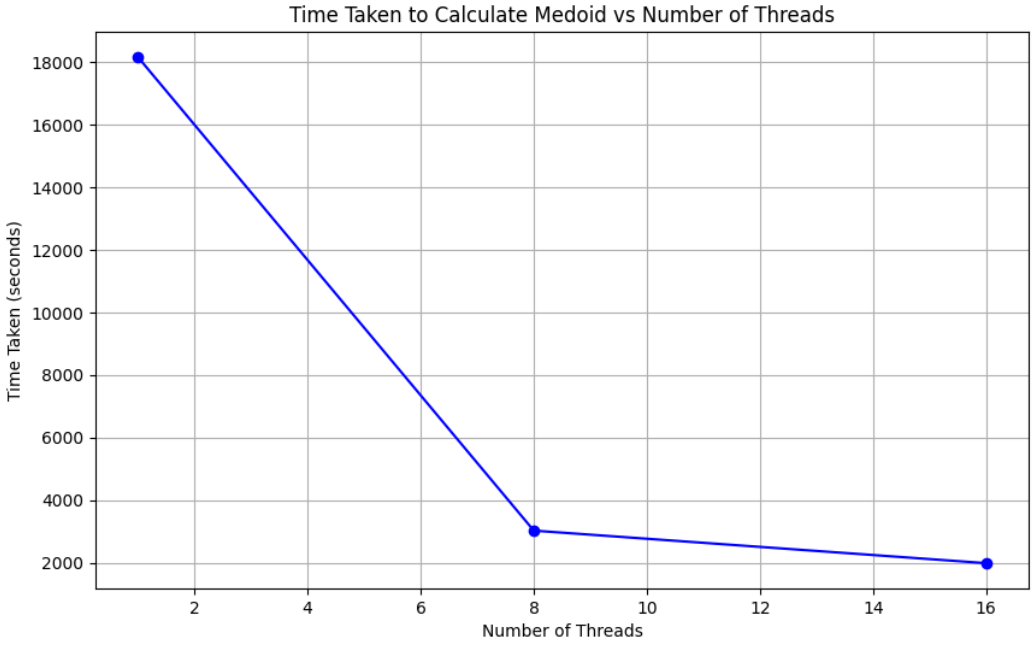
Παραλληλοποίηση

Medoid

Εξ' ορισμού, ως medoid θεωρείται το σημείο m ενός χώρου το οποίο απέχει από όλα τα υπόλοιπα το λιγότερο. Όπως είναι αντιληπτό,για datasets με μεγάλο αριθμό σημείων -έστω n το πλήθος αυτών- και διάστασης d, ο υπολογισμός του medoid έχει πολυπλοκότητα $O\left(n^2 \times d\right)$. Με αυτό κατά νου, αυτός ο τομέας του project χρήζει βελτιστοποίησης. Έτσι, με χρήση openMP threads, καταφέραμε να ρίξουμε αισθητά τον χρόνο υπολογισμού, όπως φαίνεται και στα γραφήματα παρακάτω.



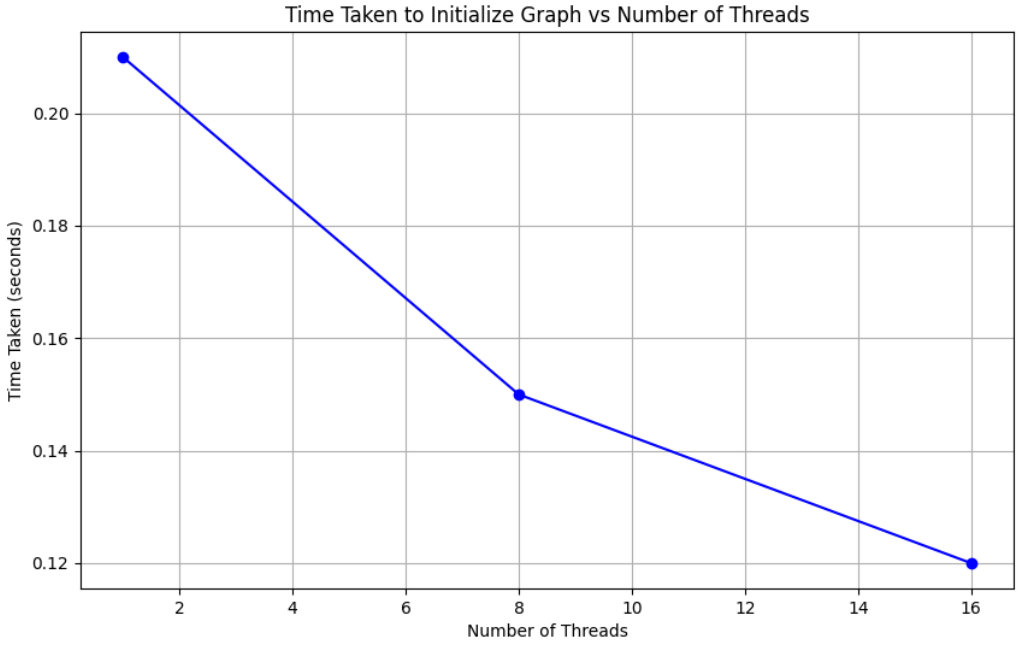
Εύρεση Medoid σε dataset 10.000 σημείων, 100 διαστάσεων (siftsmall_base.fvecs)



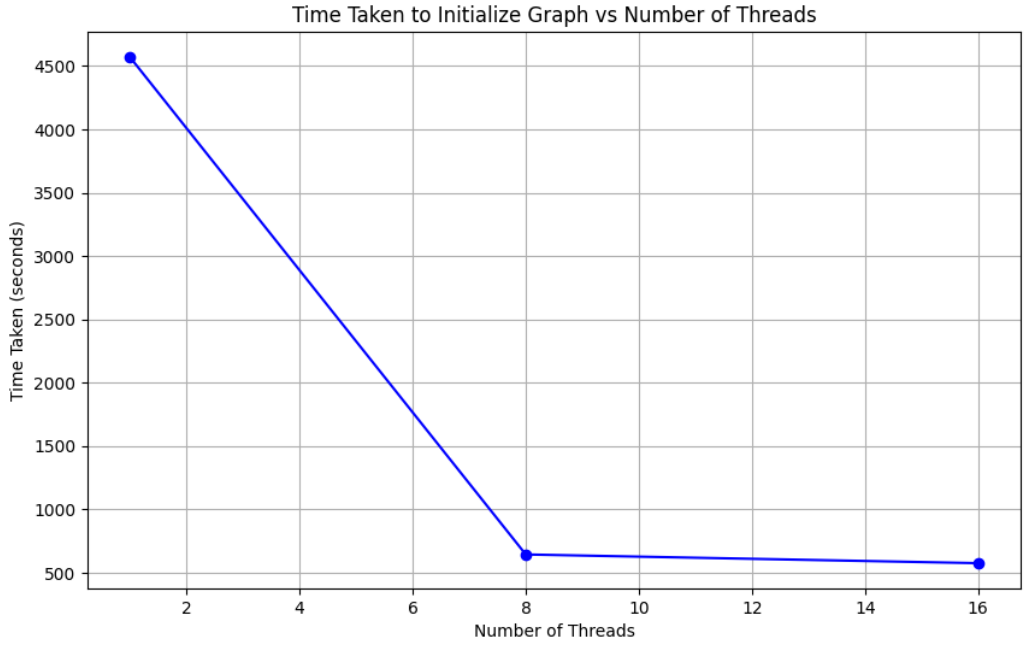
Εύρεση Medoid σε dataset 1.000.000 σημείων, 100 διαστάσεων (SIFT_base.fvecs)

Αρχικοποίηση Γράφου

Επόμενη βελτίωση κάναμε στη συνάρτηση αρχικοποίησης του γράφου, καθώς και πάλι για μεγάλα datasets, ο χρόνος δημιουργίας ήταν απαγορευτικός. Με χρήση παραλληλοποίησης openMP, καταφέραμε τις παρακάτω βελτιώσεις:



Αρχικοποίηση 100-regular Γράφου 10.000 κόμβων (siftsmall_base.fvecs)



Αρχικοποίηση 100-regular Γράφου 1.000.000 κόμβων (sift_base.fvecs)

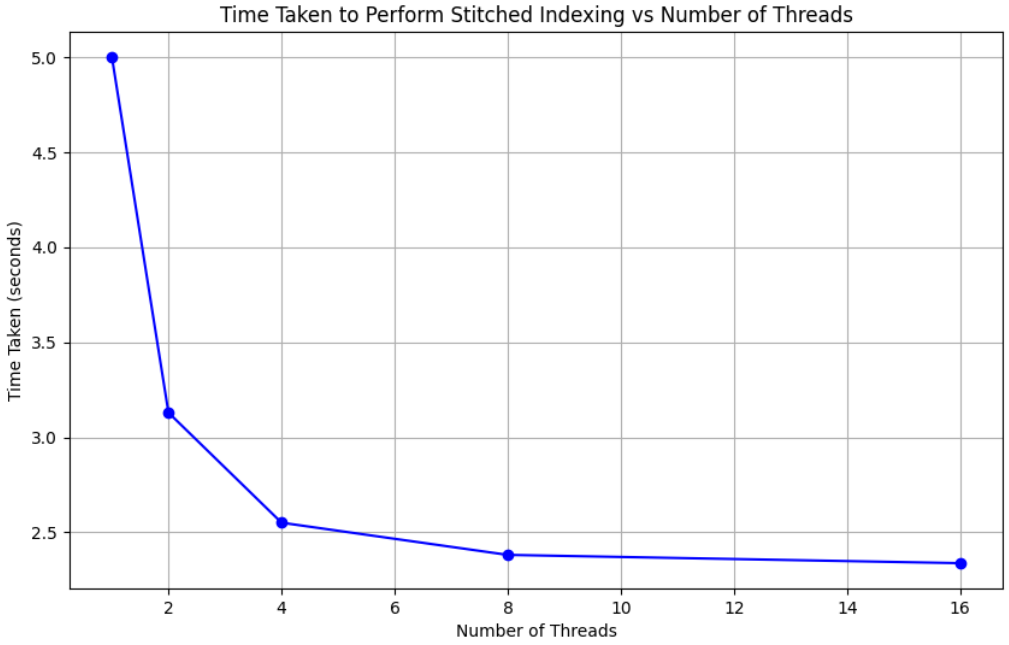
Προτεινόμενη Βελτίωση: Τυχαίο Medoid - Κενός Γράφος

Παρατηρήσαμε ότι χρήση τυχαίου medoid και αρχικοποίηση του γράφου σε κενό, όπως και στο 2ο παραδοτέο, βελτιώνει τον χρόνο κατασκευής, χωρίς να επηρεαστεί το recall.

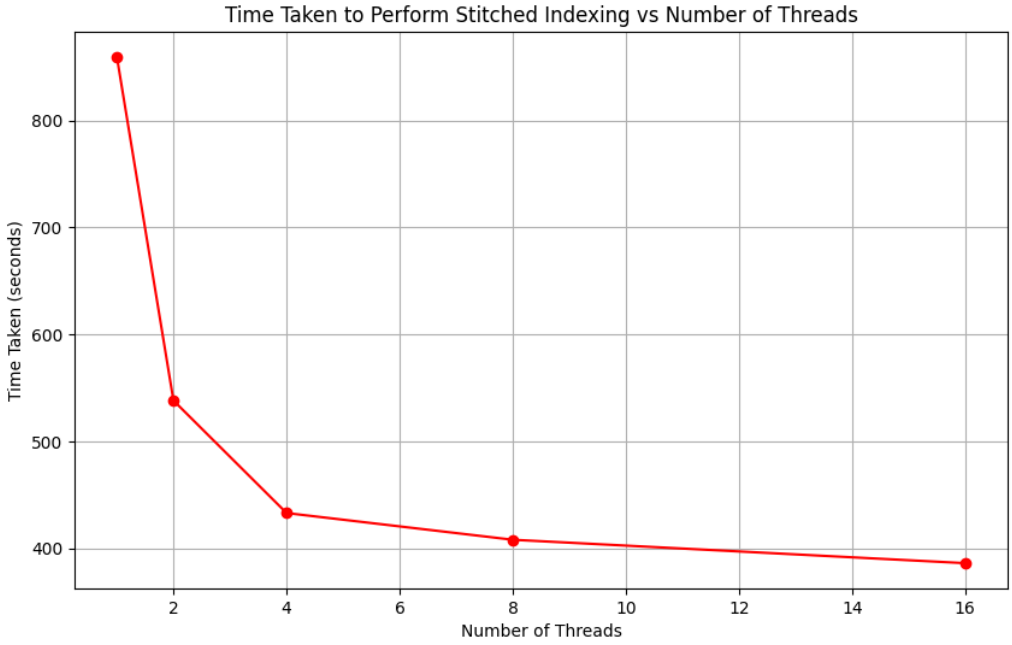
Stitched Vamana

Όπως εξηγήσαμε παραπάνω, ο αλγόριθμος του Stitched Vamana Indexing δημιουργεί για κάθε φίλτρο του έναν υπογράφο βαθμού R_small. Τα σύνολα των σημελιων των φίλτρων είνα ξένα μεταξύ, αφού καθε σημείο έχει ένα μόνο φίλτρο και συνεπώς οι υπογράφοι. Άρα,η συνάρτηση Indexing παραλληλοποιείται σχετικά εύκολα και αποδοτικά χωρίς ανάγκη συγχρονισμού. Με χρήση openMP και κάνοντας φθίνουσα ταξινόμηση στα φίλτρα με βάση το πλήθος των σημείων τους (P_f), πετύχαμε fine-grained παραλληλοποίηση με schedule(static,1) καθώς τα πιο χρονοβόρα φίλτρα εκτελούνται στην αρχή.

- a: 1.2
- L_{small}: 32
- R_{small}: 50



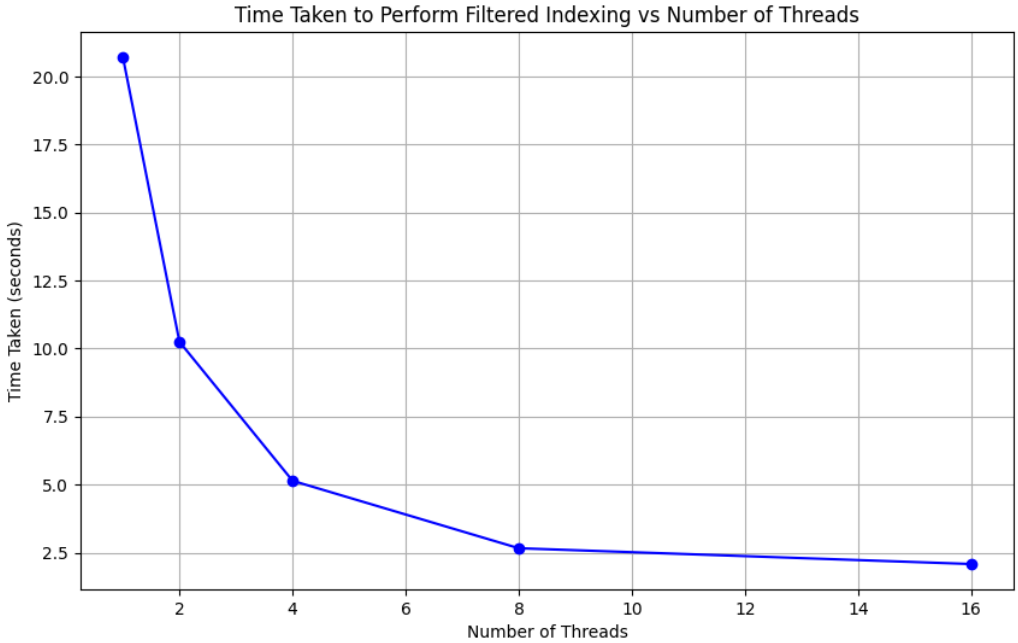
Αρχείο εισόδου 10.000 σημείων, 129 φίλτρα συνολικά (dummy-data.bin).



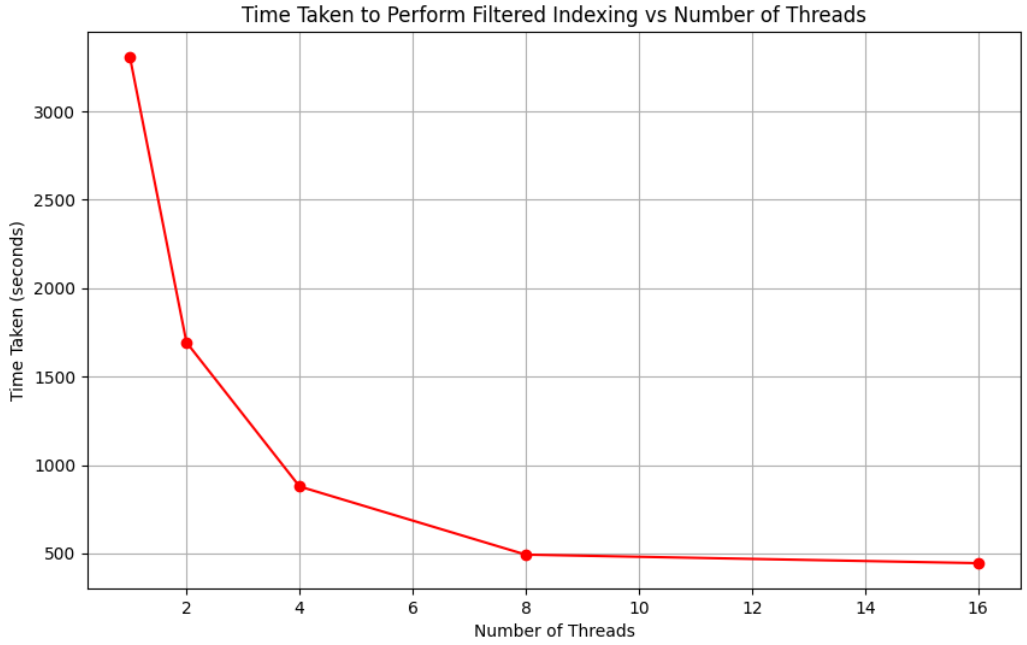
Αρχείο 1.000.000 σημείων, 1142 φίλτρα συνολικά (contest-data-release-1m.bin).

Filtered Vamana

Παρομοίως και εδώ, παραλληλοποιήσαμε τον αλγόριθμο. Δεν ήταν τόσο απλό όσο το 'stitched' καθώς απαιτεί συγχρονισμό. Ο 'filtered' φτιάχνει ανεξάρτητους υπογράφους, αλλά με αυξητικό τρόπο πάνω στον ίδιο γράφο, οπότε πρέπει να προστατέψουμε τα critical sections. Συγκριτικά με τον σειριακό βελτιώνεται πολύ, αλλά πάλι παραμένει πιο αργό.



Αρχείο εισόδου 10.000 σημείων, 129 φίλτρα συνολικά (dummy-data.bin).



Αρχείο 1.000.000 σημείων, 1142 φίλτρα συνολικά (contest-data-release-1m.bin).

Δομές Δεδομένων

Με βάση τους αλγορίθμους των papers, σε θεωρητικό επίπεδο οι μεταβλητές L και V στις GreedySearch και FilteredGreedySearch παρουσιάζονται ως sets. Έτσι και εμείς σε αρχικό στάδιο χρησιμοποιούσαμε `std::unordered_set` για να αποθηκεύουμε τους κόμβους. Η δομή αυτή δεν ταίριαζε και τόσο στην περίπτωση μας γιατί μας ενδιέφερε η ταξινόμηση και το overhead του hash table ήταν μεγάλο, επίσης δεν χρειαζόμασταν γρήγορο lookup. Γενικά μέχρι και το 2ο παραδοτέο, δεν είχαμε optimized κώδικα και η ταξινόμηση και η εύρεση του min και k_closest έπαιρναν πολύ χρονο, αφού έπρεπε να κάνουμε αντιγραφή σε vectors και μετά ταξινομηση. Μετά μέσω profiling είδαμε ότι το μεγαλύτερο μέρος της εκτέλεσης γινόταν σε αυτά τα σημεία και δοκιμάσαμε άλλες δομές. Στην αρχή `std::vector` με ταξινόμηση βάσει απόστασης και χρήση `std::unique` για απαλοιφή διπλότυπων. Έδωσε σχεδόν x3 επιτάχυνση στο πρόγραμμά μας. Έπειτα, δοκιμάσαμε την `std::set`, με δικιά μας συνάρτηση σύγκρισης βάσει απόστασης, που τελικά ήταν πιο αποδοτική από την `std::vector`, καθώς ταίριαζε περισσότερο στην περίπτωσή μας. Γι'αυτό υπάρχουν 2 υλοποιήσεις των αλγορίθμων, αυτές με τα _s είναι που χρησιμοποιούν sets.

Παρατηρήσεις-Συμπεράσματα Πειραμάτων

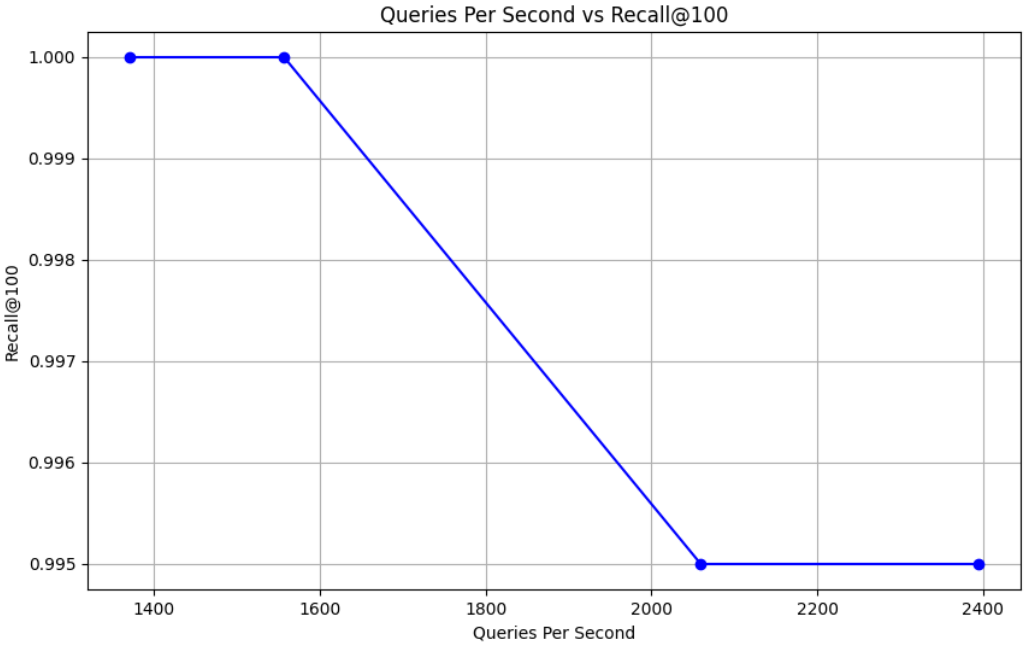
Πειραματιστήκαμε με διάφορες παραμέτρους. Προσθέσαμε μια καινούρια παράμετρο `-u` που είναι για το querying στα unfiltered σημεία. Αυτό που κάνει είναι να ορίζει το k και το L στις επιμέρους greedy_search που γίνονται για κάθε φίλτρο, όταν έχουμε unfiltered query. Μεγαλύτερες τιμές μπορούν να βελτιώσουν αρκετά το recall τους, έως και 10%. Έτσι καταλήξαμε στο ότι γίνεται να βελτιστοποιήσουμε το tradeoff μεταξύ χρόνου κατασκευής του γράφου και χρόνου εκτέλεσης ερωτημάτων. Δίνοντας μεγαλύτερες τιμές για τα k,L στο querying μπορείς να έχει ενα πιο μικρό γράφο με καλό recall, αλλά μικρότερο QPS. Οπότε ανάλογα το use case επιλέγεις τι χρειάζεσαι. Για την παράμετρο a, η τιμή 1.1 φάνηκε πιο ισορροπημένη, γενικά η αύξηση του χειροτέρευε τον χρόνο του indexing χωρις απαραίτητα να βελτιώνει το recall.

Τέλος, παραθέτουμε παρακάτω γραφήματα σύγκρισης **recall@100** και **QPS**.

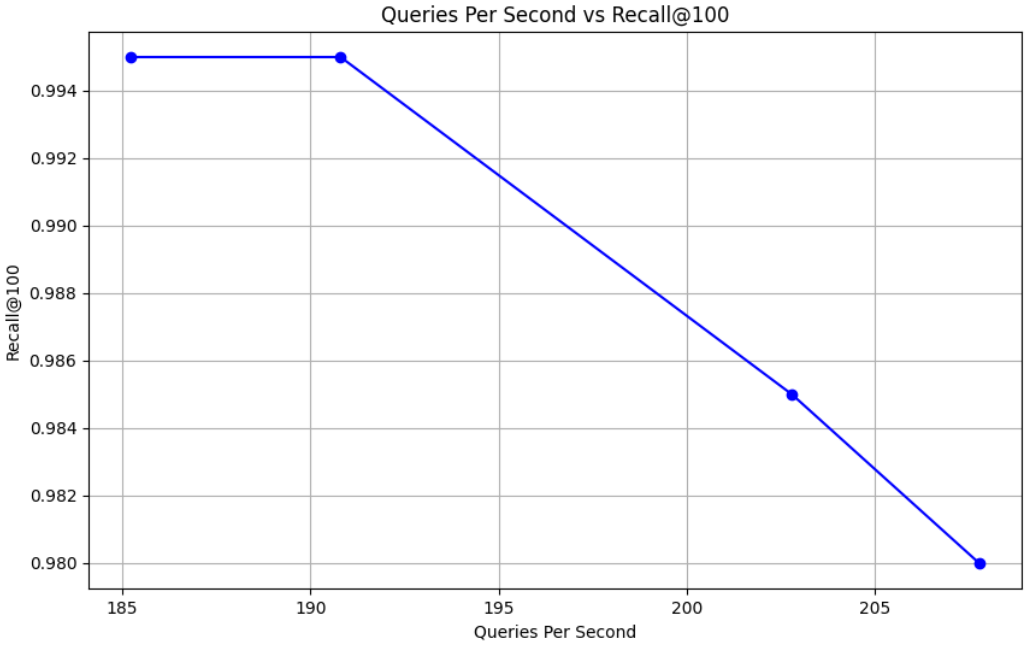
- **Filtered Vamana**

  Οι εκτελέσεις έγιναν χρησιμοποιώντας τις εξής παραμέτρους:

  - **a:** 1,2
  - **k:** 100
  - **R:** 32
  - Το **L** πήρε τις τιμές **{180,160,120,100}** αντίστοιχα.
  - **t:** 1
  - **16 threads**
  - **L_unfiltered (-u) :**10



Αρχείο εισόδου 10.000 σημείων (dummy-data.bin).

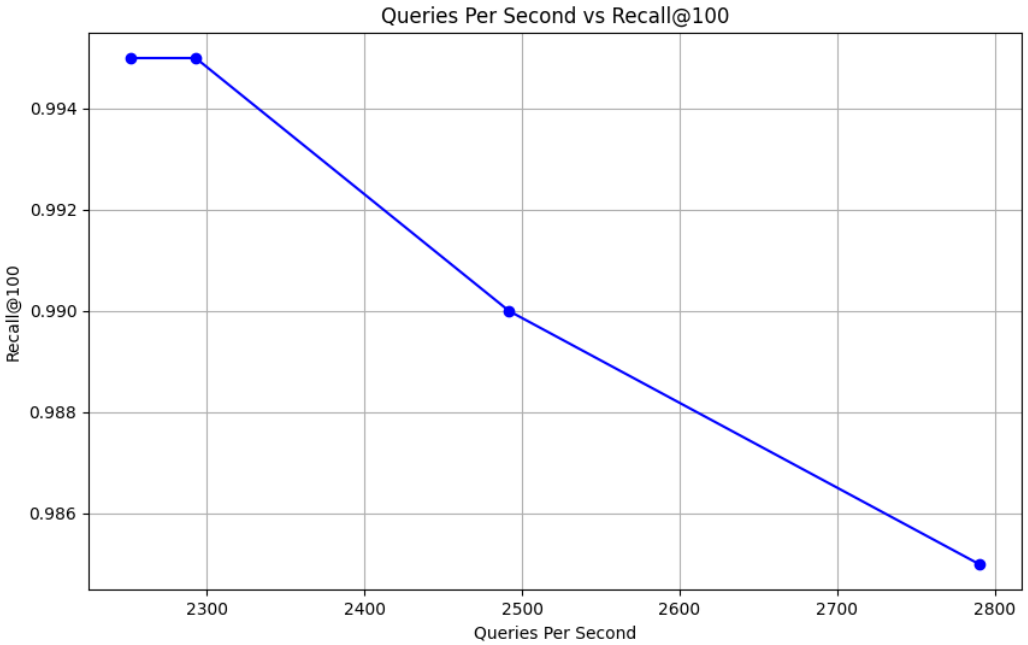


Αρχείο 1.000.000 σημείων (contest-data-release-1m.bin).

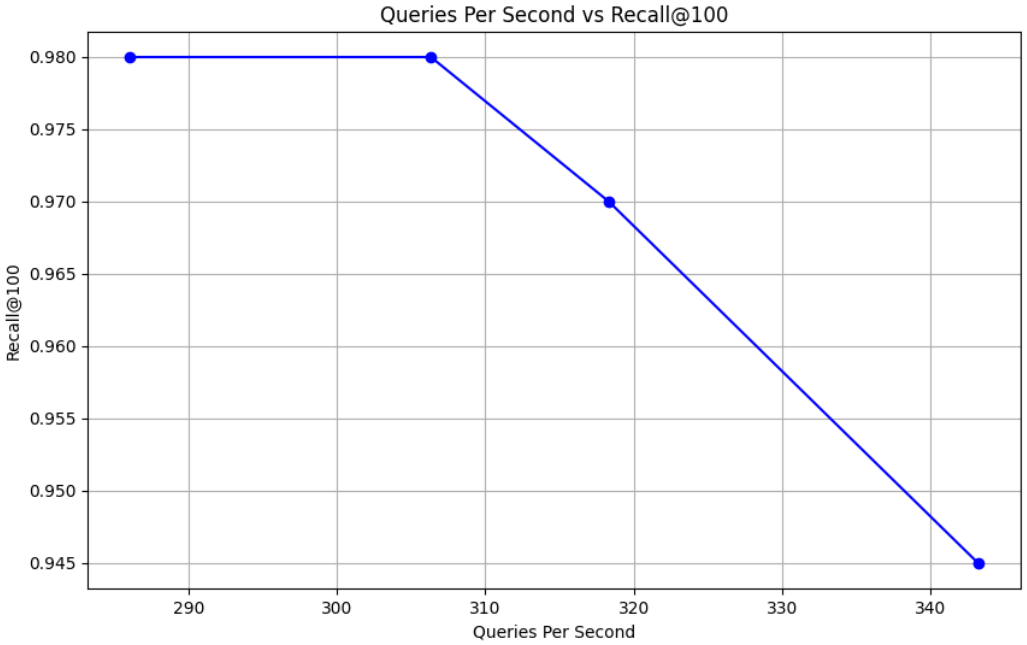
- **Stitched Vamana**

  Οι εκτελέσεις έγιναν χρησιμοποιώντας τις εξής παραμέτρους:

  - **a:** 1,2
  - **k:** 100
  - $\mathbf{R_{small}}$**:** 32
  - Το $\mathbf{L_{small}}$ πήρε τις τιμές **{96,64,32,20}** αντίστοιχα.
  - **16 threads**
  - **L_unfiltered (-u) :**10



Αρχείο εισόδου 10.000 σημείων (dummy-data.bin).

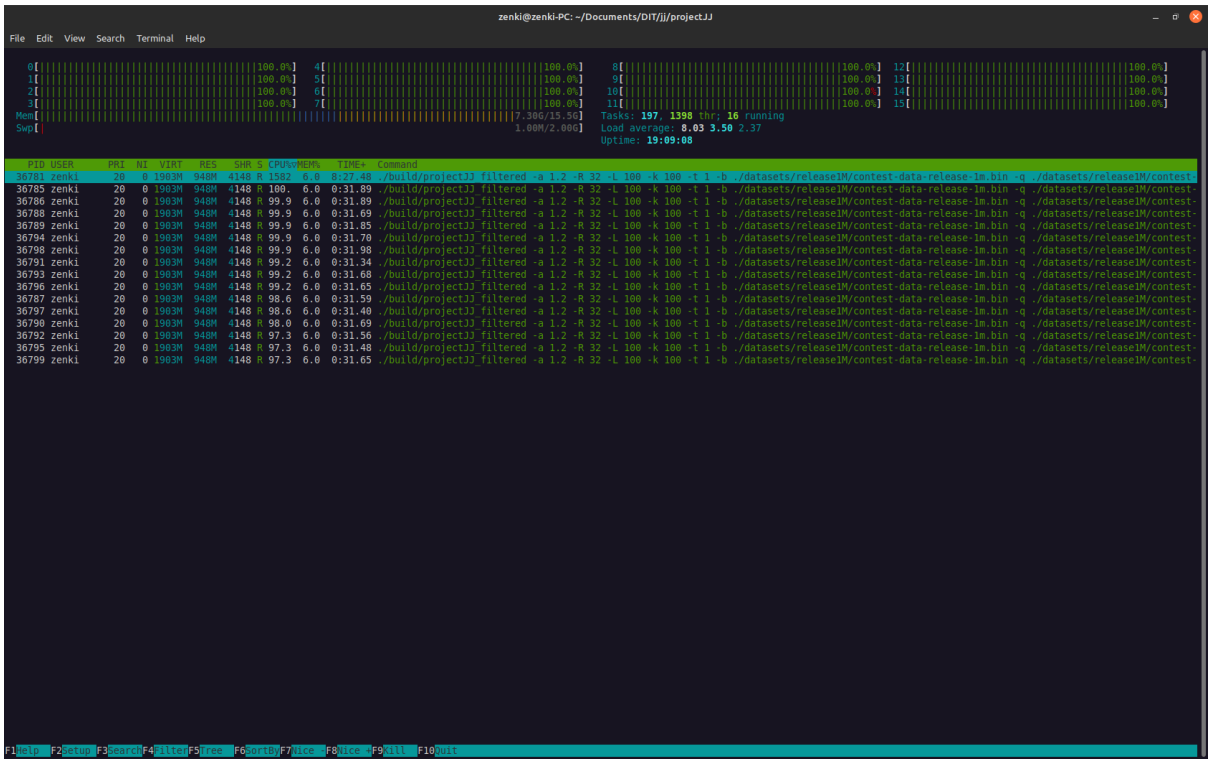


Αρχείο 1.000.000 σημείων (contest-data-release-1m.bin).

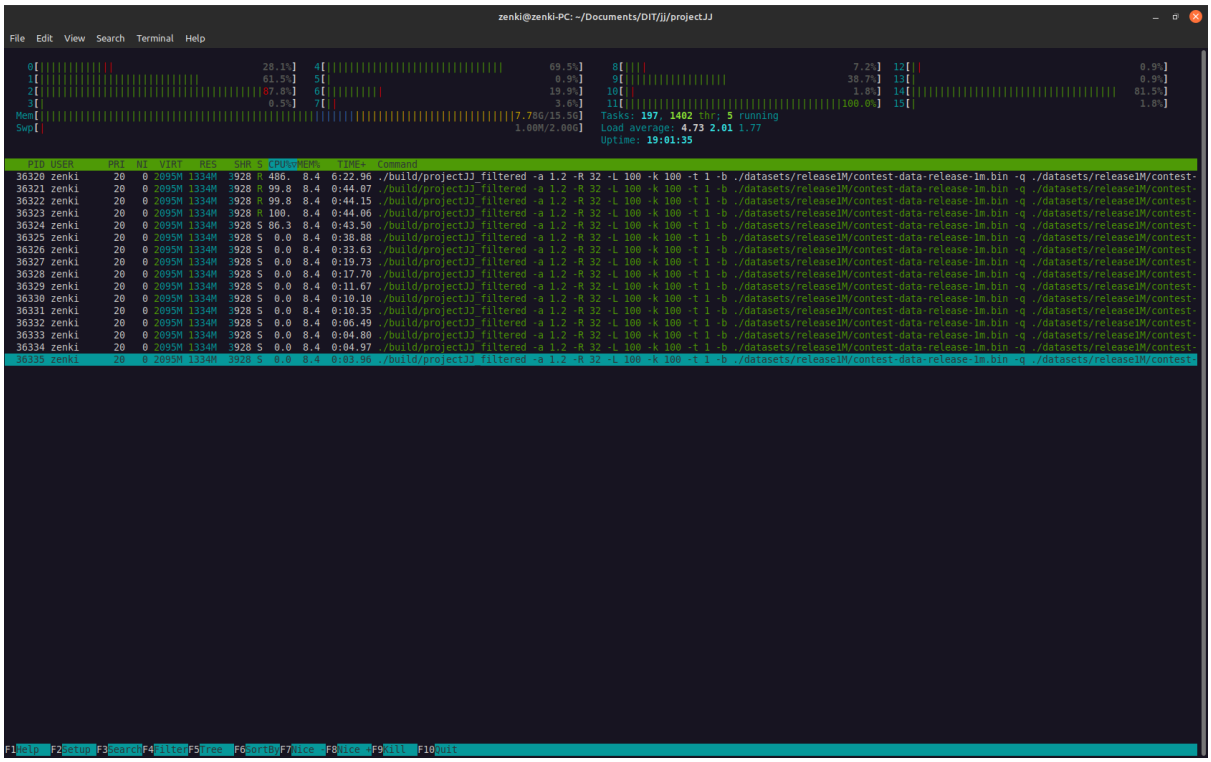
Σημείωση: Ως recall@100 παραπάνω παρουσιάσαμε τον Μ.Ο. των Filtered Recall@100 και Unfiltered Recall@100 για κάθε εκτέλεση και στο QPS μετράμε τόσο filtered όσο και unfiltered queries.

Μνήμη

Παρακάτω παραθέτουμε δύο εκτελέσεις htop, η μια για Filtered Vamana και Stitched Vamana αντίστοιχα με 16 threads.



Αρχείο 1.000.000 σημείων (contest-data-release-1m.bin), a:1.2 k:100 R:32 L:100 t:1 u:10.



Αρχείο 1.000.000 σημείων (contest-data-release-1m.bin), a:1.2 k:100 R_{small}:32 L_{small}:50.