ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Βελτιστοποίηση Ερωτημάτων Συσχετίσεων με Εφαρμογές στη Βιοϊατρική Βιβλιογραφία

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γιώργος Δ. Κεφάλας

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Μάρτιος 2020

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Υπολογιστικών Συστημάτων

Βελτιστοποίηση Ερωτημάτων Συσχετίσεων με Εφαρμογές στη Βιοϊατρική Βιβλιογραφία

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γιώργος Δ. Κεφάλας

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 10η Μαρτίου 2020.

.....	.....	.....
Νεκτάριος Κοζύρης, Καθηγητής ΕΜΠ	Γκούμας Γεώργιος. Επικ. Καθηγητής ΕΜΠ	Τσουμάκος Δημήτριος. Αναπ. Καθηγητής, Ιόνιο Παν.

(Αθήνα, Μάρτιος 2020)

.....
Γιώργος Δ. Κεφάλας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η σύγχρονη τάση στον τομέα της μηχανικής μάθησης και της εξόρυξης δεδομένων οδηγεί στην ανάγκη για καλύτερες επιδόσεις στα προβλήματα αναλυτικής επεξεργασίας άμεσης επικοινωνίας (OLAP scenarios). Οι συμβατικές βάσεις δεδομένων αποθήκευσης κατά γραμμή και κατά στήλη, καθώς και οι βάσεις γράφων, δεν μπορούν να ανταποκριθούν ικανοποιητικά στις απαιτήσεις των προβλημάτων αυτών.

Στην εργασία αυτή μελετάμε τη δομή και τη λειτουργία της GQ-Fast μιας βάσης δεδομένων βασισμένης σε δείκτες. Η GQ-Fast φτιάχτηκε για να ανταποκρίνεται σε μια κατηγορία ερωτημάτων που ονομάζουμε "ερωτήματα συσχετίσεων", τα οποία ασχολούνται με την ανάλυση γράφων σε μοτίβα δεντρικών δομών καθώς και στον έλεγχο της προσβασιμότητας μεταξύ διαφόρων κόμβων. Συγκεντρώνει τα πλεονεκτήματα της οργάνωσης κατά στήλη (column organizing), της δεικτοδότησης και της συμπίεσης δεδομένων, ενώ κατασκευάζει με έξυπνο τρόπο πίνακες γειτνίασης για να μοντελοποιήσει τα δεδομένα που της δίνονται. Το ιδιαίτερο χαρακτηριστικό της είναι η εξαγωγή πηγαίου κώδικα σε C++ για κάθε ένα από τα ερωτήματα που επιλύει. Τα ερωτήματα αυτά είναι πολύ σημαντικά για την επίλυση προβλημάτων αναλυτικής επεξεργασίας άμεσης επικοινωνίας (OLAP scenarios).

Στόχος μας είναι να περιγράψουμε τις λειτουργικές επεκτάσεις που προσθέσαμε στη GQ-Fast , καθώς και τις εφαρμογές που αυτές έχουν στην εξόρυξη δεδομένων στη Βιοϊατρική βιβλιογραφία. Για το σκοπό αυτό χρησιμοποιούμε τη βάση δεδομένων του εκδοτικού οίκου PubMed, επεξεργάζοντας τα δεδομένα της και εφαρμόζοντας σε αυτά μια ομάδα από ερωτήματα SQL τα οποία επιστρέφουν χρήσιμη ερευνητική πληροφορία με τρόπο σαφώς πιο αποδοτικό από τις συμβατικές μεθόδους που έχουμε για να την ανακαλύψουμε. Παραθέτουμε αναλυτικά πειραματικά δεδομένα εστιασμένα σε αλληλεπιδραστικά ερωτήματα που απαιτούν ταχύτατη απόκριση αναφορικά με τις επιδόσεις της GQ-Fast έναντι της βάσης PostgreSQL, όταν όλες οι βάσεις είναι φορτωμένες στη RAM του δοκιμαστικού συστήματος.

Επιδεικνύουμε τις παραπάνω επεκτάσεις μέσω ενός γραφικού περιβάλλοντος χρήστη (User Interface) που σχεδιάσαμε και υλοποιήσαμε για τον φυλλομετρητή ιστού. Στο περιβάλλον αυτό, εφαρμόζονται άμεσα οι δυνατότητες της GQ-Fast για την ταχύτατη ανάκτηση πληροφοριών σχετικά με την Βιοϊατρική βιβλιογραφία καθώς και η αποτελεσματική απεικόνισή τους.

Λέξεις Κλειδιά SQL, Relationship Queries, GQ-Fast, OLAP, C++, PubMed, Knowledge Discovery, Biomedical literature, Graph Discovery, Graph Exploration

Abstract

Recent developments in the fields of Knowledge Discovery and Data Mining lead to the need for greater perfomance in Online Analytical Processing (OLAP) scenarios. Conventional databases (row stores, column stores and graph databases) are unable to meet the expectations of modern OLAP scenarios.

We follow and extend the work on GQ-Fast , which is an indexed database that roughly corresponds to efficient encoding of annotated adjacency lists that combines salient features of column-based organization, indexing and compression. GQ-Fast is designed to execute a family of SQL queries which we call relationship queries. These queries involving aggregation, join, semi join, intersection and selection are a wide superset of fixed length graph reachability queries and of tree pattern queries. We present real-world OLAP scenarios, where efficient relationship queries are needed. For that purpose, we use the Biomedical Datasets of PubMed [1] and SemMed [2].

GQ-Fast uses a bottom-up fully pipelined query execution model, which enables (a) aggressive compression (e.g., compressed bitmaps and Huffman) and (b) avoids intermediate results that consist of row IDs (which are typical in column databases). GQ-Fast compiles query plans into executable C++ source code. Besides achieving runtime efficiency, GQ-Fast also reduces main memory requirements because, unlike column databases, GQ-Fast selectively allows dense forms of compression including heavy-weight compressions, which do not support random access.

Our aim is to contribute to the original work of GQ-Fast by adding new features and by re-designing some of its core functionalities. Through the modifications that we propose we are able to expand the set of supported queries and also improve runtime efficiency. These additions allow as to accelerate new and more complex queries for PubMed's OLAP dashboard, offering useful scientific insight in the processing of information in the Biomedical field. Our experiments show how GQ-Fast outperforms PostgreSQL by 2–4 orders of magnitude.

We present the efficiency of GQ-Fast through a Demo Browser Application that we designed. In the User Interface of our application, we utilize GQ-Fast's functionalitites in order to retrieve and process useful information concerning Biomedical Literature.

Keywords SQL, Relationship Queries, GQ-Fast, OLAP, C++, PubMed, Knowledge Discovery, Biomedical literature, Graph Exploration

[1] PubMed comprises more than 30 million citations for biomedical literature from MEDLINE, life science journals, and online books.

[2] https://skr3.nlm.nih.gov/SemMed/

Contents

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Η σύγχρονη τάση για αποθήκευση και αξιοποίηση όλο και μεγαλύτερου όγκου δεδομένων, ειδικά για λόγους όπως η εξόρυξη δεδομένων και η αναζήτηση γνώσης, οδηγεί στην ανάγκη για καλύτερες επιδόσεις στα προβλήματα αναλυτικής επεξεργασίας άμεσης επικοινωνίας (OLAP scenarios).

Η αναζήτηση γνώσης σε πεδία όπως η Βιοϊατρική βιβλιογραφία απαιτεί, συγκεκριμένα, τη βελτιστοποίηση της εκτέλεσης ερωτημάτων αναλυτικής επεξεργασίας πάνω σε βιβλιογραφικούς γράφους, όπως ο γράφος γνώσης της PubMed που χρησιμοποιούμε στην εργασία αυτή.

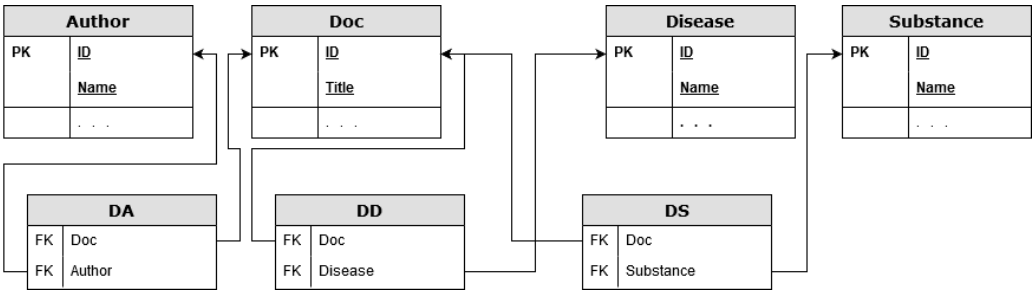
Χαρακτηριστικά στοιχεία αυτού το είδους των γράφων είναι ότι οι κόμβοι και οι ακμές τους μπορούν να αναπαρασταθούν ως πλειάδες πινάκων σε ένα σχεσιακό σχήμα μιας βάσης δεδομένων. Διακρίνουμε τους πίνακες αυτούς σε δύο κατηγορίες: τους πίνακες Οντοτήτων (Entity tables) και τους Συσχετίσεων (Relationship tables), ακολουθώντας το μοντέλο Οντοτήτων-Συσχετίσεων από τη θεωρία των βάσεων δεδομένων.

Μέσω της επέκτασης και αξιοποίησης των δυνατοτήτων της GQ-Fast , κατορθώνουμε να εκτελέσουμε ερωτήματα αναλυτικής επεξεργασίας πολύ γρήγορα, επιτρέποντας επομένως τη κατασκευή ενός συστήματος που επιτρέπει την άμεση αλληλεπίδραση ενός χρήστη με μία μηχανή αναζήτησης και αναλυτικής παρουσίασης χρήσιμης ερευνητικής πληροφορίας. Ένα σημαντικό σύνολο από τις δυνατότητες αυτές παρουσιάζεται στο δοκιμαστικό περιβάλλον χρήστη που αναπτύξαμε.

Κεφάλαιο 2

Περιγραφή του **PubMed Schema**

Η βάση δεδομένων PubMed οδηγεί σε ένα σχεσιακό σχήμα που ταιριάζει ακριβώς στη παραπάνω περιγραφή που δώσαμε για έναν γράφο γνώσης με οντότητες (κόμβους) και συσχετίσεις (ακμές). Αν και το πραγματικό σχήμα της βάσης δεδομένων που εξάγουμε από τα δεδομένα της PubMed είναι πολύ σύνθετο, στο σημείο αυτό παραθέτουμε ένα απλοποιημένο σχήμα το οποίο αρκεί για τις λειτουργίες που παρουσιάζουμε στη συνέχεια και έχει ως στόχο τη διευκόλυνση του αναγνώστη.



Σχήμα 2.1: Το (απλοποιημένο) σχήμα της βάσης PubMed.

Στο σχήμα 2.1 διακρίνουμε:

1. Τους Πίνακες Οντοτήτων:
   (a) Author
   (b) Doc
   (c) Disease
   (d) Substance
2. Τους Πίνακες Συσχετίσεων:
   (a) DA (Document - Authors)

(b) DD (Document - Diseases)

(c) DS (Document - Substances)

Συνολικά, τα 29 εκατομμύρια άρθρα που φιλοξενούνται στους διακομιστές της PubMed, μαζί με τους συγγραφείς και τους αυτόνομους όρους (ασθένειες ή χημικές ουσίες), αποτελούν τις οντότητες του σχήματός μας. Τις οντότητες συνδέουν συγκεκριμένες συσχετίσεις που φαίνονται στο σχήμα 2.1:

1. Document - Authors: κάθε εγγραφή αυτού του πίνακα αναπαριστά μία αντιστοιχία ενός συγγραφέα με ένα επιστημονικό άρθρο.

2. Document - Diseases: κάθε εγγραφή αυτού του πίνακα αναπαριστά μία αντιστοιχία μεταξύ ενός άρθρου και μιας ασθένειας στην οποία αναφέρεται.

3. Document - Substance: κάθε εγγραφή αυτού του πίνακα αναπαριστά μια αντιστοιχία μεταξύ ενός άρθρου και μιας χημικής ουσίας στην οποία αναφέρεται.

Σε κάποια σημεία της εργασίας, για μεγαλύτερη απλότητα, οι πίνακες Disease και Substance αντικαθίστανται από τον πινακα Term (ο οποίος περιέχει όλη την ορολογία της PubMed χωρίς να διακρίνει ανάμεσα σε ασθένειες και χημικές ουσίες). Σε αυτή την περίπτωση, οι πίνακες DD και DS αντικαθίστανται από τον πίνακα DT ο οποίος περιέχει όλες τις συσχετίσεις μεταξύ documents και terms.

Πάνω στους παραπάνω πίνακες μπορούμε να σχεδιάσουμε πολλά ενδιαφέροντα ερωτήματα συσχέτισης, και ακριβώς αυτό κάνουμε στην επόμενη ενότητα όπου παρουσιάζουμε αποτελέσματα από πραγματική ανάλυση και επεξεργασία δεδομένων.

Κεφάλαιο 3

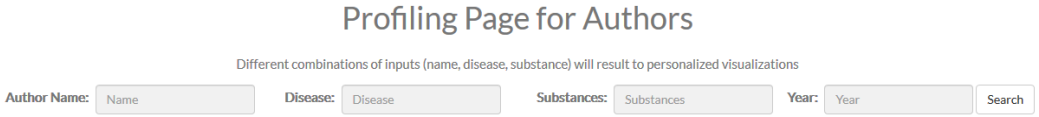
(Demo UI)

Στην ενότητα αυτή παρουσιάζουμε τις δυνατότητες που προσθέσαμε στη GQ-Fast μέσω της εφαρμογής τους σε ένα δοκιμαστικό σύστημα που αναπτύξαμε για το φυλλομετρητή ιστού. Το σύστημα αυτό επιτρέπει την αναζήτηση πληροφοριών μέσα από τον γράφο γνώσης της PubMed, καθώς και την απεικόνιση της ζητούμενης πληροφορίας με τρόπου που διευκολύνει την ανάλυσή της.

Συγκεκριμένα, παραθέτουμε στιγμιότυπα από τις διάφορες δυνατότητες του συστήματός μας σχετικά με την αναζήτηση πληροφοριών σχετικά με κάποιον συγγραφέα, σε συνδυασμό (ή όχι) με κάποια ασθένεια ή χημική ουσία.

3.1 Εισαγωγή Στοιχείων για Αναζήτηση και Αυτόματη Συμπλήρωση

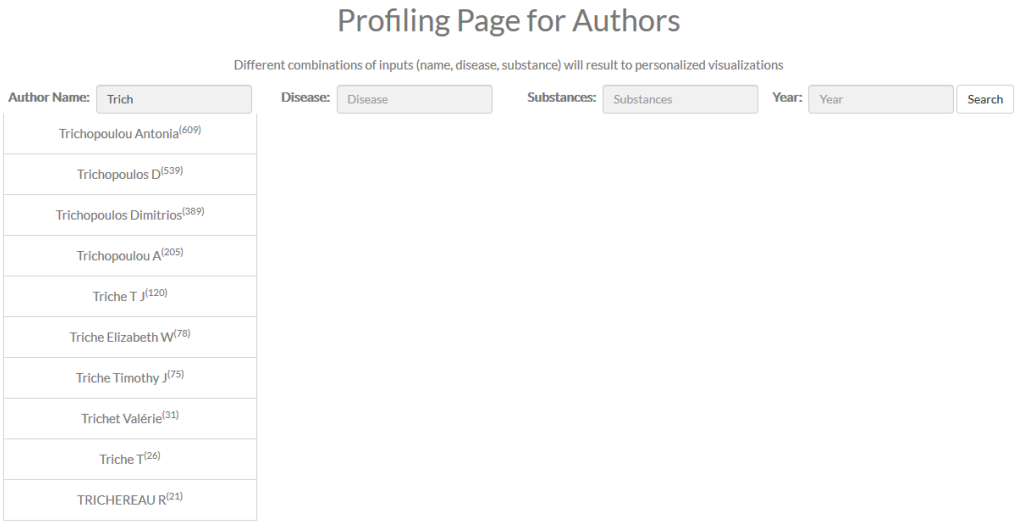
Η αρχική σελίδα που βλέπει ο χρήστης όταν ανοίγει την εφαρμογή, φαίνεται στην εικόνα 3.1:



Σχήμα 3.1: Αρχική σελίδα της εφαρμογής μας. PubMed.

Θα επικεντρωθούμε στη περίπτωση χρήσης όπου ο χρήστης ενδιαφέρεται για την ανακάλυψη πληροφοριών σχετικά με έναν συγγραφέα. Στη φόρμα αναζήτησης της εικόνας υπάρχει λειτουργία αυτόματης συμπλήρωσης η οποία εξαρτάται από τη GQ-Fast για την ταχύτητά της. Πιο συγκεκριμένα, για κάθε πλήκτρο που πατάει ο χρήστης κατά τη συμπλήρωση ενός πεδίου, επιλέγεται (ανάλογα με τις εισόδους των υπόλοιπων

πεδίων) το κατάλληλο ερώτημα (ερώτημα συσχέτισης για τη βάση δεδομένων της PubMed) που πρέπει να εκτελεστεί. Για να είναι το σύστημα εύχρηστο, είναι απαραίτητο να αξιοποιηθεί η GQ-Fast αντί για μία συμβατική βάση δεδομένων.



Σχήμα 3.2: Λίστα Autocomplete με πιθανά αποτελέσματα για το πεδίο Author.

Στην εικόνα 3.2: διακρίνουμε τα εξής:

1. Ο χρήστης συμπλήρωσε στο πεδίο "Author Name" το πρόθεμα "Trichopoul".

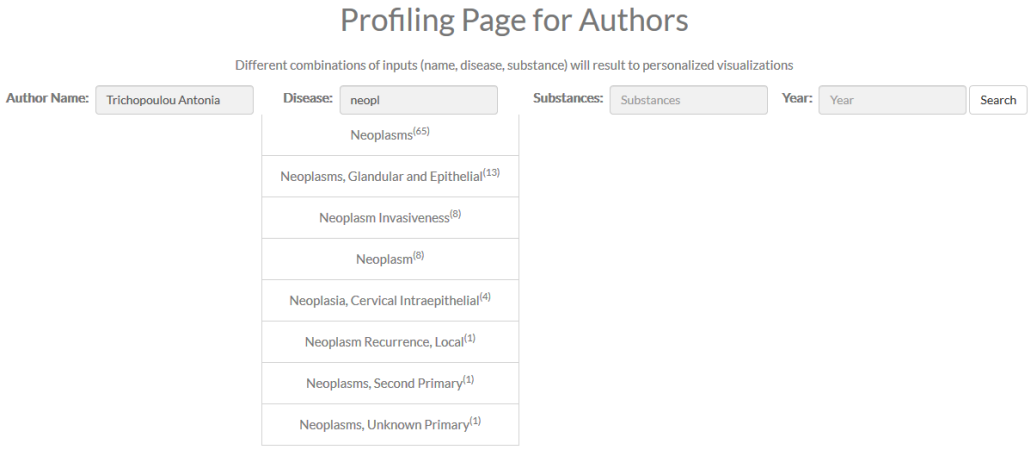
2. Η εφαρμογή μας (μέσω του API που χρησιμοποιεί) έστειλε αίτημα αντίστροφης αναζήτησης με το αλφαριθμητικό πρόθεμα που έδωσε ο χρήστης σε έναν διαχειριστή λεξικού που έχουμε κατασκευάσει για να απαντά σε αυτού του είδους τα ερωτήματα.

3. Απάντηση στο ανωτέρω αίτημα είναι μία λίστα με ονόματα συγγραφέων που ξεκινάνε με το πρόθεμα του χρήστη. Η λίστα αυτή είναι ταξινομημένη όχι αλφαβητικά, αλλά βάσει ενός δείκτη σημαντικότητας. Στη προκειμένη περίπτωση όπου δεν έχουν συμπληρωθεί άλλα πεδία της φόρμας, ο δείκτης σημαντικότητας για έναν συγγραφέα είναι απλώς ο αριθμός των δημοσιεύσεων που έχει κάνει.

Επιλέγοντας από τη λίστα τον επιθυμητό συγγραφέα, προχωράμε στη συμπλήρωση ενός άλλου όρου, όπως το πεδίο της ασθένειας, προκύπτει η εικόνα 3.3:

Στην νέα εικόνα (3.3) διακρίνουμε τα εξής:

1. Ο χρήστης συμπλήρωσε το πεδίο "Disease" με το πρόθεμα "neopl".



Σχήμα 3.3: Λίστα **Autocomplete** με πιθανά αποτελέσματα για το πεδίο **Disease**, όταν το πεδίο **Author** είναι ήδη συμπληρωμένο.

2. Όπως και προηγουμένως, εφαρμογή μας (μέσω του API που χρησιμοποιεί) έστειλε αίτημα αντίστροφης αναζήτησης με το αλφαριθμητικό πρόθεμα που έδωσε ο χρήστης.

3. Επειδή το πεδίο **author** είναι συμπληρωμένο, η λίστα με τα αποτελέσματα από την αντίστροφη αναζήτηση στο λεξικό των ασθενειών, δίνεται, μαζί με τον επιλεγμένο συγγραφέα, στη GQ-Fast για να εκτελεστεί το κατάλληλο ερώτημα που θα υπολογίσει τους δείκτες σημαντικότητας. Σε αυτή τη περίπτωση, δείκτης σημαντικότητας για κάθε ασθένεια είναι το πλήθος των εμφανίσεών της σε άρθρο του επιλεγμένου συγγραφέα. Παραθέτουμε σε SQL το ερώτημα συσχέτισης που επιστρέφει τα αποτελέσματα με τον ζητούμενο δείκτη σημαντικότητας:

SQL για το ερώτημα που επιστρέφει την **Autocomplete** λίστα για τις ασθένειες**.**

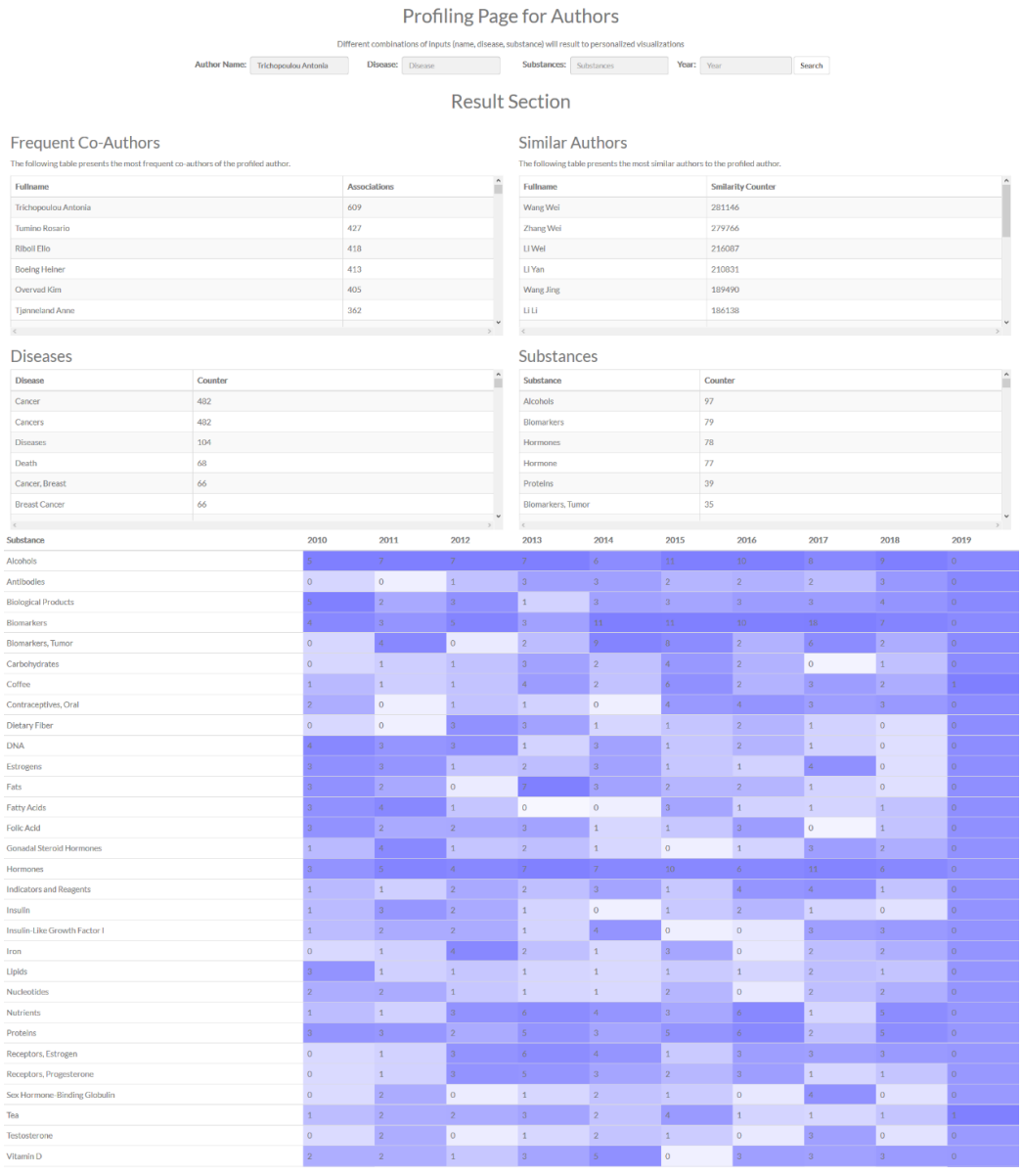
```
SELECT d.Name, COUNT(*)
FROM DD dd
JOIN Disease d ON dd.Disease = d.Id
JOIN DA da ON dd.Doc = da.Doc
JOIN Author auth ON da.Author = auth.Id
WHERE LOWER(d.Name) LIKE 'neopla%'
  AND LOWER(auth.NAME) LIKE LOWER('Trichopoulou Antonia%')
GROUP BY sd.Id
ORDER BY COUNT DESC
```

Με όλα τα απαραίτητα ευρετήρια για την αύξηση της απόδοσης της PostgreSQL υλοποιημενα και χρησιμοποιημένα:

- Στη PostgreSQL το παραπάνω ερώτημα εκτελείται σε: 4,742 ms.
- Στη GQ-Fast το παραπάνω ερώτημα εκτελείται σε: 432 ms.

3.2 Περιοχή των Αποτελεσμάτων

Από τη στιγμή που συμπληρώνεται ένα πεδίο, αυτόματα εμφανίζονται και οι πίνακες των αποτελεσμάτων από διάφορα χρήσιμα ερωτήματα συσχέτισης. Συγκεκριμένα, με το που επιλέχθηκε από τη λίστα αυτόματης συμπλήρωσης η συγγραφέας "Trichopoulou Antonia", εμφανίζεται η εικόνα 3.4.



Σχήμα 3.4: Τελική Σελίδα της Εφαρμογής όπου ο χρήστης έχει συμπληρώσει τα πεδία που επιθυμεί και εμφανίζονται τα αποτελέσματα.

Στην εικόνα 3.4 βλέπουμε τέσσερις πίνακες και ένα Heat Map. Πιο συγκεκριμένα:

1. Frequent Co-Authors: Πρόκειται για τους συγγραφείς με τους οποίους έχει συνεργαστεί πιο συχνά ο συγγραφέας του πεδίου Author Name. Το ερώτημα συσχέτισης που εκτελείται για την παραγωγή αυτού του πίνακα, σε SQL, είναι το εξής:

```
SELECT Name, COUNT(*)
FROM Author auth JOIN DA da ON auth.Id = da.Author
     JOIN DA da2 ON da.Article = da2.Article
     JOIN Author auth2 ON da2.Author = auth2.Author
WHERE auth.Name = 'Trichopoulou Antonia'
GROUP BY auth2.Id
ORDER BY count DESC LIMIT 100
```

2. Similar Authors: Αφορά τους συγγραφείς που έχουν χρησιμοποιήσει (σε μεγαλύτερο πλήθος εμφανίσεων) ορολογία την οποία χρησιμοποιεί ο επιλεγμένος συγγραφέας στις δημοσιεύσεις του. Το σχετικό ερώτημα σε SQL είναι το εξής:

```
SELECT auth2.NAME, COUNT(*)
FROM Author auth JOIN DA da ON auth.Id = da.Author
    JOIN DT a2s1 ON da.Doc = a2s1.Doc
    JOIN DT a2s2 ON a2s1.Term = a2s2.Term
    JOIN DA da2 ON a2s2.Doc = da2.Doc
    JOIN Author auth2 ON da2.Author = auth2.Id
WHERE auth.NAME = 'Trichopoulou Antonia'
GROUP BY auth2.NAME
ORDER BY count DESC LIMIT 100
```

*Σημείωση: εδώ αντί των πινάκων Substance, Disease και DD, DS, χρησιμοποιούμε το απλοποιημένο σχήμα όπου υπάρχουν μόνο οι πίνακες Term και DT (Document-Term). Αυτό το σχεσιακό σχήμα φαίνεται στην εικόνα 4.1.

3. Diseases: Τα περιεχόμενα αυτού του πίνακα εξαρτώνται από τα πεδία της φόρμας αναζήτησης που έχουν συμπληρωθεί. Στη γενική περίπτωση που έχει επιλεγεί μόνο το όνομα ενός συγγραφέα, τότε η πληροφορία που εμφανίζεται αφορά τις ασθένειες που εμφανίζονται πιο συχνά στα άρθρα του εν λόγω συγγραφέα. Σε SQL, το σχετικό ερώτημα συσχέτισης είναι το εξής:

```
SELECT d.Name, COUNT(*)
FROM Disease d JOIN DS ds on d.Id = ds.Disease
JOIN DA da on ds.Doc = da.Doc
JOIN Author auth on auth.Id = da.Author
WHERE authors.name = 'Trichopoulou Antonia'
GROUP BY d.Id
ORDER BY count desc LIMIT 100;
```

4. Substances: Τα περιεχόμενα αυτού του πίνακα εξαρτώνται από τα πεδία της φόρμας αναζήτησης που έχουν συμπληρωθεί. Στη γενική περίπτωση που έχει επιλεγεί μόνο το όνομα ενός συγγραφέα, τότε η πληροφορία που εμφανίζεται αφορά τις ασθένειες που εμφανίζονται πιο συχνά στα άρθρα του εν λόγω συγγραφέα. Σε SQL, το σχετικό ερώτημα συσχέτισης είναι το εξής:

```
SELECT s.Name, COUNT(*)
FROM Substance s JOIN DS ds on s.Id = ds.Substance
JOIN DA sa on ds.Doc = da.Doc
JOIN Author auth on auth.Id = da.Author
WHERE auth.name = 'Trichopoulou Antonia'
GROUP BY s.Id
ORDER BY count desc LIMIT 100;
```

5. Heat Map: Στο Heat Map εμφανίζονται τα συχνότερα συβστανσες που εμφανίζονται στις δημοσιεύσεις του επιλεγμένου συγγραφέα για τα τελευταία δέκα χρόνια. Ο χρωματισμός κάθε κελιού (πόσο έντονο είναι το χρώμα του) εξαρτάται από τη σύγκριση του αριθμητικού του περιεχομένου (πλήθος εμφανίσεων του εκάστοτε όρου σε δημοσιεύσεις της αντίστοιχης χρονιάς που ορίζουν οι στήλες του πίνακα) με τον μέσο όρο της στήλης. Σε SQL, το ερώτημα από το οποίο προκύπτουν οι πληροφορίες του Heat Map είναι το εξής:

```
SELECT d.Name, year, COUNT(*)
FROM Disease d JOIN DD dd on d.Id = dd.Disease
  JOIN DA da on ds.Doc = da.Doc
    JOIN Author auth on auth.Id = da.Author
    JOIN Doc dc on da.Doc = dc.Id
WHERE auth.Name = 'Trichopoulou Antonia' AND dc.Year > 2000
GROUP BY d.Name, dc.Year
ORDER BY count desc LIMIT 100
```

Εισαγωγή μιας Ασθένειας: Χρησιμοποιούμε τη φόρμα αναζήτησης, προσθέτοντας τον όρο "Neoplasms" στο πεδίο Disease. Αυτόματα, ενημερώνονται οι πίνακες των αποτελεσμάτων και συγκεκριμένα οι πίνακες Substances (3.5) και Diseases (3.6).

Ανάλογα με τις τιμές που εισάγει ο χρήστης στη φόρμα αναζήτησης, τα ερωτήματα που πρέπει να τρέξει μια βάση δεδομένων σε SQL μπορεί να γίνουν πολύ χρονοβόρα, καθιστώντας απαγορευτική τη χρήση μιας συμβατικής βάσης δεδομένων για την υποστήριξη αυτής της εφαρμογής. Χρησιμοποιώντας τη GQ-Fast , κατορθώνουμε να υπολογίσουμε και να εμφανίσουμε όλες τις παραπάνω πληροφορίες σε λιγότερο από 1 δευτερόλεπτο, ξεπερνώντας έτσι σε ταχύτητα τη PostgreSQL έως και 10 φορές.

Substances

Substance	Counter
Alcohols	15
Biomarkers	9
Nutrients	5
Indicators	5
Nutrient	5
Olive Oil	5
Oil, Olive	4

Σχήμα 3.5: Ο πίνακας με τις σχετικές χημικές ουσίες, μετά την ανανέωση της σελίδας εξαιτίας της προσθήκης του όρου "neoplasms" στο σχετικό πεδίο της φόρμας αναζήτησης.

Diseases

Disease	Counter
Neoplasms	65
Cancer	63
Cancers	63
Death	18
Cardiovascular Diseases	18
Diseases	17
Chronic Disease	11

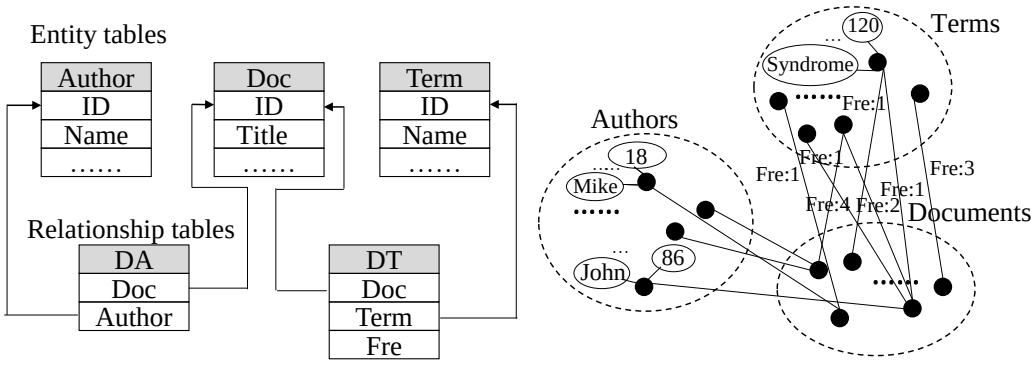
Σχήμα 3.6: Ο πίνακας με τις σχετικές ασθένειες, μετά την ανανέωση της σελίδας εξαιτίας της προσθήκης του όρου "neoplasms" στο σχετικό πεδίο της φόρμας αναζήτησης.

Κεφάλαιο 4

Τεχνολογικό Υπόβαθρο

Παλαιότερα, τα συστήματα Αναλυτικής Επεξεργασίας των Δεδομένων (Online Analytical Processing – OLAP) απευθύνονταν σε SQL ερωτήματα πάνω σε κύβους δεδομένων, η πληροφορία των οποίων οργανώνεται σε σχήμα Αστέρα (Star schema) ή Χιονονιφάδας (Snowflake schema) [7, 28]. Τα τελευταία χρόνια ωστόσο, παρατηρούνται πολλές περιπτώσεις προβλημάτων όπου τα δεδομένα οργανώνονται σε έναν γράφο, όπως τα δίκτυα ασθενειών και φαρμάκων) [16, 17] τα κοινωνικά δίκτυα [30]. Μια νέα γενιά πακέτων Συγκριτικής Προτυποποίησης (Benchmarks), όπως το Microsoft Academic Graph (MAG) Benchmark και το Berkeley Big Data Benchmark [26] [24], καθιστούν σαφή τη διάκριση ανάμεσα στα δεδομένα που αναπαρίστανται με τη μορφή γράφου και των κύβων δεδομένων. Τα συγκεκριμένα πακέτα δεδομένων και συγκριτικής προτυποποίησης, καθώς και πολλά άλλα, ανήκουν στη κατηγορία των ≪Typed Graphs≫, δηλαδή οι κόμβοι και οι ακμές του σχετικού γράφου σχετίζονται με τύπους δεδομένων που είναι εκ των προτέρων γνωστοί. Η ανάγκη για SQL ερωτήματα αναλυτικής φύσης πάνω σε τέτοιους γράφους αυξάνεται συνεχώς. Χαρακτηριστικό παράδειγμα τέτοιου προβλήματος αποτελεί η εύρεση συσχετιζόμενων ασθενειών σε ένα δίκτυο ασθενειών-φαρμάκων. Οι συμβατικές τεχνολογίες των OLAP συστημάτων δεν μπορούν να διαχειριστούν αποδοτικά τις απαιτήσεις που περιγράφουμε διότι δεν έχουν βελτιστοποιηθεί ώστε να ανακαλύπτουν μονοπάτια μεταξύ των οντοτήτων ενός γράφου [8, 9].

Η Σημασία του Σχήματος Για να μελετήσουμε τα συστήματα OLAP που εξυπηρετούν SQL ερωτήματα πάνω σε γράφους, πρέπει να ορίσουμε την αναπαράσταση των ≪typed graphs≫ (γνωστών και ως γράφων με σχήμα) σε μία βάση δεδομένων SQL. Οι κόμβοι και οι ακμές αυτών των γραφημάτων μπορούν να αναπαρασταθούν ως πλειάδες πινάκων σε ένα σχεσιακό σχήμα μιας βάσης δεδομένων. Κατηγοριοποιούμε τους πίνακες αυτούς σε δύο κατηγορίες: πίνακες Οντοτήτων (Entity tables) και Συσχετίσεων (Relationship tables), ακολουθώντας το μοντέλο Οντοτήτων-Συσχετίσεων από τη θεωρία των βάσεων δεδομένων. Ενδιαφερόμαστε κυρίως για πίνακες δυαδικών σχέσεων. Κάθε πίνακας οντοτήτων έχει ένα πρωτεύον κλειδί, ενώ κάθε πίνακας συσχετίσεων



(a) Schema (b) Graph

Σχήμα 4.1: Περαταίρω απλοποίηση της βάσης PubMed. Αντί του διαχωρισμού μεταξύ ασθενειών και χημικών ουσιών, επιλέγουμε να κρατήσουμε όλους τους όρους (MeSH Terms) σε έναν πίνακα Term. Μαζί με το σχήμα της βάσης (α) δίνεται και ο αντίστοιχος γράφος (β). Κάθε πίνακας οντοτήτων αντιστοιχεί σε έναν τύπο κόμβων, ενώ κάθε πίνακας συσχέτισης αντιστοιχεί σε έναν τύπο ακμών (που συνδέουν δύο κόμβους) [21].

διαθέτει δύο εξωτερικά κλειδιά τα οποία δείχνουν σε πρωτεύον κλειδί ενός πίνακα οντοτήτων. Ένα typed graph προκύπτει αντιστοιχίζοντας κάθε πίνακα οντοτήτων σε ένα σύνολο κόμβων, και κάθε πίνακα συσχετίσεων σε ένα σύνολο ακμών μεταξύ δύο κόμβων. Για παράδειγμα, στο Σχήμα 1 (α) διακρίνονται μερικοί πίνακες οντοτήτων και συσχετίσεων από τη βάση δεδομένων της PubMed και στο Σχήμα 1 (β) φαίνεται ο σχετικός "typed graph". Οι πλειάδες του πίνακα συσχέτισης DT αναπαρίστανται ως ακμές μεταξύ ενός Document (πλειάδα/οντότητα του πίνακα οντοτήτων Doc) και ενός Term (πλειάδα/οντότητα του πίνακα οντοτήτων Term).

Ερωτήματα Συσχέτισης Πρόκειται για μια οικογένεια ερωτημάτων τα οποία καλύπτουν πολλές ανάγκες αναλυτικής επεξεργασίας σε δεδομένα γραφημάτων και, επιπροσθέτως, επιδέχονται βελτιστοποίηση που αγγίζει ακόμα και τάξεις μεγέθους καλύτερη ταχύτητα εκτέλεσης. Ένα ερώτημα συσχέτισης, ουσιαστικά εκτελεί διάσχιση ενός "typed graph", πραγματοποιώντας παράλληλα ένα σύνολο από λειτουργίες. Η διάσχιση ξεκινά από τις εναρκτήριες οντότητες και καταλήγει στις οντότητες τερματισμού (όπου με τον όρο οντότητες αναφερόμαστε ταυτόχρονα και σε πλειάδες ενός πίνακα οντοτήτων αλλά και σε κόμβους του γράφου). Ανεπίσημα, ένα ερώτημα συσχέτισης αποτελείται από τρία στάδια:

1. Υπολογισμός Εναρκτήριων Οντοτήτων (Context Computation) : αναγνώριση ενός συνόλου οντοτήτων που ικανοποιούν κάποια κριτήρια που δίνει ο χρήστης.

2. Πλοήγηση στον Γράφο (Path Navigation) : Περιήγηση του γράφου, ξεκινώντας από τις εναρκτήριες οντότητες του προηγούμενου βήματος και ακολουθώντας

τις ακμές που ορίζουν οι σχεσιακοί πίνακες όταν εφαρμόζονται λειτουργίες συνένωσης.

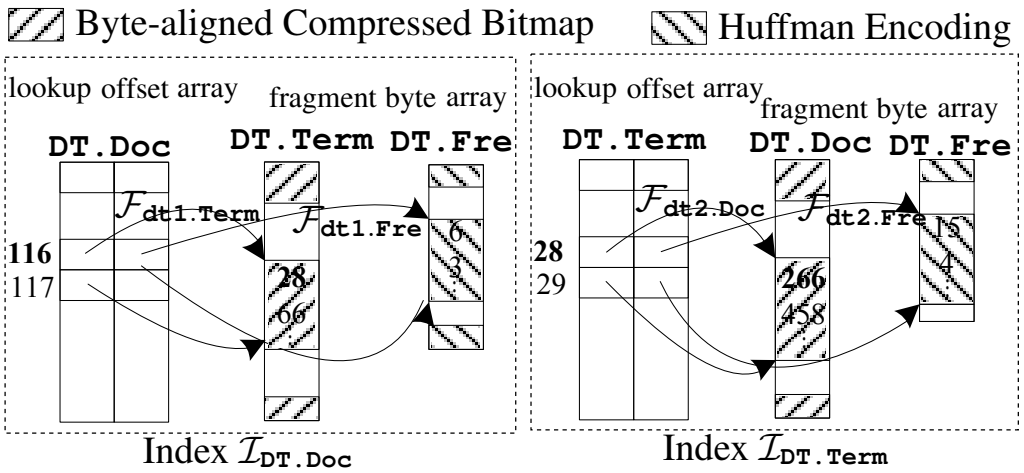
3. Συνάθροιση των Διαδρομών (Path Aggregation) : Η σημασία των οντοτήτων τερματισμού μεταφράζεται ως μετρική η οποία προκύπτει από την εφαρμογή συναθροιστικών συναρτήσεων στα γνωρίσματα των οντοτήτων (πλειάδες-κόμβοι) που συναντήθηκαν κατά τη διάσχιση του εκάστοτε μονοπατιού στον γράφο.

Τα μέρη (1) και (3) είναι προαιρετικά. Τα ερωτήματα συσχετίσεων έχουν ιδιαίτερη αξία στην ανάλυση γράφων (graph analytics) και συναντώνται πολύ συχνά σε ερευνητικά προβλήματα. Ένα χαρακτηριστικό ερώτημα συσχέτισης στο οποίο απαντάει το σύστημα που θα περιγράψουμε στη συνέχεια, είναι το εξής: ένας χρήστης αναζητά έγγραφα d_j που είναι παρόμοια με ένα δοθέν έγγραφο d_0 που έχει πρωτεύον κλειδί (ID) $d^{ID}{}_0$ στον γράφο γνώσης της PubMed. Εκφράζουμε την ομοιότητα μεταξύ δύο οποιονδήποτε εγγράφων ως το πλήθος των όρων (terms) που είναι κοινοί και στα δύο έγγραφα, δηλαδή το πλήθος των μονοπατιών με κόμβους $Doc \rightarrow Term \rightarrow Doc$, τα οποία ξεκινούν από το έγγραφο d_0 και καταλήγουν σε κάποιο έγγραφο d_j.

Query SD (Similar Documents).

```
SELECT dt2.Doc, COUNT(*) AS similarity
FROM DT dt1 JOIN DT dt2 ON dt1.Term = dt2.Term
WHERE dt1.Doc = d_0^ID
GROUP BY dt2.Doc
```

Το ερώτημα SD είναι ένα απλό ερώτημα συσχέτισης το οποίο διασχίζει τον γράφο γνώσης της PubMed και εφαρμόζει μια συναθροιστική συνάρτηση υπολογισμού του πλήθους των διαφορετικών μονοπατιών [21]. Στη συνέχεια θα παρουσιάσουμε πιο σύνθετα και απαιτητικά ερωτήματα συσχετίσεων. Είναι δύσκολο να απαντηθεί αποδοτικά ακόμη και αυτό το απλό ερώτημα συσχέτισης εξαιτίας του μεγέθους του γράφου της PubMed: περίπου 30 εκατομμύρια κόμβοι και ένα δισεκατομμύριο ακμές με αρκετά γνωρίσματα στις πλειάδες που τους αντιστοιχούν. Δοθείσης της αναλυτικής φύσης των ερωτημάτων συσχετίσεων, οι column-oriented βάσεις δεδομένων είναι πολύ πιο αποδοτικές από τις βάσεις row-stores και graph databases. Ακόμη και τότε, όμως, η επιτευχθείσα επίδοση δεν ανταποκρίνεται στις απαιτήσεις των ερωτημάτων που γίνονται ζωντανά σε διαδραστικές εφαρμογές. Το ερώτημα SD χρειάζεται 61.6 και 19.17 δευτερόλεπτα για να εκτελεστεί, αντίστοιχα, στις column-oriented βάσεις δεδομένων Monet DB και Vertica, ενώ για στη βάση PostgreSQL (row-stores) χρειάζεται 741.2 δευτερόλεπτα και στη βάση δεδομένων για γράφους Neo4j 49.3 δευτερόλεπτα [21]· όταν όλες οι βάσεις δεδομένων τρέχουν στη αποκλειστικά στη μνήμη RAM. Οι επιδόσεις αυτές χειροτερεύουν πολύ όταν τα μονοπάτια συνένωσης έχουν μεγαλύτερο μήκος και οι συναθροιστικές συναρτήσεις χρησιμοποιούν πολλές από τις ιδιότητες που ανακτώνται κατά μήκος των μονοπατιών, ή ακόμα και όταν οι εναρκτήριες οντότητες περιγράφονται από τις ιδιότητές τους αντί του πρωτεύοντος κλειδιού του σχετικού πίνακα οντοτήτων.



Σχήμα 4.2: Παράδειγμα Τμημάτων και της επεξεργασίας τους για την εκτέλεση του ερωτήματος SD. Τα τμήματα: $\pi_{Term}\sigma_{Doc=116}DT$ και $\pi_{Fre}\sigma_{Doc=116}DT$ κωδικοποιούνται με ***Bit-Aligned Compressed Array*** και Κωδικοποίηση ***Huffman***, αντίστοιχα.

GQ-Fast Η GQ-Fast αποθηκεύει και επεξεργάζεται μόνο δείκτες – δεν ασχολείται με την αρχική πληροφορία των λογικών πινάκων. Ο διαχειριστής του συστήματος έχει τη δυνατότητα να φορτώσει μια σχέση $R(C_1, C_2, ..., C_n)$ και να δηλώσει τα ευρετήρια που πρέπει να κατασκευαστούν για κάθε πρωτεύων ή εξωτερικό κλειδί C, χρησιμοποιώντας αυτό το κλειδί ως πεδίο ευρετηριοποίησης. Ως αποτέλεσμα η GQ-Fast θα δημιουργήσει τα ευρετήρια $\mathcal{I}_{R.C}$. Στο Σχήμα 9.2 διακρίνονται οι δύο δείκτες $\mathcal{I}_{\text{DT.Doc}}$ και $\mathcal{I}_{\text{DT.Term}}$ που αντιστοιχούν στα δύο εξωτερικά κλειδιά του πίνακα DT. Η διαδικασία της κατασκευής και χρησιμοποιήσης των ευρετηρίων της GQ-Fast περιγράφεται αναλυτικά στο κεφάλαιο 13.

Κατά την εκτέλεση, ο επεξεργαστής ερωτημάτων της GQ-Fast θα χρησιμοποιήσει το σχετικό ευρετήριο κατά την αναζήτηση (της προβολής των) πλειάδων της σχέσης R που έχουν την ιδιότητα $C = c$. Για παράδειγμα, το ευρετήριο $\mathcal{I}_{\text{DT.Doc}}$ μπορεί να χρησιμοποιηθεί για να βρεθούν οι όροι που σχετίζονται με το έγγραφο 116 (δηλαδή να εφαρμοστεί η ακόλουθη συνάρτηση της σχεσιακής άλγεβρας: ($\pi_{\text{Term}}\sigma_{\text{Doc=116}}\text{DT}$), ή για να βρεθούν τα ζεύγη όρου και συχνότητας που σχετίζονται με το έγγραφο 116, ($\pi_{\text{Term, Fre}}\sigma_{\text{Doc=116}}\text{DT}$).

Εσωτερικά, ένα ευρετήριο της GQ-Fast έχει δύο σκέλη: έναν πίνακα αναζήτησης και ένα σύνολο από τμήματα. Μπορούμε να θεωρήσουμε, χωρίς βλάβη της γενικότητας, ότι η C_1 είναι η στήλη που αντιστοιχεί στο πεδίο ευρετηριοποίησης. Τότε, για κάθε στήλη $C_j \in \{C_2, \ldots, C_n\}$ και για κάθε τιμή $t \in C_1$ υπάρχει ένα τμήμα $\pi_{C_j}\sigma_{C_1=t}(R)$, το οποίο διατηρεί την αρχική σειρά των τιμών της αντίστοιχης πλειάδας. Στο Σχήμα 9.2 το τμήμα $\pi_{\text{Term}}\sigma_{\text{Doc=116}}\text{DT}$ (με περιεχόμενα 28, 66, κλπ.) και

το τμήμα $\pi_{\text{Fre}}\sigma_{\text{Doc}=116}\text{DT}$ (με περιεχόμενα 6, 3, κλπ.) περιέχουν τις τιμές των όρων και των συχνοτήτων, αντίστοιχα, για το έγγραφο με ID 116.

Για να μειώσει τη κατανάλωση χώρου στο σύστημα, η GQ-Fast μπορεί να συμπιέσει αυτόνομα κάθε θραύσμα. Η ιδιαίτερη παρατήρηση πίσω από τη συμπίεση των θραυσμάτων, είναι ότι όταν ένα ερώτημα συσχέτισης αποκτά πρόσβαση σε ένα θραύσμα, μπορούμε να είμαστε σίγουροι ότι όλα τα δεδομένα του θα χρησιμοποιηθούν και, κατ' επέκταση, δεν υπάρχει ανάγκη για τυχαία προσπέλαση (Random Access) σε κάποιο θραύσμα · γεγονός που επιτρέπει τη κωδικοποίηση των θραυσμάτων με μεγάλο βαθμό συμπίεσης, όπως η κωδικοποίηση Huffman. Στο σημείο αυτό σημειώνουμε ότι το μέγεθος ενός συνηθισμένου θραύσματος είναι σχετικά μικρό (σε σύγκριση με τη σχετική στήλη) και μπορεί να χωρέσει στην L1 ή, στη χειρότερη περίπτωση, στην L2 περιοχή της προσωρινής μνήμης (Cache). Η αποκωδικοποίηση επομένως δεν επιβραδύνει την εκτέλεση με πολλαπλές αναφορές στη μνήμη RAM.

Δοθείσης μίας τιμής η οποία αναφέρεται σε μία στήλη για την οποία υπάρχει ευρετήριο, ο σχετικό πίνακας αναζήτησης πρέπει να διαθέτει έναν δείκτη για το κάθε σχετικό θραύσμα, μαζί με το μέγεθος του θραύσματος. Ένας πίνακας αναζήτησης μπορεί να κατασκευαστεί με πολλούς τρόπους (π.χ. με έναν πίνακα κατακερματισμού). Η GQ-Fast κάνει οικονομία χώρου και χρόνου κατασκευάζοντας τους πίνακες αναζήτησης ως γραμμικούς πίνακες που χρησιμοποιούν την υπόθεση του πυκνού αναγνωριστικού κλειδιού (dense ID assumption), σύμφωνα με την οποία τα αναγνωριστικά (IDs) ενός πίνακα οντοτήτων είναι διαδοχικοί ακέραιοι, ξεκινώντας από το 0. Βάσει αυτής της υπόθεσης, όλα τα θραύσματα του ίδιου τύπου μπορούν να καταχωρηθούν διαδοχικά σε έναν πίνακα ως εξής: ένας πίνακας αναζήτησης για ένα ευρετήριο πάνω στο πεδίο R.C_1 είναι ένας δισδιάστατος πίνακας $\mathcal{I}_{\text{R.C}_1}$ μεγέθους $v \times (n-1)$, όπου v το πλήθος των μοναδικών τιμών στη στήλη R.C_1 και n το πλήθος των στηλών στη σχέση R. Η διεύθυνση εκκίνησης ενός τμήματος$\pi_{C_j}\sigma_{C_1=t}(R)$ αποθηκεύεται στη θέση $\mathcal{I}_{\text{R.C}_1}[t][j-1]$ και το μέγεθός του υπολογίζεται βάσει της αρχικής διεύθυνσης και του επόμενου τμήματος.

Επεξεργασία Ερωτημάτων Η GQ-Fast τρέχει αποκλειστικά πάνω σε δείκτες, χρησιμοποιώντας αρχιτεκτονική σωλήνωσης και λογική bottom-up για να αποφύγει το μεγάλο μέγεθος των ενδιάμεσων αποτελεσμάτων. Επιπρόσθετα, επιστρατεύει μια γεννήτρια C++ κώδικα για τα πλάνα εκτέλεσης των ερωτημάτων.

Χαρακτηριστικό παράδειγμα των παραπάνω αποτελεί ο κώδικας που παράγεται για το ερώτημα SD, το οποίο χρησιμοποιεί τον πίνακα DT με τις στήλες Document (0η στήλη) και Term (1η στήλη). Στις γραμμές 2-4 η GQ-Fast χρησιμοποιεί το ευρετήριο$\mathcal{I}_{\text{DT.Doc}}$ για να βρει το τμήμα $\pi_{\text{Term}}\sigma_{\text{Doc}=116}$ $(\text{DT} \mapsto \text{dt1})$ της στήλης Terms, ξεκινώντας από τη θέση $\mathcal{I}_{\text{DT.Doc}}[116][1]$ με μέγεθος $l_{\text{dt1.Term}}$. Η GQ-Fast αποκωδικοποιεί το τμήμα στον (ήδη ορισμένο) πίνακα $\mathcal{A}_{\text{dt1.Term}}$ και επιστρέφει το πλήθος των στοιχείων$n_{\text{dt1.Term}}$. Στη συνέχεια (γραμμές 5-9), για κάθε term ID $v_{\text{dt1.Term}} \in \mathcal{A}_{\text{dt1.Term}}$, η GQ-Fast

Generated Code: Ο κώδικας σε C++ που παράγεται για το Query SD

```
1  R ← ∅
2  F_dt1.Term ← I_DT.Doc[116][1]
3  l_dt1.Term ← I_DT.Doc[116 + 1][1] − F_dt1.Term
4  A_dt1.Term, n_dt1.Term ← decodeBB(F_dt1.Term, l_dt1.Term)
5  for i ← 0 to n_dt1.Term − 1 do
6      v_dt1.Term ← A_dt1.Term[i]
7      F_dt2.Doc ← I_DT.Term[v_dt1.Term][0]
8      l_dt2.Doc ← I_DT.Term[v_dt1.Term + 1][0] − F_dt2.Doc
9      A_dt2.Doc, n_dt2.Doc ← decodeBB(F_dt2.Doc, l_dt2.Doc)
10     for j ← 0 to n_dt2.Doc − 1 do
11         v_dt2.Doc ← A_dt2.Doc[j]
12         R[v_dt2.Doc] ← R[v_dt2.Doc] + 1
13 return R
```

χρησιμοποιεί το ευρετήριο $\mathcal{I}_{\text{DT.Term}}$ για να βρει τα τμήματα του πίνακα Δοςυμεντς $\pi_{\text{dt2.Doc}}\sigma_{\text{dt2.Term}=v_{\text{dt1.Term}}}$ (DT$\mapsto$dt2) ξεκινώντας από τη θέση $\mathcal{I}_{\text{DT.Term}}[v_{\text{dt1.Term}}][0]$ (βλ. 9.2). Τα τμήματα που προκύπτουν αποκωδικοποιούνται στον πίνακα $\mathcal{A}_{\text{dt2.Doc}}$. Τέλος, η GQ-Fast ελέγχει όλα τα τμήματα του πίνακα Documents για να ενημερώσει τον πίνακα $\mathcal{R}$ (γραμμές 11-12), ο οποίος περιέχει τη τελική τιμή (αποτέλεσμα του ερωτήματος) για κάθε έγγραφο (Document).

Κεφάλαιο 5

Περιορισμοί της Αρχικής Έκδοσης της **GQ-Fast**

Παρά τη μεγάλη ταχύτητα εκτέλεσης, η αρχική έκδοση της GQ-Fast διέθετε περιορισμένες λειτουργίες και, κατ' επέκταση, δεν μπορούσε να υποστηρίξει το σύνολο των ερωτημάτων που χρειαζόμαστε ώστε να φτιάξουμε εφαρμογές όπως αυτή που αναπτύξαμε παραπάνω. Πιο συγκεκριμένα, τα ερωτήματα που ήταν δυνατό να επεξεργαστεί δεν μπορούσαν να χρησιμοποιούν τους όρους "IN" και "HAVING" της SQL, ενώ το η συνθήκη του όρου "WHERE" δεν μπορούσε να περιέχει πάνω από έναν διαφορετικό πίνακα.

Πηγή των περιορισμών αυτών ήταν η αρχιτεκτονική της GQ-Fast . Αναλυτικότερα, ένα ερώτημα SQL πρέπει αρχικά να μετασχηματιστεί σε δεντρική μορφή σχεσιακής άλγεβρας (παραδείγματα αυτής της μορφής είναι τα RQNA[1] δέντρα των πρωτότυπων ερωτημάτων που διατυπώνουμε αργότερα στην εργασία, όπως το σχήμα 6.1). Ακολούθως, ένα υποπρόγραμμα της GQ-Fast που ονομάζεται RQNA Normalizer φροντίζει να βελτιστοποιήσει το RQNA δέντρο βάσει κάποιων κανόνων που έχουμε κτασκευάσει (παρόμοια με τη λειτουργία των optimizers των βάσεων δεδομένων). Στη συνέχεια, η δεντρική μορφή μετασχηματίζεται σε φυσικά πλάνα, τα οποία αποτελούνται από μία λίστα φυσικών τελεστών οι οποίοι αργότερα αντιστοιχίζονται με συγκεκριμένα κομμάτια από κώδικα C++ (μέσω ενός υποπρογράμματος που λέγεται Code Generator, περισσότερες πληροφορίες σχετικά με τον αλγόριθμο της διαδικασίας στον αλγόριθμο: 3). Οι υπάρχοντες φυσικοί τελεστές δεν μπορούν να περιγράψουν τις απαραίτητες ενέργειες για ένα σύνθετο ερώτημα συσχέτισης, όπως τα ερωτήματα του δοκιμαστικού συστήματος που περιγράψαμε παραπάνω.

Χωρίς να επεκταθούμε σε λεπτομέρειες της λειτουργίας ή της υλοποίησής τους, αναφέρουμε συνοπτικά τους φυσικούς τελεστές της αρχικής έκδοσης της GQ-Fast :

[1] Η γραμματική πίσω από τις εκφράσεις RQNA φαίνεται στο σχήμα: 14.2

- Selection Operator
- Join & Semi Join Operator
- Threading Operator
- Aggregation Operator

Περισσότερες λεπτομέρειες για τη δομή και τη λειτουγία των φυσικώ τελεστών της GQ-Fast βρίσκονται στην ενότητα 14.2.

Join Operator Ο τελεστής συνένωσης μεταφράζεται πρακτικά από το φυσικό πλάνο εκτέλεσης ως μία δομή επανάληψης (for loop) πάνω στις τιμές κάποια μεταβλητής που έχει γίνει διαθέσιμη σε κάποιο προηγούμενο στάδιο του φυσικού πλάνου από έναν άλλον τελεστή (Select, Project). Επειδή ο τελικός κώδικας που παράγεται αποτελείται ουσιαστικά από διαδοχικά φωλιασμένους βρόχους, δεν υποστηρίζεται η ύπαρξη ενός τελεστή επιλογής (Select) μετά από κάποιο Join. Συνεπώς, σε κάποιο ερώτημα δύναται να υπάρχει μόνο ένας τελεστής Select και, επομένως, επιτρέπεται ο καθορισμός των τιμών (μέσω λογικών συνθηκών) μόνο σε έναν πίνακα της βάσης. Αυτό σημαίνει ουσιαστικά ότι ένα ερώτημα όπως το παρακάτω δεν μπορεί να απαντηθεί:

Query SDY (Similar Documents in a specific Year). Το ερώτημα SDY λειτουργεί ακριβώς όπως το ερώτημα SD που διατυπώσαμε νωρίτερα, με τη διαφορά ότι ο χρήστης καθορίζει ποια χρονιά πρέπει να έχουν δημοσιευθεί τα άρθρα των αποτελεσμάτων (μέσω μιας παραπάνω παραμέτρου).

```
SELECT dt2.Doc, COUNT(*) AS similarity
FROM DT dt1
JOIN DT dt2 ON dt1.Term = dt2.Term
JOIN Doc d AS d ON dt2.Doc = d.Id
WHERE dt1.Doc = d_0^ID AND d.year = year_0
GROUP BY dt2.Doc
ORDER BY similarity DESC
```

Πολυνημάτωση: Ο φυσικός τελεστής που διέθετε η GQ-Fast για να εισάγει στο φυσικό πλάνο multi-threaded κώδικα (Threading Operator) ήταν κατασκευασμένος με τέτοιο τρόπο ώστε η μετάβαση από μονονηματική σε πολυνηματική εκτέλεση να επιτρέπεται μόνο μετά το εναρκτήριο Selection (Selection Operator). Αυτός ο περιορισμός είναι πολύ σημαντικός για τις επιδόσεις των ερωτημάτων, ωστόσο δεν επηρεάζει τις δυνατότητες της GQ-Fast .

Εφαρμογή Λογικών Εκφράσεων: Ένας από τους βασικούς περιορισμούς, ήταν η έλλειψη ενός τελεστή που επιτρέπει την εφαρμογή μιας λογικής συνθήκης ανάμεσα στα διαδοχικά Join Operators του φυσικού πλάνου. Χαρακτηριστικό παράδειγμα ενός ερωτήματος που χρειάζεται τη δυνατότητα αυτή είναι η περίπτωση του SDY όταν θέλουμε να αποκλείσουμε από τον πίνακα dt2 κάποιο συγκεκριμένο Term (οπότε θα προσθέταμε στο Where clause : AND $dt2.Term$!= Id_1).

Κεφάλαιο 6

Νέοι Φυσικοί Τελεστές

Το βασικότερο μέρος των αλλαγών στη δομή και τη λειτουργία της GQ-Fast βασίζεται πάνω στη προσθήκη των τελεστών: Hash Join Operator & On-The-Fly Select Operator. Παρακάτω αναλύουμε τους τελεστές αυτούς και δίνουμε παραδείγματα των δυνατοτήτων προσφέρουν.

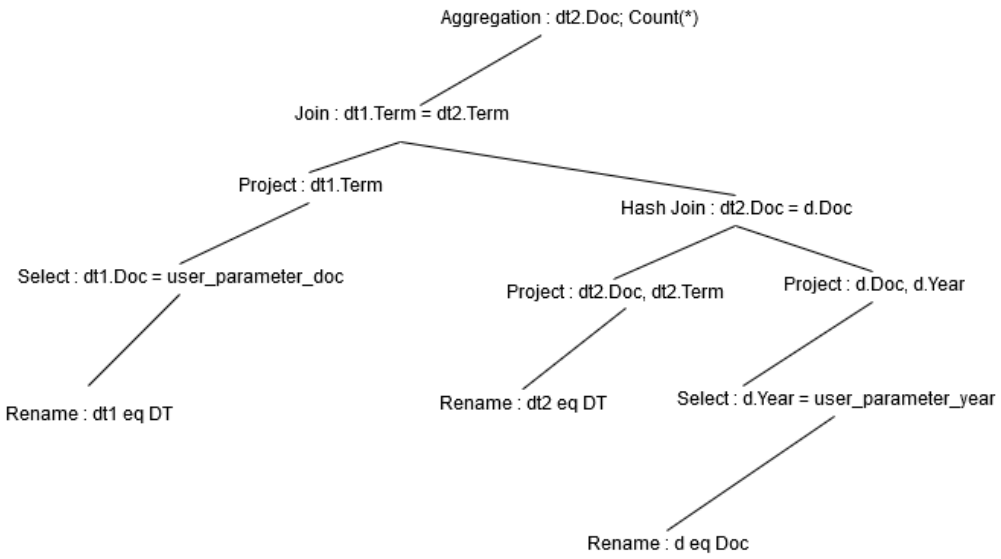
6.1 Ο Τελεστής **Hash Join**

Μέχρι τώρα, ένα RQNA δέντρο επιτρεπόταν να έχει μόνο έναν Selection Operator, ο οποίος βρίσκεται πάντα στο «αριστερότερο φύλλο» του δέντρου. Με τον Hash Join Operator επιτρέπουμε την ύπαρξη και άλλων Selection Operators. Η λογική πίσω από αυτή την δυνατότητα είναι η εξής: το RQNA subtree που περιέχει τον Selection Operator χωρίζεται από το υπόλοιπο δέντρο και υπολογίζεται ξεχωριστά. Θέτουμε ως ρίζα σε αυτό το υπόδεντρο τον τελεστή Hash Map Operator που φτιάξαμε και ουσιαστικά υπολογίζει τα tuples που πρέπει να συμμετέχουν στο Join με το αριστερό υπόδεντρο του κεντρικού RQNA δέντρου, και τα αποθηκεύει σε ένα Hash Map (ο υπολογισμός αυτός γίνεται προτού ξεκινήσει η εκτέλεση του κυρίως δέντρου και έτσι λύνεται το πρόβλημα να γίνουν όλοι αυτοί οι υπολογισμοί μέσα στα φωλιασμένα for loops του κυρίως RQNA δέντρου).

Αποτέλεσμα του Hash Join Operator είναι η υποστήριξη ερωτημάτων όπως το SDY που περιγράψαμε παραπάνω. Στο σχήμα 6.1 παρουσιάζουμε το RQNA δέντρο, ενώ ακολουθεί η λίστα με το φυσικό πλάνο εκτέλεσης.

SELECTION OPERATOR (DY) - d.Year
JOIN OPERATOR (DY) - Dummy join for Selection
HASHMAP OPERATOR (DY) - key:d.Doc

SELECTION OPERATOR (SD) - dt1.Doc
JOIN OPERATOR (SD) - Dummy join for Selection
THREADING OPERATOR (SD)
JOIN OPERATOR (SD) - dt1.Term = dt2.Term
HASH JOIN OPERATOR (SDY) - dt1.Doc = d.Doc in HashMap
AGGREGATION OPERATOR (SD) - dt2.Doc; Count(*)



Σχήμα 6.1: Το δέντρο RQNA του ερωτήματος SDY.

Επειδή η εκτέλεση του φυσικού πλάνου για κάθε υπόδεντρο που περιέχει Selection Operator διαχωρίζεται από το κεντρικό RQNA Tree, είναι σημαντικό να υπάρχει μια λειτουργία που αναγνωρίζει ποιες μεταβλητές του «χωρισμένου» πλάνου εκτέλεσης πρέπει να «επιβιώσουν» και να αποθηκευτεί η τιμή τους σε κατάλληλο Hash Map. Για το λόγο αυτό, ανάμεσα στα στάδια του Physical Planner και του Code Generator προσθέσαμε τον Hash Table Virtualizer, ένα υποπρόγραμμα που κάνει ακριβώς αυτή τη δουλειά, ενώ φροντίζει να δίνει μοναδικά (εικονικά) ονόματα στις μεταβλητές που μεταφέρει στο Hash Map κάθε υπόδεντρου που ενώνεται με το κυρίως RQNA Tree με Hash Join, ώστε να μην υπάρχουν συγκρούσεις ονομάτων στο τέλος.

Μαζί με τον Hash Join Operator, αναπτύξαμε και τον Hash Semi Join Operator, με την ίδια λογική που υπάρχει στην SQL τόσο ο Join όσο και ο Semi Join τελεστής.

Ένας αρχιτεκτονικός περιορισμός, είναι ότι στην υλοποίησή μας δεν επιτρέπονται φωλιασμένα Hash Join Operators. Εάν ένας κόμβος του RQNA Tree έχει τη πράξη της συνένωσης (Join), και αργότερα αναγνωριστεί αυτή η πράξη ως ένα Hash Join, τότε δεν επιτρέπεται να γίνει το ίδιο και σε κάποιον κόμβο-παιδί αυτού του κόμβου.

6.2 Ο Τελεστής **On-The-Fly-Select**

Ο τελεστής αυτός μπορεί να παρεμβάλλεται σε οποιοδήποτε σημείο των φυσικών πλάνων και να εισάγει μία λογική συνθήκη (για τον κώδικα σε C++ αυτό μεταφράζεται ως μια δομή επιλογής). Με τον τελεστή αυτό πετυχαίνουμε δύο πράγματα:

- Επιτρέπονται παραπάνω από μία λογικές συνθήκες στο φυσικό πλάνο. Μέχρι πρότινος, μόνο ο Selection Operator υποστήριζε την ύπαρξη λογικής συνθήκης και δεν γινόταν να χρησιμοποιηθεί παρά μόνο στην αρχή του φυσικού πλάνου. Ακόμα και με τον Hash Join Operator, δεν είναι πρακτικό να διαχωρίζεται ένα κομμάτι από το RQNA Tree και να επιστρατεύεται ένα ολόκληρο Hash Map για να εισαχθεί μια απλή συνθήκη ελέγχου. Επίσης, σε αντίθεση με τον Selection Operator, η λογική συνθήκη μπορεί να είναι όσο σύνθετη είναι οποιαδήποτε συνθήκη σε SQL.

- Δίνουμε τη δυνατότητα στον RQNA Normalizer να τοποθετήσει τη λογική συνθήκη του RQNA δέντρου στη βέλτιστη (για το φυσικό πλάνο) θέση. Αυτό σημαίνει ότι η GQ-Fast θα αναγνωρίσει τις μεταβλητές που εμφανίζονται στη λογική συνθήκη και θα φροντίσει ώστε η συνθήκη να μπει αμέσως μετά τον ορισμό όλων των μεταβλητών αυτών στον κώδικα της C++.

Πιο κάτω περιγράφουμε αρκετά καινούργια ερωτήματα συσχέτισης στα οποία μπορεί να εφαρμοστεί ο On-The-Fly-Select Operator. Στο σημείο αυτό, αναφέρουμε ενδεικτικά την εξής περίπτωση που θα μπορούσε να χρειαστεί: έστω ότι στο ερώτημα SDY θέλουμε να προσθέσουμε μία λογική συνθήκη που συνδυάζει με κάποιο τρόπο τις μεταβλητές: dt2.Term και d.Year. Σε αυτή τη περίπτωση (μπορεί να μην προκύπτει κάποιο λογικό ερώτημα έτσι για τα δεδομένα που έχουμε, ωστόσο μας ενδιαφέρει η τεχνική πλευρά της απαίτησης) είναι απαραίτητος ο On-The-Fly-Select Operator και μάλιστα πρέπει να εισαχθεί αμέσως κάτω από την ρίζα του RQNA δέντρου (Aggregation Operator).

Στη σύγχρονη, λοιπόν, μορφή της, η GQ-Fast διαθέτει τους εξής φυσικούς τελεστές:

1. Selection Operator
2. Join & Semi Join Operator
3. Hash Join & Hash Semi Join Operators
4. Hash Map Operator
5. Threading Operator
6. Aggregation Operator

Κεφάλαιο 7

Νέα Ερωτήματα Συσχέτισης

Σε αυτή την ενότητα παρουσιάζουμε μερικά ερωτήματα συσχέτισης που σχεδιάσαμε ώστε αφενός να επιδεικνύουν τη συμβολή των επεκτάσεων της GQ-Fast και αφετέρου να επιστρέφουν χρήσιμη ερευνητική πληροφορία σε πιθανές ανάγκες ενός χρήστη που μελετά τη Βιοϊατρική βιβλιογραφία. Αντί να διαχωρίζουμε τις σχέσεις Substances καιDiseases, για τα επόμενα ερωτήματα θεωρούμε τη σχέση Terms, η οποία περιέχει όλους τους όρους που υπάρχουν στο σύστημα γνώσης της PubMed.

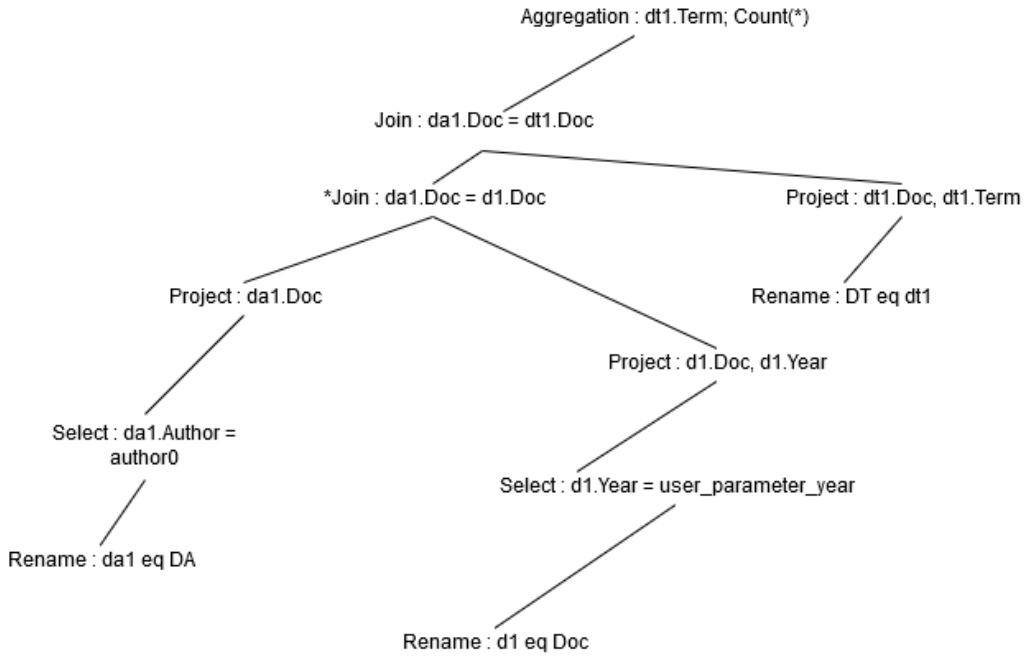
Το πιο απλό ερώτημα το οποίο χρησιμοποιεί τον Hash Join Operator έχει ήδη διατυπωθεί παραπάνω: πρόκειται για το SDY. Ακολουθούν πιο σύνθετα ερωτήματα που συχνά απαιτούν τη διαδοχική χρήση των νέων τελεστών μέσα από σύνθετα λογικά πλάνα. Στα δέντρα RQNA που παραθέτουμε μαζί με τα ερωτήματα, σημειώνουμε με άστρο τους τελεστές JOIN οι οποίοι κατά την επεξεργασία του ερωτήματος θα αντιστοιχιστούν με τον τελεστή Hash Join Operator (και όχι του συμβατικού Index Based Join).

7.1 **ATY**

Το ερώτημα ATY (Author's Terms in a specific Year) επιστρέφει τους όρους οι οποίοι υπάρχουν στις δημοσιεύσεις ενός συγκεκριμένου συγγραφέα σε μία συγκεκριμένη χρονιά. Είσοδο, λοιπόν, του ερωτήματος αποτελούν το πρωτεύον κλειδί του συγγραφέα και ένας αριθμός που αντιστοιχεί στο επιθυμητό έτος. Παραθέτουμε το ερώτημα σε SQL:

Query ATY (Author's Terms in a specific Year).

```
SELECT dt1.term, COUNT (*) As similarity
FROM DA da1 JOIN Doc d1 ON da1.doc = d1.article
JOIN DT dt1 ON da1.article = dt1.doc
WHERE da1.author = author'id AND d1.year = year
GROUP BY dt1.term
ORDER BY similarity DESC
```



Σχήμα 7.1: Το δέντρο RQNA του ερωτήματος ATY.

Το αντίστοιχο RQNA Tree του ATY φαίνεται στο σχήμα 7.1, ενώ το φυσικό πλάνο είναι το εξής:

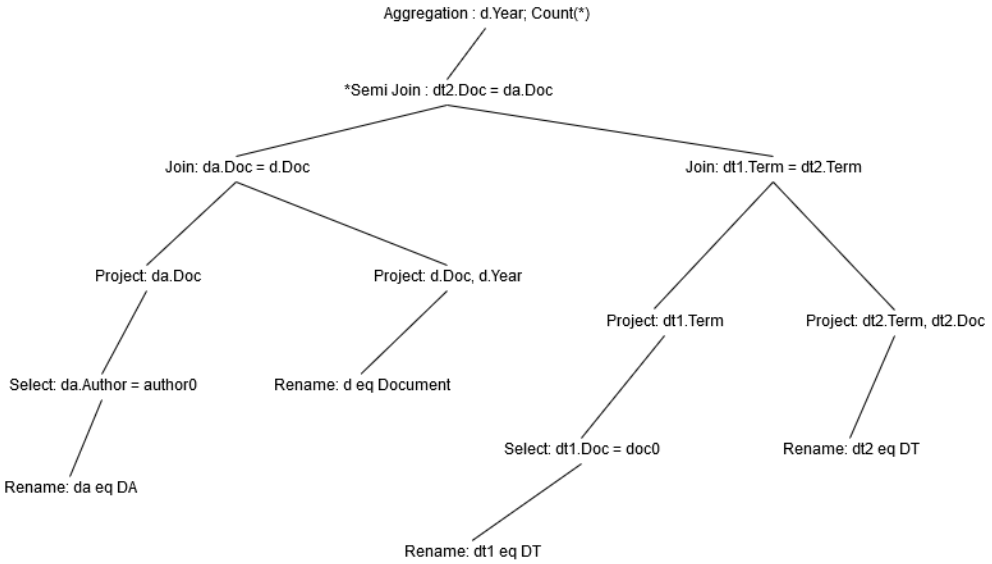
```
SELECTION˙OPERATOR (DY) - d1.Year
HASHMAP˙OPERATOR (DY) - Will be replaced with HashMap
--------------------------------------------------------------------------------
SELECTION˙OPERATOR (AT) - da1.Author
HASHJOIN OPERATOR (ATY) - da1.doc = d1.Doc in HashMap
THREADING˙OPERATOR (AT)
JOIN˙OPERATOR (AT) - da1.Doc = dt1.Doc
AGGREGATION˙OPERATOR (AT) - dt1.Term; Count(*)
```

7.2 DAY

Το ερώτημα DAY (similar Documents of an Author per Year) επιστρέφει το πλήθος των άρθρων, τα οποία σχετίζονται με ένα συγκεκριμένο άρθρο (η ομοιότητα υπολογίζεται όπως και στο ερώτημα SD) – για έναν συγγραφέα ανά έτος. Παραθέτουμε το ερώτημα σε SQL:

Query ATY (Author's Terms in a specific Year).

```
SELECT d.year, Count(*)
FROM DA da JOIN Doc d ON da.article = d.article
WHERE da.author = author˙id AND da.article IN
(
    SELECT dt2.article
    FROM DT dt1 JOIN DT dt2
```



Σχήμα 7.2: Το δέντρο RQNA του ερωτήματος DAY.

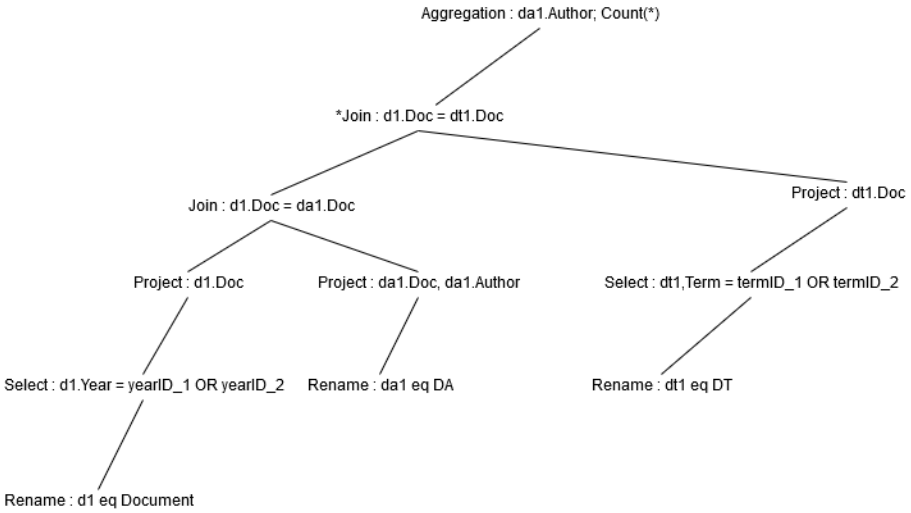
```
    ON dt1.term = dt2.term
    WHERE dt1.doc = document^ID
)
GROUP BY d.year
```

Το αντίστοιχο RQNA Tree του DAY φαίνεται στο σχήμα 7.2, ενώ το φυσικό πλάνο είναι το εξής:

```
SELECTION˙OPERATOR - dt1.Doc = doc0
THREADING˙OPERATOR
JOIN˙OPERATOR - dt1.Term = dt2.Term
HASHMAP˙OPERATOR - dt2.Doc as key
--------------------------------------------------------------------------------
SELECTION˙OPERATOR -da.Author = author0
THREADING˙OPERATOR
JOIN˙OPERATOR -da.Doc = d.Doc
(HASH) JOIN˙OPERATOR -d.Doc = dt2.Doc
AGGREGATION˙OPERATOR -d.Year; Count(*)
```

7.3 **A2X**

Το ερώτημα A2X εξετάζει τη περίπτωση εισαγωγής πολλαπλών τιμών σε έναν Selection Operator. Δέχεται ως είσοδο δύο αριθμούς που συμβολίζουν χρονολογία και δύο (αναγνωριστικά κλειδιά από) Terms. Επιστρέφει τους συγγραφείς που χρησιμοποιήσαν τους όρους αυτούς σε άρθρα τα οποία δημοσιεύτηκαν σε μία από τις δύο δοθείσες χρονιές (η συναθροιστική συνάρτηση υπολογίζει το πλήθος αυτών των άρθρων). Τα έγγραφα που περιέχουν και τους δύο όρους όρους (terms) της εισόδου, προσμετρο-



Σχήμα 7.3: Το δέντρο RQNA του ερωτήματος A2X.

ύνται δύο φορές στο συντελεστή κάθε συγγραφέα. Παραθέτουμε το ερώτημα A2X σε SQL:

Query A2X.

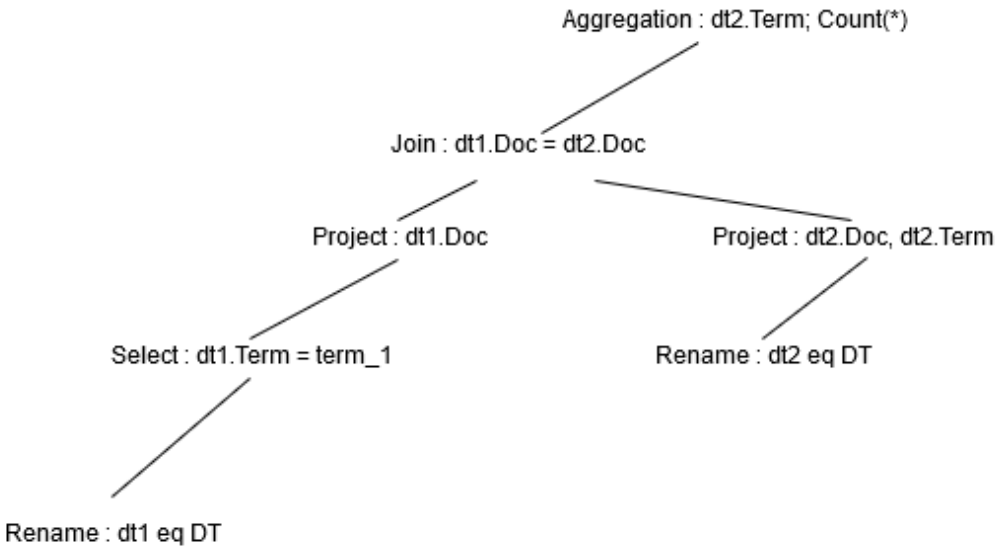
```
SELECT da.author, COUNT(*) AS similarity
FROM Doc d JOIN DA da ON d.article = da.article
JOIN DT dt ON da.article = dt.article
WHERE (d.Year =year_1 OR d.year =year_2)
   AND (dt.Term =term_1 OR dt.Term = term_2)
GROUP BY da.author
```

Το αντίστοιχο RQNA Tree του A2X φαίνεται στο σχήμα 7.3, ενώ το φυσικό πλάνο είναι το εξής:

```
SELECTION˙OPERATOR - dt1.Term = term1 OR term2
HASHMAP˙OPERATOR - dt1.Doc as key
--------------------------------------------------------------------------------
SELECTION˙OPERATOR -d1.Year = year1 OR year2
THREADING˙OPERATOR
JOIN˙OPERATOR -d1.Doc = da1.Doc
(HASH) JOIN˙OPERATOR -d1.Doc = dt1.Doc
AGGREGATION˙OPERATOR -da1.Author; Count(*)
```

7.4 ST

Στόχος του συγκεκριμένου ερωτήματος είναι να βρει παρόμοια Terms: δίνεται ένα term, και επιστρέφονται αυτά που συνυπάρχουν πιο συχνά μαζί του σε όλα τα άρθρα της PubMed. Το ερώτημα αυτό δεν χρησιμοποιεί κάποιον από τους τελεστές που προσθέσαμε στη GQ-Fast , ωστόσο είναι ιδιαίτερα σημαντικό για το δοκιμαστικό πε-



Σχήμα 7.4: Το δέντρο RQNA του ερωτήματος ST.

ριβάλλον χρήστη που παρουσιάσαμε, καθώς παραλλαγές του χρησιμοποιήθηκαν για τους πίνακες Diseases και Substances των αποτελεσμάτων.

Query ST (Similar Terms).

```
SELECT dt2.Term, COUNT(*)
FROM DT dt1
    JOIN DT dt2 ON dt1.Doc = dt2.Doc
WHERE dt1.Term = term_1
GROUP BY dt2.Term
ORDER BY count DESC
```

Το αντίστοιχο RQNA Tree του ST φαίνεται στο σχήμα 7.4:

Κεφάλαιο 8

Πειραματικά Αποτελέσματα

Σε αυτή την ενότητα παραθέτουμε τα δεδομένα και τα αποτελέσματα από μερικές αντιπροσωπευτικές δοκιμές των ερωτημάτων που διατυπώσαμε παραπάνω. Σημειώνουμε ότι ως Terms θεωρήσαμε μόνο τη σχέση Substances από το Datamart Schema (2.1) της PostgreSQL.

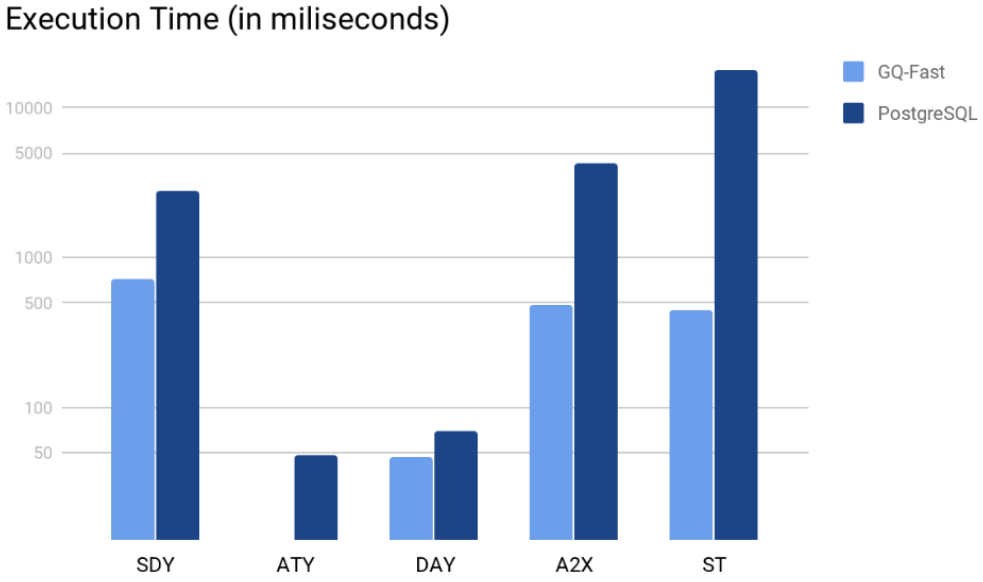
Όλες οι δοκιμές έγιναν σε φορητό υπολογιστή με επεξεργαστή Intel Core i7 - 7700HQ (256 KB L1 cache, 1MB L2 cache και 6MB L3 cache). Κρατήθηκαν μόνο οι χρόνοι της δεύτερης εκτέλεσης κάθε παραδείγματος (αφορά τη PostgreSQL ώστε τα δεδομένα να βρίσκονται ήδη στη RAM). Για κάθε ερώτημα που έτρεξε στη PostgreSQL, φροντίσαμε ώστε να υπάρχουν και να χρησιμοποιούνται όλα τα ευρετήρια που μπορούν να βελτιώσουν τον χρόνο εκτέλεσης.

Δεδομένα από PubMed (χρησιμοποιήθηκαν όλες οι επιστημονικές δημοσιεύσεις από το 1996 και μετά):

- Articles : 14319018
- Document-Term tuples : 56533389
- Document-Author tuples : 70198290

Στο σχήμα 8.1 φαίνονται οι χρονικές διαφορές όταν εκτελέσαμε τα ερωτήματα που παρουσιάσαμε στη GQ-Fast και στη PostgreSQL.

Παρουσιάζουμε επίσης τον πίνακα (15.2) με τους χρόνους εκτέλεσης των ερωτημάτων του Demo User Interface που παρουσιάσαμε προηγουμένως, για τη GQ-Fast και για τη PostgreSQL. Σημειώνουμε ότι για τη περίπτωση του Author's Autocomplete πήραμε το σενάριο όπου έχει ήδη επιλεγεί μία ασθένεια, και αντίστοιχα για τη



Σχήμα 8.1: Σύγκριση των χρονικών επιδόσεων της GQ-Fast και της PostgreSQL.

περίπτωση του Disease's Autocomplete θεωρήσαμε το σενάριο όπου έχει ήδη επιλεγεί ένας συγγραφέας.

Δεδομένα που χρησιμοποιήθηκαν από τη βάση δεδομένων PubMed για τον πίνακα του Demo:

- Articles : 29137786
- Document-Substances tuples : 101010816
- Document-Diseases tuples : 53253398
- Document-Authors tuples : 114339706

Πίνακας 8.1: Συγκεντρωτικό Πίνακας με τους χρόνους εκτέλεσης των GQ-Fast & PostgreSQL για τα ερωτήματα συσχέτισης που χρησιμποιήθηκαν στη δοκιμαστική εφαρμογή φυλλομετρητή που αναπτύξαμε.

Query	**GQ-Fast (ms)**	**PostgrSQL (ms)**
Author's Autocomplete	2288	24596
Disease's Autocomplete	432	4742
Frequent Co-Authors	40	216
Similar Authors	64412	1260000
Diseases	360	8350
Substances	448	17675
Heat Map	76	100

Chapter 9

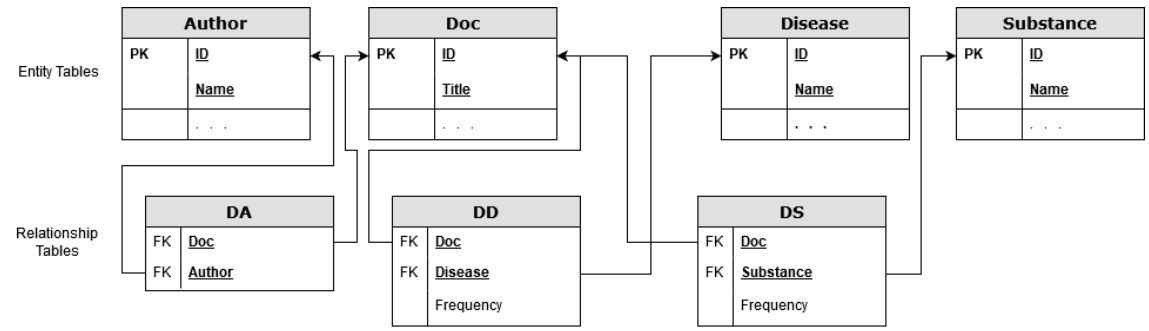
Introduction

The focus of past OLAP systems was on SQL queries on data cubes, whose data is modeled as star/snowflake SQL schemas [7, 28]. However, in recent years, an avalanche of graph data emerged, such as disease-drug networks (chem/bio-informatics) [16, 17] and social networks (Web) [30]. A new generation of benchmarks, such as the Microsoft Academic Graph (MAG) Benchmark [26] and the Berkeley Big Data Benchmark [24] make clear the distinction of these data from data cubes (such as the old TPC-H benchmark). The particular data sets and benchmarks, as well as many others, are essentially *typed graphs*, i.e., graphs where vertices and edges are associated with types known in advance. There is an increasing demand to perform analytic SQL queries over such graphs; e.g., discovering related diseases in a disease-drug network graph. Traditional, SQL OLAP technologies do not handle such demands well because they are not sufficiently optimized for finding paths among entities [8, 9].

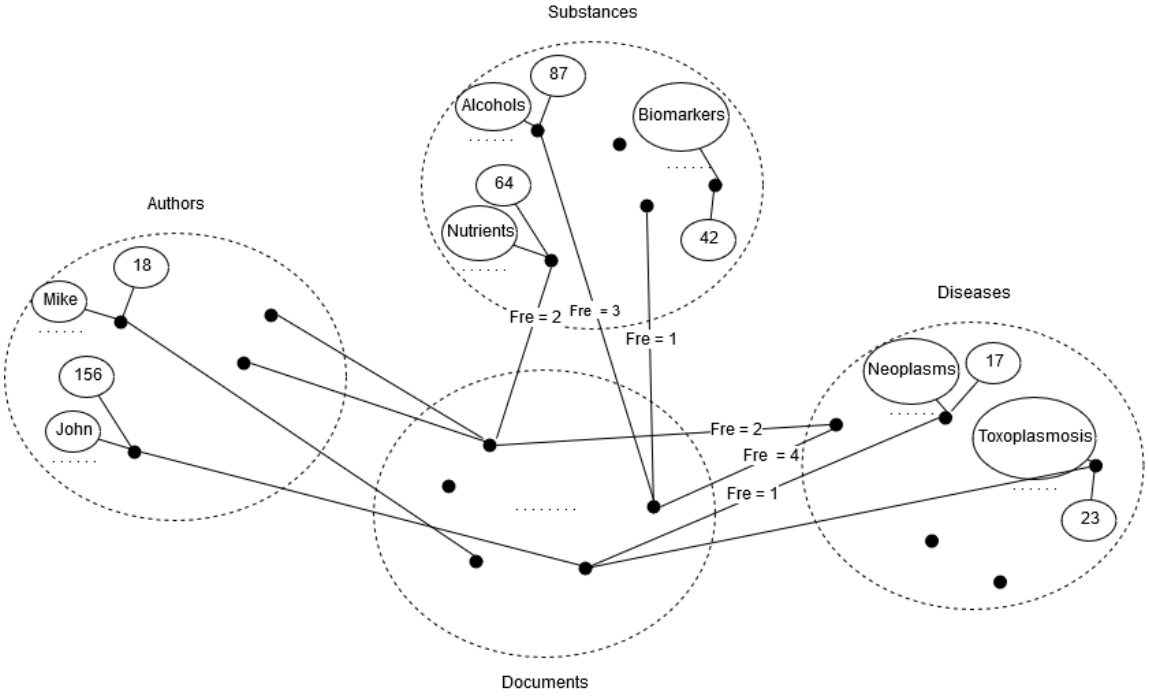
Schema. Towards SQL-based OLAP on graphs, we first define the representation of *typed graphs* (also known as graphs with schema, e.g., [14]) in an SQL database. The nodes and edges of a typed graph are represented as tuples of relational tables. We classify the tables into two categories: *Entity tables* and *Relationship tables*, following the database E/R model [13]. We focus on *binary* relationships. Each entity table has a primary key column, called the *ID* column, while each relationship table has two foreign key columns pointing to ID columns of entity tables[1]. Hence, the tuples comprise a typed graph [29, 32]: Each entity table corresponds to a type of vertices, while each relationship table corresponds to a type of edges. Columns in entity tables and relationship tables correspond to attributes of vertices and edges, respectively. For example, consider the premier public biomedical database PubMed[2]. Figure 9.1(a) shows its schema, and Figure 9.1(b) presents a corresponding typed graph. The tuples of the relationship table DT stand for edges from a document tuple/entity/node to a term tuple/entity/node.

[1] In order to capture many-to-one relationships efficiently, we also allow entity tables to have foreign keys [13]. We neglect this possibility, as it does not essentially change any consideration.

[2] http://www.ncbi.nlm.nih.gov/pubmed



(a) Schema.



(b) Graph.

Figure 9.1: PubMed Schema and Corresponding Graph. Each entity table corresponds to a type of vertices, while each relationship table corresponds to edges linking corresponding types of vertices.

Relationship Queries. We identify a class of queries, called *relationship queries*, which cover many analytics needs on graph data and, in addition, they are amenable to orders-of-magnitude speed optimization. Informally, a relationship query contains three steps: (i) *Context Computation*: The context is a collection of entities whose properties satisfy the user given conditions; we call these entities "source entities". At least one condition concerning one entity table is required. However, multiple conditions concerning different entity tables are allowed. (ii) *Graph Discovery*: Depending on the relationship query and

the user given conditions, we discern two types of navigation and processing in the graph:

- Graph Navigation: Navigation from source entities to target entities via joins over relationship tables. Essentially, a sequence of Joins can create a Spanning Tree of the Subgraph of the discovered entities. In the most simple case where there are only consecutive Joins and each one is based on a key that it's predecessor introduced, we can think of the total operation as navigating through a path of the PubMed Graph.

- Path Convergence: We could say that each family of entities (entity table) defines the start of a navigation process of (a). At some point, a Join operation will connect two different Spanning Trees through a common entity table; a process we call convergence. Different Spanning Trees (or, in the simplest case, paths) can coexist, however, all of them must eventually converge into one that leads to a final type of target entities (entity table) that may or may not take part in an Aggregation function of the following Aggregation step. A Spanning Tree that is defined by a sequence of Joins can work as a filtering process by limiting the valid entities (by allowing only those that it has discovered) of the Spanning Tree (or Navigation Path) that it is converged into [3].

(iii) *Aggregation*: The importance of the target entities is computed by applying aggregation functions over attributes collected along the navigation paths, which are accessed in the first step. The first and third steps are optional. Relationship queries are common in graph analytics. For example, all the queries evaluated in [22] are relationship queries. We illustrate a relationship query on the PubMed schema, which will serve as one of the running examples.

Schema Convention For the following example queries, we can think of the entity table Term to either be the entity table of Substances, or the entity table of Diseases, or the Union of those tables. Accordingly, the relationship table DT (Document-Term), can be the relationship table of DS, or the relationship table of DD, or the Union of those tables.

Query SD (Similar Documents). Assume a user wants to find documents d_j that are similar to a given document d_0 with ID d_0^{ID} in the PubMed graph. Similarity between documents d_0 and d_j is measured by the number of terms associated to both of them, i.e., the number of paths with type $\text{Doc} \rightarrow \text{Term} \rightarrow \text{Doc}$ that start at d_0 and end at d_j.

```
SELECT dt2.Doc, COUNT(*) AS similarity
FROM DT dt1 JOIN DT dt2 ON dt1.Term = dt2.Term
WHERE dt1.Doc = d0^ID
GROUP BY dt2.Doc
```

[3] Choosing the final Spanning Tree (or, in the simplest case, Navigation Path) that contains the entity on which Aggregation is based, is done arbitrarily and does not change the qualitative nature of a relationship query. However, this is an important decision for the efficiency of the physical evaluation of the query.

The Query SD is a simple relationship query: It navigates via typed paths $\mathrm{Doc} \rightarrow \mathrm{Term} \rightarrow \mathrm{Doc}$ and then aggregates the number of paths reaching each target. More complex (and performance-challenging) relationship queries are presented in Section 10.

It is challenging to answer even this simple query efficiently, due to the large size of the graph: thirty million vertices and one billion edges with several attributes. Given the analytical nature of relationship queries, column-oriented database systems are much more efficient than row-stores and graph database systems. [2,3,15,27]. Nevertheless, the obtained performance is often insufficient for online queries and interactive applications. Query SD takes 61.6 and 19.17 seconds on the column databases MonetDB [15] and Vertica [19], 741.2 seconds on the row database PostgreSQL, and 49.3 seconds on the graph database Neo4j, even though we fully cached the data in main memory in all the cases. Performance gets far worse when the join paths are longer, the aggregations involve many attributes of the paths or the source entities themselves are specified by their properties and connections, rather than their IDs.

To improve the performance, we propose an index-only fully pipelined database called GQ-Fast. GQ-Fast answers Query SD in 1.068 seconds. As the queries become more complex, its performance ratio to the other systems widens. Moreover, GQ-Fast generally requires less memory.

GQ-Fast achieves such superior performance by employing a code generator to produce efficient fully pipelined source code running upon a new compressed fragment-based index, as outlined in the following paragraphs.

Database Structure. A GQ-Fast database physically stores only indices – it does not store the logical tables. Generally, the administrator may load a relation $R(C_1, C_2, ..., C_n)$ and specify that for each ID or foreign key attribute C a respective index should be built, using C as the *indexed column*. In response, GQ-Fast will make an index $\mathcal{I}_{R.C}$ for each such attribute. Figure 9.2 shows the two indices $\mathcal{I}_{\mathrm{DS.Doc}}$ and $\mathcal{I}_{\mathrm{DS.Substance}}$ that correspond to the two foreign keys of the table DS. During runtime, the GQ-Fast query processor will use the index to find (projections of) tuples of R that have a given $C = c$ value. For example, the index $\mathcal{I}_{\mathrm{DS.Doc}}$ can be used to find the terms associated to document 116 (i.e., $\pi_{\mathrm{Substance}}\sigma_{\mathrm{Doc}=116}\mathrm{DS}$) or to find the term/frequency pairs associated to document 116 (i.e., $\pi_{\mathrm{Term,\ Fre}}\sigma_{\mathrm{Doc}=116}\mathrm{DS}$).

Internally, a GQ-Fast index has two components: a lookup table and a set of fragments. Let us say, w.l.o.g., that C_1 is the indexed column. Then for each column $C_j \in \{C_2, \ldots, C_n\}$ and for each value $t \in C_1$ there is a fragment $\pi_{C_j}\sigma_{C_1=t}(R)$, which retains the original order of the values. In Figure 9.2, the fragment $\pi_{\mathrm{Term}}\sigma_{\mathrm{Doc}=116}\mathrm{DS}$ (with contents 28, 66, etc.) and the fragment $\pi_{\mathrm{Fre}}\sigma_{\mathrm{Doc}=116}\mathrm{DS}$ (with contents 6, 3, etc.) provide the substances and frequencies associated to document 116, respectively.

To reduce space costs, GQ-Fast compresses individual fragments. *The key observation behind compressing fragments is that when a relationship query accesses a fragment, all of its data will be used. There is no need for random access within the fragment.* Based on this, GQ-Fast allows very compressed encodings of each individual fragment, such as Huffman encoding. Note that the typical fragment is relatively small (compared to the column) and can typically fit in the L1 cache or, at least, in the L2 cache. Hence, its

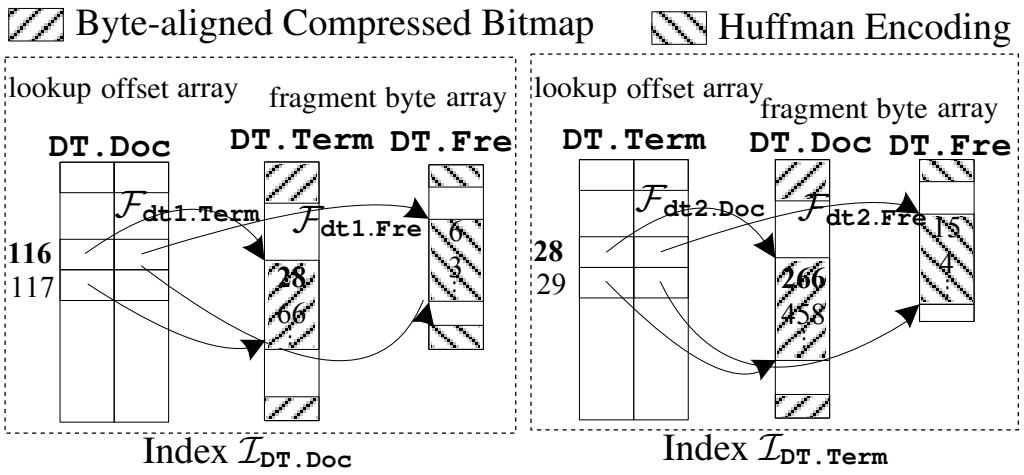


Figure 9.2: Example of Fragments and Query Processing for Query SD. Fragments $\pi_{Term}\sigma_{Doc=116}DT$ and $\pi_{Fre}\sigma_{Doc=116}DT$ are encoded with *bit-aligned compressed array* and *Huffman encoding*.

decoding is not penalized with multiple random access to the RAM.

Given a value for the indexed column, the lookup table must be able to provide a pointer to the respective fragment, along with the size of each fragment. A lookup table can be built in many known ways; e.g., as a hash table. GQ-Fast saves space and response time by building *lookup tables as offset arrays that utilize the dense ID assumption*, according to which the IDs of an entity table are consecutive integers, starting from 0. Under this assumption, all fragments of the same type are listed consecutively in an array: A GQ-Fast lookup table for an index on $\mathrm{R.C_1}$ is a two-dimensional array $\mathcal{I}_{\mathrm{R.C_1}}$ of size $v \times (n-1)$, where v is the number of unique values in $\mathrm{R.C_1}$ and n is the number of columns in R. The starting address of the fragment $\pi_{C_j}\sigma_{C_1=t}(R)$ is stored in $\mathcal{I}_{\mathrm{R.C_1}}[t][j-1]$ and its size can be calculated using the starting address of the next fragment.

Query Processing. GQ-Fast query plans run exclusively on indices. They employ a *bottom-up pipelined execution model*, illustrated next, to avoid large intermediate results. In addition, GQ-Fast employs a C++ code generator for query plans.

As an example, consider the generated code for Query SD, which uses table DT with columns Document (0^{th} column) and Term (1^{st} column). In Lines 2–4, GQ-Fast uses index $\mathcal{I}_{\mathrm{DT.Doc}}$ to find the Terms' fragment $\pi_{\mathrm{Term}}\sigma_{\mathrm{Doc}=116}\ (\mathrm{DT} \mapsto \mathrm{dt1})$ starting at position $\mathcal{I}_{\mathrm{DT.Doc}}\ [116][1]$ with size $l_{\mathrm{dt1.Term}}$. GQ-Fast decodes the fragment into the (preallocated) array $\mathcal{A}_{\mathrm{dt1.Term}}$ and returns the number of elements $n_{\mathrm{dt1.Term}}$. Afterwards (Lines 5–9), for each term ID $v_{\mathrm{dt1.Term}} \in \mathcal{A}_{\mathrm{dt1.Term}}$, GQ-Fast uses index $\mathcal{I}_{\mathrm{DT.Term}}$ to find the Documents fragment $\pi_{\mathrm{dt2.Doc}}\sigma_{\mathrm{dt2.Term}=v_{\mathrm{dt1.Term}}}\ (\mathrm{DT} \mapsto \mathrm{dt2})$ starting at position offset $\mathcal{I}_{\mathrm{DT.Term}}[v_{\mathrm{dt1.Term}}][0]$ (see Figure 9.2). GQ-Fast decodes the identified fragments into $\mathcal{A}_{\mathrm{dt2.Doc}}$. Finally, GQ-Fast scans all Documents fragments to update the array $\mathcal{R}$ (Lines 11–12), which holds the counts per document.

Notice that (1) GQ-Fast can afford to have $\mathcal{R}$ be an array (as opposed to a hash

Generated Code: Generated code for Query SD

```
1  R ← ∅
2  F_dt1.Term ← I_DT.Doc[116][1]
3  l_dt1.Term ← I_DT.Doc[116 + 1][1] − F_dt1.Term
4  A_dt1.Term, n_dt1.Term ← decodeBB(F_dt1.Term, l_dt1.Term)
5  for i ← 0 to n_dt1.Term − 1 do
6      v_dt1.Term ← A_dt1.Term[i]
7      F_dt2.Doc ← I_DT.Term[v_dt1.Term][0]
8      l_dt2.Doc ← I_DT.Term[v_dt1.Term + 1][0] − F_dt2.Doc
9      A_dt2.Doc, n_dt2.Doc ← decodeBB(F_dt2.Doc, l_dt2.Doc)
10     for j ← 0 to n_dt2.Doc − 1 do
11         v_dt2.Doc ← A_dt2.Doc[j]
12         R[v_dt2.Doc] ← R[v_dt2.Doc] + 1
13 return R
```

table) because of the dense ID assumption; (2) The execution is pipelined in a sense that it iterates over the fragments and their elements. The memory footprint is small as it is dictated by the max. size of fragments and not of the overall column size.

Chapter 10

Data Schema and Queries

10.1 Data Schema

We classify relational tables in GQ-Fast in two categories according to the entities and the relationships of the E/R model [13]: *entity tables* (e.g., Author in Figure 9.1(a)) and *relationship tables* (e.g., DS, DA). Each entity table $\mathbb{E}$ has an ID (primary key) attribute and several attributes $M_1, ..., M_n$. Each tuple $t \in \mathbb{E}$ corresponds to a real-life entity. A relationship table $\mathbb{R}$ has two *foreign key attributes* F_1 and F_2 referencing the IDs of respective entity tables[1], i.e., $F_1 \rightsquigarrow \mathbb{E}_1.ID$ and $F_2 \rightsquigarrow \mathbb{E}_2.ID$, where $\rightsquigarrow$ is *reference*. The combination $(f_1, f_2) \in F_1 \times F_2$ is unique. A relationship table may also have *measure attributes* $M_1, ..., M_m$ (e.g., $\mathrm{DT.Fre}$).

E/R $\rightarrow$ Graph. Mapping E/R schemas to graph models is a well-studied topic [5, 29]. We use the following two steps to map our schema to a typed graph [14]: (i) Each entity table $\mathbb{E}(M_1, ..., M_n)$ refers to a type of vertices $\mathbb{V}$, and each entity $t \in \mathbb{E}$ refers to one vertex $v \in \mathbb{V}$. The attributes $M_1, ..., M_n$ in the entity table are mapped to properties of vertices; and (ii) each relationship table $\mathbb{R}(F_1, F_2, M_1, ..., M_m)$ refers to edges $\mathbb{E}$ crossing two types of vertices $\mathbb{V}_1 \times \mathbb{V}_2$, where $\mathbb{V}_1$ and $\mathbb{V}_2$ are translated from entities $\mathbb{E}_1$ and $\mathbb{E}_2$ and $F_1 \rightsquigarrow \mathbb{E}_1.ID$ and $F_2 \rightsquigarrow \mathbb{E}_2.ID$.

Graph $\rightarrow$ Relational Schema with Entities and Relationships. Mapping a graph to a relational schema has been studied for several years [10,31]. We first show how to convert a typed graph to our E/R schema, then describe general graphs. Mapping a typed graph to our schema has the following two steps: First, store vertices of the same type into one entity table. Each attribute of the vertices becomes one column in the table. Second, store edges that have the same type into the same relationship table. Edges of the same type have source (resp. target) nodes that have the same type.

GQ-Fast supports SQL schemas. It classifies tables into *entity* tables $E_1, \ldots, E_m$ and *relationship* tables $R_1, \ldots, R_n$. The naming relates to the well-known E/R schema design technique [13]. Intuitively, entity tables correspond to entities of the E/R design, whereas the relationship tables correspond to many-to-many relationships.

[1] In this paper, we focus on relationship tables with two foreign keys.

Entity tables have to follow a single convention: Each GQ-Fast entity table must have an integer primary key (aka ID) attribute. In practice, this convention does not limit generality, since database schema designers often follow it. This convention is also recommended when translating an E/R design into a schema [13]. In our examples, the ID attributes are always named ID.

In the general case, each relationship table $R(F_1, \ldots, F_d, M_1, \ldots, M_k)$ has d *foreign key attributes* $F_1, \ldots, F_d$, where each foreign key F_i refers to the ID of an entity E_i. The intuition, according to E/R design, is that the foreign key F_i corresponds to the connection between the many-to-many relationship (that corresponds to the relationship table R) and the entity E_i. To make this connection clear in our examples, the foreign key attribute F_i has the same name as the entity E_i. E.g., in the PubMed example, the $\mathrm{Substance}$ foreign key of the DS table points to the ID of the $\mathrm{Substance}$ table. The relationship table may also have k attributes $M_1, \ldots, M_k$ that are not foreign keys. They correspond to the attributes of the many-to-many relationship in the E/R design. We call $M_1, \ldots, M_k$ *measure attributes*. For example, the *Fre* attribute of the DS table is a measure attribute. As is typical in E/R-based schema design, a one-to-many relationship between entities does not have a corresponding relationship table. Instead, it is captured by including a foreign key attribute to the entity table corresponding to the entity on the "many" side of the relationship. For example, the entity table Doc has a *Journal* foreign key attribute, which captures the many-to-one relationship between documents and journals.

10.2 Relationship Queries

As explained in 9, a relationship query proceeds in three steps: (i) *Context Selection*: Defining the "source" entities from which graph discovery will start. (ii) *Graph Discovery* To reach target entities from the context, queries "navigate" between entities via join operations; and (iii) *Relevance Computation:* The relevance between each target entity and the context is computed by applying aggregation functions over measure attributes collected in the second step.

In its algebraic form, a relationship query involves σ (selection), π (projection), $\bowtie$ (join), $\ltimes$ (semi-join) operators and an optional γ (aggregation) at the end, and must satisfy the follow restrictions: (i) join and semijoin conditions are equalities between (primary or foreign) key attributes and (ii) aggregations group-by on a primary key or foreign key. The set of relationship queries includes graph reachability (path finding) queries, where the edges are defined by foreign keys. More generally, it includes tree pattern queries, followed by aggregation. The first restriction does not narrow down the scope of relationship query applications as it only requires that navigation on a graph should be performed via connected edges, which is a natural requirement for graph navigation. The second restriction allows GQ-Fast to use an array to maintain the aggregation results instead of using a map.

Example Queries. We now illustrate a number of relationship queries using the datasets of some GQ-Fast applications: PubMed and SemMedDB. These queries were used in our

experiments and are implemented in our interactive demo system.

Even though the definition of relationship queries includes a larger set of queries, we focus on these queries, because they illustrate accurately the use cases for which GQ-Fast was designed and achieves the best speedup compared to other database systems: queries with long join paths involving many-to-many relationships. In the following examples, we use $E_1 \rightarrow E_2$ to visualize a join from table E_1 to table E_2 and $\circlearrowleft_E$ to visualize an intersection on table E.

Example Queries in PubMed

FSD (Frequency-Time-aware Document Similarity). Query FSD computes time-aware and frequency-aware cosine similarity. The cosine similarity is computed as follows: Each document d is associated with a vector $t^d = [t_1^d, \ldots, t_n^d]$, where n is the number of terms across all documents. The cosine similarity between two documents x and y is defined as $\sum_{i=1,\ldots,n} t_i^x t_i^{y2}$ In contrast to Query SD in the Introduction, Query FSD raises the similarity degree of documents that are chronologically close. The navigation path of Query FSD can be visualized as $\mathrm{d1} \rightarrow \mathrm{dt1} \rightarrow \mathrm{dt2} \rightarrow \mathrm{d2}$.

```
SELECT dt2.Doc, SUM(dt1.Fre * dt2.Fre) / (abs(d1.Year-d2.Year)+1)
FROM (((Doc d1 JOIN DT dt1 ON d1.ID = dt1.Doc)
      JOIN DT dt2 ON dt1.Term = dt2.Term)
     JOIN Doc d2 ON d2.ID = dt2.Doc)
WHERE d1.ID = d0^ID
GROUP BY dt2.Doc
```

AD (Authors' Discovery). Query AD finds the authors who published papers that pertain to the terms identified by $t_1^{ID}, \ldots, t_n^{ID}$ (e.g., authors that published papers related to the terms "neoplasms" and "statins") and counts the number of papers per author. The navigation path of Query AD can be visualized as $\circlearrowleft_{dt} \rightarrow \mathrm{da}$.

```
SELECT da.Author, COUNT(*)
FROM DA da
WHERE da.Doc IN
  (SELECT dt.Doc FROM DT dt WHERE dt.Term = t1^ID)
  INTERSECT
  ...
  INTERSECT
  (SELECT dt.Doc FROM DT dt WHERE dt.Term = tn^ID)
GROUP BY da.Author
```

FAD (Co-Occurring Terms Discovery). Query FAD is similar to Query AD. It finds other terms that co-occur in documents about terms identified by $t_1^{ID}, \ldots, t_n^{ID}$ along with the number of occurrences (e.g., terms that co-occur in documents about "neoplasms" and "statins" and how often). The navigation path of Query FAD can be visualized as $\circlearrowleft_{dt} \rightarrow \mathrm{dt1}$.

[2] In practice, the queries also normalize for the sizes of t^x and t^y and, in later examples, the sizes of measures. The examples exclude the normalization since they do not present any important additional aspect to the exhibited query pattern.

```
SELECT dt1.Term, Sum(dt1.Fre)
FROM DT dt1
WHERE dt1.Doc IN
  (SELECT dt.Doc FROM DT dt1 WHERE dt1.Term = t_1^ID)
  INTERSECT
  ...
  INTERSECT
  (SELECT dt.Doc FROM DT dtn WHERE dtn.Term = t_n^ID)
GROUP BY dt1.Term
```

AS (Author Similarity). Query AS finds the authors whose publications relate to the Mesh terms in the publications of a given author, identified by the id a^{ID}. The navigation path of Query AS can be visualized as $\mathrm{da1}\rightarrow\mathrm{dt1}\rightarrow\mathrm{dt2}\rightarrow\mathrm{d}\rightarrow\mathrm{da2}$.

```
SELECT da2.Author,COUNT(*)
FROM (((DA da1 JOIN DT dt1 ON da1.Doc=dt1.Doc)
       JOIN DT dt2 ON dt1.Term = dt2.Term)
      JOIN Doc d ON dt2.Doc=d.ID)
JOIN DA da2 ON dt2.Doc=da2.Doc
WHERE da1.Author = a^ID
```

Furthermore, we could evaluate each discovered author through a weight/similarity score: first we compute the similarity of the publications using the cosine of the term frequencies and then weigh recent publications heavier.

```
SELECT da2.Author,SUM(dt1.Fre × dt2.Fre)/(2017-d.Year)
FROM (((DA da1 JOIN DT dt1 ON da1.Doc=dt1.Doc)
       JOIN DT dt2 ON dt1.Term = dt2.Term)
      JOIN Doc d ON dt2.Doc=d.ID)
JOIN DA da2 ON dt2.Doc=da2.Doc
WHERE da1.Author = a^ID
```

SDY (Similar Documents in a specific Year). Query SDY is similar to query SD; Their only difference is that in query SDY, a parameter $year_0$ limits the result documents to the user's specified year of publishing .The navigation path of Query SDY can be visualized as two paths : (i) $P_1 = \mathrm{dt1}\rightarrow\mathrm{dt2}$, and (ii) $P_2 = \mathrm{d}\rightarrow\mathrm{dt2}$. Path P_2 converges into P_1 at $\mathrm{dt2}$.

```
SELECT dt2.Doc, COUNT(*) AS similarity
FROM DT dt1
  JOIN DT dt2 ON dt1.Term = dt2.Term
  JOIN Doc d AS d ON dt2.Doc = d.Id
WHERE dt1.Doc = d_0^ID AND d.year = year_0
GROUP BY dt2.Doc
ORDER BY similarity DESC
```

ATY (Author's Terms in a specific Year). Query ATY finds the terms of an author that appear on that author's publications of a specific year. The wieight of each terms is measured by its number of occurrences. The navigation path of Query ATY can be visualized as two paths : (i) $P_1 = \mathrm{da1}\rightarrow\mathrm{dt1}$, and (ii) $P_2 = \mathrm{d1}\rightarrow\mathrm{da1}$; P_2 essentially filters the tuples of $da1$, so that only docments published in $year_0$ are allowed in the graph discovery . Path P_2 converges into P_1 at $\mathrm{da1}$.

```
SELECT dt1.term, COUNT (*) As similarity
FROM DA da1 JOIN Doc d1 ON da1.doc = d1.article
  JOIN DT dt1 ON da1.article = dt1.doc
```

```
WHERE da1.author = author_id AND d1.year = year
GROUP BY dt1.term
ORDER BY similarity DESC
```

DAY (similar Documents of an Author per Year). Query DAY calculates the number of articles of a given author ($author_{id}$), per year, that are similar (share at least one term) with a specific article $document_{id}$. The navigation path of Query DAY can be visualized as two paths : (i) $P_1 = \mathrm{da}{\rightarrow}\mathrm{d}$, and (ii) $P_2 = \mathrm{dt1}{\rightarrow}\mathrm{dt2}{\rightarrow}\mathrm{d}$. Path P_2 converges into P_1 at d.

```
SELECT d.year, Count(*)
FROM DA da JOIN Doc d ON da.article = d.article
WHERE da.author = author_id AND da.article IN
(
    SELECT dt2.article
    FROM DT dt1 JOIN DT dt2
    ON dt1.term = dt2.term
    WHERE dt1.doc = document'id
)
GROUP BY d.year
```

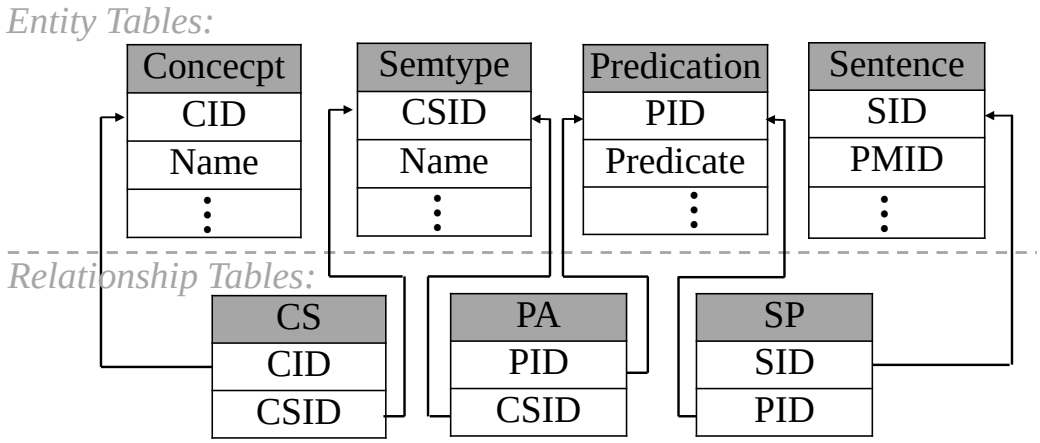
A2X (Authors that used certain terms in documents published in specific years). Query A2X displays the ability of the Selection Operator to support multiple selection values. It recieves as input two integers donoting the preffered years, and two terms. Query A2X returns the authors that used one or both of those terms in articles that were published in one of the given years. A similarity counter is computed for each author, depending of the number of articles; articles that contain both of the given terms are counted twice for the similarity counter. The navigation path of Query A2X can be visualized as two paths : (i) $P_1 = \mathrm{d}{\rightarrow}\mathrm{da}{\rightarrow}\mathrm{dt}$, and (ii) $P_2 = \mathrm{dt}{\rightarrow}\mathrm{dt}$ (P_2 only identifies valid entities of documents that contain the selected terms. Path P_2 converges into P_1 at dt.

```
SELECT da.Author, COUNT(*) AS similarity
FROM Doc d JOIN DA da ON d.Doc = da.Doc
JOIN DT dt ON da.Doc = dt.Doc
WHERE (d.Year =year_1 OR d.year =year_2)
   AND (dt.Term =term_1 OR dt.Term = term_2)
GROUP BY da.author
```

Example Queries in SemMedDB

CS (Concept Similarity). As a use case of Knowledge.Bio, Query CS finds the concepts that are most relevant to a given concept, e.g., "Atropine", where c^{ID} is the concept ID of "Atropine". The navigation path of Query CS can be visualized as $\mathrm{c1} \rightarrow \mathrm{p1} \rightarrow \mathrm{s1} \rightarrow \mathrm{s2} \rightarrow \mathrm{p2} \rightarrow \mathrm{c2}$.

```
SELECT c2.CID, COUNT(*)
FROM CS c2, PA p2, SP s2
WHERE s2.PID = p2.PID
  AND p2.CSID = c2.CSID AND s2.SID IN
    (SELECT s1.SID
     FROM CS c1, PA p1, Sp s1
     WHERE s1.PID = p1.PID AND p1.CSID = c1.CSID
       AND c1.CID = c^ID)
GROUP BY CID
```



CS(CID, CSID) : Concept_Semtype(concept_id, concept_semtype_id)
PA(PID, CSID) : Predication_Argument(predication_id, concept_semtype_id)
SP(SID, PID) : Sentence_Predication(sentence_id, predication_id)

Figure 10.1: SemMedDB Database Schema. CS, PA and SP are relationship tables, whereas the others are entity tables.

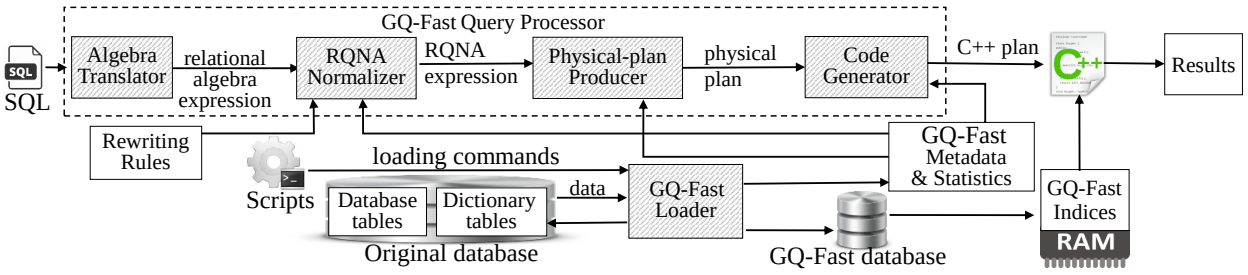


Figure 10.2: Architecture of Relationship Query Processing in GQ-Fast. The *GQ-Fast Loader* produces the GQ-Fast database and metadata, and the *GQ-Fast Query Processor* generates code answering a given relationship query.

The running time of this query on an Amazon Relational Database Service (Amazon RDS) with MySQL was 25 minutes. GQ-Fast reduced the running time for that query to less than 1 second [21].

10.3 Further Examples

Relationship queries can be found in a variety of applications. Some examples are outlined in the following list.

Potential Virus Discovery in Network Security [12, 18]: Consider a database documenting virus infections in a computer network with tables for "virus" entities, "host IP" entities, and virus instance - host IP relationships. To discover potential virus infections for a host who has reported a virus s, a relationship query first selects the set of hosts associated with s, and then retrieves and aggregates all the virus infections know for these hosts. The viruses with the high scores might also hide in the host computer.

Friend Suggestion in Social Networks [25]: Consider a database with "user" entities, "tweet" entities, and a relationship associating tweets with users. For example, the relationship captures the information that a user read or shared a tweet. To provide friend suggestions for a given user u, a relationship query first discovers his/her tweets, then returns a sorted list of users based on their association with the discovered tweets.

Chapter 11

User Stories (Demo UI)

Utilizing GQ-Fast's runtime efficiency, we designed an interactive user interface (Demo UI) for searching and browsing PubMed's data. In the following demonstration, we explore various pieces of information about authors in the biomedical field. More specifically, through a search form with autocomplete support, we select a specific author and, optionally, choose a substance and/or a disease. After doing so, a result section appears, showing the results of the relationship queries that correspond to the search form's selections.

In the following sections we present images from out demonstrative application, along with explanations about their content and the relationship queries that were conscripted.

11.1 Search Form

Initially, our interface consists of the following input form of the picture 11.1

Figure 11.1: Search Form of our Demo User Interface

We focus on the use case where the user is interested in discovering knowledge about an author. In the search form above, there is an autocomplete utility for the fields: Author Name, Disease and Substance. Depending on the user's input in each of these fields, as well as the existing content of the other fields, our application decides on a specific relationship query that runs on PubMed data and returns the autocomplete list. In order for the autocomplete utility to be interactive, the application depends on GQ-Fast's runtime efficiency for the autocomplete queries.

The autocomplete function is based on the Reverse Index Manager (a sub application of GQ-Fast). The Reverse Index Manager utilizes the Trie data structure to create Tree-based indexes for PubMed's Authors, Diseases and Substances. Its role is to answer inverse search queries, where a prefix string of a field is given, and the result must be a list of all of the index's strings that begin with the given prefix, along with an importance counter. There are two types of importance metrics:

- Non-contextual Importance: The number of occurrences of a PubMed entity (i.e. the number of published papers is the non-contextual importance for an author).
- Contextual Importance: The number of occurrences of a PubMed entity, on account with other restrictions that result from the input data of the rest of the fields of the search form. For example: If the Disease field contains the term "neoplasms", then the importance counter of the autocomplete list's of authors is the number of published documents of each author that contain the term "neoplasms".

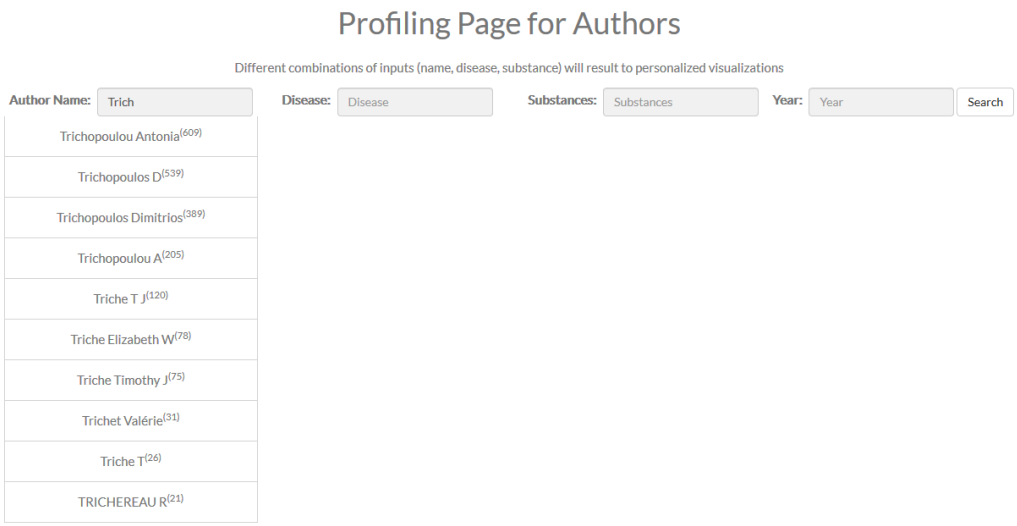


Figure 11.2: Autocomplete List for Authors that automatically generates after user's input.

In the image11.2, we observe the following:

- The user filled the field "Author Name" with the prefix: "Trich".
- Our application (through its API) sends a reverse search request with the user's prefix string to GQ-Fast's Reverse Index Manager.
- The reply from the Reverse Index Manager consists of a sorted list containing tuples of author names and their importance counter. The list is sorted on the importance counter, and in this case where none of the other form fields is filled, we can see each author's non-contextual importance (the number of published documents).

After selecting an author from the autocomplete list, the user proceeds to typing into the Disease field (image 11.3).

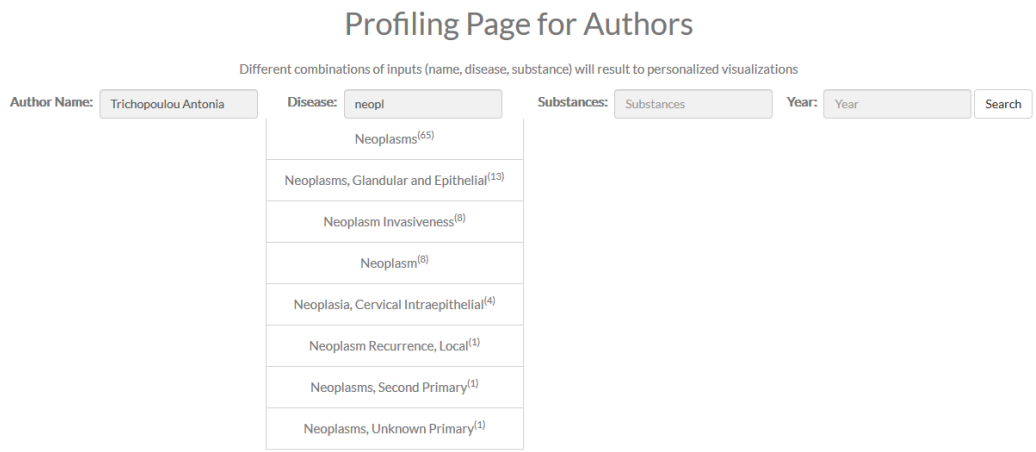


Figure 11.3: Autocomplete List for Diseases that automatically generates after user's input.

In the image 11.3 observe the following:

- The user filled the field "Disease" with the prefix: "neopl".
- Again, our application sends an inverse index search requests to the Reverse Index Manager with the prefix string "neopl".
- ince the Author field is already filled, the autocomplete list with the results is sorted on the counter of contextual importance. This means that the Reverse Index Manager, after discovering the disease names that begin with the user's prefix, run the appropriate GQ-Fast query that returned the number of occurrences of each disease to the selected author's publications. The corresponding SQL query for this utility is the following:

```
SELECT Name, COUNT(*)
FROM DD dd
JOIN Disease d ON dd.Disease = d.Id
JOIN DA da ON dd.Doc = da.Doc
JOIN authors auth ON da.Author = auth.Id
WHERE LOWER(Name) LIKE 'neopla%' AND LOWER(auth.Name) LIKE ↩
    LOWER('Trichopoulou Antonia%')
GROUP BY d.Id
ORDER BY COUNT DESC
```

- PostgreSQL computes the above query in: 4,742 ms
- GQ-Fast computes the above query in: 432 ms

11.2 Result Section

Right after a selection from the autocomplete list of a field of the search form, the result section is automatically loaded. This section contains tables that result from various relationship queries that correspond to the user's input. Specifically for our example where the selected author is "Trichopoulou Antonia" and the selected disease is the term "neoplasms" (while all the other fields are left empty), the result is the image 11.4.

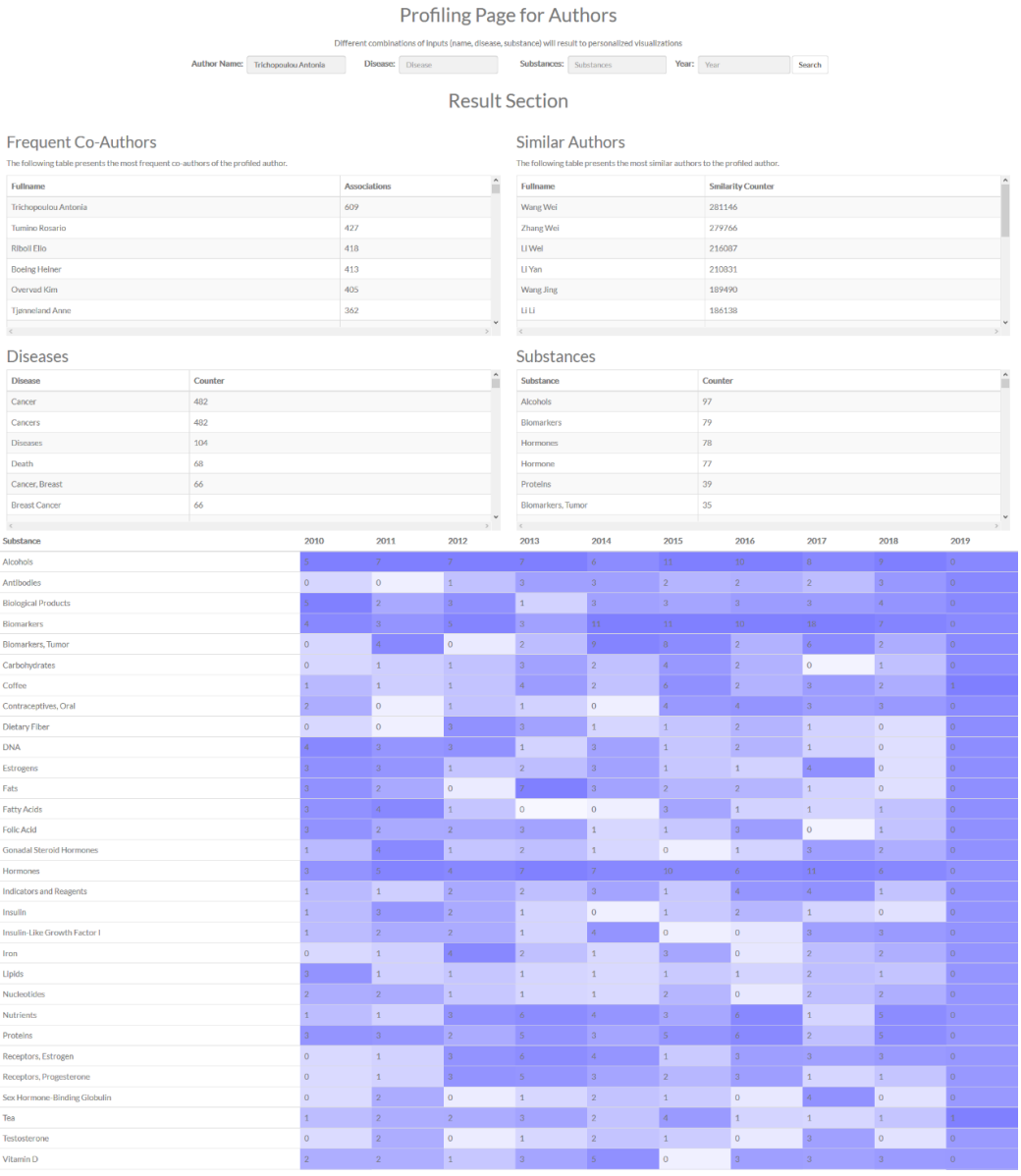


Figure 11.4: Screen with result tables according to the user's total input.

We can see 4 tables and a Heat Map. In greater detail:

1. Frequent Co-Authors: The authors that are the most often collaborators of the selected author on PubMed's publications. The SQL for this table's relationship query is the following:

```
SELECT Name, COUNT(*)
FROM Author auth JOIN DA da ON auth.Id = da.Author
      JOIN DA da2 ON da.Article = da2.Article
      JOIN Author auth2 ON da2.Author = auth2.Author
WHERE auth.Name = 'Trichopoulou Antonia'
GROUP BY auth2.Id
ORDER BY count DESC LIMIT 100
```

2. Similar Authors: Authors that have used the same terminology with the selected author of the search form. The importance field of this table represents the number of documents of an author that contain one or more terms that the selected author has used. The corresponding SQL of the relationship query for this table is the following:

```
SELECT auth2.NAME, COUNT(*)
FROM Author auth JOIN DA da ON auth.Id = da.Author
    JOIN DT a2s1 ON da.Doc = a2s1.Doc
    JOIN DT a2s2 ON a2s1.Term = a2s2.Term
    JOIN DA da2 ON a2s2.Doc = da2.Doc
    JOIN Author auth2 ON da2.Author = auth2.Id
WHERE auth.NAME = 'Trichopoulou Antonia'
GROUP BY auth2.NAME
ORDER BY count DESC LIMIT 100
```

3. Diseases: The contents of this table depend of the input in the fields of the search form. In the general case where only an author has been selected and all the other fields are empty, the information of the table represents the most common diseases that appear in the selected author's documents. The corresponding SQL of the relationship query is the following:

```
SELECT d.Name, COUNT(*)
FROM Disease d JOIN DS ds on d.Id = ds.Disease
JOIN DA da on ds.Doc = da.Doc
JOIN Author auth on auth.Id = da.Author
WHERE authors.name = 'Trichopoulou Antonia'
GROUP BY d.Id
ORDER BY count desc LIMIT 100;
```

4. Substances: The contents of this table depend of the input in the fields of the search form. In the general case where only an author has been selected and all the other fields are empty, the information of the table represents the most common substances that appear in the selected author's documents. The corresponding SQL of the relationship query is the following:

```
SELECT s.Name, COUNT(*)
FROM Substance s JOIN DS ds on s.Id = ds.Substance
JOIN DA sa on ds.Doc = da.Doc
JOIN Author auth on auth.Id = da.Author
WHERE auth.name = 'Trichopoulou Antonia'
```

```
GROUP BY s.Id
ORDER BY count desc LIMIT 100;
```

5. Heat Map: The Heat Map depicts the most common substances that appear in the selected author's documents in the last ten years. The intensity of the color of each cell depends on the comparison of its content with the column's mean. The SQL of the corresponding relationship query for this table is the following:

```
SELECT d.Name, year, COUNT(*)
FROM Disease d JOIN DD dd on d.Id = dd.Disease
  JOIN DA da on ds.Doc = da.Doc
    JOIN Author auth on auth.Id = da.Author
    JOIN Doc dc on da.Doc = dc.Id
WHERE auth.Name = 'Trichopoulou Antonia' AND dc.Year > 2000
GROUP BY d.Name, dc.Year
ORDER BY count desc LIMIT 100
```

At this point, our first use-case scenario is over. An interactive search form with autocomplete support and intelligent representations in its autocomplete lists guides the user while he enters the name of the author that he is interested in. Right after the author's selection and in less than a second, a result section appears with useful and case-specific information. We extend the case scenario with the selection of a disease:

Through the search form, we type the word neoplasms in the "Disease" field (and select the appropriate term from the autocomplete list). The result page is automatically updated. The new result section differs from the previous one on the tables: Diseases (11.5) and Substances (11.6).

11.3 Execution Time for Demo's Queries

Depending on the user's input(s) in the search form, the relationship queries that have to run can be excessively time consuming for a conventional database, making the application too slow for human-machine interaction. With GQ-Fast, the autocomplete lists and the result tables are most often loaded in less than 1 second, outperforming PostgreSQL by 1-3 orders of magnitude. Table 15.2 compares GQ-Fast and PostgreSQL on all the queries shown in the Demo User Interface.[1][2]

[1] Author's autocomplete in the case where a disease term is already selected.
[2] Disease's autocomplete in the case where an author is already selected.

Diseases

Disease	Counter
Neoplasms	65
Cancer	63
Cancers	63
Death	18
Cardiovascular Diseases	18
Diseases	17
Chronic Disease	11

Figure 11.5: Updated Table of Diseases after user's change in the content of the search form.

Substances

Substance	Counter
Alcohols	15
Biomarkers	9
Nutrients	5
Indicators	5
Nutrient	5
Olive Oil	5
Oil, Olive	4

Figure 11.6: Updated Table of Substances after user's change in the content of the search form.

Chapter 12

Architecture

Applications use GQ-Fast as an OLAP-oriented database that accompanies their original transaction-oriented databases. Figure 10.2 gives an overview of GQ-Fast's architecture. It has two parts: *GQ-Fast Database Generation* and *GQ-Fast Query Processing*.

GQ-Fast Database Generation. The *GQ-Fast Loader* receives loading commands, retrieves data from one or multiple relational databases, and creates GQ-Fast indices along with relevant metadata, containing information about fragments and their encodings. This phase is done offline. The schema of the GQ-Fast database has to follow certain conventions (see Section 10). GQ-Fast data is stored in main memory data structures (see Section 13).

When loading data into GQ-Fast, users should specify (i) the columns to be indexed, upon which GQ-Fast builds lookup tables. Then, GQ-Fast organizes the values in other columns as fragments; and (ii) an encoding method for each column excluding indexed columns. Section 13 provides detailed guidelines for choosing proper encoding methods for different columns.

GQ-Fast Query Processing. The *GQ-Fast Query Processor* receives an SQL query and outputs its result. It consists of several subcomponents. The *Algebra Translator* translates an SQL query into a relational algebra expression, which is then transformed into a *Relationship Query Normalized Algebra (RQNA) expression* (see Section 14.1) by the *RQNA Normalizer* using rewriting rules. Given an SQL query q in its algebraic format, the RQNA Normalizer applies the following rewriting rules to transform it into RQNA: (1) push every possible selection and projection down to the corresponding tables; projections are upon selections, (2) rewrite into a *left-deep*.

The *RQNA Normalizer* also verifies whether an SQL query is a relationship query by checking the restrictions according to metadata. Afterwards, the *Physical-plan Producer* transforms the RQNA expression into a physical-level plan. The *Code Generator* consumes the physical plan and metadata and produces C++ code, which is then compiled and ran on the GQ-Fast index to get final results. GQ-Fast can also prepare a query statement, and then execute it multiple times (as JDBC does), changing the parameters each time.

Chapter 13

GQ-Fast Index Structure

This section first presents the GQ-Fast index structure and analyzes different encoding methods, then describes how to build indices for both entity and relationship tables. Finally, a discussion of how to support incremental updates is provided.

Consider a relationship table $R(D_1, D_2, M_1, \ldots, M_m)$. The GQ-Fast data structure is optimized towards two goals: (i) rapidly evaluating $\pi_A \sigma_{D_i=c}(R)$, where A may be any attribute $D_j, j \neq i$ or M_j and (ii) minimizing space by using compressed data structures. The most important mechanism towards compression is *fragments* that encode each $\pi_A \sigma_{D_i=c}(R)$ using techniques such as compressed bitmaps and Huffman encoding.

For each table of a graph database, GQ-Fast stores two *indices* $\mathcal{I}_{R.D_1}$ and $\mathcal{I}_{R.D_2}$.[1] The only storage pertaining to R are these indices. The $\mathcal{I}_{R.D_1}$ index consists of a *lookup data structure* on D_1 and *fragments* corresponding to the other attributes. Given an ID c and an attribute A, a *lookup table* uses the data structure to return (i) a pointer to a byte array that encodes the fragment $\pi_A \sigma_{D_i=c}(R)$ and (ii) the size of the byte array, which is required by the algorithms that decode fragments.

Index Structure. Given a relation $R(C, C_1, C_2, ..., C_n)$, assuming the indexed column is C, GQ-Fast builds one index $\mathcal{I}_{R.C}$ (shown in Figure 13.1) for R. A GQ-Fast index has one lookup table for the indexed column C, and organizes values in columns $C_1, ..., C_n$ in fragments. We assume that $|R| = h$, consequently the column C contains IDs in the interval $[0, h-1]$. The lookup table $\mathcal{P}_C$ is a 2D array of size $(h+1) \times n$ and stores offsets into the respective fragments array designating the beginning of a fragment. All fragments are stored consecutively and byte-aligned in one *fragment byte array* per column. Specifically, $\mathcal{P}[t][m]$ stores the offset where fragment $\pi_{C_m} \sigma_{C=t}(R)$ starts in C_m's fragment byte array, where C_m is the $(m+1)$-th column of R. If a value $t \in C$ has no associated values in columns $C_1, ..., C_n$, then all fragments $\pi_* \sigma_{C=t}(R)$ are empty. The size of a fragment is defined implicitly as the difference between two consecutive offsets, which is why the size of the first dimension of $\mathcal{P}$ is $h+1$. For further space savings, offsets are encoded with the minimum number of bytes. In the following, we

[1] The administrator may elect to store only one of the two indices but then some queries will be not be amenable to GQ-Fast's efficient processing.

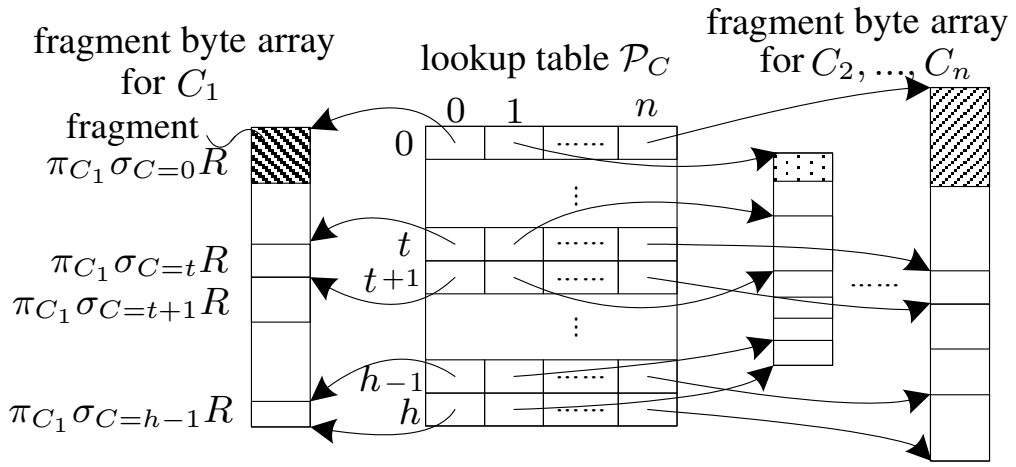


Figure 13.1: Index $\mathcal{I}_{R.C}$. The lookup table $\mathcal{P}_c$ stores offsets of fragments. Fragments of different columns have different encodings.

present various encodings for fragments utilized in this paper.

Retrieve a fragment $\pi_A\sigma_{F_i=c}(R)$**.** GQ-Fast first obtains an offset-arrays $\mathcal{P} = \mathcal{I}_{R.F_i}[c]$ by a random access, and its neighbour $\mathcal{P}_{next} = \mathcal{I}_{R.F_i}[c+1]$. Then GQ-Fast gets the start address of the fragment $\mathcal{F}_{R.A} = \mathcal{P}[A]$ and its length l=$P_{next}[A] - \mathcal{F}_{R.A}$. If this fragment is encoded by an encoding method **E**, GQ-Fast decodes it by using a macro $decode\mathbf{E}(\mathcal{F}_{R.A}, l : \mathcal{A}_{R.A}, n)$ to produce a decoded array $\mathcal{A}_{R.A}$ and the number of elements n within it. In the following, we introduce encodings for fragments utilized in this paper.

Fragments Encoding Methods. In a GQ-Fast index $\mathcal{I}_{R.C}$ of relation $R(C, C_1, C_2, ..., C_n)$, all the values associated with $t \in C$ in column C_i are organized as a fragment $\pi_{C_i}\sigma_{C=t}R$. GQ-Fast compresses fragments with different compression methods. GQ-Fast does not have any restrictions on compression methods, as long as fragments can be decompressed without accessing other fragments. It allows a wide range of encoding methods, including those that do not support random access within a fragment. GQ-Fast currently uses the following four methods for encoding single fragments.

- *Uncompressed Array (UA):* An uncompressed array stores the original numerical values in their declared type.
- *Bit-aligned Compressed Array (BCA):* Assume a foreign key attribute points to the IDs of an entity, which range from 0 to $h-1$. Then each foreign key value needs $\lceil \log_2 h \rceil$ bits. Consequently, a fragment $\pi_A\sigma_{F=c}R$ with size n requires $\lceil \frac{n \cdot \lceil \log_2 h \rceil}{8} \rceil$ bytes (including alignment-induced padding).
- *Byte-aligned Compressed Bitmap (BB):* Given an array of values $[v_1, \ldots, v_n]$, the equivalent uncompressed bit vector is a sequence of bits, such that the bits at the positions $v_1, \ldots, v_n$ are 1 and all other bits are 0. GQ-Fast uses the byte-aligned method to compress bit vectors [4]. The first bit of a byte is a flag that declares whether (i) the next

Uncompressed Array (UA)	$32 \cdot N \cdot \lceil \log_{2^{32}} D \rceil$
Bit-aligned Compressed Array (BCA)	$8 \cdot \lceil \frac{N \cdot \lceil \log_2 D \rceil}{8} \rceil$
Byte-aligned Compressed Bitmap (BB)	$N \cdot (8 \cdot \lceil \log_{128} \frac{D-N}{N} \rceil)$
Huffman	$8 \cdot \lceil \frac{N \cdot E_D + D}{8} \rceil$

Table 13.1: Space Analysis of Encoding Methods

seven bits are part of a number that also uses consequent bytes or (ii) the remaining seven bits actually represent the length number by themselves.

- *Huffman-encoded Array (Huffman):* GQ-Fast employs Huffman encoding with an array-based encoding of the Huffman tree [11,20] to avoid tree traversals (i.e., random access on the heap). This can speed up decoding due to CPU L1/L2 caching effects.

13.1 Compression Quality Analysis

Table 13.1 summarizes the space needed by each fragment. Assume that each fragment contains N elements, the domain size of the column containing this fragment is D, $E_D = -\sum_{i=1}^{D} p_i \log p_i$ is the entropy of the column, and p_i is the probability of occurrence of element i[2]. In our experiments, GQ-Fast chooses an optimal encoding for each column with minimal space cost by using the formulas in Table 13.1, where N is set to be the average fragment size on each column.

Different fragments of the same column may be most compactly encoded with different methods. For example, a fragment of more documents id's of a term should be encoded with BB, otherwise, it is less suitable to apply BCA. Though applying different encodings for different fragments can achieve minimal space cost, the penalty is that we need to remember the encoding method for each fragment, which increases the space cost. To balance this trade-off, we apply the same encoding (the one with minimal space cost for the fragment with average size) for fragments in the same column. Note that, fragments in different columns still benefit from applying different encodings. And only one encoding type is required to store for each column, which can be stored in the metadata.

Building GQ-Fast Indices. For an entity table $E(ID, M_1, \ldots, M_m)$, GQ-Fast chooses the ID column as the indexed column and creates one index $\mathcal{I}_{E.ID}$. Note that in an entity table, a fragment contains only a single value.

For a relationship table $R(F_1, F_2, M_1, \ldots, M_m)$ with two foreign keys F_1 and F_2, GQ-Fast chooses both F_1 and F_2 as indexed columns, which means GQ-Fast builds two indices $\mathcal{I}_{R.F_1}$ and $\mathcal{I}_{R.F_2}$ according to different indexed columns. The reason is that a relationship table refers to a collection of (potentially undirected) edges in graphs, and it is necessary to provide an efficient way to obtain fragments for both source vertices (in column F_1) and destination vertices (in column F_2). For scenarios where relationship

[2]We report the lower bound of the space needed by Huffman. The space needed by Huffman is bounded by $[8\lceil \frac{N \cdot E_D + D}{8} \rceil, 8\lceil \frac{N \cdot E_D + N + D}{8} \rceil)$ [23].

tables have more than two foreign keys, say $a > 2$, to fully index all the foreign key columns (if needed) GQ-Fast builds a indices, which may require a large amount of space.

Incremental Updates. GQ-Fast's compact storage strategy (storing all the fragments of the same attribute in one big fragment array and using offsets to refer to them) can significantly reduce space costs at the expense of incremental updates. To support incremental updates, GQ-Fast could (i) store each fragment independently and (ii) maintain explicit pointers for them. Theoretically, GQ-Fast will then require additional $N(64 - \lceil \log_2 N \rceil)$ bits, where N is the total number of distinct values in the indexed column.

As fragments may be encoded using Huffman encoding, it is challenging to maintain the optimality of Huffman-encoded fragments after massive updates. Dynamic Huffman encoding should be applied, which remains optimal as the weights change.

Chapter 14

GQ-Fast Query Processing

The GQ-Fast Query Processor (Figure 10.2) transforms a given query into an RQNA expression, which is then transformed into a plan of physical operators (e.g., the plan in Figure 14.3 corresponds to the RQNA expression in Figure 14.2(e)), which is then used together with metadata for C++ code generation. This section formally describes RQNA expressions, presents physical operators and key intuitions in the translation of RQNA expressions into plans, and describes how the GQ-Fast code generator translates plans into code, essentially by mapping each physical operator to an efficient code snippet and stitching these snippets together.

14.1 RQNA Expression

To efficiently answer relationship queries, GQ-Fast first translates them into RQNA (Relationship Query Normalized Algebra) expressions (Figure 14.1). In the simplest case, an RQNA expression is a left-deep series of joins with a selection and aggregation: In Line 6 the RQNA expression starts with a selection $\sigma_c(T \mapsto v)$ of qualifying entities – we call them the *context* entities[1]. Subsequently, the RQNA expression performs a series of left-deep joins (Line 3) that navigate to entities related to the qualifying entities. Optionally, an RQNA expression may group-by the key attribute k (Line 1), followed by multiple aggregations.

In more complex cases, an SQL query (as shown in later examples) may contain nested queries using SQL's IN syntax, where "IN" translates to semijoins (Line 7). Nested queries are themselves relationship queries (Line 9) without aggregation or the result of an intersection (Line 10). Figure 14.2 shows the RQNA expressions for all the queries evaluated in this paper.

(Lines 3,4) allow for even more complex cases, where the RQNA expression is not left-deep. This means that the joined operand is not just a selection operation on entities, but a

[1]The condition may be set to true, setting the context to all entities.

$$
\begin{array}{lcll}
RQNA & \Rightarrow & \gamma_{k;f_1(.)\mapsto N_1,\ldots,f_n(.)\mapsto N_{n_\gamma}}\, Join & (1)\\
 & & \text{attributes named } k \text{ are primary or foreign keys} & \\
 & | & Join & (2)\\
Join & | & Join \bowtie_{j.k_1=v.k_2} (\pi_{\bar{A}}(T \mapsto v)) & (3)\\
 & | & Join \bowtie_{j.k_1=v.k_2} Join & (4)\\
 & | & Join \ltimes_{j.k_1=v.k_2} Join & (5)\\
 & & j \text{ is a variable defined by } Join & \\
 & | & \pi_{\bar{A}}(\sigma_c(T \mapsto v)) & (6)\\
 & | & \pi_{\bar{A}}((T \mapsto v) \ltimes_{v.k_1=x.k_2} Context) & (7)\\
 & & x \text{ is a variable defined by } Context & \\
 & | & \pi_{\bar{A}}(\sigma_c(Join \mapsto v)) & (8)\\
Context & \Rightarrow & \pi_{v.k} Join & (9)\\
 & | & \pi_{v.k}\sigma_{c_1}(T_1 \mapsto v) \cap \ldots \cap \pi_{v.k}\sigma_{c_n}(T_n \mapsto v) & (10)
\end{array}
$$

Figure 14.1: Grammar Describing RQNA Expressions

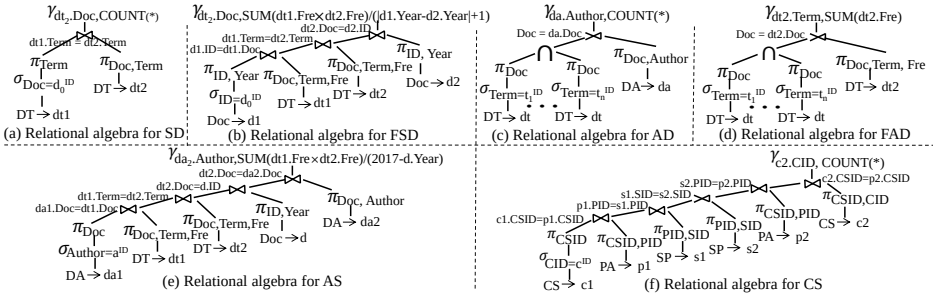


Figure 14.2: Relational Algebra Expressions for Queries on PubMed/SemMedDB

whole RQNA expression that lacks the Aggregation step. We can say that the $\bowtie_{j.k_1=v.k_2}$ operation of (Line 3) defines a converging point for two Spanning Trees, based on the Relationship Query definition of Chapter 9. For example, we can consider the Physical Algebraic Plan of query A2X, as the following Lists of Physical Operators (evaluated in the given order):

SELECTION˙OPERATOR - dt1.Term = term1 OR term2
HASHMAP˙OPERATOR - dt1.Doc as key

SELECTION˙OPERATOR -d1.Year = year1 OR year2
THREADING˙OPERATOR
JOIN˙OPERATOR -d1.Doc = da1.Doc
(HASH) JOIN˙OPERATOR -d1.Doc = dt1.Doc
AGGREGATION˙OPERATOR -da1.Author; Count(*)

The selection operation that provides the context entities (Line 6), filters entities autonomously, regardless of the attributes of other entities that they may be connected with

during the step of Gragh Discovery (as defined in Relationship Query definition in: 9). (Line 8) enables for mid-navigation filtering of sets of entities. More specifically, a Spanning Tree (or, in the simlest case, a path) of the graph that has been discovered due to sequences of Joins, can be filtered as a whole with a logical expression c that contains attributes from multiple entities of the Spanning Tree.

14.2 Physical Operators

This section explains the physical operators' syntax and semantics, neglecting for now the bottom-up pipelined execution aspects.

Fragment-based Join. The operator $\overset{\rightarrow r.A_1,\ldots,r.A_n}{\ltimes}_{B;R\mapsto r} \mathcal{I}_{R.B'}L$ receives as input the result of an expression L that produces a column B, generally among others. For each value $b \in B$, the operator uses the index $\mathcal{I}_{R.B'}$ to retrieve (and decompress) the fragments $\pi_{A_i}\sigma_{r.B'=b}$ $(R \mapsto r)$ for $i = 1, \ldots, n$. Intuitively, L would be the left operand of a conventional join and $R \mapsto r$ would be the right side. Conceptually, one may think that the fragments are combined into a result table whose schema has the attributes $A_1, \ldots, A_n$ (and also the attributes of L). However, in reality, the decompressed fragments are not combined into rows. In adherence to the late binding technique [1,2] of column-oriented processing, the ordering of the items in the fragments dictates how they can be combined into tuples. The $\overset{\rightarrow}{\ltimes}$ operator is useful for executing both selections and joins of the RQNA expressions:

- A projection/join combination $\pi_{attrs(L),r.A_1,\ldots,r.A_n}(L \ltimes_{B=r.B'} R \mapsto r)$ where B is an attribute of L and B' is a foreign key of a relationship table R or B' is the ID of an entity table R, translates to $L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\ltimes}_{B;R\mapsto r} \mathcal{I}_{R.B'}$.
- A projection/selection combination $\pi_{r.A_1,\ldots,r.A_n}\sigma_{r.B'=c}(R \mapsto r)$, where c is a constant and B' is a foreign key of a relationship table R or B' is the ID of an entity table R, translates to $\{[B : c]\} \overset{\rightarrow r.A_1,\ldots,r.A_n}{\ltimes}_{B;R\mapsto r} \mathcal{I}_{R.B'}$. Essentially, GQ-Fast reduces the selection into a join, by considering the left-hand-side argument to be a table with a single tuple and a single attribute B, whose value is c.

Fragment-based Semijoin. The operator $\overset{\rightarrow r.A_1,\ldots,r.A_n}{\rtimes}_{B;R\mapsto r} \mathcal{I}_{R.B'}L$ operates similarly to the fragment-based join but returns only attributes from $(R \mapsto r)$ if there is a matching tuple in L. It is introduced in the plan when the RQNA expression has an expression $\pi_{r.A_1,\ldots,r.A_n}((R \mapsto r) \rtimes_{B=r.B'} L)$. The operator maintains a lookup structure for values from the B column of L; for each value $b \in B$, the operator checks the lookup structure to find out whether that particular value b was already received earlier. If L is relatively large, it is best to use a boolean array, despite the fact that the query needs to initialize all array elements to *false*. Otherwise, a hash set or a tree is preferable.

Hash Join. In Figure 14.1, (Line 4) defines a case where Fragment-based Join cannot be applied, since the right operand does not define entities that can be accessed through fragments, but a whole result table (with many attributes from different entities of a Spanning

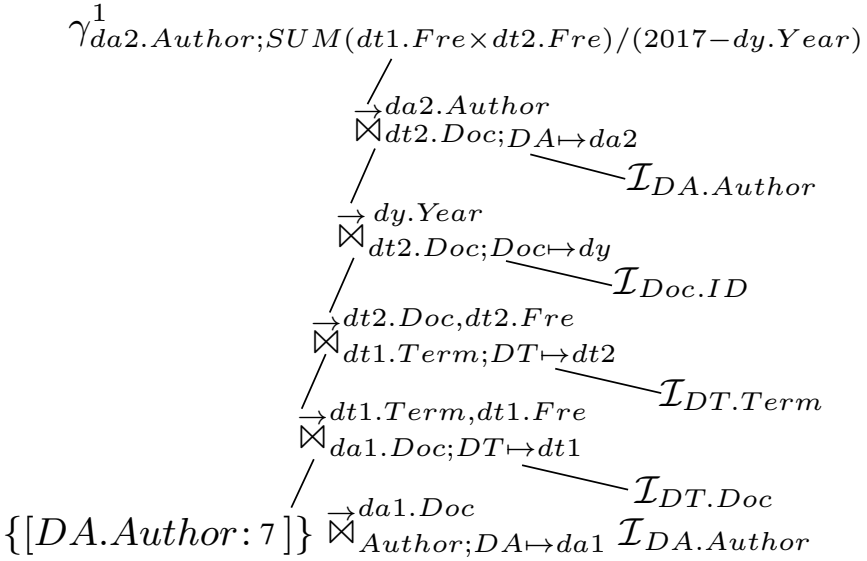


Figure 14.3: Physical Algebraic Plan for Query AS

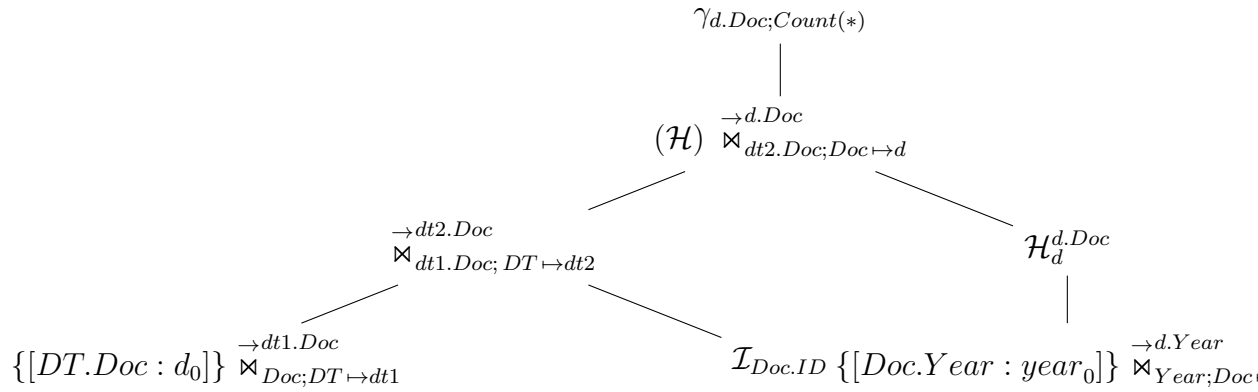


Figure 14.4: Physical Algebraic Plan for Query SDY. The Hash Join Operator is marked with $(\mathcal{H})$.

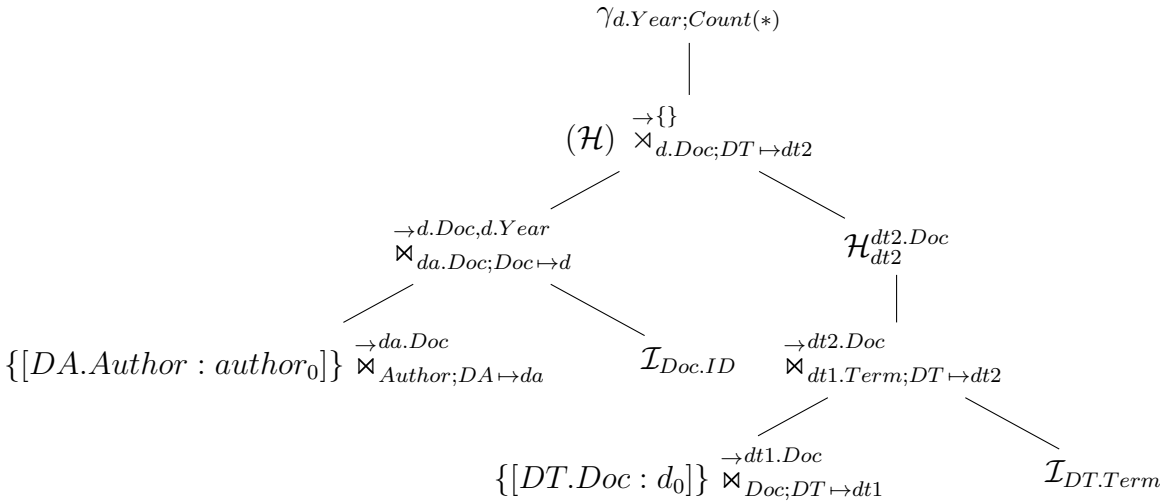


Figure 14.5: Physical Algebraic Plan for Query DAY. The Hash Semi Join Operator is marked with $(\mathcal{H})$.

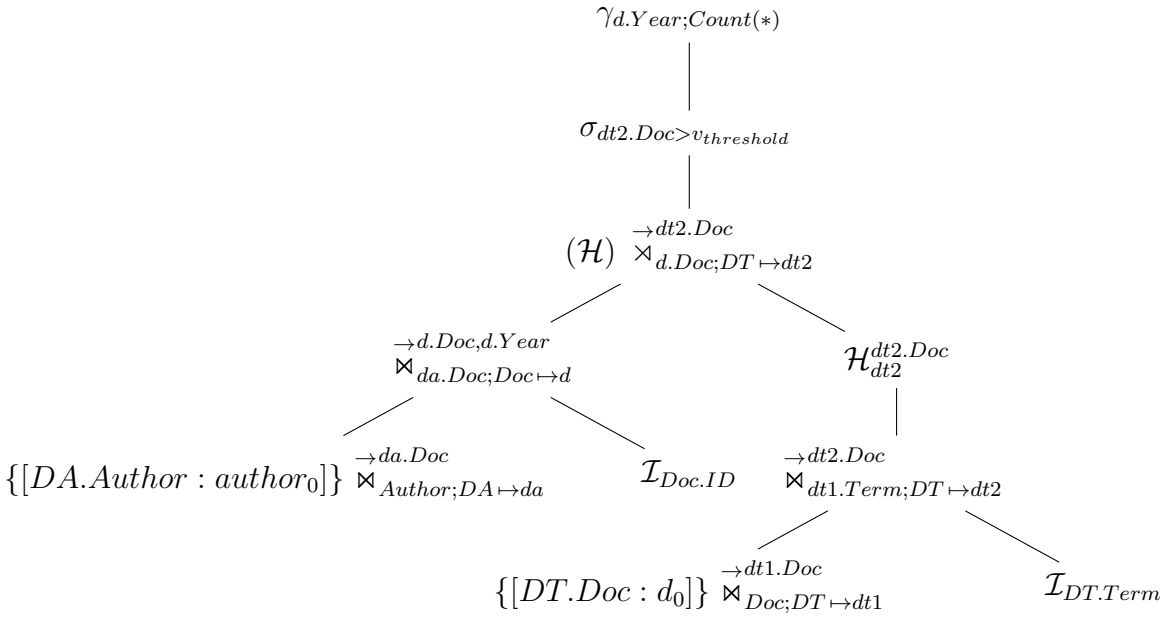


Figure 14.6: Physical Algebraic Plan for Query DAY with an On-The-Fly-Select Operator right before the Aggregation. The Hash Semi Join Operator is marked with $(\mathcal{H})$. In this version of the DAY query, GQ-Fast recieves as input a threshold value for the logical expression of the On-The-Fly-Select.

Tree) of an RQNA sub-expression of a sequence of Fragment-based Joins. Hash Join Operator $\overset{\rightarrow r.A_1,\ldots,r.A_n}{\ltimes_{B;R\mapsto r}} L$ recieves as input the result of an RQNA expression L that produces a column B, generally among others. For each value $b \in B$, the operator retrieves the result table of the right operand (an expression that can be a full RQNA query and, consequently, may contain attributes from different combinations of entities) from a Hash Map (which is created by the Hash Map Operator). The keys of the Hash Map are also one of the attributes of the left operand on which the Join is based. For example, in query SDY, the key of the Hash Map that needs to be used, is the attribute $dt2.Doc$. The values of the Hash Map in this example consist of the attributes : $d.Doc$ and $d.Year$ [2]. The branch of the Hash Join that creates the Hash Map (the decision of wich play that role can be made arbitrarily and is usually based on efficiency) is evaluated autonomously before the evaluation of the rest of the physical plans in GQ-Fast, since at the time of the application of the Hash Map Operator, the whole Hash Map needs to be computed. Figure 14.4 shows a Physical Plan with a Hash Join $((\mathcal{H}) \overset{\rightarrow dt2.Doc}{\ltimes_{d.Doc;Doc\mapsto d}})$; the key of the corresponding Hash Map is the attribute $d.Doc$, and the join operation checks wether $dt2.Doc$ exists in the Hash Map and, if it does, proceeds with a for loop for each set of values that are retrieved from the Hash Map. In this case, no values are retrived (in fact, a Hash Set can be used instead o a Hash Map) and therefore the Hash Join operator actually works as a filter in this example.

Hash Semi Join. Similarly to Semi Join, the $\overset{\rightarrow r.A_1,\ldots,r.A_n}{\ltimes_{B;R\mapsto r}} L$ operator maintains a lookup structure for values from the B column of L; for each value $b \in B$, the operator checks the lookup structure to find out whether that particular value b was already received earlier. If it is the first encounter of b, the operator retrieves the values $r.A_1, \ldots, r.A_n$ from the corresponding Hash Map exactly like Hash Join Operator does.

Hash Map Operator. Hash Join & Hash Semi Join operators basically seperate a branch of the Join operation to be evaluated autonomously, as we describe above. Internally, the Query Processor creates a new RQNA expression from that branch and instead of an Aggregation, it places a Hash Map Operator ($H_{r_i\mapsto L}$) at the beginning of the list of the Physical Plans. The attribute (which must be a primary or foreign eky of an entity) r_1 is the key of the Hash Map, and L is the RQNA sub-expression that is evaluated to a set of rows that populate the Hash Map. The Hash Map Operator is responsible for creating and populating the Hash Map that will be utilized by the corresponding Hash Join or Hash Semi Join. Figure 14.4 shows a Hash Map operator ($\mathcal{H}_d^{d.Doc}$) that creates a Hash Map with the attribute $d.Doc$ as the key, and a set of lists of d.Doc values for each key. Figure 14.5 shows a Hash Map operator ($\mathcal{H}_{dt2}^{d.Doc}$) that creates a Hash Map with the attribute $dt2.Doc$ as the key, and and an empty set of values for each key.

On-The-Fly-Select. In (Line 8), a mid-expression filter is defined. On-The-Fly-Select operator checks whether the logical expression of the selection (σ_c) is true for the Spanning

[2]If we replace $d.Doc$ with $dt2.Doc$ in the Aggregation Operator then none of d's attributes are needed for the evaluation of the RQNA expression after the Join and we can ignore them. In this case, instead of a Hash Map, a Hash Set can be used.

Tree that has been discovered up to this point. If the logical expression is false, the Spanning Tree itself (as a mid-term result for the final aggregation) is discarded. Figure 14.6 shows a case where On-The-Fly-Select operator filters the valid nodes that are discovered (Spanning Tree) based on the logical expression $dt2.Doc > v_{threshold}$[3] Also, in Figure 14.6 the operator $\overset{\rightarrow dt2.Doc}{\rtimes}_{d.Doc;DT \mapsto dt2}$ does not retrieve an empty list from the Hash Map, but the attribute $dt2.Doc$, since it is needed for the application of the On-The-Fly-Select.

Threading Operator. For performance purposes GQ-Fast uses the Threading Operator to indicate to the Code Generator that the rest of the operators of the physical plan list must be implemented in a multi-threaded environment. Optimally, the Threading Operator should be incorporated in the Physical Plans by GQ-Fast's Physical Plan Producer, after dynamically evaluating the corresponding RQNA query and deciding the operator's most efficient usage (like the number of threads to be used and the right place in the physical plan list)[4]. At the moment, GQ-Fast automatically inserts a Threading Operator after the Selection Operator (which is always the first operator) in a list of physical operators if there is also a Join or Semi Join Operator present.

Aggregation. Recall, in relationship queries the single group-by attribute $r.D$ is the foreign key of a relationship table or the ID of an entity. In either case, the range of $r.D$ is the same as the range of the underlying entity ID. Consequently, the γ^1 operator's superscript 1 signifies the assumption that the domain of G is small enough to allow for the allocation of an array, whose size is the domain of $r.D$ and each entry is a number, initialized to zero. Every time a "tuple" from r is processed, this array is updated at $r.D$ accordingly. In addition, an array of booleans registers which values of $r.D$ were actually found. For example, the aggregation $\gamma^1_{Author;\frac{SUM(DT_2.Fre \times DT_1.Fre)}{(2017-DY.Year)}}$ of the Query AS, as shown in Figure 14.3, initializes an array and a boolean register array with sizes of *domain size of* Author.

14.3 Code Generator

The GQ-Fast code generator algorithm (Part 1: Algorithm 3 and Part 2: Algorithm 4) has two main components: to-be-emitted source code (lines of the pseudocode that are underlined with dashes); and control commands that determine which code pieces should be emitted. The input of the code generator is (1) the physical plan and (2) GQ-Fast metadata, which specifies the encoding of each fragment. The Code Generator has two phases: (1) initialize necessary buffers (Lines 3–9); and (2) emit code pieces for each physical operator (Algorithm of Physical Operators' evaluation). More precisely, in the first phase, the GQ-Fast code generator initializes an array R to store final aggregation results and several boolean arrays for duplicate-checking in semijoin operations. In the second phase, it emits code for selection operators $\{[B : c]\}$ $\overset{\rightarrow r.A_1,\ldots,r.A_n}{\ltimes}_{B;R \mapsto r}$ $\mathcal{I}_{R.B'}$ (Lines

[3]This is an example that presents the functionalities of GQ-Fast and is not necessarily useful as an RQNA Query.

[4]In Chapter 16 we discuss more about the potential of Threading Operator

5–7 of Algorithm 4). The getDecodeFragment macro emits code for (1) retrieving a fragment (Lines 12–14) and (2) calling the corresponding decode macro (Lines 15–22); encoding information is obtained from metadata. For each join $L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\bowtie_{B;R\mapsto r}} \mathcal{I}_{R.B'}$ and semijoin $L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\rtimes_{B;R\mapsto r}} \mathcal{I}_{R.B'}$ operator (Lines 8–20 of Algorithm 4), the generator first checks whether the previous operator operates on an entity table and the operated columns are the same. In that case a for-loop can be avoided (Line 11). For a semijoin operator, one more duplicate-checking step should be added (Line 16). The remaining steps (Lines 17-20) of join/semijoin are identical to the selection operator. For each intersection operator $\overset{\rightarrow\alpha}{\cap}_{F_1,\ldots,F_m}$, the generator first identifies whether all the fragments are encoded with the same bitmap encoding (metadata). If so ($\alpha = 0$), the generator emits code to perform intersection directly on encoded fragments (Lines 36–40). For each Hash Join or Semi Join operator, the Code Generator emits code that accesses the corresponding Hash Map and retrieves the values that may be needed by following operators (Lines 22–23 for Hash Join and 26–29 for Hash Semi Join). In the case of the On-The-Fly-Select σ_c, an If clause is inserted in the code with the logical expression defined by the operator (Line 33); the logical expression may contain any attribute of entities discovered by operators up to this point. Otherwise, it emits code to perform intersection on decoded fragments (Lines 42–46). Then, the code generator either emits aggregation code pieces (Lines 48–50) for an aggregation operator $\gamma^{1\alpha}_{r.D;\alpha(s(A_1,\ldots,A_n))}$, or emits code for the creation of the appropriate Hash Map (Line 31) ;this step cocnludes one of the lists of physical plans that is evaluated autonomously).

Algorithm 3: Code Generator

Input: a list of physical operators $\mathbb{O}$ and metadata $\mathbb{M}$;
Output: executable C++ code;
// Initialize arrays in global
Initialize an array R ($|R| = |r.D|$) for $\gamma^1_{r.D;\alpha(s(A_1,\ldots,A_n))}$;
for *each semijoin operator* $L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\rtimes_{B;R\mapsto r}} \mathcal{I}_{R.B'}$ **do**
    a boolean array BA ($|BA| = |R.B'|$) with false values;
for *each hash semijoin operator* $L \overset{\rightarrow r.A_1,\ldots,r.A_n}{\rtimes_{B;R\mapsto r}}$ **do**
    Initialize a boolean array BA ($|BA| = |R.B'|$) with false values;
for *each Hash Map operator* $H_{r_i\mapsto L}$ **do**
    Initialize the appropriate Hash Map H_L for the results $(r.A_1, \ldots, r.A_n)$ of L;
Run Algorithm for Evaluation of Physical Operators;
Macro getDecodedFragment($\mathcal{P}_r, r.A, c$)
fragment $\mathcal{F}_{r.A}$= $\mathcal{P}_r[column(A)]$;
offset-array $next = \mathcal{I}_{R.B'}[c+1]$;
length $l_{r.A} = next[column(A)] - \mathcal{F}_{r.A}$;
if $\mathcal{F}_A$ *is UA encoded* **then**
    decodeUA($\mathcal{F}_{r.A}$, $l_{r.A}$: $\mathcal{A}_{r.A}$, $n_{r.A}$)
if $\mathcal{F}_A$ *is BCA encoded* **then**
    decodeBCA($\mathcal{F}_{r.A}$, $l_{r.A}$: $\mathcal{A}_{r.A}$, $n_{r.A}$)
else if $\mathcal{F}_A$ *is BB encoded* **then**
    decodeBB($\mathcal{F}_{r.A}$, $l_{r.A}$: $\mathcal{A}_{r.A}$, $n_{r.A}$)
else if $\mathcal{F}_A$ *is Huffman encoded* **then**
    decodeHuffman($\mathcal{F}_{r.A}$, $l_{r.A}$: $\mathcal{A}_{r.A}$, $n_{r.A}$)

Algorithm 4: Evaluation of Physical Operators

Input: a list of physical operators $\mathbb{O}$ and metadata $\mathbb{M}$;
Output: executable C++ code;
// Produce code
for *each physical operator* $o \in \mathbb{O}$ **do**
  if $o = \{[B : c]\} \overset{\rightarrow r.A_1,...,r.A_n}{\bowtie_{B;R \mapsto r}} \mathcal{I}_{R.B'}$ **then**
    offset-array $\mathcal{P}_r = \mathcal{I}_{R.B'}[c]$;
    for(each column $r.A_i$){
    getDecodedFragment($\mathcal{P}_r, r.A_i, c$);
  else if $o = L \overset{\rightarrow r.A_1,...,r.A_n}{\bowtie_{B;R \mapsto r}} \mathcal{I}_{R.B'} || o = L \overset{\rightarrow r.A_1,...,r.A_n}{\rtimes_{B}} \mathcal{I}_{R.B'}$ **then**
    //Let o' be the last Join or Semi operator of o
    if $o.B = o'.B$ *AND* $o'.R$ *is an entity table* **then**
      $v_B = \mathcal{A}_B[i_B]$;
    else
      for($i_B = 0; i_B < n_B; i_B + +$) {
        $v_B = \mathcal{A}_B[i_B]$;
    if $o = L \overset{\rightarrow r.A_1,...,r.A_n}{\rtimes_{B}} \mathcal{I}_{R.B'}$ **then**
      if($BA[v_B] = false$) {
    offset-array $\mathcal{P}_r = \mathcal{I}_{R.B'}[v_B]$;
    for(each column $r.A_i$){
    getDecodedFragment($\mathcal{P}_r, r.A_i, v_B$);
    }
  else if $o = L_1 \overset{\rightarrow r.A_1,...,r.A_n}{\bowtie_{B;R \mapsto r}} L_2$ **then**
    for each set of attribute values stored in H_{L_2} when accessing the Map with the corresponding key attribute of L_1) {
      Retrieve { $r.A_1, \ldots, r.A_n$ } values from $H_{L_2}[v_B]$;
  else if $o = L_1 \overset{\rightarrow r.A_1,...,r.A_n}{\rtimes_{B;R \mapsto r}} L_2$ **then**
    if($BA[v_B] = false$) {
    offset-array $\mathcal{P}_r = \mathcal{I}_{R.B'}[v_B]$;
    for each set of attribute values stored in H_{L_2} when accessing the Map with the corresponding key attribute of L_1) {
      Retrieve { $r.A_1, \ldots, r.A_n$ } values from $H_{L_2}[v_B]$;
    }
  else if $o = H_{r_i \mapsto L}$ **then**
    Store all values $r.A_1, \ldots, r.A_n$ in H_{L_2} ;
  else if σ_c **then**
    if(Logical Expression of c) {
  else if $o = \overset{\rightarrow \alpha}{\cap}_{L_1,...,L_m}$ **then**
    if $\alpha == 0$ **then**
      for(each $L_i = \{[B : c]\} \overset{\rightarrow A}{\bowtie_{B;R \mapsto r}} \mathcal{I}_{R.B'}$) {
      offset-array $\mathcal{P}_r = \mathcal{I}_{R.B'}[c]$;
      fragment $\mathcal{F}_{r.A}$= $\mathcal{P}_r[column(A)]$;
      }
      $I \leftarrow$Bitwise($\mathcal{F}_{r_1.A},...,\mathcal{F}_{r_m.A}$);
    else
      for(each $L_i = \{[B : c]\} \overset{\rightarrow r.A}{\bowtie_{B;R \mapsto r}} \mathcal{I}_{R.B'}$) {
      offset-array $\mathcal{P}_r = \mathcal{I}_{R.B'}[c]$;
      getDecodedFragment($\mathcal{P}_r, r.A, c$);
      }
      $I \leftarrow$Merge($\mathcal{A}_{r_1.A},...,\mathcal{A}_{r_m.A}$);
  else if $o = \gamma^1_{r.D;\alpha(s(A_1,...,A_n))}$ **then**
    for($i_{r.D} = 0; i_{r.D} < n_{r.D}; i_{r.D} + +$){
      $R[i_{r.D}] = \alpha(s(A_1, \ldots, A_n))$;
    }
Emit corresponding close braces;

Memory Requirements. Query execution requires $4 \cdot |r.D| + \sum_{i=1}^{k} \cdot |r_i.B'|$ bytes, where $|r.D|$ is the domain size of $r.D$ for an aggregation operator, k is the number of semijoin operators, and $|r_i.B'|$ is the domain size of $r_i.B'$ for the i^{th} semijoin operator.

Parallel Computing. GQ-Fast can use multiple cores/threads to perform parallel computation. The fact that in GQ-Fast (1) each fragment is independent of others, so GQ-Fast can assign fragments to different threads; and (2) the query processing is more CPU-bounded than memory-bounded especially when decompressing encoded fragments. We would like to mention one important technical detail of how to handle two kinds of global arrays, i.e., boolean arrays for each semijoin operation, and a numerical array for aggregation operator. To guarantee the correctness of GQ-Fast, it applies spinlock [6] in each array slot, which is more efficient than just using one spinlock on the entire boolean array.

Chapter 15

Experimental Results

All of the following experiments were performed on a computer with an Intel Core i7 - 7700HQ (256 KB L1 cache, 1MB L2 cache and 6MB L3 cache) processor, 32GB DDR4 RAM (2400 MHz) and a Samsung NVMe M.2 Hard Drive (970 Evo Plus), running Ubuntu 18.04.3 for the GQ-Fast experiments and Windows 10 for the PostgreSQL experiments. Generated C++ code was compiled with gcc++ 7.4.0, using -O3 optimization. For PostgreSQL, we ensured that all the indexes that can boost the database's efficiency were implemented and utilized. For all our experiments in 15.1 and 15.2, we used GQ-Fast's Uncrompressed Array as the encoding method.

15.1 Example Relationship Queries

In this section we present the execution time (for representative inputs of the PubMed data) of the relationship queries: SDY, ATY, DAY, A2X and ST.

Dataset Our dataset consists of all of PubMed's data from 1996 up to September of 2019. The table DS (Document Substances) is used as the Table DT for the execution of the relationship queries. More specifically, our dataset includes:

- Articles : 14319018
- Authors : 13589207
- Document-Term tuples : 56533389
- Document-Author tuples : 70198290

The execution times are presented in the Table: 15.1, and figure 15.2 contains the corresponding graph with logarithmic scale.

Table 15.1: Summary Table with Example Queries' execution times for GQ-Fast & PostgreSQL.

Query	GQ-Fast (ms)	PostgrSQL (ms)
SDY	723	2777
ATY	13	48
DAY	47	70
A2X	479	4270
ST	448	17675

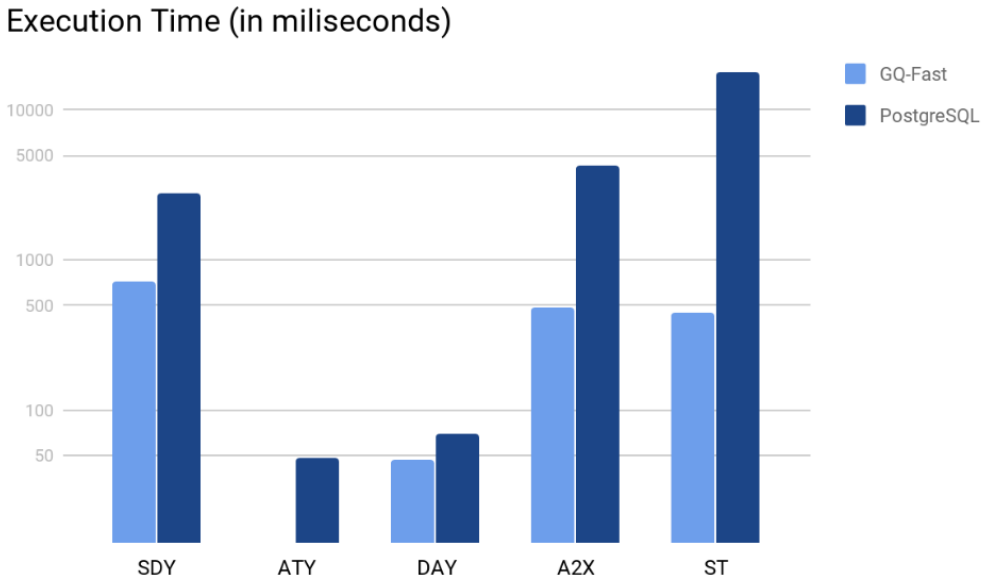


Figure 15.1: Comparison of execution times between GQ-Fast and PostgreSQL for the relationship queries: SDY, ATY, DAY, A2X and ST in logarithmic scale.

Table 15.2: Summary Table with Demo Queries' execution times for GQ-Fast & PostgreSQL.

Query	GQ-Fast (ms)	PostgrSQL (ms)
Author's Autocomplete	2288	24596
Disease's Autocomplete	432	4742
Frequent Co-Authors	40	216
Similar Authors	64412	1260000
Diseases	360	8350
Substances	448	17675
Heat Map	76	100

15.2 Demo User Interface

In this section we present the execution time of the queries that where utilized in order to compute the data tables and autocomplete lists of the example scenario in section 3.

Dataset Our dataset consists of all of PubMed's data up to September of 2019.

- Articles : 29137786
- Authors : 16345241
- Document-Substances tuples : 101010816
- Document-Diseases tuples : 53253398
- Document-Authors tuples : 114339706

The execution times are presented in the Table: 15.2, and figure 15.2 contains the corresponding graph with logarithmic scale.

15.3 Space and Runtime Efficiency of Different Encoding Methods

In this section we present the execution time for some of PubMed's queries, along with the Index sizes for different encoding methods that were tested in our system. More specifically, Table 15.3 shows the execution time for the encoding methods : Uncompressed Array (UA), Huffman-encoded Array (HU), Bit-aligned Compressed Array (BCA) and Byte-aligned Compressed Bitmap (BB). Table 15.4 shows the size of each Index from PubMed's data for the same encoding methods. The dataset for this section is the same with Section 15.2, and we chose the tables Substances and DS to take the place of Terms and DT.

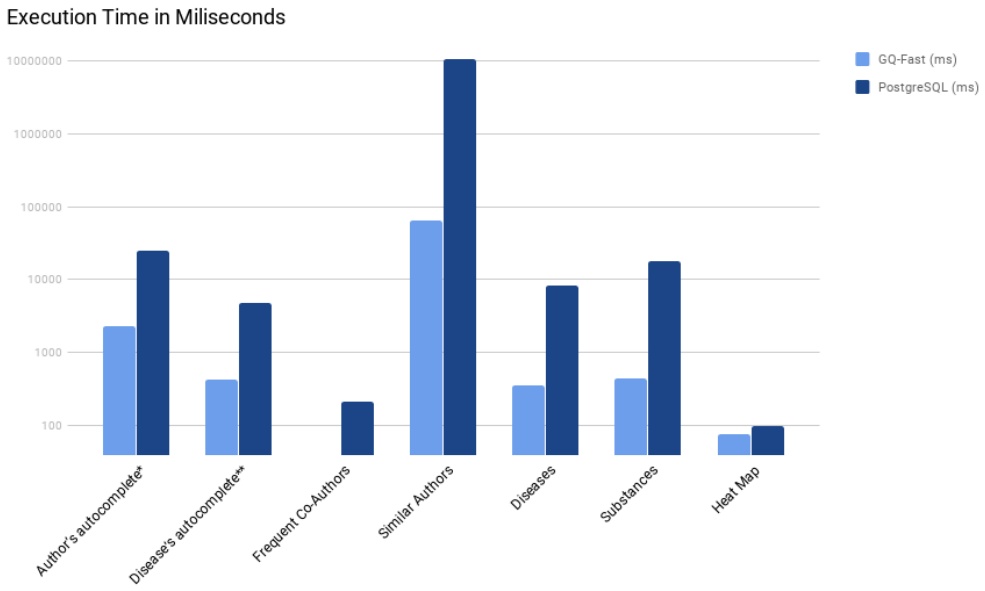


Figure 15.2: Comparison of execution times between GQ-Fast and PostgreSQL for the queries behind the Demo User Interface in logarithmic scale.

Table 15.3: Execution Time for GQ-Fast Queries for different encoding methods (Uncompressed Array, Huffman-encoded Array, Bit-aligned Compressed Array and Byte-aligned Compressed Bitmap).

Query	**UA (ms)**	**HU (ms)**	**BCA (ms)**	**BB (ms)**
SD	89	174	79	83
SDY	227	444	217	221
ATY	9	496	9	10
DAY	112	348	109	108
A2X	127	2596	97	122
ST	4	30	3	4

Table 15.4: Index Size for PubMed's Data Set for different Encoding Methods (Uncompressed Array, Huffman-encoded Array, Bit-aligned Compressed Array and Byte-aligned Compressed Bitmap).

Query	**UA**	**HU**	**BCA**	**BB**	**Fragments**
DT.Term:	1.5 GB	1.2 GB	680 MB	1.5 GB	247421
DT.Doc:	1.2 GB	1.4 GB	1.3 GB	1.8 GB	18060121
Doc.ID:	1.0 GB	1.0 GB	1.2 GB	1.2 GB	29137786
Doc.Year:	1.2 GB	615 MB	408 MB	660 MB	219
DA.Doc:	1.8 GB	1.5 BG	1.6 GB	1.8 GB	28510300
DA.Author:	2.0 BG	1.5 GB	1.3 GB	1.6 GB	16345237
Total Space:	8.7 GB	7.2 GB	6.44 GB	8.56 GB	-

We can see in Table 15.3 that the compression methods of Bit-aligned Compressed Array (BCA) and Byte-aligned Compressed Bitmap (BB), provide slightly better runtime efficienct than the Uncompressed Array (UA)[1]. At the same time, the Bit-aligned Compressed Array (BCA) saves approximately 26% of space according to Table 15.4. On the other hand, Huffman-encoded Array (HU) results in much greater execution times for GQ-Fast's queries. We believe that this happens because the compression methods fails to provide small enough framgents to the processor in order to fit them in the L1 or L2 cache.

[1]Larger datasets than the one we used would result in even greater differences between the encoding methods

Chapter 16

Future Work

In the future we will try to investigate methods to incorporate the creation and positioning of the Threading Operator in the Physical Planner of GQ-Fast. More specifically, we aim to dynamically calculate the optimal number of occurrences for the Threading operator and where is it best to place it inside the list of physical plans (of each autonomously evaluated RQNA sub-query).

More specifically, a Threading Operator notifies the Code Generator to execute the operators that succeed the Treading Operator with multiple threads. If more Threading Operators appear later in the same list of physical operators, then each thread will have to generate and control multiple threads for the remaining physical operators. The process of deciding the right places for Threading Operators inside the physical plans is highly complex, but can also prove excessively beneficial for the efficiency of GQ-Fast .

An RQNA Query may consist of various autonomous branches that are distinguished as the right child of a Hash Join or Hash Semi Join parent node. In this case, the branch can be considered as an autonomous RQNA sub-query, and will be executed separately. GQ-Fast should analyze the sub-query and apply the optimal number (and in the right place) of Threading Operators dynamically, by taking into account other autonomous RQNA sub-queries that may exist in the original query and that may be able to run in parallel. If, for example, GQ-Fast runs on an processor with 8 cores, and an RQNA query contains 3 Hash Join branches that can be autonomously executed in parallel, then there are 42 different ways to distribute the cores to the queries. Our aim is for an efficient way for GQ-Fast to analyze and distribute processing cores to each autonomous part of an RQNA query in order to achieve even better performance.

Also, we will design more complex and scientifically interesting queries in order to better process data sets like PubMed's graph and, subsequently, better analyze, retrieve and depict useful information for scientists and academics.

Appendix A

Κλήσεις του **Demo UI** στο **Rest API**

Στην εφαρμογή που παρουσιάσαμε στην ενότητα 3, η εφαρμογή φυλλομετρητή (front end) αξιοποιεί ένα Resful API το οποίο αναπτύξαμε. Σε αυτό το παράρτημα παρουσιάζουμε τα βασικά endpoints τα οποία χρησιμοποιήσαμε για τη παραγωγή των αποτελεσμάτων που φαίνονται στις εικόνες από τα δοκιμαστικά σενάρια χρήσης της εφαρμογής.

Α΄.1 **Autocomplete feature**

Υλοποιήσαμε ένα endpoint: baseURL/autocomplete στο οποίο το front end μπορεί να αποστέλλει ένα διάνυσμα (vector) από τιμές/παραμέτρους των pubmed entities: author, substance, disease, year, ένα prefix string από ένα ζητούμενο dimension (entity της GQ-Fast) καθώς και την επιλογή του συγκεκριμένου dimension από το σύνολο των προηγούμενων τιμών (π.χ. author). Το API εκτελεί το κατάλληλο Case Analysis και ζητάει από τη GQ-Fast να εκτελέσει:

- Inverse Index Search πάνω στο ανάστροφο ευρετήριο του ζητούμενου dimension βάσει του prefix string των παραμέτρων. Αν για παράδειγμα το ζητούμενο dimension είναι το author, και το prefix string που δίνεται περιέχει "Trichop", τότε θα γίνει αναζήτηση για όλα τα ονόματα συγγραφέων τα οποία ξεκινάνει με αυτό το string.
- (Προαιρετικό) Σε περίπτωση που υπάρχουν μη κενές παράμετροι στο vector, εκτελείται κατάλληλο query στη GQ-Fast το οποίο επιστρέφει για κάθε αποτέλεσμα της αναζήτησης έναν αριθμό που χρησιμοποιείται ως δείκτης σημαντικότητας (contextual importance) για την ταξινόμηση των αποτελεσμάτων στον front end. Σε περίπτωση που δεν εκτελεστεί κάποιο query, κάθε τιμή που επιστρέφεται από το ευρετήριο συνοδεύεται από έναν δείκτη non-contextual importance που μπορεί να χρησιμοποιηθεί για την ταξινόμηση των αποτελεσμάτων. Για παράδειγμα, εάν κατά την αναζήτηση για τα ονόματα συγγραφέων δίνεται και μία ασθένεια στις παραμέτρους, τότε ο δείκτης σημαντικότητας στα αποτελέσματα

Table A.1: Παράμετροι του baseURL/autocomplete endpoint.

Parameter	**Type**
Search Dimension	String
Prefix	String
Author (id)	Int
Substance (id)	Int
Disease (id)	Int
Year	Int

Table A.2: Παράμετροι του {baseURL}table endpoint.

Parameter	**Type**
Table Dimension	String
Author (id)	Int
Substance (id)	Int
Disease (id)	Int
Year	Int

συμβολίζει τη σχέση του κάθε συγγραφέα με τη συγκεκριμένη ασθένεια (π.χ. πόσες φορές έχει συμπεριλάβει την ασθένεια ένας συγγραφέας στις δημοσιεύσεις του).

Η αίτηση θα πρέπει να περιέχει στο body του μηνύματος προς το API τις τιμές για τα πεδία του πίνακα: A.1.

A′.2 Table Request feature

Υλοποιήσαμε ένα endpoint: baseURL/table στο οποίο το front end μπορεί να αποστέλλει ένα διάνυσμα (vector) από τιμές/παραμέτρους (π.χ. $author_{id} = 100$, $disease_{id} = 200$) μαζί με ένα ζητούμενο dimension (π.χ. authors ή substances) για το οποίο χρειάζεται τον κατάλληλο πίνακα δεδομένων. Το ποιος θα είναι αυτός ο πίνακας προκύπτει από το αντίστοιχο case analysis που γίνεται στον back end (Rest API). Ο πίνακας που επιστρέφεται ως αποτέλεσμα είναι έτοιμος για παρουσίαση στον χρήστη καθώς το API έχει φροντίσει να αντικαταστήσει τις τιμές κλειδιών (IDs) που επιστρέφει η GQ-Fast με τα αντίστοιχα ονόματα των PubMed Entities.

Η αίτηση θα πρέπει να περιέχει στο body του μηνύματος προς το API τις τιμές για τα πεδία του πίνακα: A.2.

Table A.3: Παράετροι του {baseURL}heatmap endpoint.

Parameter	**Type**
Heat Map Dimension 1	String
Heat Map Dimension 2	String
Author (id)	Int
Substance (id)	Int
Disease (id)	Int
Year	Int

A′.3 HeatMap Request feature

Υλοποιήσαμε ένα endpoint: baseURL/table στο οποίο το front end μπορεί να αποστέλλει ένα διάνυσμα (vector) από τιμές/παραμέτρους (π.χ. $author_{id} = 100$, $disease_{id} = 200$) μαζί με δύο ζητούμενα dimensions (π.χ. authors και substances) για το οποίο χρειάζεται τον κατάλληλο πίνακα τύπου Heat Map. Το ποιος θα είναι αυτός ο πίνακας προκύπτει από το αντίστοιχο case analysis που γίνεται στον back end (API).

Η αίτηση θα πρέπει να περιέχει στο body του μηνύματος προς το API τις τιμές για τα πεδία του πίνακα: A.3.

A′.4 Case Analysis

Στα API endpoints που περιγράψαμε, κάθε κλήση που γίνεται στο API από τον φυλλομετρητή περιέχει ένα σύνολο από παραμέτρους. Ανάλογα με το ποιες είναι αυτοί οι παράμετροι και το περιεχόμενό τους, προκύπτει και ένα διαφορετικό αποτέλεσμα το οποίο πρέπει να επιστρέψει ένα endpoint. Αυτό σημαίνει ότι το Rest API που αναπτύξαμε, προτού ξεκινήσει να επικοινωνεί με τη GQ-Fast και τη PostgreSQL προκειμένου να συλλέξει και να επιστρέψει τις απαντήσεις που ζητάει η χρήστης, πρέπει πρώτα να αποφασίσει ποιες είναι αυτές οι απαντήσεις. Ανάλογα με το αποτέλεσμα αυτής της μελέτης (δηλαδή του Case Analysis), προκύπτει ένα ξεχωριστό σύνολο από ερωτήματα συσχέτισης που πρέπει να τρέξουν στη GQ-Fast . Για παράδειγμα, εάν κατά τη κλήση του endpoint: baseURL/table δοθεί ως ζητούμενη διάσταση: "Authors" και η παράμετρος "Substances" δεν είναι κενή αλλά περιέχει τη τιμή "Alcohols", τότε θα πρέπει να εκτελεστεί στη GQ-Fast ένα ερώτημα όπου επιστρέφει τους συγγραφείς που έχουν χρησιμοποιήσει το χημικό στοιχείο "Alcohols" τις περισσότερες φορές στις δημοσιεύσεις τους. Εάν γίνει ξανά η ίδια κλήση αλλά αυτή τη φορά έχει συμπληρωθεί και το πεδίο Year, τότε το ερώτημα που πρέπει να εκτελεστεί θα πρέπει να ασχοληθεί μόνο με τις δημοσιεύσεις που έγιναν στη δοθείσα χρονιά προκειμένου να βρει τους σχετικούς συγγραφείς.

Bibliography

[1] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[2] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column oriented database systems. *PVLDB*, 2(2):1664–1665, 2009.

[3] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, pages 967–980, 2008.

[4] G. Antoshenkov. Byte-aligned bitmap compression. In *DCC*, page 476, 1995.

[5] S. Bordoloi and B. Kalita. Designing graph database models from existing relational databases. *IJCA*, 74(1), 2013.

[6] J. Catozzi and S. Rabinovici. Operating system extensions for the teradata parallel VLDB. In *VLDB*, pages 679–682, 2001.

[7] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[8] C. Chen, X. Yan, F. Zhu, J. Han, and S. Y. Philip. Graph OLAP: a multi-dimensional framework for graph data analysis. *KAIS*, 21(1):41–63, 2009.

[9] C. Chen, X. Yan, F. Zhu, J. Han, and P. S. Yu. Graph OLAP: Towards online analytical processing on graphs. In *ICDM*, pages 103–112, 2008.

[10] P. Chen. Entity-relationship modeling: historical events, future trends, and lessons learned. In *Software pioneers*, pages 296–310. Springer, 2002.

[11] K. Chung and J. Wu. Level-compressed huffman decoding. *TCOM*, 47(10):1455–1457, 1999.

[12] E. J. Franczek, J. T. Bretscher, and R. W. Bennett III. Computer virus screening methods and systems, Nov. 16 1999. US Patent 5,987,610.

[13] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[14] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, pages 367–378. IEEE, 2003.

[15] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *DEBU*, 35(1):40–45, 2012.

[16] H. Kilicoglu, M. Fiszman, A. Rodriguez, D. Shin, A. Ripple, and T. C. Rindflesch. Semantic MEDLINE: a web application for managing the results of PubMed searches. In *SMBM*, volume 2008, pages 69–76, 2008.

[17] H. Kilicoglu, D. Shin, M. Fiszman, G. Rosemblat, and T. C. Rindflesch. SemMedDB: a PubMed-scale repository of biomedical semantic predications. *Bioinformatics*, 28(23):3158–3160, 2012.

[18] J. Knight. Method and system for remote network security management, Apr. 28 2004. US Patent App. 10/834,443.

[19] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. *VLDB*, 5(12):1790–1801, 2012.

[20] J. Li, H. Tseng, C. Lin, Y. Papakonstantinou, and S. Swanson. Hippogriffdb: Balancing I/O and GPU bandwidth in big data analytics. *PVLDB*, 9(14):1647–1658, 2016.

[21] C. Lin, B. Mandel, Y. Papakonstantinou, and M. Springer. Fast in-memory sql analytics on typed graphs. *Proc. VLDB Endow.*, 10(3):265–276, Nov. 2016.

[22] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang. G-SQL: Fast query processing via graph exploration. *PVLDB*, 9(12), 2016.

[23] G. Navarro and N. Brisaboa. New bounds on D-ary optimal codes. *Information Processing Letters*, 96(5):178–184, 2005.

[24] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178. ACM, 2009.

[25] M. Roth, A. Ben-David, D. Deutscher, G. Flysher, I. Horn, A. Leichtberg, N. Leiser, Y. Matias, and R. Merom. Suggesting friends using the implicit social graph. In *ACM SIGKDD*, pages 233–242, 2010.

[26] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B.-j. P. Hsu, and K. Wang. An overview of Microsoft Academic Service (MAS) and applications. In *WWW*, pages 243–246. ACM, 2015.

[27] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[28] P. Vassiliadis and T. Sellis. A survey of logical models for OLAP databases. *ACM Sigmod Record*, 28(4):64–69, 1999.

[29] D. W. Wardani and J. Kiing. Semantic mapping relational to graph model. In *IC3INA*, pages 160–165. IEEE, 2014.

[30] F. Xia, Y. Li, C. Yu, H. Ma, and W. Qian. Bsma: A benchmark for analytical queries over social media data. *VLDB*, 7(13):1573–1576, 2014.

[31] Z. Xu, S. Zhang, and Y. Dong. Mapping between relational database schema and OWL ontology for deep annotation. In *WI*, pages 548–552. IEEE, 2006.

[32] S. Zhou. Exposing relational database as RDF. In *IIS*, volume 2, pages 237–240. IEEE, 2010.