ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΗΛΕΚΤΡΟΝΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΑΝΑΝΕΩΣΗ ΤΟΥ KNOWLEDGE GRAPH ΤΗΣ ΕΛΛΗΝΙΚΗΣ DBPEDIA
ΑΝΑΠΤΥΞΗ ΣΗΜΑΣΙΟΛΟΓΙΚΗΣ ΕΦΑΡΜΟΓΗΣ ΙΣΤΟΥ

του

ΠΑΠΑΖΟΓΛΟΥ ΚΩΝΣΤΑΝΤΙΝΟΥ
Αρ. Μητρώου:134016

Επιβλέπων Καθηγητής: Μπράτσας Χαράλαμπος

Υποβλήθηκε ως απαιτούμενο για την απόκτηση του πτυχιακού διπλώματος στο
τμημα Μηχανικων Πληροφορικης και
Ηλεκτρονικων συστηματων

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως πτυχιακή εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Η παρούσα εργασία αποτελεί πνευματική ιδιοκτησία του φοιτητή Παπλαζογλου Κωνσταντίνου που την εκπόνησε. Στο πλαίσιο της πολιτικής ανοικτής πρόσβασης, ο συγγραφέας/δημιουργός εκχωρεί στο Διεθνές Πανεπιστήμιο της Ελλάδος άδεια χρήσης του δικαιώματος αναπαραγωγής, δανεισμού, παρουσίασης στο κοινό και ψηφιακής διάχυσης της εργασίας διεθνώς, σε ηλεκτρονική μορφή και σε οποιοδήποτε μέσο, για διδακτικούς και ερευνητικούς σκοπούς, άνευ ανταλλάγματος. Η ανοικτή πρόσβαση στο πλήρες κείμενο της εργασίας, δεν σημαίνει καθ' οιονδήποτε τρόπο παραχώρηση δικαιωμάτων διανοητικής ιδιοκτησίας του συγγραφέα/δημιουργού, ούτε επιτρέπει την αναπαραγωγή, αναδημοσίευση, αντιγραφή, πώληση, εμπορική χρήση, διανομή, έκδοση, μεταφόρτωση (downloading), ανάρτηση (uploading), μετάφραση, τροποποίηση με οποιονδήποτε τρόπο, τμηματικά ή περιληπτικά της εργασίας, χωρίς τη ρητή προηγούμενη έγγραφη συναίνεση του συγγραφέα/δημιουργού.

Η έγκριση της πτυχιακής εργασίας από το Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του Διεθνούς Πανεπιστημίου της Ελλάδος, δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων του συγγραφέα, εκ μέρους του Τμήματος.

Περίληψη

Η πτυχιακή παρουσιάζει την ανάκτηση, λειτουργία και οπτικοποίηση των σημασιολογικών δεδομένων στο Ελληνικό Γραφικό Γνώσης DBpedia. Το κύρια κίνητρο ήταν η ανάγκη γεφύρωσης του χάσματος μεταξύ της αξίας των συνδεδεμένων δεδομένων και της δυσκολίας πρόσβασης σε αυτά από τους χρήστες. Αυτό οφείλεται κυρίως στις άκαμπτες, μη διαδραστικές διεπαφές SPARQL και την ανεπάρκεια της ελληνικής οντολογίας. Έχουμε σχεδιάσει και υλοποιήσει μια σύγχρονη Εφαρμογή Σημασιολογικού Ιστού με αρχιτεκτονική τριών επιπέδων για να λύσουμε αυτές τις δυσκολίες.

Στο επίπεδο των δεδομένων, χρησιμοποιήθηκε ένας Καθολικός Διακομιστής Virtuoso για τη διαχείριση και φιλοξενία των εκατομμυρίων αρχείων απο την DBpedia Databus. Το ενδιάμεσο επίπεδο κατασκευάστηκε χρησιμοποιώντας τη γλώσσα προγραμματισμού Golang με χρήση και του εγγενούς μοντέλο ταυτόχρονης εκτέλεσης. Η εκτέλεση σύνθετων ερωτημάτων παράλληλα μείωσε σημαντικά τον χρόνο απόκρισης και βελτιώνοντας έτσι την απόδοση του συστήματος.

Η διεπαφή χρήστη κατασκευάστηκε με βάση τη βιβλιοθήκη React.js· παρέχει δυναμικούς πίνακες αποτελεσμάτων με αυτόματη ενσωμάτωση μικρογραφιών και διαδραστικούς γεωγραφικούς χάρτες. Αυτή η νέα εφαρμογή περιλαμβάνει επίσης έναν ενσωματωμένο επεξεργαστή ερωτημάτων, ο οποίος μπορεί άμεσα να δώσει ανατροφοδότηση για συντακτικά λάθη, και Σημασιολογικό Φιλτράρισμα για αναζήτηση σε κειμενικές περιγραφές.

Όλο αυτό το σύστημα είναι ενσωματωμένο χρησιμοποιώντας το Docker και επιτρέπει τη φορητότητα και την ευκολία εγκατάστασης και ανέβασμα του κωδικα. Αυτό μας οδηγεί στο τελικό προϊόν που είναι ένα ισχυρό, γρήγορο και φιλικό προς τον χρήστη εργαλείο για πρόσβαση στο Ελληνικό Γραφικό Γνώσης DBpedia. Αυτό δείχνει τη συνένωση μεταξύ των σύγχρονων εργαλείων ανάπτυξης λογισμικού και των προτύπων του Σημασιολογικού Ιστού.

Λέξεις-Κλειδιά: DBpedia, Σημασιολογικός Ιστός, Γράφημα Γνώσης, Διασυνδεδεμένα Δεδομένα, SPARQL, Golang, React.js, Virtuoso.

Abstract

The thesis presents the retrieval, operation and visualization of the semantic data in the Greek Bibliography of Knowledge DBpedia. The main motivation was the need to bridge the gap between the value of linked data and the difficulty for users to access it. This is mainly due to the rigid, non-interactive SPARQL interfaces and the inadequacy of the Greek ontology. We've designed and implemented a modern three-tier architecture Semantic Web App to address these challenges.

At the data level, a Catholic Virtuoso Server was used to manage and host the millions of files from DBpedia Databus. The intermediate level was constructed using the Golang programming language using the native concurrent execution model. Running complex queries in parallel significantly reduced the response time thus improving the performance of the system.

The user interface is built on the React.js library; it provides dynamic result tables with automatically embedded thumbnails and interactive geographical maps. This new application also includes a built-in query editor, which can instantly provide feedback for syntactic errors, and Semantic Filtering for searching in textual descriptions.

This whole system is built using Docker and allows for portability and ease of installing and uploading code. This leads us to the final product which is a powerful, fast and user-friendly tool for accessing the DBpedia. This shows the convergence between modern software development tools and the standards of the Semantic Web.

Keywords: DBpedia, Semantic Web, Knowledge Graph, Linked Data, SPARQL, Golang, React.js, Virtuoso.

2026

ΕΥΧΑΡΙΣΤΙΕΣ

Αρχικά, θα ήθελα να ευχαριστήσω ιδιαιτέρως τον επιβλέπων καθηγητή μου
για την βοήθεια, τις συμβουλές και την συνεχή ανταπόκριση του
σε όλο το διάστημα της εκπόνηση της διπλωματικής μου εργασίας.
Επιπλέον, θα ήθελα να πω ένα ευχαριστώ σε όλους τους καθηγητές του
Τμήματος για το σύνολο των γνώσεων που μου προσέφεραν κατά την διάρκεια των σπουδών μου,
καθώς και για την υποστήριξή τους όποτε μου χρειάστηκε.
Τέλος, θα ήθελα να ευχαριστήσω τα αδέρφια μου αλλά και τους γονείς μου για την διαρκή
ψυχολογική στήριξή τους δίνοντας μου δύναμη όποτε τη χρειαζόμουν.

ΠΕΡΙΕΧΟΜΕΝΑ

ΚΕΦΑΛΑΙΟ 1: ΕΙΣΑΓΩΓΗ

1.1 ΕΙΣΑΓΩΓΗ

Η ταχεία ανάπτυξη του Παγκόσμιου Ιστού (World Wide Web) τα τελευταία χρόνια έχει οδηγήσει στη μετάβαση από τον παραδοσιακό "Ιστό των Εγγράφων" (Web of Documents) ή την πληροφορία που είναι για ανθρώπινη ανάγνωση, στον “Ιστό των Δεδομένων” (Web of data) ή Σημασιολογικό Ιστό “Scemantic Web Στο νέο αυτό πεδίο, τα δεδομένα είναι δικτυωμένα και δομημένα ώστε να επιτρέπουν στις μηχανές να κατανοούν, να επεξεργάζονται και να αξιοποιούν νέες πληροφορίες. Κεντρικό ρόλο σε αυτό παίζει το έργο των Συνδεδεμένων Ανοιχτών Δεδομένων (LOD). Το κεντρικό στοιχείο του Συνδεδεμένου Νέφους Δεδομένων είναι το DBpedia - ένα ερευνητικό έργο για τη συλλογή δομημένων πληροφοριών από τη Wikipedia και τη δημοσίευσή τους ως ένα τεράστιο Γράφημα Γνώσης, το οποίο επιτρέπει την ανάκτηση δομημένων πληροφοριών εντός της Wikipedia.

Το DBpedia δεν κρατά τη γνώση κολλημένη σε ελεύθερο κείμενο αλλά την αντιμετωπίζει ως τριπλέτες (υποκείμενο - κατηγόρημα - αντικείμενο), βασισμένο στο πρότυπο RDF (Πλαίσιο Περιγραφής Πόρων) για την εκτέλεση περίπλοκων ερωτημάτων μέσω κωδικα SPARQL. Συγκεκριμένα, το Ελληνικό DBpedia (Ελληνικός Κόμβος) είναι ένα σχήμα για τη μετατροπή της Ελληνικής Wikipedia [13] σε σημασιολογικά δεδομένα . Παρά την τεράστια αξία αυτών των δεδομένων για την εκπαίδευση, την έρευνα και την ανάπτυξη εφαρμογών (π.χ., συστήματα ερωταποκρίσεων, έξυπνες αναζητήσεις), η πρόσβαση σε αυτά παραμένει δύσκολη για τον μέσο χρήστη. Τα παραδοσιακά SPARQL Endpoints (τα σημεία όπου γίνονται οι ερωτήσεις) είναι πολύ αυστηρές διεπαφές προγραμματισμού (APIs) που επιστρέφουν απλό κείμενο ή αρχεία JSON, χωρίς οπτική αναπαράσταση. Επίσης, η ποιότητα και η πληρότητα των δεδομένων είναι κοινά ζητήματα για την ελληνική έκδοση του DBpedia, επίσης γνωστά ως ελλείπουσες ιδιότητες, καθώς βασίζεται σε εθελοντές συντάκτες για να συμπληρώσουν τα infoboxes της Wikipedia.

Αυτή η εργασία επιδιώκει να ξεπεράσει αυτή την ανεπάρκεια εισάγοντας και αναπτύσσοντας ένα ολοκληρωμένο, σύγχρονο σύστημα (Εφαρμογή Ιστού) που αναζωογονεί τον τρόπο με τον οποίο αλληλεπιδρούμε με το Γράφημα Γνώσης του Ελληνικού DBpedia και βοηθά σε μια σημασιολογική μορφή δεδομένων που είναι εύκολα προσβάσιμη, οπτικά ελκυστική και χρηστική.

1.2 ΣΚΟΠΟΣ

Ο κύριος σκοπός της παρούσας εργασίας είναι η σχεδίαση, ανάπτυξη και αξιολόγηση μιας σύγχρονης Σημασιολογικής Εφαρμογής Ιστού (Semantic Web Application), η οποία θα παρέχει στο χρήστη ένα φιλικό, χρήσιμο, διαδραστικό και υψηλών επιδόσεων περιβάλλον για την εξερεύνηση της πληροφορίας της Ελληνικής DBpedia.

Για την επίτευξη του κεντρικού αυτού σκοπού, τέθηκαν και υλοποιήθηκαν οι ακόλουθοι επιμέρους στόχοι:

1. Συλλογή και Αποθήκευση Δεδομένων: Η αυτόματη λήψη των πιο πρόσφατων συνόλων δεδομένων (datasets) της Ελληνικής DBpedia και η εισαγωγή τους σε ένα τοπικό σύστημα διαχείρισης σημασιολογικών βάσεων database (Virtuoso Universal Server).Ενώ χρησιμοποιεί

τεχνολογίες απομόνωσης αναπτυξης (Docker) για την εύκολη χρήση , εγκατάσταση και αναπαραγωγή όλου του συστήματος.

2. Ανάπτυξη Υψηλών Επιδόσεων (High-Performance Backend): Η δημιουργία ενός ενδιάμεσου επιπέδου (Middleware / API) με τη γλώσσα προγραμματισμού Golang. Στόχος ήταν η ασφαλής επικοινωνία με τη βάση (επίλυση ζητημάτων CORS , Cross-Origin Resource Sharing) και η δραματική βελτίωση της ταχύτητας ανάκτησης δεδομένων, αξιοποιώντας και τεχνολογίες όπως τον Ταυτοχρονισμό (Concurrency) της Go για την παράλληλη εκτέλεση σύνθετων ερωτημάτων SPARQL (π.χ. εμφάνιση στατιστικών σε πραγματικό χρόνο).
3. Οπτικοποίηση και Εμπειρία Χρήστη (Frontend & UX): Η ανάπτυξη μιας δυναμικής διεπαφής χρήστη με τη βιβλιοθήκη React.js. Στόχος η μετατροπή των "ψυχρών" δεδομένων RDF σε οπτική πληροφορία, υλοποιώντας δυναμικούς πίνακες αποτελεσμάτων που ενσωματώνουν αυτόματα μικρογραφίες εικόνων, συνδέσμους πλοήγησης και σημεία (Pins) στο χαρτη Google Maps βάσει των γεωγραφικών συντεταγμένων (geo:lat, geo:long).
4. Ανάπτυξη Εργαλείου (SPARQL Editor): Η υλοποίηση ενός ενσωματωμένου επεξεργαστή ερωτημάτων (Query Editor) με χαρακτηριστικά αντίστοιχα ενός Compiler. Το σύστημα αναλύει και προβάλλει τα ακριβή συντακτικά σφάλματα που επιστρέφει ο Virtuoso Server απευθείας στην οθόνη του χρήστη, διευκολύνοντας την εκμάθηση της γλώσσας SPARQL.
5. Επίλυση Ζητημάτων Ελλιπών Δεδομένων (Semantic Filtering): Η δημιουργία έτοιμων, βελτιστοποιημένων ερωτημάτων (Predefined Queries) που χρησιμοποιούν προηγμένες τεχνικές αναζήτησης κειμένου. Μέσω "έξυπνων" φίλτρων, το σύστημα αναζητά πληροφορίες μέσα στις περιγραφές (abstracts), παρακάμπτοντας ελλείψεις στις οντολογικές συνδέσεις της βάσης, εξασφαλίζοντας έτσι ότι ο χρήστης θα λάβει ουσιαστικά αποτελέσματα.

1.3 ΔΟΜΗ ΤΗΣ ΕΡΓΑΣΙΑΣ

Το παρόν τεύχος δομείται σε επτά (7) διακριτά κεφάλαια.
Περιγράφουν αναλυτικά τη θεωρία, την αρχιτεκτονική και την υλοποίηση του συστήματος. Συγκεκριμένα:

- **Στο Κεφάλαιο 1 (Εισαγωγή)**, παρουσιάζεται το αντικείμενο της εργασίας, το πρόβλημα που καλείται να λύσει, καθώς και οι βασικοί στόχοι της υλοποίησης.
- **Στο Κεφάλαιο 2 (Θεωρητικό Υπόβαθρο)**, παρατίθενται οι απαραίτητες θεωρητικές έννοιες. Γίνεται εισαγωγή στον Σημασιολογικό Ιστό, στα Διασυνδεδεμένα Δεδομένα (Linked Data), στην αρχιτεκτονική της Wikipedia και της DBpedia, καθώς και στο πρότυπο RDF και τη γλώσσα ερωτημάτων SPARQL.
- **Στο Κεφάλαιο 3 (Αρχιτεκτονική και Τεχνολογίες Συστήματος)**, αναλύεται η αρχιτεκτονική των τριών επιπέδων (3-Tier Architecture) που ακολουθήθηκε, καθώς και οι τεχνολογίες που επιλέχθηκαν για κάθε επίπεδο (Virtuoso, Golang, React, Docker), εξηγώντας τους λόγους επιλογής τους.
- **Στο Κεφάλαιο 4 (Εξαγωγή και Αποθήκευση Δεδομένων)**, περιγράφεται η μεθοδολογία άντλησης των δεδομένων από το Databus της DBpedia μέσω αυτοματοποιημένων σεναρίων (scripts), καθώς και η μαζική φόρτωσή τους (bulk loading) στον Virtuoso.
- **Στο Κεφάλαιο 5 (Υλοποίηση Συστήματος)**, πραγματοποιείται εις βάθος τεχνική ανάλυση του κώδικα. Παρουσιάζονται η λειτουργία του API, η βελτιστοποίηση με τις Go Routines για ταυτόχρονη εκτέλεση ερωτημάτων, οι αλγόριθμοι χειρισμού σφαλμάτων και η στρατηγική σημασιολογικού φιλτραρίσματος (semantic text filtering) για τη διόρθωση των ελλιπών δεδομένων.

- **Στο Κεφάλαιο 6 (Αποτελέσματα και Αξιολόγηση)**, παρουσιάζεται η τελική μορφή της εφαρμογής μέσα από στιγμιότυπα οθόνης (screenshots) και παραδείγματα χρήσης. Επιπλέον, γίνεται αξιολόγηση της απόδοσης (performance evaluation) της ταυτόχρονης εκτέλεσης σε σχέση με τη σειριακή.
- **Στο Κεφάλαιο 7 (Συμπεράσματα και Μελλοντικές Επεκτάσεις)**, συνοψίζονται τα συμπεράσματα που προέκυψαν από την εκπόνηση της πτυχιακής εργασίας και προτείνονται ιδέες για τη μελλοντική εξέλιξη και αναβάθμιση του συστήματος.

ΚΕΦΑΛΑΙΟ 2: ΘΕΩΡΗΤΙΚΟ ΥΠΟΒΑΘΡΟ

2.1 ΕΙΣΑΓΩΓΗ

Το παρόν κεφάλαιο εισάγει τις βασικές έννοιες και τεχνολογίες πάνω στις οποίες δομείται η παρούσα πτυχιακή εργασία. Γίνεται μια ιστορική και τεχνολογική αναδρομή από τον παραδοσιακό Ιστό στον Σημασιολογικό Ιστό, αναλύεται η δομή της Wikipedia και η αρχιτεκτονική του εγχειρήματος της DBpedia, και τέλος, παρουσιάζονται τα πρότυπα αναπαράστασης (RDF) και ερωτημάτων (SPARQL) που καθιστούν δυνατή την αξιοποίηση των Διασυνδεδεμένων Δεδομένων.

2.2 ΑΠΟ ΤΟ WEB OF DOCUMENTS ΣΤΟ WEB OF DATA (ΣΗΜΑΣΙΟΛΟΓΙΚΌΣ ΙΣΤΌΣ)

Η αρχιτεκτονική του Παγκόσμιου Ιστού (World Wide Web), κατά την αρχική της σύλληψη, σχεδιάστηκε με γνώμονα τη διανομή και διασύνδεση εγγράφων πολυμέσων για ανθρώπινη κατανάλωση. Το μοντέλο αυτό, γνωστό ως Ιστός των Εγγράφων (Web of Documents), βασίστηκε σε τρεις θεμελιώδεις τεχνολογίες: το πρωτόκολλο HTTP για τη μεταφορά, τα URIs (Uniform Resource Identifiers) για την ταυτοποίηση των πόρων, και τη γλώσσα σήμανσης HTML (HyperText Markup Language) για την οπτική παρουσίαση της πληροφορίας.

Μολονότι η HTML αποδείχθηκε εξαιρετικά επιτυχής στον καθορισμό της δομής και της μορφοποίησης ενός εγγράφου για τους φυλλομετρητές (browsers), χαρακτηρίζεται από ένα εγγενές "σημασιολογικό κενό" (semantic gap). Ενδείξεις όπως οι επικεφαλίδες ή οι παράγραφοι καθοδηγούν αποκλειστικά τον τρόπο οπτικής απεικόνισης, χωρίς να παρέχουν καμία μετα-πληροφορία (metadata) σχετικά με το εννοιολογικό περιεχόμενο των δεδομένων. Κατά συνέπεια, η πληροφορία παραμένει μεν *μηχανικά αναγνώσιμη* (machine-readable) – ως σύνολο αδόμητων συμβολοσειρών (strings) – αλλά δεν καθίσταται *μηχανικά κατανοητή* (machine-understandable).

Καθώς ο όγκος της διαθέσιμης πληροφορίας στον Ιστό αυξήθηκε εκθετικά, η απουσία εννοιολογικής δομής οδήγησε στο φαινόμενο της υπερφόρτωσης πληροφορίας (information overload). Οι συμβατικές μηχανές αναζήτησης, λειτουργώντας κυρίως μέσω αλγορίθμων ανάκτησης πληροφορίας (Information Retrieval) που βασίζονται στο συντακτικό ταίριασμα λέξεων-κλειδιών (keyword matching) και στη στατιστική ανάλυση κειμένων, αδυνατούν να επιλύσουν λογικές ασάφειες, να διαχωρίσουν ομώνυμους όρους ή να συσχετίσουν δεδομένα από ετερογενείς πηγές προκειμένου να απαντήσουν σε σύνθετα ερωτήματα.

Η θεωρητική θεμελίωση για την υπέρβαση αυτών των περιορισμών διατυπώθηκε το 2001 στο οραματικό άρθρο των Tim Berners-Lee, James Hendler και Ora Lassila [2], όπου εισήχθη η έννοια του Σημασιολογικού Ιστού (Semantic Web). Ο Σημασιολογικός Ιστός δεν προτάθηκε ως ένα ανεξάρτητο, παράλληλο δίκτυο, αλλά ως μια δομική επέκταση του υφιστάμενου Ιστού, στην οποία η πληροφορία διαθέτει αυστηρά καθορισμένο νόημα (well-defined meaning). Απώτερος σκοπός αυτής της προσέγγισης είναι η δημιουργία ενός περιβάλλοντος όπου λογισμικά προγράμματα (intelligent agents) θα δύνανται να πλοηγούνται αυτόματα, να συλλέγουν δεδομένα και να διεξάγουν αυτοματοποιημένη συλλογιστική (automated reasoning) προς όφελος του χρήστη.

Αυτή η παραδειγματική μετατόπιση οδήγησε στην εξέλιξη του Ιστού των Εγγράφων στον **Ιστό των Δεδομένων (Web of Data)**. Σε αυτό το νέο επίπεδο αφαίρεσης, η ατομική μονάδα πληροφορίας παύει να είναι η ιστοσελίδα. Αντίθετα, στο επίκεντρο τίθενται οι "οντότητες" (π.χ. πρόσωπα, οργανισμοί, τοποθεσίες, έννοιες) και οι μεταξύ τους σχέσεις. Η μετάβαση αυτή συνοψίζεται βιβλιογραφικά στη φράση *"Things, not strings"* (Οντότητες, όχι απλές συμβολοσειρές).

Η θεμελιώδης διαφορά μεταξύ των δύο προσεγγίσεων εντοπίζεται στον μηχανισμό διασύνδεσης:

- **Στον Ιστό των Εγγράφων**, οι υπερσύνδεσμοι (HTML links) είναι αδιαφανείς (untyped). Υποδεικνύουν τη δυνατότητα μετάβασης από ένα έγγραφο σε ένα άλλο, χωρίς να προσδιορίζουν τη φύση της σχέσης μεταξύ τους.
- **Στον Ιστό των Δεδομένων**, οι υπερσύνδεσμοι είναι αυστηρά τυποποιημένοι (typed links). Βασιζόμενοι στο μοντέλο δεδομένων RDF (Resource Description Framework), περιγράφουν σαφώς την εννοιολογική σχέση μεταξύ δύο πόρων. Για παράδειγμα, μια διασύνδεση δεν υποδεικνύει απλώς ότι η σελίδα για την "Αθήνα" συνδέεται με τη σελίδα για την "Ελλάδα", αλλά κωδικοποιεί ρητά τη σημασιολογική σχέση:

  [Αθήνα] -> <αποτελεί πρωτεύουσα της> -> [Ελλάδας].

Προκειμένου να διασφαλιστεί η βιωσιμότητα του Ιστού των Δεδομένων και να αποτραπεί η δημιουργία απομονωμένων αποθετηρίων πληροφορίας (data silos), ο Tim Berners-Lee διατύπωσε το 2006 ένα σύνολο κατευθυντήριων γραμμών, γνωστό ως αρχές των **Διασυνδεδεμένων Ανοικτών Δεδομένων (Linked Open Data - LOD)** [14]. Οι τέσσερις θεμελιώδεις αρχές για τη δημοσίευση δεδομένων ορίζουν ότι:

1. Πρέπει να χρησιμοποιούνται αναγνωριστικά (URIs/IRIs) για την ονοματοδοσία και τον μοναδικό προσδιορισμό των οντοτήτων.
2. Τα εν λόγω URIs πρέπει να βασίζονται στο πρωτόκολλο HTTP, επιτρέποντας στους χρήστες και στα συστήματα να ανακτούν (dereference) τις πληροφορίες.
3. Κατά την προσπέλαση ενός URI, πρέπει να επιστρέφεται τυποποιημένη και χρήσιμη πληροφορία, ακολουθώντας τα πρότυπα του W3C (όπως το RDF και η γλώσσα επερωτήσεων SPARQL).
4. Τα δεδομένα πρέπει να περιέχουν συνδέσμους (URIs) προς άλλες εξωτερικές, σχετικές οντότητες, προωθώντας τη διασυνδεσιμότητα (interlinking) και επιτρέποντας την αλυσιδωτή ανακάλυψη γνώσης μέσω του γραφήματος (graph traversal).

Η αυστηρή υιοθέτηση των παραπάνω αρχών οδήγησε στη συγκρότηση του Παγκόσμιου Γραφήματος Διασυνδεδεμένων Δεδομένων (LOD Cloud), μέλος του οποίου αποτελεί και η DBpedia. Πάνω σε αυτό το ισχυρό θεωρητικό και τεχνολογικό υπόβαθρο βασίζεται η αρχιτεκτονική της παρούσας πτυχιακής εργασίας, αξιοποιώντας την οντολογική δομή των δεδομένων για την εκτέλεση πολύπλοκων και στοχευμένων ερωτημάτων.

.

2.3 Η WIKIPEDIA ΚΑΙ Ο ΡΟΛΟΣ ΤΗΣ ΣΤΗΝ ΑΝΟΙΚΤΗ ΓΝΩΣΗ

Η Wikipedia δεν αποτελεί απλώς τη μεγαλύτερη διαδικτυακή εγκυκλοπαίδεια, αλλά ίσως το πιο επιτυχημένο και μακροβιότερο παράδειγμα συλλογικής νοημοσύνης και πληθοπορισμού (crowdsourcing) στην ιστορία του Παγκόσμιου Ιστού. Από την ίδρυσή της, βασίστηκε στη φιλοσοφία του Web 2.0, μετατρέποντας τους απλούς αναγνώστες σε ενεργούς συντάκτες και δημιουργούς περιεχομένου. Σήμερα, στεγάζει σχεδόν το σύνολο της καταγεγραμμένης ανθρώπινης γνώσης, φιλοξενώντας δεκάδες εκατομμύρια άρθρα τα οποία ενημερώνονται διαρκώς και διατίθενται σε εκατοντάδες διαφορετικές γλώσσες και διαλέκτους. Η προσφορά της στην εκδημοκρατικοποίηση της γνώσης είναι αδιαμφισβήτητη, καθώς παρέχει ελεύθερη και άμεση πρόσβαση σε πληροφορίες, καταργώντας τα γεωγραφικά και οικονομικά σύνορα.

2.3.1 Το Σημασιολογικό Χάσμα και ο Περιορισμός των Αδόμητων Δεδομένων

Ωστόσο, η χρησιμότητα αυτού του κολοσσιαίου όγκου δεδομένων εξαρτάται απόλυτα από τον "αναγνώστη" του. Αν προσεγγίσουμε τη Wikipedia μέσα από τα "μάτια" ενός υπολογιστικού συστήματος ή ενός αλγορίθμου, η κατάσταση περιπλέκεται δραματικά. Αυτό οφείλεται στο γεγονός ότι η γνώση που αποθηκεύεται στα άρθρα είναι, στην πλειονότητά της, αδόμητη (unstructured data) και προορίζεται αποκλειστικά για ανθρώπινη κατανάλωση.
Τι σημαίνει αυτό στην πράξη; Όταν ένας άνθρωπος διαβάζει την πρώτη πρόταση στο λήμμα για τον Αριστοτέλη, κατανοεί αμέσως, διαισθητικά, ότι πρόκειται για έναν Έλληνα φιλόσοφο που γεννήθηκε στα Στάγειρα και υπήρξε μαθητής του Πλάτωνα. Ο ανθρώπινος εγκέφαλος εξάγει αυτόματα τις οντότητες (Αριστοτέλης, Στάγειρα, Πλάτων) και τις μεταξύ τους σχέσεις (γεννήθηκε σε, μαθήτευσε υπό). Για έναν αλγόριθμο, όμως, η ίδια πρόταση δεν φέρει καμία απολύτως εγγενή σημασιολογία. Αποτελεί απλώς μια επίπεδη αλληλουχία χαρακτήρων (string) κωδικοποιημένη σε μορφή UTF-8, ενθυλακωμένη μέσα σε ετικέτες μορφοποίησης HTML (όπως <p> ή <b>), οι οποίες υποδεικνύουν στον φυλλομετρητή πώς να εμφανίσει το κείμενο, αλλά όχι *τι σημαίνει* το κείμενο.

Αυτό το φαινόμενο ονομάζεται "Σημασιολογικό Χάσμα" (Semantic Gap). Για να μπορέσει μια μηχανή να "βγάλει νόημα", να συσχετίσει έννοιες και να εξάγει δομημένες απαντήσεις (π.χ. σε ένα ερώτημα όπως "Βρες μου όλους τους αρχαίους Έλληνες φιλοσόφους που γεννήθηκαν στη Μακεδονία"), απαιτούνται εξαιρετικά περίπλοκες και υπολογιστικά δαπανηρές τεχνικές Επεξεργασίας Φυσικής Γλώσσας (Natural Language Processing - NLP). Ακόμη και με τα σύγχρονα μοντέλα μηχανικής μάθησης, η εξαγωγή πληροφορίας από ελεύθερο κείμενο (Information Extraction) είναι επιρρεπής σε σφάλματα, καθώς η ανθρώπινη γλώσσα χαρακτηρίζεται από ασάφεια, συνώνυμα, ειρωνεία και πολύπλοκη συντακτική δομή.

2.3.2 Τα Πλαίσια Πληροφοριών (Infoboxes) ως Γέφυρα Δομημένης Γνώσης

Μέσα σε αυτόν τον ωκεανό αδόμητου κειμένου, υπήρξε ένα συγκεκριμένο αρχιτεκτονικό στοιχείο της Wikipedia που άλλαξε ριζικά τον τρόπο με τον οποίο οι μηχανές προσεγγίζουν την πληροφορία της: τα Πλαίσια Πληροφοριών (Infoboxes). Τα Infoboxes είναι οι χαρακτηριστικοί, συνοπτικοί πίνακες που εμφανίζονται συνήθως στην πάνω δεξιά γωνία των άρθρων (ή στην κορυφή των σελίδων στις κινητές συσκευές) και έχουν ως σκοπό να συνοψίσουν τα βασικότερα, αντικειμενικά στοιχεία μιας οντότητας, προσφέροντας στον αναγνώστη μια "γρήγορη ματιά" (quick facts).
Από τεχνική άποψη, τα Infoboxes δεν είναι απλό κείμενο. Αποτελούν ένα εξαιρετικό παράδειγμα ημι-δομημένης (semi-structured) πληροφορίας. Στον υποκείμενο κώδικα (Wikitext) ενός άρθρου, τα δεδομένα του Infobox εισάγονται μέσω προτύπων (templates) και είναι αυστηρά οργανωμένα με τη μορφή ζευγών "ιδιότητας - τιμής" (attribute-value pairs). Για παράδειγμα, στο λήμμα μιας χώρας, η πληροφορία κωδικοποιείται ρητά με ζεύγη όπως:

- Πρωτεύουσα : Αθήνα
- Πληθυσμός : 10.423.054
- Νόμισμα : Ευρώ

Αυτή η προσέγγιση μετατρέπει ένα κομμάτι του άρθρου σε μια μορφή που μοιάζει εκπληκτικά με τις εγγραφές μιας σχεσιακής βάσης δεδομένων ή με τη λογική των εγγράφων JSON. Η σχέση μεταξύ της οντότητας (της σελίδας) και του χαρακτηριστικού της είναι ρητή, ξεκάθαρη και εύκολα αναγνώσιμη από έναν αλγόριθμο ανάλυσης (parser), χωρίς να απαιτείται η παραμικρή χρήση τεχνητής νοημοσύνης ή κατανόησης φυσικής γλώσσας.

2.3.3 Το Εφαλτήριο για τον Σημασιολογικό Ιστό

Η συνειδητοποίηση της αξίας αυτών των ημι-δομημένων τμημάτων αποτέλεσε ορόσημο για την Επιστήμη των Υπολογιστών. Ερευνητές αντιλήφθηκαν ότι εάν μπορούσαν να γράψουν αυτοματοποιημένα σενάρια (scripts) τα οποία θα σάρωναν και τα εκατομμύρια άρθρα της Wikipedia, εξάγοντας αποκλειστικά τα ζεύγη ιδιότητας-τιμής από τα Infoboxes, θα μπορούσαν να κατασκευάσουν τη μεγαλύτερη ανοιχτή, δομημένη βάση δεδομένων στον κόσμο.
Αυτά τα ζεύγη ιδιότητας-τιμής αντιστοιχίζονταν άμεσα στη λογική των "Τριάδων" (Υποκείμενο - Κατηγόρημα - Αντικείμενο) που απαιτεί το πρότυπο RDF του Σημασιολογικού Ιστού. Το Υποκείμενο είναι το ίδιο το άρθρο (π.χ. Ελλάδα), το Κατηγόρημα είναι η ιδιότητα του Infobox (π.χ. Πρωτεύουσα) και το Αντικείμενο είναι η τιμή (π.χ. Αθήνα).
Αυτή ακριβώς η διαδικασία μαζικής εξόρυξης και μετασχηματισμού της ημι-δομημένης πληροφορίας των Infoboxes αποτέλεσε τον πυρήνα, το θεμέλιο και το απόλυτο εφαλτήριο για τη δημιουργία του έργου **DBpedia** [1]. Μέσω αυτής της έξυπνης αξιοποίησης του πληθοπορισμού, η Wikipedia μετουσιώθηκε από μια απλή ψηφιακή εγκυκλοπαίδεια σε μια τεράστια, παγκόσμια και διασυνδεδεμένη βάση γνώσης (Knowledge Base), την οποία οι υπολογιστές μπορούν πλέον όχι μόνο να διαβάσουν, αλλά να κατανοήσουν και να επερωτήσουν με μαθηματική ακρίβεια.

2.3.4 Η ΑΡΧΙΤΕΚΤΟΝΙΚΉ "LAYER CAKE" ΤΟΥ ΣΗΜΑΣΙΟΛΟΓΙΚΟΎ ΙΣΤΟΎ

Η υλοποίηση του οράματος του Σημασιολογικού Ιστού δεν αποτελεί μια μονολιθική τεχνολογία, αλλά βασίζεται σε μια αυστηρά ιεραρχική στοίβα προτύπων, γνωστή στη βιβλιογραφία ως "Semantic Web Layer Cake" (Αρχιτεκτονική Διαστρωμάτωσης του Σημασιολογικού Ιστού). Η αρχιτεκτονική αυτή, η οποία προτάθηκε από τον Tim Berners-Lee και συντηρείται από το W3C, ακολουθεί μια σπονδυλωτή λογική (modular approach): κάθε επίπεδο (layer) αξιοποιεί τις δομές του ακριβώς κατώτερου στρώματος και παρέχει αναβαθμισμένες, πιο περίπλοκες λειτουργίες στο αμέσως ανώτερο. Η δομή αυτή περιγράφει τον σταδιακό μετασχηματισμό του Ιστού από ένα δίκτυο αδόμητων εγγράφων σε μια παγκόσμια, μηχανικά κατανοητή βάση δεδομένων [2].
Αναλύοντας την πυραμίδα από τη βάση προς την κορυφή, διακρίνονται τα εξής επίπεδα:

1. Επίπεδο Ταυτοποίησης και Κωδικοποίησης (URI/IRI & Unicode): Στο θεμελιώδες επίπεδο της αρχιτεκτονικής συναντάμε το Unicode και τους Ενιαίους Αναγνωριστικούς Πόρους (URIs/IRIs). Το πρότυπο Unicode εξασφαλίζει την παγκόσμια συμβατότητα και τη σωστή αναπαράσταση των χαρακτήρων ανεξαρτήτως φυσικής γλώσσας (πολυγλωσσικότητα). Αντίστοιχα, τα URIs/IRIs διασφαλίζουν την παγκόσμια και μοναδική ταυτοποίηση κάθε φυσικής ή αφηρημένης οντότητας (π.χ. προσώπων, εννοιών, τοποθεσιών) στον Παγκόσμιο Ιστό, επιλύοντας το πρόβλημα της ονοματοδοσίας.

2. Συντακτικό Επίπεδο (XML & XML Schema): Ακριβώς από πάνω, η γλώσσα XML (eXtensible Markup Language) και τα σχήματά της (XML Schemas) παρέχουν έναν κοινό, ευέλικτο συντακτικό μηχανισμό. Το επίπεδο αυτό προσφέρει μια ιεραρχική δομή (δέντρου) για τη σειριοποίηση (serialization) και την ανταλλαγή των δεδομένων μεταξύ διαφορετικών συστημάτων, χωρίς ωστόσο να τους προσδίδει οποιαδήποτε σημασιολογία.

3. Επίπεδο Μοντέλου Δεδομένων (RDF): Το σημείο καμπής μεταξύ του παραδοσιακού και του Σημασιολογικού Ιστού βρίσκεται στο RDF (Resource Description Framework). Το επίπεδο αυτό ορίζει το βασικό μοντέλο δεδομένων, το οποίο έχει τη μορφή κατευθυνόμενου γράφου (directed graph).

Η πληροφορία αναπαρίσταται αυστηρά μέσω τριπλετών (Υποκείμενο - Κατηγόρημα - Αντικείμενο), μετατρέποντας τον Ιστό από μια αποθήκη εγγράφων σε ένα δίκτυο διασυνδεδεμένων εννοιών.

4. Επίπεδο Οντολογιών και Λεξιλογίων (RDFS & OWL): Για να μπορέσουν τα συστήματα να κατανοήσουν τη δομή των δεδομένων RDF, απαιτούνται ειδικά "λεξιλόγια". Το RDFS (RDF Schema) επιτρέπει τον ορισμό βασικών ιεραρχιών κλάσεων και ιδιοτήτων (π.χ. δήλωση ότι το "Μουσείο" είναι υποκλάση του "Κτιρίου"). Η γλώσσα OWL (Web Ontology Language) επεκτείνει αυτές τις δυνατότητες προσφέροντας ισχυρή εκφραστικότητα, επιτρέποντας τον καθορισμό περιορισμών, καρδικοτήτων (cardinality), ισοδυναμιών και ασυμβατοτήτων (disjointness) μεταξύ των κλάσεων.

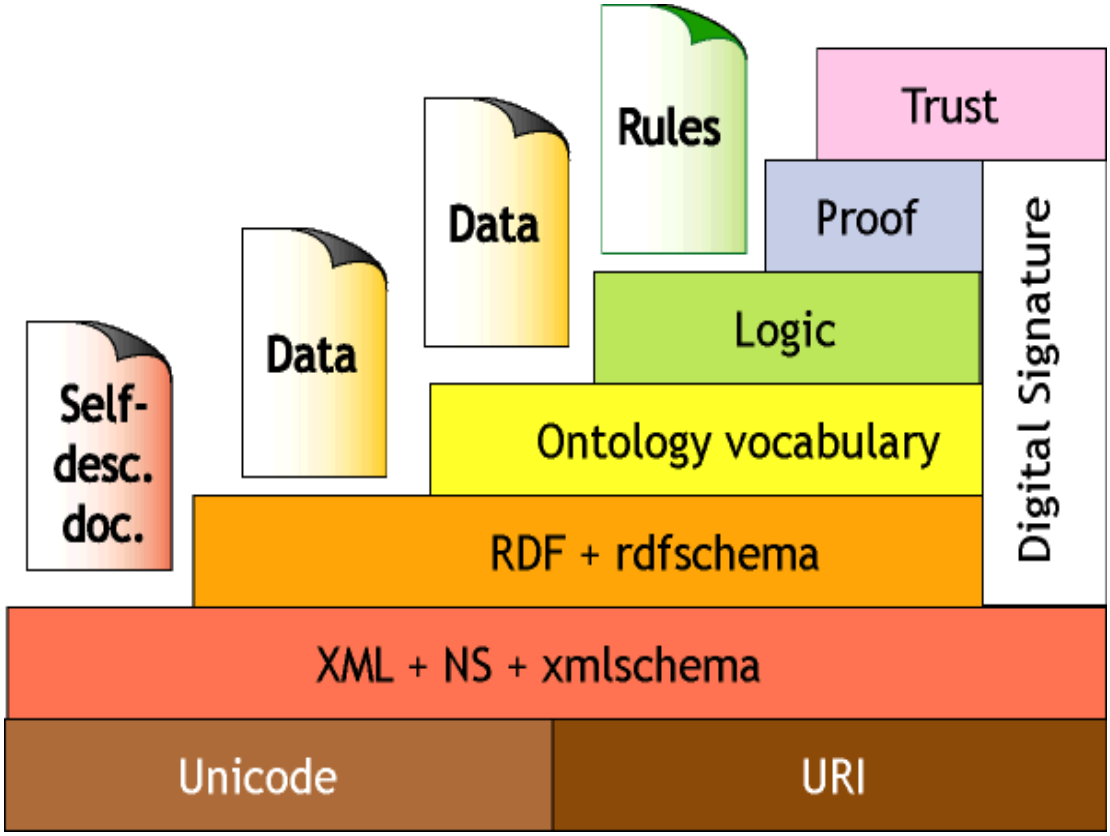
5. Επίπεδο Επερωτήσεων (SPARQL): Παράλληλα με το επίπεδο των οντολογιών, τοποθετείται η γλώσσα SPARQL. Είναι ο τυποποιημένος μηχανισμός (Query Language & Protocol) για την αναζήτηση, άντληση και διαχείριση της πληροφορίας που βρίσκεται αποθηκευμένη με μορφή τριπλετών RDF. Αποτελεί το εργαλείο μέσω του οποίου οι τελικές εφαρμογές (όπως ο Semantic Browser της παρούσας εργασίας) αλληλεπιδρούν με το Γράφημα Γνώσης.

6. Επίπεδο Λογικής και Εξαγωγής Συμπερασμάτων (Logic / RIF): Σε αυτό το στρώμα (Rule Interchange Format), οι κανόνες λογικής εφαρμόζονται πάνω στις υπάρχουσες οντολογίες (OWL) προκειμένου να παραχθεί αυτοματοποιημένη συλλογιστική (automated reasoning). Ο υπολογιστής δεν ανακτά απλώς αποθηκευμένα δεδομένα, αλλά συνάγει *νέα, έμμεση γνώση*. Για παράδειγμα, αν γνωρίζει ότι η οντότητα Α βρίσκεται στην οντότητα Β, και η Β βρίσκεται στην οντότητα Γ, εξάγει το λογικό συμπέρασμα ότι η Α βρίσκεται και στη Γ, χωρίς αυτό να έχει καταχωρηθεί ρητά.

7. Επίπεδο Εμπιστοσύνης και Απόδειξης (Proof & Trust): Στην κορυφή της πυραμίδας εδράζονται οι έννοιες της Απόδειξης και της Εμπιστοσύνης. Εφόσον οποιοσδήποτε μπορεί να δημοσιεύσει δεδομένα στον Ιστό, το επίπεδο της "Απόδειξης" (Proof) καταγράφει την προέλευση των πληροφοριών (provenance) και τη λογική διαδρομή που ακολουθήθηκε για την εξαγωγή ενός

συμπεράσματος. Τέλος, το επίπεδο της "Εμπιστοσύνης" (Trust) ενσωματώνει τεχνικές ψηφιακών υπογραφών και κρυπτογραφίας, επιτρέποντας στα
συστήματα να ελέγχουν την αξιοπιστία της πηγής (authoritativeness) πριν αποδεχθούν μια πληροφορία ως αληθή.

Μέσω αυτής της διαστρωμάτωσης, διασφαλίζεται ότι ο Σημασιολογικός Ιστός αποτελεί ένα ανθεκτικό, **διαλειτουργικό και κλιμακούμενο** (scalable) οικοσύστημα, ικανό να υποστηρίξει από απλές εφαρμογές αναζήτησης έως προηγμένα συστήματα τεχνητής νοημοσύνης.



Εικόνα 1 :Διάγραμμα layer cake W3C Semantic
Πηγή : Web standards https://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html

2.4 DBPEDIA: ΕΞΑΓΟΝΤΑΣ ΔΟΜΗΜΈΝΗ ΓΝΩΣΗ

Το ερευνητικό εγχείρημα **DBpedia**, το όνομα του οποίου προκύπτει από τον συνδυασμό των λέξεων Database και Wikipedia, Ήταν ο καρπός της συνεργασίας ανάμεσα σε ερευνητές από τα Πανεπιστήμια της Λειψίας και του Βερολίνου (Free University), καθώς και της OpenLink Software. Η κεντρική ιδέα πίσω από την DBpedia είναι η αυτοματοποιημένη εξαγωγή (**parsing**) της δομημένης πληροφορίας που ήδη υπάρχει στη Wikipedia, ώστε αυτή να γίνει προσβάσιμη στον Παγκόσμιο Ιστό μέσα από τα πρότυπα του Σημασιολογικού Ιστού [14].
Στην πράξη, η DBpedia λειτουργεί ως ένας κεντρικός κόμβος (hub) στο ευρύτερο «σύννεφο» των Διασυνδεδεμένων Δεδομένων (**LOD Cloud**). Αυτό που την κάνει τόσο σημαντική είναι η ικανότητά της να γεφυρώνει διαφορετικούς κόσμους: συνδέει τις δικές της οντότητες με εξωτερικές πηγές δεδομένων, όπως το GeoNames (για τοποθεσίες) ή το MusicBrainz (για τη μουσική). . Ο κεντρικός πυρήνα του Linked Open Data Cloud (LOD Cloud**)** Πρόκειται για ένα παγκόσμιο δίκτυο από χιλιάδες βάσεις δεδομένων που είναι διασυνδεδεμένες μεταξύ τους.

Όταν δημοσιεύουμε δεδομένα στην Ελληνική DBpedia, αυτά συνδέονται αυτόματα με άλλες πηγές, όπως:

- **Geonames:** Για ακριβή γεωγραφικά δεδομένα.
- **VIAF (Virtual International Authority File):** Για ταυτοποίηση προσώπων και συγγραφέων.

Wikidata: Για διασταύρωση στοιχείων σε πραγματικό χρόνο. Αυτή η διασύνδεση επιτρέπει την εκτέλεση "Ομοσπονδιακών Ερωτημάτων" (Federated Queries), όπου μια εφαρμογή μπορεί να αντλεί ταυτόχρονα πληροφορίες από την Αθήνα, το Βερολίνο και το Λονδίνο σαν να ήταν μία ενιαία βάση.Το αποτέλεσμα αυτής της ενιαίεις δικτύωσης είναι η δημιουργία ενός τεράστιου, ενιαίου **Γραφήματος Γνώσης (Knowledge Graph)**, το οποίο επιτρέπει την αναζήτηση και τη σύνδεση πληροφοριών που προηγουμένως ήταν απομονωμένες.

2.5 ΤΟ ΠΡΟΤΥΠΟ RDF (RESOURCE DESCRIPTION FRAMEWORK)

Η υποκείμενη αρχιτεκτονική δεδομένων της DBpedia διαφοροποιείται ριζικά από τα παραδοσιακά Συστήματα Διαχείρισης Σχεσιακών Βάσεων Δεδομένων (RDBMS), όπως η MySQL ή η PostgreSQL. Στις σχεσιακές βάσεις, η πληροφορία οργανώνεται σε αυστηρά προκαθορισμένους πίνακες (tables) με συγκεκριμένες στήλες, γεγονός που επιβάλλει ένα άκαμπτο σχήμα (rigid schema). Αυτή η προσέγγιση, αν και αποδοτική για κλειστά εταιρικά συστήματα, κρίνεται ανεπαρκής για την κλίμακα, την ετερογένεια και τη συνεχή εξέλιξη της πληροφορίας στον Παγκόσμιο Ιστό. Για την αντιμετώπιση αυτής της πρόκλησης, η κοινοπραξία W3C (World Wide Web Consortium) θέσπισε το πρότυπο **RDF (Resource Description Framework)** [6], το οποίο προσφέρει απόλυτη σχηματική ευελιξία (schema-less flexibility).

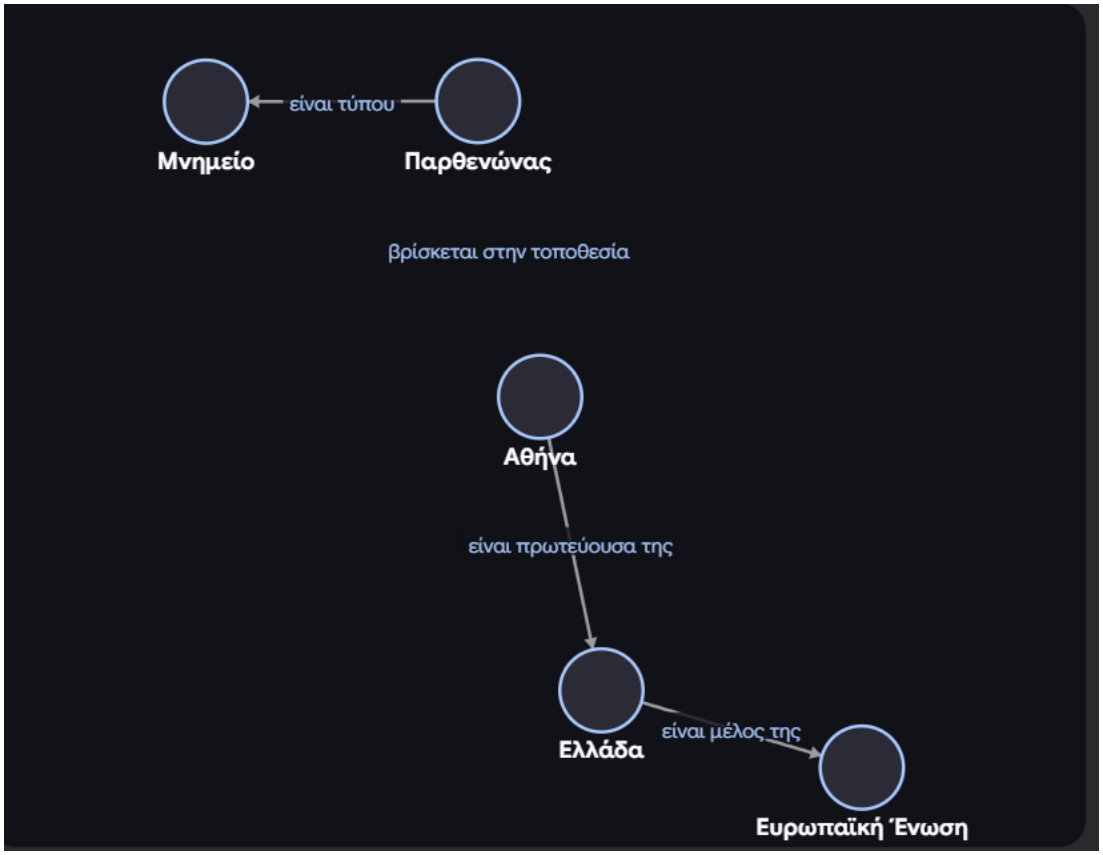
Στο μοντέλο RDF, η πληροφορία δεν αποθηκεύεται σε γραμμές πινάκων, αλλά αναπαρίσταται ως ένα σύνολο λογικών προτάσεων, γνωστών ως **Τριάδες (Triples)**. Η αρχιτεκτονική της τριάδας ακολουθεί τη φυσική δομή της ανθρώπινης γλώσσας και αποτελείται πάντα από τρία διακριτά μέρη:

1. **Υποκείμενο (Subject):** Αντιπροσωπεύει τον κύριο πόρο, την οντότητα ή την έννοια για την οποία γίνεται η δήλωση (π.χ. ο "Παρθενώνας" ή ο "Οδυσσέας Ελύτης"). Το υποκείμενο οφείλει να ταυτοποιείται με μοναδικό τρόπο.
2. **Κατηγόρημα (Predicate / Property):** Ορίζει το χαρακτηριστικό, την ιδιότητα ή τη σχέση που αποδίδεται στο υποκείμενο. Παραδείγματα κατηγορημάτων αποτελούν οι έννοιες "βρίσκεται στην τοποθεσία", "έχει ημερομηνία γέννησης" ή "ανήκει στην κατηγορία". Το κατηγόρημα είναι ο μηχανισμός που προσδίδει αυστηρή *σημασιολογία* στη σύνδεση των δεδομένων.
3. **Αντικείμενο (Object):** Αποτελεί την τιμή της ιδιότητας που ορίστηκε από το κατηγόρημα. Το αντικείμενο μπορεί να έχει δύο μορφές: είτε να είναι ένας *άλλος ανεξάρτητος πόρος* (π.χ. η πόλη "Αθήνα", δημιουργώντας έτσι μια σχέση μεταξύ δύο οντοτήτων), είτε να είναι ένα *απλό αλφαριθμητικό δεδομένο (Literal)*, όπως η συμβολοσειρά "438 π.Χ.", ο ακέραιος αριθμός 438, ή ένα γεωγραφικό στίγμα. Τα Literals συχνά συνοδεύονται από τύπους δεδομένων (Datatypes, π.χ. xsd:integer) ή ετικέτες γλώσσας (Language Tags, π.χ. @el για τα ελληνικά), διασφαλίζοντας τη διαλειτουργικότητα των συστημάτων.

Ένα από τα σημαντικότερα προβλήματα στα κατανεμημένα συστήματα είναι η σύγκρουση ονοματολογίας (naming collisions) και η πολυσημία (π.χ. η λέξη "Αθήνα" μπορεί να αναφέρεται στην πρωτεύουσα της Ελλάδας ή στην πόλη Athens της Πολιτείας Τζόρτζια των Η.Π.Α.). Το πρότυπο RDF επιλύει οριστικά αυτό το πρόβλημα επιβάλλοντας τη χρήση **Ενιαίων Αναγνωριστικών Πόρων (URI - Uniform Resource Identifiers)** για όλα τα υποκείμενα και τα κατηγορήματα.

Για παράδειγμα, το URI http://el.dbpedia.org/resource/Αθήνα αποτελεί ένα παγκόσμια μοναδικό αναγνωριστικό (Globally Unique Identifier) που παραπέμπει αποκλειστικά στην ελληνική πρωτεύουσα,

μηδενίζοντας τις πιθανότητες σύγχυσης για τη μηχανή. Προκειμένου να διατηρηθεί η αναγνωσιμότητα του κώδικα και να μειωθεί ο όγκος των δεδομένων, συνηθίζεται η συμπύκνωση αυτών των μεγάλων URI μέσω **Προθεμάτων (Prefixes)**. Έτσι, το παραπάνω URI μπορεί να γραφτεί συνοπτικά ως dbr:Αθήνα, ενώ οι ιδιότητες της οντολογίας συμβολίζονται ως dbo: (π.χ. dbo:location).
Η πραγματική δυναμική του προτύπου RDF, ωστόσο, αναδεικνύεται όταν εκατομμύρια μεμονωμένες τριάδες συνενώνονται. Καθώς το Αντικείμενο (Object) μιας τριάδας μπορεί να αποτελέσει το Υποκείμενο (Subject) μιας άλλης, δημιουργείται ένα τεράστιο, διασυνδεδεμένο δίκτυο πληροφοριών. Μαθηματικά, το σύνολο αυτών των τριάδων μεταφράζεται σε έναν **Κατευθυνόμενο Γράφο με Ετικέτες (Directed Labeled Graph)**, όπου οι κόμβοι (nodes) είναι οι πόροι/τιμές και οι ακμές (edges) είναι τα κατηγορήματα. Αυτός ο διευρυνόμενος ιστός πληροφορίας είναι αυτό που ονομάζουμε **Γράφημα Γνώσης (Knowledge Graph)**, το οποίο επιτρέπει σε γλώσσες όπως η SPARQL να εκτελούν πολύπλοκες διασχίσεις (graph traversals) και να εξάγουν κρυμμένα μοτίβα που θα ήταν αδύνατο να εντοπιστούν σε παραδοσιακούς πίνακες.



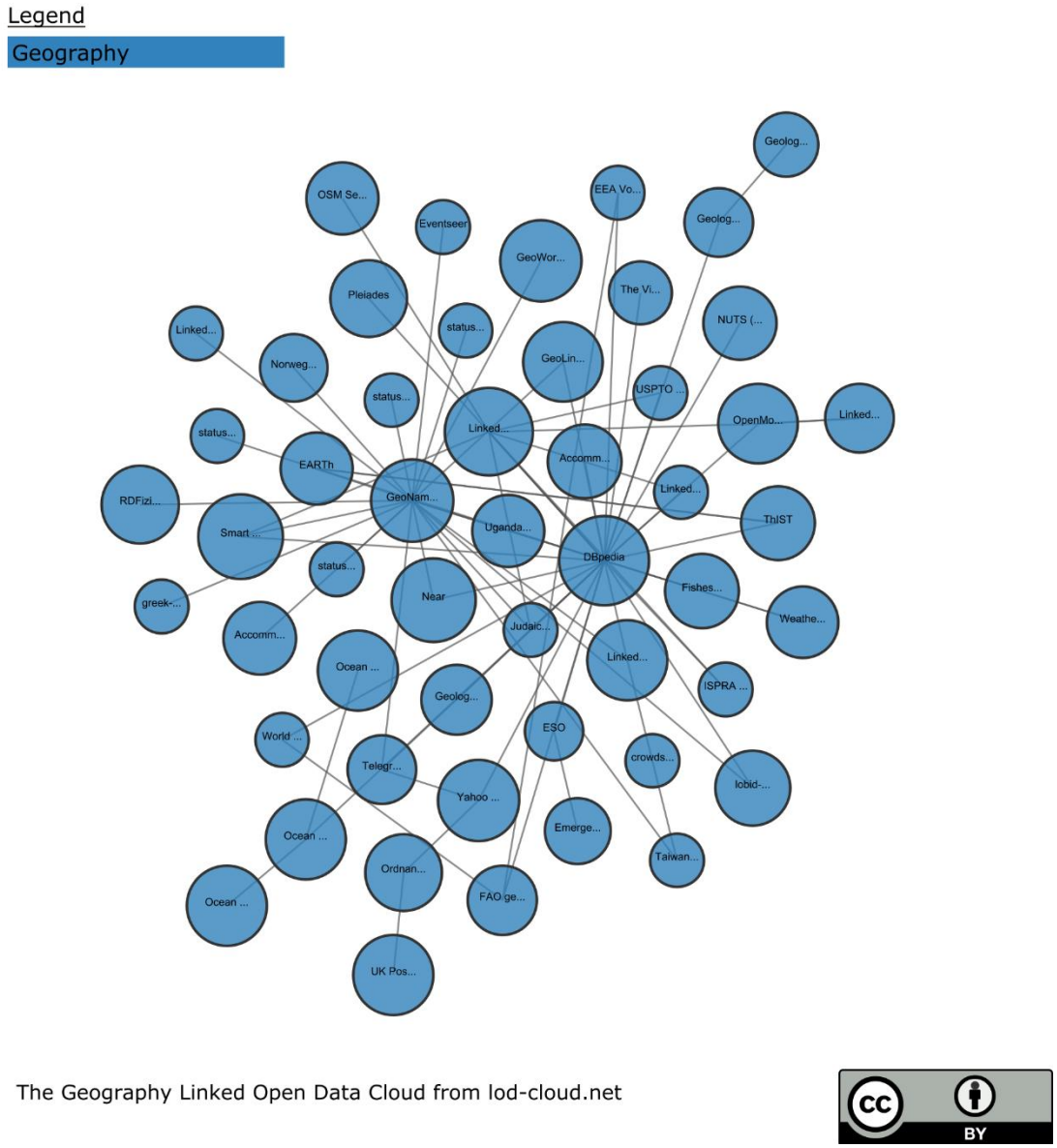
Εικόνα 2 Παράδειγμα Γράφου με Τριάδες RDF.
Πηγή:Δημιουργία του συγγραφέα

Κάθε γραμμή στην Εικόνα 2 αναπαριστά μια 'Τριάδα' (Triple):
Υποκείμενο->Κατηγόρημα->Αντικείμενο.Αποτελείται απο 5 κομβους και 4 τριάδες.

2.6 Ο ΕΛΛΗΝΙΚΟΣ ΚΌΜΒΟΣ (GREEK DBPEDIA) ΚΑΙ ΟΙ ΠΡΟΚΛΉΣΕΙΣ ΤΟΥ

Ένα από τα σημαντικότερα χαρακτηριστικά της DBpedia είναι η δυνατότητά της να αντλεί και να συγχωνεύει πληροφορίες από πολλές διαφορετικές γλωσσικές εκδόσεις (Internationalization). Σε αυτό το πλαίσιο, η Ελληνική DBpedia (el.dbpedia.org) επικεντρώνεται αποκλειστικά στην εξαγωγή

σημασιολογικών δεδομένων από την ελληνική έκδοση της Wikipedia. Ουσιαστικά, λειτουργεί ως ένας ψηφιακός κόμβος που αναδεικνύεται σε σημείο αναφοράς για τον ελληνικό πολιτισμό, την ιστορία και τη γλώσσα μας στο παγκόσμιο νέφος των Διασυνδεδεμένων Ανοικτών Δεδομένων (LOD Cloud).



Εικόνα 3 Επίσημο γράφημα LOD Γεωγραφίας 10-Apr-2026,
Πηγή: https://lod-cloud.net/versions/2011-09-19/lod-cloud.pdf

Η Ελληνική DBpedia (el.dbpedia.org) αποτελεί το εθνικό και γλωσσικό υποσύνολο του παγκόσμιου εγχειρήματος εξαγωγής πληροφορίας DBpedia. Πρόκειται για ένα μεγάλης κλίμακας Γράφημα Γνώσης (Knowledge Graph), το οποίο δημιουργείται μέσω της αυτόματης εξαγωγής δομημένων δεδομένων (structured data) αποκλειστικά από την ελληνική έκδοση της Wikipedia (el.wikipedia.org).

Ενώ η Wikipedia είναι σχεδιασμένη για να διαβάζεται από ανθρώπους, περιέχοντας κυρίως ελεύθερο κείμενο, η DBpedia λειτουργεί ως το "σημασιολογικό της αντίγραφο". Αξιοποιώντας έναν ειδικό μηχανισμό (DBpedia Information Extraction Framework), σαρώνει την ελληνική Wikipedia, εντοπίζει τα πλαίσια πληροφοριών (Infoboxes), τις κατηγορίες, τις γεωγραφικές συντεταγμένες και τους εσωτερικούς συνδέσμους των άρθρων, και τα μετατρέπει σε μηχανικά αναγνώσιμη μορφή, δηλαδή σε τριπλέτες τεχνολογίας RDF.

Η σημασία του Ελληνικού Κόμβου είναι τεράστια, καθώς αποτελεί τον κεντρικό πυλώνα (hub) για το νέφος των Ελληνικών Ανοικτών Διασυνδεδεμένων Δεδομένων (Greek LOD Cloud). Επιτρέπει σε έξυπνα συστήματα, αλγορίθμους και ερευνητές να εκτελούν πολύπλοκα ερωτήματα (μέσω της γλώσσας SPARQL) πάνω στην ελληνική ιστορία, γεωγραφία και πολιτισμό. Λειτουργίες όπως "Βρες μου όλα τα μουσεία της Ελλάδας που ιδρύθηκαν τον 19ο αιώνα" είναι πλέον εφικτές, κάτι που θα ήταν αδύνατο με μια απλή παραδοσιακή αναζήτηση λέξεων-κλειδιών.

Ωστόσο, η ποιότητα και η πληρότητα αυτού του ελληνικού Γραφήματος Γνώσης (Knowledge Graph) είναι άρρηκτα συνδεδεμένες με τη δομή της ίδιας της ελληνικής Wikipedia, καθώς και με τους περιορισμούς του μηχανισμού εξαγωγής πληροφοριών (DBpedia Information Extraction Framework). Στην πράξη, κατά την αναζήτηση και χρήση των δεδομένων, αναδεικνύονται κρίσιμα εμπόδια, τα οποία κατηγοριοποιούνται ως εξής:

Ελλείψεις και Αδυναμία Χαρτογράφησης (Missing Properties & Mapping Failures): Η εξαγωγή δομημένων δεδομένων βασίζεται στα "Πρότυπα" (Infoboxes) που τοποθετούν οι συντάκτες στα άρθρα. Πολλά ελληνικά άρθρα είτε δεν περιλαμβάνουν καθόλου Infoboxes, είτε αυτά είναι ημιτελή. Επιπλέον, για να ενταχθεί μια πληροφορία στην επίσημη οντολογία (dbo:), απαιτείται η χειροκίνητη χαρτογράφηση (mapping) του ελληνικού προτύπου από εθελοντές της κοινότητας (Mappings Wiki). Όταν αυτή η χαρτογράφηση απουσιάζει, τα δεδομένα εξάγονται μεν, αλλά καταλήγουν στον αδόμητο χώρο ονομάτων dbp: (raw properties). Για παράδειγμα, συχνά λείπουν ή δεν έχουν χαρτογραφηθεί σωστά κρίσιμα στοιχεία, όπως οι γεωγραφικές συντεταγμένες ή η ιδιότητα της χώρας (dbo:country).

Ασυνέπειες στην Οντολογία (Ontology Inconsistency): Στην ελληνική Wikipedia επικρατεί το πρόβλημα της "πολυφωνίας" των συντακτών. Διαφορετικοί χρήστες δημιουργούν ή χρησιμοποιούν διαφορετικά πρότυπα και ορολογίες για την ίδια ακριβώς έννοια (π.χ. σε ένα πρότυπο χρησιμοποιείται η παράμετρος χώρα, σε άλλο το κράτος και σε τρίτο η τοποθεσία). Αυτή η έλλειψη τυποποίησης δυσχεραίνει τη δημιουργία καθολικών ερωτημάτων SPARQL, καθώς ο χρήστης πρέπει να γνωρίζει εκ των προτέρων όλες τις πιθανές παραλλαγές μιας ιδιότητας για να ανακτήσει το σύνολο των αποτελεσμάτων.

Γλωσσολογικές Ιδιαιτερότητες της Ελληνικής (Linguistic Challenges): Σε αντίθεση με την αγγλική, η ελληνική είναι μια γλώσσα με πλούσια μορφολογία, ισχυρή κλίση (πτώσεις, γένη, αριθμοί) και πληθώρα συνωνύμων. Στα ακατέργαστα δεδομένα της DBpedia, η ίδια οντότητα μπορεί να συναντάται με διαφορετικές μορφές (π.χ. "Αθήνα", "στην Αθήνα", "της Αθήνας"). Αυτό προκαλεί σοβαρά προβλήματα κατά την ακριβή σύγκριση συμβολοσειρών (string matching) μέσα στα ερωτήματα, μειώνοντας την ακρίβεια της αναζήτησης.

Προβλήματα Εσωτερικής και Εξωτερικής Διασύνδεσης (Interlinking Issues): Η αξία του Σημασιολογικού Ιστού έγκειται στις συνδέσεις (owl:sameAs). Στον ελληνικό κόμβο παρατηρείται συχνά ελλιπής διασύνδεση τόσο εσωτερικά (μεταξύ ελληνικών οντοτήτων) όσο και εξωτερικά (με άλλες βάσεις όπως το GeoNames, Wikidata, YAGO, ή την Αγγλική DBpedia). Αποτέλεσμα αυτού είναι πολλές οντότητες να παραμένουν "απομονωμένες" στο γράφημα, καθιστώντας αδύνατη την πλοήγηση (graph traversal) μέσω ερωτημάτων JOIN.

Εξαιτίας των παραπάνω δομικών κενών, καθίσταται σαφές ότι η απλή και συμβατική άντληση δεδομένων (χρήση αποκλειστικά dbo ιδιοτήτων) συχνά δεν αρκεί, καθώς οδηγεί σε εξαιρετικά χαμηλή ανάκληση (recall) αποτελεσμάτων.Προκειμένου να προσπελαστούν αυτοί οι αυστηροί δομικοί περιορισμοί του ελληνικού κόμβου, κρίθηκε αναγκαία η εφαρμογή πιο σύνθετων τεχνικών προγραμματισμού. Συγκεκριμένα, αξιοποιήθηκε σε μεγάλο βαθμό η αναζήτηση ελεύθερου κειμένου εντός της περίληψης των άρθρων (dbo:abstract), σε συνδυασμό με τεχνικές Σημασιολογικού Φιλτραρίσματος (Semantic Filtering) και Κανονικών Εκφράσεων (Regular Expressions). Με αυτή την προσέγγιση (fallback strategy), το σύστημα καταφέρνει να εξάγει λανθάνουσα πληροφορία που δεν

είναι ρητά δηλωμένη στην οντολογία, διασφαλίζοντας την ακρίβεια, την ποιότητα και τον πλούτο των αποτελεσμάτων που παραδίδονται στον τελικό χρήστη.

2.7 Η ΓΛΏΣΣΑ ΕΡΩΤΗΜΆΤΩΝ SPARQL

Για να καταστεί δυνατή η άντληση, η διαχείριση και ο μετασχηματισμός της πληροφορίας από βάσεις δεδομένων που ακολουθούν το πρότυπο RDF (τα λεγόμενα Triplestores ή Quadstores), απαιτείται η χρήση ενός εξειδικευμένου εργαλείου: της γλώσσας **SPARQL (SPARQL Protocol and RDF Query Language)**. Πρόκειται για το επίσημο, διεθνές πρότυπο του οργανισμού W3C (World Wide Web Consortium), το οποίο αποτελεί τον ακρογωνιαίο λίθο και τη "γέφυρα" επικοινωνίας στον Σημασιολογικό Ιστό. Η SPARQL δεν είναι απλώς μια γλώσσα ερωτημάτων, αλλά ένα ολοκληρωμένο πρωτόκολλο επικοινωνίας (μεταφορά δεδομένων μέσω HTTP) που επιτρέπει σε κατανεμημένα συστήματα να ανταλλάσσουν γνώση με κοινά αποδεκτό τρόπο.
Αν και με μια πρώτη ματιά η σύνταξη της SPARQL θυμίζει έντονα την κλασική SQL των Σχεσιακών Βάσεων Δεδομένων (εξαιτίας της χρήσης δεσμευμένων λέξεων όπως SELECT, WHERE, FILTER και ORDER BY), η υποκείμενη αρχιτεκτονική και φιλοσοφία της είναι ριζικά διαφορετική. Αυτή η διαφοροποίηση πηγάζει από τον τρόπο με τον οποίο τα δύο μοντέλα αντιλαμβάνονται την οργάνωση των δεδομένων.

Στον παραδοσιακό σχεσιακό προγραμματισμό, η SQL "επερωτά" αυστηρά δομημένους πίνακες, στήλες και σειρές, βασιζόμενη σε προκαθορισμένα σχήματα (rigid schemas) και στην Υπόθεση του Κλειστού Κόσμου (Closed World Assumption). Αντίθετα, η SPARQL σχεδιάστηκε για ένα περιβάλλον χωρίς αυστηρό σχήμα (schema-less), υιοθετώντας την Υπόθεση του Ανοικτού Κόσμου (Open World Assumption), όπου η πληροφορία έχει τη μορφή ενός τεράστιου, ρευστού και διασυνδεδεμένου ιστού.

Λόγω αυτής της δομής, η SPARQL δεν ερευνά σε συγκεκριμένες "στήλες", αλλά λειτουργεί αποκλειστικά με τον μηχανισμό της **Αντιστοίχισης Μοτίβων Γραφήματος (Graph Pattern Matching)**. Ο προγραμματιστής δεν ορίζει το *πού* ακριβώς θα ψάξει η βάση, αλλά περιγράφει το "σχήμα" (template) της πληροφορίας που αναζητά.

Ο πυρήνας αυτού του μηχανισμού είναι τα **Βασικά Μοτίβα Γραφήματος (Basic Graph Patterns - BGPs)**. Ένα BGP είναι ουσιαστικά ένα σύνολο από τριάδες RDF, στις οποίες ένα ή περισσότερα στοιχεία (Υποκείμενο, Κατηγόρημα, Αντικείμενο) έχουν αντικατασταθεί από Μεταβλητές (οι οποίες στη SPARQL συμβολίζονται με το πρόθεμα ? ή $). Όταν το ερώτημα υποβάλλεται στον κινητήρα του Triplestore (όπως ο Virtuoso), ο αλγόριθμος διασχίζει τον Γράφο Γνώσης και προσπαθεί να εντοπίσει όλα τα δυνατά υπο-γραφήματα (sub-graphs) που "κουμπώνουν" συντακτικά στο πρότυπο που έδωσε ο χρήστης, αντικαθιστώντας τις μεταβλητές με πραγματικά URIs ή Literals.

Για παράδειγμα, το νοητικό μοτίβο *"Βρες μου όλες τις οντότητες (X) που είναι μουσεία και βρίσκονται στην πόλη της Αθήνας (Y)"* μεταφράζεται στον γράφο ως ένα τρίγωνο σχέσεων. Αν η μηχανή βρει κόμβους που ικανοποιούν ταυτόχρονα αυτές τις συνδέσεις, τους επιστρέφει ως τελικό αποτέλεσμα, αγνοώντας την υπόλοιπη, άσχετη τοπολογία του δικτύου.

2.7.1 ΔΟΜΉ ΚΑΙ ΒΑΣΙΚΆ ΜΟΤΊΒΑ

Ο πυρήνας κάθε ερωτήματος SPARQL είναι το **Βασικό Μοτίβο Γραφήματος (Basic Graph Pattern - BGP)**. Αυτό αποτελείται από ένα σύνολο προτύπων τριπλετών (triple patterns), τα οποία μοιάζουν με τις τριπλέτες RDF (Υποκείμενο - Κατηγόρημα - Αντικείμενο), με τη διαφορά ότι

οποιοδήποτε από τα τρία μέρη μπορεί να αντικατασταθεί από μια μεταβλητή (η οποία συμβολίζεται με το πρόθεμα ?, π.χ. ?museum).

Ένα τυπικό ερώτημα SPARQL δομείται στους εξής βασικούς άξονες:

- **PREFIX:** Αποτελεί τη δήλωση των συντομεύσεων για τα ονόματα χώρου (URIs). Για παράδειγμα, η δήλωση PREFIX dbo: <http://dbpedia.org/ontology/> επιτρέπει τη χρήση του dbo:country αντί για το πλήρες και δυσανάγνωστο URI, διατηρώντας τον κώδικα καθαρό.
- **WHERE:** Αποτελεί τον κορμό του ερωτήματος, όπου ορίζονται τα πρότυπα των τριάδων (BGP) που πρέπει να ταιριάξουν με τα δεδομένα του Knowledge Graph προκειμένου να επιστραφούν αποτελέσματα.
- **FILTER:** Επειδή τα δεδομένα της DBpedia είναι συχνά πολυγλωσσικά και αδόμητα, το σημασιολογικό φιλτράρισμα είναι απαραίτητο. Για παράδειγμα, η εντολή FILTER (LANG(?name) = 'el') χρησιμοποιήθηκε στην παρούσα εργασία για την απομόνωση αποκλειστικά των ελληνικών ονομασιών. Αντίστοιχα, συναρτήσεις όπως το REGEX επιτρέπουν την πολύπλοκη αναζήτηση εντός ελεύθερου κειμένου.
- **OPTIONAL:** Μια από τις ισχυρότερες δομές της SPARQL. Επιτρέπει στην αναζήτηση να προχωρήσει ακόμα κι αν κάποια πληροφορία λείπει, χωρίς να απορρίπτεται ολόκληρο το αποτέλεσμα (προσομοιάζοντας τη λογική του LEFT OUTER JOIN της SQL). Στην περίπτωσή μας, ήταν σωτήρια για την ανάκτηση εικόνων (dbo:thumbnail) ή χαρτών, καθώς αυτά τα στοιχεία δεν υπάρχουν σε όλα τα άρθρα της Wikipedia. Αν δεν γινόταν χρήση του OPTIONAL, ένα άρθρο χωρίς εικόνα θα "απορριπτόταν" τελείως από το σύνολο των αποτελεσμάτων.
- **UNION:** Επιτρέπει τη συνένωση εναλλακτικών μοτίβων γραφήματος, δίνοντας τη δυνατότητα στο ερώτημα να ταιριάξει δεδομένα που μπορεί να βρίσκονται σε διαφορετικές διαδρομές (π.χ. αν ένα νησί συνδέεται με την Ελλάδα είτε μέσω της ιδιότητας dbo:country είτε μέσω της dbo:location).

2.7.2 ΤΥΠΟΙ ΕΡΩΤΗΜΆΤΩΝ

Η αρχιτεκτονική της γλώσσας SPARQL σχεδιάστηκε με γνώμονα τη μέγιστη ευελιξία κατά την αλληλεπίδραση με τα Γραφήματα Γνώσης. Σε αντίθεση με την παραδοσιακή SQL, η οποία είναι πρακτικά εγκλωβισμένη στην επιστροφή δισδιάστατων πινάκων, η SPARQL δεν περιορίζεται αποκλειστικά σε επίπεδες δομές. Προσφέρει τέσσερις θεμελιωδώς διαφορετικούς τύπους ερωτημάτων (Query Forms), έκαστος βελτιστοποιημένος για συγκεκριμένες ανάγκες διαχείρισης, μετασχηματισμού και εξερεύνησης της πληροφορίας:

1. SELECT (Προβολή Αποτελεσμάτων σε Πίνακα)

Είναι ο πλέον διαδεδομένος και ευρέως χρησιμοποιούμενος τύπος ερωτήματος, ειδικά κατά την ανάπτυξη διαδικτυακών εφαρμογών. Η λειτουργία SELECT αναλαμβάνει να εξάγει τις τιμές των μεταβλητών που ικανοποιούν τα μοτίβα γραφήματος (Graph Patterns) και να τις προβάλει (projection) σε μια αυστηρά δομημένη μορφή. Το αποτέλεσμα που επιστρέφεται στο δίκτυο δεν είναι γράφος, αλλά μια "Ακολουθία Λύσεων" (Solution Sequence) σε μορφή πίνακα.

Στα σύγχρονα συστήματα ανάπτυξης (όπως στην παρούσα εφαρμογή), τα αποτελέσματα του SELECT τυποποιούνται συνήθως σε μορφή **JSON (SPARQL Query Results XML/JSON Format)**. Αυτή η δισδιάστατη, επίπεδη δομή είναι ιδανική για να τροφοδοτήσει άμεσα συστήματα διεπαφής (Frontends) όπως η React.js, προκειμένου να κατασκευαστούν πίνακες, λίστες ή σημεία σε έναν γεωγραφικό χάρτη.

2. CONSTRUCT (Δυναμικός Μετασχηματισμός Γραφημάτων)

Η εντολή CONSTRUCT αποτελεί ίσως το ισχυρότερο χαρακτηριστικό του Σημασιολογικού Ιστού. Αντί να επιστρέφει έναν πίνακα με κείμενα, **επιστρέφει έναν ολοκαίνουργιο, έγκυρο γράφο**

RDF. Ο προγραμματιστής ορίζει ένα "πρότυπο γραφήματος" (Graph Template), και η μηχανή παράγει νέες τριάδες (Triples) αντικαθιστώντας τις μεταβλητές με τα αποτελέσματα της αναζήτησης.
Αυτός ο τύπος είναι το απόλυτο εργαλείο για διαδικασίες Εξαγωγής, Μετασχηματισμού και Φόρτωσης (ETL) και για την **Ευθυγράμμιση Οντολογιών (Ontology Alignment)**. Για παράδειγμα, εάν ένα σύστημα χρησιμοποιεί την οντολογία FOAF (Friend of a Friend) για τα πρόσωπα, αλλά η βάση δεδομένων είναι χτισμένη με την οντολογία της DBpedia (dbo:), ένα ερώτημα CONSTRUCT μπορεί να "μεταφράσει" δυναμικά τα δεδομένα εν πτήσει (on-the-fly) από το ένα λεξιλόγιο στο άλλο, εξασφαλίζοντας την πλήρη διαλειτουργικότητα μεταξύ διαφορετικών πληροφοριακών συστημάτων.

3. ASK (Λογική Επαλήθευση και Βελτιστοποίηση Δικτύου)

Η λειτουργία ASK χρησιμοποιείται αυστηρά για λογική αξιολόγηση (Boolean Verification). Δεν επιστρέφει τα δεδομένα που βρέθηκαν, αλλά απαντά αποκλειστικά με μια λογική τιμή (**True ή False**), υποδεικνύοντας εάν το ζητούμενο μοτίβο υπάρχει έστω και μία φορά εντός του Γραφήματος Γνώσης.
Από πλευράς Μηχανικής Λογισμικού, αποτελεί ένα εξαιρετικό εργαλείο βελτιστοποίησης. Αν μια εφαρμογή χρειάζεται απλώς να ελέγξει εάν "Ο Παρθενώνας υπάρχει στη βάση ως Μνημείο", η χρήση του ASK είναι απείρως πιο αποδοτική από το SELECT. Ελαχιστοποιεί τον υπολογιστικό φόρτο του διακομιστή (καθώς ο αλγόριθμος αναζήτησης σταματά στην πρώτη επιτυχή εύρεση) και εκμηδενίζει το δικτυακό κόστος (network bandwidth overhead), επιστρέφοντας ένα ελάχιστο ωφέλιμο φορτίο (payload) λίγων μόνο bytes.

4. DESCRIBE (Σημασιολογική Εξερεύνηση)

Ο τύπος DESCRIBE είναι το κατεξοχήν εργαλείο εξερεύνησης (exploratory query) σε άγνωστα περιβάλλοντα δεδομένων. Όταν ένας προγραμματιστής γνωρίζει το URI μιας οντότητας (π.χ. dbr:Αριστοτέλης), αλλά αγνοεί παντελώς τη δομή, τα κατηγορήματα και το σχήμα (schema) που τη συνοδεύουν, χρησιμοποιεί το DESCRIBE.
Το σύστημα επιστρέφει έναν γράφο RDF που περιγράφει τη συγκεκριμένη οντότητα. Ωστόσο, αξίζει να σημειωθεί ότι ο ακριβής αλγόριθμος για το "τι συνιστά περιγραφή" δεν είναι αυστηρά καθορισμένος από το W3C, αλλά εναπόκειται στην υλοποίηση του εκάστοτε εξυπηρετητή (Virtuoso, Apache Jena κ.ά.). Συνήθως, εφαρμόζεται η μέθοδος της **Συνοπτικής Οριοθετημένης Περιγραφής (Concise Bounded Description - CBD)**, επιστρέφοντας όλες τις εξερχόμενες ακμές της οντότητας και, προαιρετικά, κάποιους κενούς κόμβους (blank nodes) που σχετίζονται άμεσα με αυτήν.
Για την καλύτερη κατανόηση των παραπάνω, ο Ακόλουθος Πίνακας συνοψίζει τα χαρακτηριστικά του κάθε τύπου ερωτήματος.

Συνοπτικός Πίνακας Συγκριτικής Αξιολόγησης

Τύπος Ερωτήματος	Μορφή Επιστρεφόμενου Αποτελέσματος	Κύρια Περίπτωση Χρήσης (Use Case)
SELECT	Πίνακας / Ακολουθία Λύσεων (JSON, CSV, XML)	Τροφοδοσία Διεπαφών (UI), Δημιουργία Λιστών & Αναφορών.
CONSTRUCT	Γράφημα Γνώσης (RDF/XML, Turtle, N-Triples)	Μετασχηματισμός Οντολογιών, Δημιουργία Νέων Υπο-γραφημάτων.
ASK	Λογική Τιμή (Boolean: True / False)	Προ-έλεγχος,Ύπαρξης, Επικύρωση,Μείωση Δικτυακού Φόρτου.
DESCRIBE	Γράφημα Γνώσης (Αλγόριθμος CBD)	Εξερεύνηση άγνωστων Οντοτήτων, Debugging, Ανακάλυψη Σχήματος.

2.7.3 ΤΡΟΠΟΠΟΙΗΤΕΣ ΚΑΙ ΣΥΝΑΡΤΗΣΕΙΣ ΣΥΝΆΘΡΟΙΣΗΣ

Κατά τη διαχείριση Γραφημάτων Γνώσης κολοσσιαίου μεγέθους, όπως αυτό της Ελληνικής DBpedia το οποίο απαριθμεί δεκάδες εκατομμύρια τριάδες, η απλή ανάκτηση δεδομένων δεν επαρκεί. Η εκτέλεση ενός γενικού ερωτήματος χωρίς περιορισμούς θα είχε ως αποτέλεσμα την επιστροφή εκατομμυρίων εγγραφών, γεγονός που θα προκαλούσε ανεπανόρθωτη υπερφόρτωση του δικτύου (network bottleneck) και εξάντληση της διαθέσιμης μνήμης RAM (Out-Of-Memory errors) τόσο στον διακομιστή όσο και στον υπολογιστή του τελικού χρήστη.

Για την αντιμετώπιση αυτού του προβλήματος, το πρότυπο της SPARQL ενσωματώνει τους **Τροποποιητές Ακολουθίας Λύσεων (Solution Sequence Modifiers)**. Αυτοί οι τροποποιητές εφαρμόζονται στο τελικό σύνολο των αποτελεσμάτων, λίγο πριν αυτά αποσταλούν στον πελάτη, και περιλαμβάνουν τις εξής κομβικές εντολές:

- **ORDER BY:** Αναλαμβάνει την ταξινόμηση των αποτελεσμάτων βάσει μίας ή περισσότερων μεταβλητών, είτε σε αύξουσα (ASC) είτε σε φθίνουσα (DESC) σειρά. Η ταξινόμηση μπορεί να είναι αλφαβητική, αριθμητική ή χρονολογική. Από αλγοριθμική σκοπιά, το ORDER BY εισάγει ένα υπολογιστικό κόστος $O(n \log n)$, καθιστώντας την ύπαρξη των ευρετηρίων (B-Trees) που συζητήθηκαν σε προηγούμενα κεφάλαια, απολύτως κρίσιμη για την ταχύτητα του συστήματος.
- **LIMIT και OFFSET:** Η συνδυαστική χρήση αυτών των δύο εντολών αποτελεί τον ακρογωνιαίο λίθο για την υλοποίηση της **Σελιδοποίησης (Pagination)** στις σύγχρονες διαδικτυακές εφαρμογές. Η εντολή OFFSET καθορίζει το πλήθος των αρχικών αποτελεσμάτων που πρέπει να παρακαμφθούν, ενώ η εντολή LIMIT θέτει το αυστηρό άνω όριο στο πλήθος των εγγραφών που θα επιστραφούν (π.χ., το "παράθυρο" των επόμενων 10 αποτελεσμάτων). Με αυτόν τον τρόπο, η εφαρμογή "σπάει" την τεράστια πληροφορία σε μικρά, διαχειρίσιμα τμήματα δεδομένων (chunks), προστατεύοντας τον διακομιστή Virtuoso από επιθέσεις ή τυχαία υπερφόρτωση.

Παράλληλα, με την έλευση του προτύπου **SPARQL 1.1**, η γλώσσα μετεξελίχθηκε από ένα απλό εργαλείο ανάκτησης σε ένα ισχυρό σύστημα Αναλυτικής Επεξεργασίας (Online Analytical Processing - OLAP). Εισήχθησαν οι **Συναρτήσεις Συνάθροισης (Aggregates)**, οι οποίες επιτρέπουν την εκτέλεση μαθηματικών και στατιστικών υπολογισμών απευθείας στον κινητήρα της βάσης δεδομένων, αντί να απαιτείται η μεταφορά των δεδομένων και ο υπολογισμός τους στο επίπεδο της εφαρμογής (Backend/Frontend).

Στις συναρτήσεις αυτές περιλαμβάνονται οι SUM (άθροισμα), MIN (ελάχιστο), MAX (μέγιστο) και AVG (μέσος όρος), οι οποίες λειτουργούν συνδυαστικά με τον τελεστή ομαδοποίησης **GROUP BY**, επιτρέποντας τη συμπύκνωση πολυδιάστατων δεδομένων σε συνοπτικές αναφορές. Ιδιαίτερης σημασίας για την παρούσα διπλωματική εργασία είναι η συνάρτηση **COUNT**, η οποία καταμετρά το πλήθος των εγγραφών που ικανοποιούν ένα μοτίβο. Μάλιστα, η χρήση της παραλλαγής COUNT(DISTINCT ?var) είναι ζωτικής σημασίας στα περιβάλλοντα γραφημάτων, προκειμένου να αποφεύγονται οι διπλοεγγραφές που προκύπτουν από το Καρτεσιανό γινόμενο των πολλαπλών σχέσεων (π.χ. όταν μια οντότητα ανήκει σε πολλές κατηγορίες ταυτόχρονα).

Όπως θα αναλυθεί διεξοδικά στα κεφάλαια της υλοποίησης, ο δυναμικός υπολογισμός του COUNT αποτέλεσε δομικό συστατικό της αρχιτεκτονικής του συστήματος, εξυπηρετώντας τον ασύγχρονο υπολογισμό του συνολικού όγκου δεδομένων (Total Results) στο παρασκήνιο, στοιχείο απαραίτητο για τη σωστή λειτουργία των αλγορίθμων σελιδοποίησης.

Πάνω σε αυτό το θεωρητικό, μαθηματικό και αρχιτεκτονικό υπόβαθρο, η παρούσα διπλωματική εργασία απομακρύνεται από τη σφαίρα της θεωρίας και εστιάζει στην πράξη. Στα επόμενα κεφάλαια θα αναλυθεί ο τρόπος με τον οποίο αυτά τα δομικά στοιχεία ενορχηστρώθηκαν μηχανικά, προκειμένου να χτιστεί ένα σύγχρονο, γρήγορο, ασφαλές και οπτικά διαδραστικό επίπεδο εφαρμογής: ένας πλήρης Φυλλομετρητής Σημασιολογικού Ιστού (Semantic Web Browser) που καθιστά την περίπλοκη γνώση της μηχανής προσβάσιμη στον τελικό χρήστη.

ΚΕΦΑΛΑΙΟ 3: ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΚΑΙ ΤΕΧΝΟΛΟΓΙΕΣ ΣΥΣΤΗΜΑΤΟΣ

3.1 ΕΙΣΑΓΩΓΗ ΚΕΦΑΛΑΙΟΥ

Μια σύγχρονη εφαρμογή του Σημασιολογικής Εφαρμογής Ιστού (Semantic Web Application) απαιτεί τον προσεκτικό σχεδιασμό της αρχιτεκτονικής της και τη σωστή επιλογή με τα κατάλληλα τεχνολογικά εργαλεία. Στο παρόν κεφάλαιο αναλύεται η αρχιτεκτονική των τριών επιπέδων (3-Tier Architecture) που υιοθετήθηκε για τον διαχωρισμό των αρμοδιοτήτων του συστήματος, καθώς και οι επιμέρους τεχνολογίες που χρησιμοποιήθηκαν. ο Virtuoso Universal Server για τη διαχείριση των δεδομένων, η γλώσσα Golang για την ανάπτυξη της αρχιτεκτονική του backend επιπέδου (Backend) API επικοινωνίας και η βιβλιοθήκη React.js για τη διεπαφή χρήστη (Frontend), όλα με την διαμόρφωση ρυθμίσεων και ανάπτυξης αυτών (configuration και deployment) με τη τεχνολογία Docker .

.

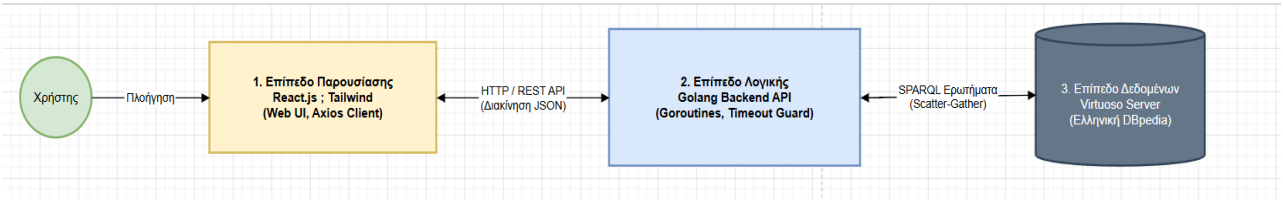
3.2 ΑΡΧΙΤΕΚΤΟΝΙΚΉ ΤΡΙΏΝ ΕΠΙΠΈΔΩΝ (3-TIER ARCHITECTURE)

Η δομή της εφαρμογής σχεδιάστηκε και αναπτύχθηκε βασιζόμενη στο πρότυπο της Αρχιτεκτονικής Τριών Επιπέδων. Το συγκεκριμένο αρχιτεκτονικό μοτίβο αποτελεί θεμελιώδη και καθιερωμένη πρακτική στον τομέα της Μηχανικής Λογισμικού (Software Engineering) για τη δημιουργία κατανεμημένων εφαρμογών πελάτη-εξυπηρετητή (client-server). Η βασική φιλοσοφία του προτύπου έγκειται στην εννοιολογική και φυσική αποσύνδεση (decoupling) των λειτουργικών υποσυστημάτων.

Αυτή η στρατηγική απομόνωση προσφέρει πολλαπλά πλεονεκτήματα: διευκολύνει την ανεξάρτητη κλιμάκωση (scalability) κάθε επιπέδου ανάλογα με τον υπολογιστικό φόρτο, απλοποιεί τη διαδικασία εντοπισμού και διόρθωσης σφαλμάτων (debugging), και εξασφαλίζει υψηλή συντηρησιμότητα (maintainability). Επιπλέον, εξασφαλίζει την ευελιξία του συστήματος, καθώς επιτρέπει τη μελλοντική αντικατάσταση ή αναβάθμιση μιας τεχνολογίας σε ένα επίπεδο (π.χ. αλλαγή της τεχνολογίας του Frontend) χωρίς να επηρεάζεται η ομαλή λειτουργία των υπολοίπων.

Η αρχιτεκτονική της παρούσας εφαρμογής δομείται αυστηρά στα εξής τρία διακριτά επίπεδα:

- **Επίπεδο Δεδομένων (Data Tier):** Αποτελεί το θεμέλιο του συστήματος. Είναι υπεύθυνο για τη φυσική αποθήκευση, τη διαχείριση και την ταχύτατη ανάκτηση των πληροφοριών. Στην προκειμένη περίπτωση, το επίπεδο αυτό φιλοξενεί το Γράφημα Γνώσης (Knowledge Graph) της Ελληνικής DBpedia και τον μηχανισμό εκτέλεσης ερωτημάτων SPARQL.
- **Επίπεδο Λογικής / Εφαρμογής (Application / Middle Tier):** Αποτελεί τον «εγκέφαλο» του συστήματος και λειτουργεί ως διαμεσολαβητής (proxy) μεταξύ του χρήστη και της βάσης δεδομένων. Σε αυτό το επίπεδο υλοποιείται η επιχειρησιακή λογική (business logic), η διαχείριση, ανάλυση και επικύρωση των εισερχόμενων ερωτημάτων (request handling), η κεντρική διαχείριση σφαλμάτων, καθώς και οι πολύπλοκοι μηχανισμοί ταυτοχρονίας (concurrency) για τη βελτιστοποίηση της απόδοσης.
- **Επίπεδο Παρουσίασης (Presentation Tier):** Είναι το ορατό τμήμα της εφαρμογής, το οποίο αναλαμβάνει την αλληλεπίδραση με τον τελικό χρήστη. Υλοποιείται αποκλειστικά στο περιβάλλον του φυλλομετρητή (web browser), προσφέροντας μια δυναμική, αποκρίσιμη (responsive) και γραφικά αναβαθμισμένη διεπαφή για την υποβολή ερωτημάτων και την οπτικοποίηση των σημασιολογικών αποτελεσμάτων.



Εικόνα 4 Διάγραμμα ροής συστήμτος , Πηγή:Δημιουργία του συγγραφέα

3.3 ΕΠΙΠΕΔΟ ΔΕΔΟΜΈΝΩΝ: VIRTUOSO UNIVERSAL SERVER

Ο Virtuoso Universal Server, ο οποίος αναπτύχθηκε από την εταιρεία OpenLink Software, αποτελεί τον πυρήνα του επιπέδου δεδομένων (Data Layer) της παρούσας αρχιτεκτονικής. Στο πλαίσιο του συστήματος, αναλαμβάνει τον κρίσιμο ρόλο του διακομιστή τριπλετών (Triple Store), όντας υπεύθυνος για την αποθήκευση, ευρετηρίαση και αποτελεσματική διαχείριση των εκατομμυρίων σημασιολογικών τριπλετών που συγκροτούν το γράφημα γνώσης της Ελληνικής DBpedia.

Η αρχιτεκτονική καινοτομία του Virtuoso έγκειται στην πολυμοντελική (multi-model) φύση του, καθώς αποτελεί έναν από τους πρώτους διακομιστές που υιοθέτησαν τη φιλοσοφία της "Καθολικής Πλατφόρμας" (Cross-Platform Universal Server). Αντί να περιορίζεται στον παραδοσιακό ρόλο ενός Συστήματος Διαχείρισης Βάσεων Δεδομένων (DBMS), συνδυάζει οργανικά ετερογενείς λειτουργίες σε μία ενιαία λύση διακομιστή:

- **Διαχείριση Δεδομένων:** Υποστηρίζει ταυτόχρονα το σχεσιακό μοντέλο (RDBMS), την εγγενή αποθήκευση εγγράφων XML (Native XML Storage) και την αυτόχθονη αποθήκευση δεδομένων RDF (Native Triple Store).
- **Υπηρεσίες Δικτύου:** Ενσωματώνει πλήρη λειτουργικότητα διακομιστή Ιστού (Web Server) και συστήματος αρχείων (File Server), επιτρέποντας την απευθείας φιλοξενία και διανομή πόρων.

Το σύστημα διακρίνεται για την υψηλή διαλειτουργικότητά του, υποστηρίζοντας ένα ευρύ φάσμα σύγχρονων πρωτοκόλλων Διαδικτύου και πρόσβασης δεδομένων, όπως XML, XPATH, XSLT, SOAP, WSDL, WebDAV, καθώς και τα καθιερωμένα πρότυπα επερωτήσεων SQL και SPARQL. Παράλληλα, εξασφαλίζει πλήρη ανεξαρτησία πλατφόρμας (platform independence), καθώς υποστηρίζεται από τα κυριότερα σύγχρονα λειτουργικά συστήματα (εκδόσεις Windows, διανομές Linux, macOS, Unix-based συστήματα κ.ά.).

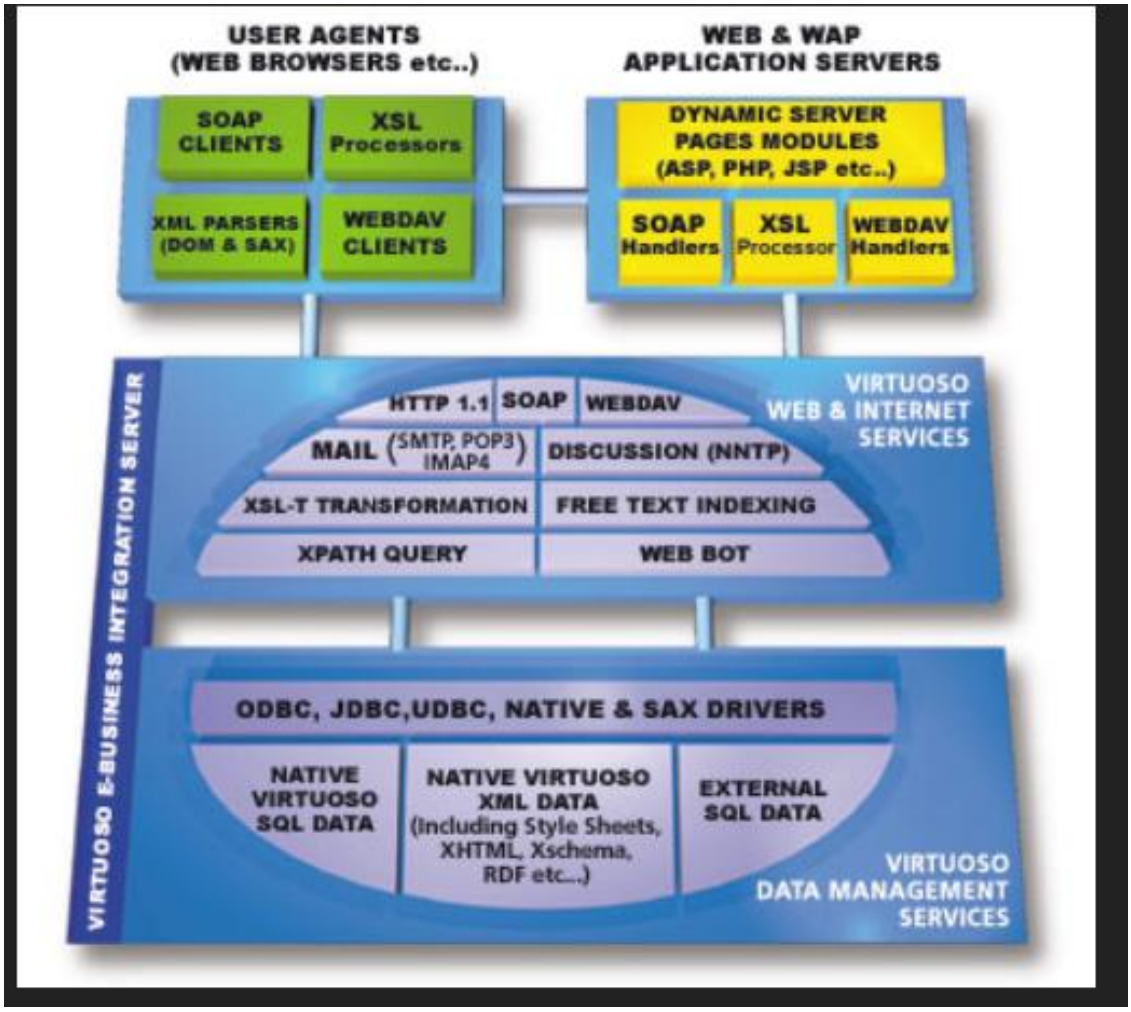
Στην εποχή του κατανεμημένου υπολογισμού, ένα από τα ισχυρότερα χαρακτηριστικά του Virtuoso είναι η λειτουργία του ως Μηχανή Εικονικών Βάσεων Δεδομένων υψηλής απόδοσης. Το σύστημα ενσωματώνει ενδιάμεσο λογισμικό καθολικής πρόσβασης (Universal Data Access Middleware), το οποίο παρέχει διαφανή (transparent) πρόσβαση σε προϋπάρχουσες, ετερογενείς πηγές δεδομένων.

Μέσω μίας μόνο κεντρικής σύνδεσης, ο Virtuoso δύναται να διασυνδέσει ταυτόχρονα τις εφαρμογές-πελάτες με εξωτερικές, παραδοσιακές μηχανές βάσεων δεδομένων (όπως Oracle, Microsoft SQL Server, IBM DB2, Informix κ.ά.) χρησιμοποιώντας καθιερωμένα πρωτόκολλα όπως ODBC, JDBC και OLE-DB. Με αυτόν τον τρόπο, ενοποιεί τα κατανεμημένα δεδομένα και επιτρέπει στο σύστημα να τα

αντιμετωπίζει και να τα επερωτά ως *μία ενιαία λογική μονάδα*, χωρίς να απαιτείται η φυσική μετακίνησή τους (data replication).

Πέρα από την αποθήκευση και την ομοσπονδιακή διαχείριση, το σύστημα εκθέτει τις λειτουργίες του δυναμικά μέσω Υπηρεσιών Ιστού (Web Services) και τελικών σημείων (SPARQL Endpoints). Αυτή η αρχιτεκτονική επιλογή επιτρέπει την πλήρη αξιοποίηση της υπάρχουσας πληροφοριακής υποδομής ενός οργανισμού. Τα δεδομένα καθίστανται άμεσα διαθέσιμα για κατανάλωση από σύγχρονες εφαρμογές ιστού (όπως η παρούσα διεπαφή σε React.js) απευθείας, χωρίς καμία απολύτως ανάγκη υποκατάστασης ή απόσυρσης των παλαιότερων υποστηρικτικών συστημάτων (legacy systems).

Συνοψίζοντας, ο Virtuoso δεν δρα απλώς ως ένας αποθηκευτικός χώρος, αλλά ως μια κεντρική γέφυρα διαλειτουργικότητας, η οποία ενοποιεί τον παραδοσιακό Ιστό των Εγγράφων με τον σύγχρονο Ιστό των Δεδομένων.



Εικόνα 5 Αρχιτεκτονική δομή του Virtuoso Universal Server. Πηγή: Προσαρμογή από την επίσημη τεκμηρίωση της OpenLink Software (OpenLink Virtuoso Product Documentation, Κεφάλαιο 5.5).

Από την παραπάνω ανάλυση καθίσταται σαφές ότι ο Virtuoso δεν αποτελεί μια απλή Σχεσιακή Βάση Δεδομένων (RDBMS), αλλά ένα προηγμένο, υβριδικό Πολυμοντελικό Σύστημα Βάσης Δεδομένων (Multi-model Database System). Η αρχιτεκτονική αυτή του παρέχει τη μοναδική ικανότητα να διαχειρίζεται ταυτόχρονα και αποδοτικά γραμμικά σχεσιακά δεδομένα (SQL), ιεραρχικά έγγραφα

(XML), καθώς και δεδομένα με μορφή δικτύου/γράφου (RDF) [8]. Αυτή η ευελιξία τον καθιστά την ιδανική υποδομή για εφαρμογές του Σημασιολογικού Ιστού.

Ειδικότερα για τις ανάγκες της παρούσας πτυχιακής εργασίας, η επιλογή του Virtuoso έναντι ανταγωνιστικών λύσεων (όπως το Apache Jena ή το GraphDB) βασίστηκε στα ακόλουθα συγκριτικά πλεονεκτήματα:

- **Εγγενής Υποστήριξη RDF και SPARQL (Native Support):**

Ο Virtuoso αποτελεί το αδιαμφισβήτητο βιομηχανικό πρότυπο (industry standard) για τη φιλοξενία ανοικτών σημασιολογικών δεδομένων μεγάλης κλίμακας. Δεν είναι τυχαίο ότι το ίδιο ακριβώς λογισμικό έχει επιλεγεί από το W3C και την κοινότητα των Linked Open Data για τη φιλοξενία του κεντρικού, παγκόσμιου κόμβου της DBpedia, διασφαλίζοντας την απόλυτη συμβατότητα με τα δεδομένα που χρησιμοποιήθηκαν.

- **Υψηλές Επιδόσεις και Ευρετηρίαση (Performance & Indexing):**

Ο διακομιστής διαθέτει εξαιρετικά βελτιστοποιημένες δομές μνήμης και εξειδικευμένες μεθόδους ευρετηρίασης (indexing) για σημασιολογικές τριπλέτες (Υποκείμενο, Κατηγόρημα, Αντικείμενο – Subject, Predicate, Object). Η δομή αυτή επιτρέπει την ταχύτατη επεξεργασία του μηχανισμού Αντιστοίχισης Προτύπων Γραφήματος (Graph Pattern Matching), καθιστώντας δυνατή την εκτέλεση εξαιρετικά πολύπλοκων ερωτημάτων (όπως κανονικές εκφράσεις και φιλτράρισμα κειμένου) σε δισεκατομμύρια εγγραφές, εντός κλασμάτων του δευτερολέπτου [17].

- **Μαζική Φόρτωση Δεδομένων (Bulk Loading):**

Η κατασκευή ενός τοπικού σημασιολογικού κόμβου (local endpoint) απαιτεί την εισαγωγή terabytes δεδομένων. Ο Virtuoso παρέχει ισχυρά ενσωματωμένα εργαλεία μαζικής φόρτωσης. Αξιοποιώντας προγραμματιστικές διεπαφές και εντολές του συστήματος (όπως η συνάρτηση ld_dir() και ο εκτελεστής rdf_loader_run()), κατέστη δυνατή η αυτοματοποιημένη, παράλληλη και ταχύτατη ανάλυση (parsing) και εισαγωγή εκατομμυρίων τριπλετών από μεγάλα συμπιεσμένα αρχεία μορφής N-Triples (.nt) και Turtle (.ttl), δομώντας τον πυρήνα της βάσης της εφαρμογής.

3.4 ΕΠΙΠΕΔΟ ΛΟΓΙΚΗΣ ΕΦΑΡΜΟΓΗΣ: GOLANG (GO)

Το ενδιάμεσο επίπεδο (Application / Middle Tier) αποτελεί τον "εγκέφαλο" της εφαρμογής, καθώς γεφυρώνει τη Διεπαφή Χρήστη (React.js) με το Επίπεδο Δεδομένων (Virtuoso). Για την ανάπτυξη του Διεπαφής Προγραμματισμού Εφαρμογών (RESTful API), επιλέχθηκε εξ ολοκλήρου η γλώσσα προγραμματισμού **Go (Golang)**. Η Go, η οποία αναπτύχθηκε στα εργαστήρια της Google, αποτελεί μια μεταγλωττισμένη (compiled), στατικά τυποποιημένη (statically typed) γλώσσα, η οποία σχεδιάστηκε ειδικά για την επίλυση προβλημάτων σε κατανεμημένα συστήματα μεγάλης κλίμακας.

Η επιλογή της Go έναντι άλλων δημοφιλών γλωσσών (όπως η Python, η Node.js ή η Java) τεκμηριώνεται από το γεγονός ότι προσφέρει την ωμή υπολογιστική ταχύτητα των γλωσσών χαμηλού επιπέδου (όπως η C/C++), διατηρώντας ταυτόχρονα την ευκολία σύνταξης και τη διαχείριση μνήμης (μέσω Garbage Collector) των γλωσσών υψηλού επιπέδου [7]. Οι κύριοι άξονες που κατέστησαν την Go ιδανική για το παρόν σύστημα Σημασιολογικού Ιστού αναλύονται παρακάτω:

- **Ταυτοχρονισμός (Concurrency) και Ασύγχρονη Εκτέλεση (Goroutines):**

Το μεγαλύτερο αρχιτεκτονικό πλεονέκτημα της Go είναι το εγγενές, πανίσχυρο μοντέλο ταυτοχρονίας της. Σε παραδοσιακά περιβάλλοντα (όπως η Java ή η PHP), κάθε ταυτόχρονο αίτημα δημιουργεί ένα νέο νήμα λειτουργικού συστήματος (OS thread), το οποίο δεσμεύει 1 έως 2 MB μνήμης RAM, οδηγώντας γρήγορα σε εξάντληση των πόρων του διακομιστή (bottleneck). Αντιθέτως, η Go εισάγει τα **"Goroutines"**, τα οποία αποτελούν ελαφρά εικονικά νήματα που διαχειρίζεται ο εσωτερικός

χρονοπρογραμματιστής της γλώσσας (M:N scheduler) και απαιτούν μόλις ~2KB αρχικής μνήμης. Στο τρέχον σύστημα, αυτή η τεχνολογία αποτέλεσε τον πυρήνα για την υλοποίηση του μοτίβου **Scatter-Gather**. Μέσω της δομής συγχρονισμού sync.WaitGroup, το API μπορεί να διασπά ένα πολύπλοκο αίτημα (π.χ. τον υπολογισμό των συνολικών νησιών, μουσείων και προσώπων στο τελικό σημείο /stats) και να εκτελεί πολλαπλά ερωτήματα SPARQL ταυτόχρονα. Το API "περιμένει" την ολοκλήρωση όλων των νημάτων πριν συνθέσει την τελική απάντηση, μειώνοντας δραματικά τον συνολικό χρόνο απόκρισης σε σχέση με τη σειριακή εκτέλεση.

- **Αρχιτεκτονική Αντίστροφου Διαμεσολαβητή (Reverse Proxying) και Εγγενές HTTP:**

Η πρότυπη βιβλιοθήκη δικτύου της Go (net/http) είναι τόσο ισχυρή και βελτιστοποιημένη, που καθιστά εντελώς περιττή τη χρήση βαριών εξωτερικών πλαισίων (frameworks) για τη δημιουργία διακομιστών Web. Στην παρούσα υποδομή, η Golang υλοποιεί τον ρόλο του **Αντίστροφου Διαμεσολαβητή**. Το Frontend (React) δεν επικοινωνεί ποτέ απευθείας με τη βάση δεδομένων. Αντίθετα, υποβάλλει το αίτημα στο Go Backend. Η Go παραλαμβάνει το αίτημα, ελέγχει την εγκυρότητά του, το "μεταφράζει" και το προωθεί στον διακομιστή Virtuoso (ο οποίος λειτουργεί σε απομονωμένη θύρα). Μόλις ο Virtuoso επιστρέψει τα ακατέργαστα δεδομένα (τα οποία συχνά αγγίζουν αρκετά Megabytes σε μορφή JSON), η Go αναλαμβάνει την ταχύτατη αποσειριοποίηση (deserialization), το φιλτράρισμα και την προώθησή τους πίσω στον πελάτη, λειτουργώντας ως "ασπίδα" για τη βάση δεδομένων.

- **Επίλυση Προβλημάτων Ασφαλείας (CORS) και Έλεγχος Πρόσβασης:**

Οι σύγχρονοι φυλλομετρητές ιστού (Web Browsers) εφαρμόζουν αυστηρές πολιτικές ασφαλείας. Μία από τις κυριότερες είναι η Πολιτική Προέλευσης (Same-Origin Policy), η οποία εμποδίζει μια εφαρμογή (π.χ. το React UI που εκτελείται στη θύρα 3000) να αντλήσει δεδομένα από μια υπηρεσία που βρίσκεται σε διαφορετική θύρα ή τομέα (π.χ. το API στη θύρα 8080). Το Backend στην Go επιλύει αυτό το ζήτημα διαχειριζόμενο κεντρικά την **Κοινή Χρήση Πόρων Διαφορετικής Προέλευσης (CORS - Cross-Origin Resource Sharing)**. Συγκεκριμένα, το API αναχαιτίζει τα προπαρασκευαστικά αιτήματα των φυλλομετρητών (HTTP OPTIONS preflight requests) και εισάγει δυναμικά τις απαραίτητες κεφαλίδες ελέγχου (όπως το Access-Control-Allow-Origin και Access-Control-Allow-Methods), επιτρέποντας την ασφαλή και ελεγχόμενη ροή των πληροφοριών.

- **Ρητή Διαχείριση Σφαλμάτων (Explicit Error Handling):**

Σε αντίθεση με γλώσσες που βασίζονται στο μοντέλο σύλληψης εξαιρέσεων (try/catch), η Go αντιμετωπίζει τα σφάλματα ως απλές επιστρεφόμενες τιμές. Αυτή η φιλοσοφία επιβάλλει στον προγραμματιστή να ελέγχει ρητά κάθε πιθανό σημείο αστοχίας (π.χ. αποτυχία σύνδεσης δικτύου, συντακτικό λάθος στη SPARQL, εξάντληση ορίου χρόνου). Στην παρούσα εφαρμογή, αυτό το χαρακτηριστικό αξιοποιήθηκε για την κατασκευή λεπτομερών HTTP απαντήσεων σφάλματος (HTTP Status Codes 400 ή 500), τροφοδοτώντας το Frontend με ακριβή διαγνωστικά μηνύματα, τα οποία στη συνέχεια παρουσιάζονται φιλικά στον τελικό χρήστη.

Μέσα από τον συνδυασμό αυτών των τεχνολογιών, η Golang διασφαλίζει ότι το σύστημα δεν αποτελεί απλώς έναν αγωγό μεταφοράς δεδομένων, αλλά έναν εύρωστο, ασφαλή και κλιμακώσιμο (scalable) κόμβο διαχείρισης σημασιολογικής πληροφορίας.

3.5 ΕΠΙΠΕΔΟ ΠΑΡΟΥΣΙΑΣΗΣ: REACT.JS & TAILWIND CSS

Το Επίπεδο Παρουσίασης (Presentation Layer) αποτελεί το τελικό σημείο αλληλεπίδρασης μεταξύ του συστήματος και του τελικού χρήστη. Για την υλοποίηση της Διεπαφής Χρήστη (User Interface - UI), υιοθετήθηκε η αρχιτεκτονική της Εφαρμογής Μίας Σελίδας (Single Page Application - SPA). Σε αυτό

το μοντέλο, η εφαρμογή φορτώνει αρχικά ένα μοναδικό έγγραφο HTML και, στη συνέχεια, ανανεώνει δυναμικά μόνο τα τμήματα της σελίδας που απαιτούν αλλαγή, προσφέροντας μια εξαιρετικά ομαλή εμπειρία πλοήγησης που προσομοιάζει σε εφαρμογή επιφάνειας εργασίας (desktop application).

Η ανάπτυξη του επιπέδου παρουσίασης βασίστηκε στη βιβλιοθήκη **React.js** (αναπτυγμένη από τη Meta), υποστηριζόμενη από επιμέρους εργαλεία για τη διαχείριση της επικοινωνίας και της αισθητικής. Η επιλογή της συγκεκριμένης στοίβας τεχνολογιών (tech stack) τεκμηριώνεται στους ακόλουθους άξονες:

- **Αρχιτεκτονική Βάσει Στοιχείων (Component-Based Architecture):** Το πλαίσιο του React επιτρέπει τη διάσπαση της διεπαφής σε μικρά, αυτόνομα και επαναχρησιμοποιήσιμα προγραμματιστικά τμήματα, γνωστά ως "Συστατικά" (Components). Στην παρούσα εφαρμογή, στοιχεία όπως ο επεξεργαστής ερωτημάτων (QueryEditor), ο πίνακας αποτελεσμάτων (ResultsTable) και ο πίνακας στατιστικών (StatsDashboard) αναπτύχθηκαν ως διακριτά συστατικά. Αυτή η αρθρωτή προσέγγιση (modularity) εξασφαλίζει την ενθυλάκωση (encapsulation) της λογικής, διευκολύνει τη συντήρηση του κώδικα και επιτρέπει την εύκολη μελλοντική επέκταση του συστήματος [16].
- **Διαχείριση Απόδοσης μέσω Εικονικού DOM (Virtual DOM):** Κατά την εκτέλεση ερωτημάτων SPARQL, ο διακομιστής συχνά επιστρέφει χιλιάδες εγγραφές (RDF bindings) σε μορφή JSON. Η άμεση και συνεχής αλλοίωση του πραγματικού DOM (Document Object Model) του φυλλομετρητή για την απόδοση αυτών των δεδομένων αποτελεί μια εξαιρετικά δαπανηρή υπολογιστικά διαδικασία, η οποία οδηγεί σε "πάγωμα" (blocking) της διεπαφής. Το React επιλύει αυτό το πρόβλημα διατηρώντας ένα ελαφρύ αντίγραφο του DOM στη μνήμη (Virtual DOM). Μέσω ενός εξειδικευμένου αλγορίθμου συμφιλίωσης (reconciliation algorithm), υπολογίζει τη βέλτιστη διαδρομή και εφαρμόζει στο πραγματικό DOM μόνο τις απολύτως απαραίτητες αλλαγές, διασφαλίζοντας ταχύτατη απόδοση (rendering) ακόμη και κάτω από βαρύ φορτίο δεδομένων.
- **Ασύγχρονη Επικοινωνία (Axios HTTP Client):** Για τη διασύνδεση της διεπαφής με το ενδιάμεσο λογισμικό (Golang Backend), επιλέχθηκε η βιβλιοθήκη Axios. Πρόκειται για έναν HTTP πελάτη (client) που βασίζεται σε Υποσχέσεις (Promises) και υποστηρίζει πλήρως την ασύγχρονη εκτέλεση (async/await). Η χρήση του Axios κρίθηκε επιβεβλημένη, καθώς παρέχει εγγενείς μηχανισμούς διαχείρισης σφαλμάτων (error handlers) και χρόνων λήξης (timeouts). Αυτό το χαρακτηριστικό αξιοποιήθηκε κεντρικά για την υλοποίηση της "Ανατροφοδότησης Σφαλμάτων Μεταγλωττιστή": εάν το ερώτημα SPARQL περιέχει συντακτικό λάθος ή υπερβεί τον επιτρεπτό χρόνο εκτέλεσης (όπως αναλύθηκε στον μηχανισμό Timeout Guard), το Axios συλλαμβάνει την εξαίρεση και η React ενημερώνει δυναμικά το UI, προστατεύοντας την εφαρμογή από πιθανή κατάρρευση (crash).
- **Σχεδιασμός και Εμπειρία Χρήστη (Tailwind CSS):** Για τη μορφοποίηση και την αισθητική απόδοση της εφαρμογής, απορρίφθηκε η χρήση παραδοσιακών αρχείων CSS υπέρ του πλαισίου **Tailwind CSS**. Το Tailwind ακολουθεί τη φιλοσοφία των "κλάσεων χρησιμότητας" (utility-first CSS), επιτρέποντας τη σύνθεση της εμφάνισης απευθείας εντός του κώδικα HTML/JSX. Η προσέγγιση αυτή εξαλείφει το πρόβλημα του αδρανούς κώδικα (CSS bloat), αποτρέπει τις συγκρούσεις ονοματοδοσίας κλάσεων και εξασφαλίζει ομοιομορφία. Επιπλέον, διευκόλυνε την ταχεία υλοποίηση ενός πλήρως αποκρίσιμου σχεδιασμού (responsive design), εγγυώμενο ότι η διεπαφή συμμορφώνεται με τα σύγχρονα πρότυπα Εμπειρίας Χρήστη (UX), ανεξαρτήτως της συσκευής πρόσβασης του τελικού χρήστη.

Μέσω του συνδυασμού αυτών των τεχνολογιών, το Επίπεδο Παρουσίασης μετατρέπει την ακατέργαστη, σημασιολογική πληροφορία της βάσης δεδομένων σε ένα δυναμικό, ασφαλές και φιλικό προς τον χρήστη περιβάλλον οπτικοποίησης.

3.6 ΥΠΟΔΟΜΉ ΚΑΙ ΕΝΘΥΛΆΚΩΣΗ (DOCKER CONTAINERS)

Στη σύγχρονη Μηχανική Λογισμικού, μία από τις κυριότερες προκλήσεις κατά τη φάση της ανάπτυξης και της ανάπτυξης (deployment) είναι η διασφάλιση της ομοιομορφίας του περιβάλλοντος εκτέλεσης. Το παραδοσιακό πρόβλημα «λειτουργεί στον υπολογιστή μου» (it works on my machine) προκύπτει από ασυμβατότητες μεταξύ λειτουργικών συστημάτων, διαφορετικών εκδόσεων βιβλιοθηκών ή ρυθμίσεων συστήματος. Για την οριστική επίλυση αυτών των ζητημάτων, ολόκληρη η αρχιτεκτονική της παρούσας εφαρμογής (Virtuoso, Go Backend, React Frontend) υλοποιήθηκε με τη χρήση της τεχνολογίας ενθυλάκωσης **Docker** [15].

3.6.1 Η ΦΙΛΟΣΟΦΊΑ ΤΗΣ ΕΝΘΥΛΆΚΩΣΗΣ ΈΝΑΝΤΙ ΤΗΣ ΕΙΚΟΝΙΚΟΠΟΊΗΣΗΣ

Σε αντίθεση με τις παραδοσιακές Εικονικές Μηχανές (Virtual Machines), οι οποίες απαιτούν την εγκατάσταση ενός πλήρους λειτουργικού συστήματος (Guest OS) πάνω από έναν Hypervisor, το Docker αξιοποιεί την εικονικοποίηση σε επίπεδο λειτουργικού συστήματος (OS-level virtualization). Τα κοντέινερ (containers) μοιράζονται τον πυρήνα (kernel) του λειτουργικού συστήματος του ξενιστή, γεγονός που τα καθιστά εξαιρετικά ελαφριά, με ταχύτερους χρόνους εκκίνησης και ελάχιστες απαιτήσεις σε υπολογιστικούς πόρους. Κάθε επίπεδο της εφαρμογής μας απομονώνεται σε ένα δικό του κοντέινερ, το οποίο περιέχει μόνο τις απαραίτητες εξαρτήσεις για τη λειτουργία του, διασφαλίζοντας ότι η εφαρμογή θα συμπεριφέρεται με τον ίδιο ακριβώς τρόπο σε οποιοδήποτε περιβάλλον (τοπικό υπολογιστή, διακομιστή ή νέφος).

3.6.2 ΑΝΆΛΥΣΗ ΤΗΣ ΣΤΟΊΒΑΣ DOCKER COMPOSE

Για την ενορχήστρωση και τον συντονισμό των επιμέρους υπηρεσιών, χρησιμοποιήθηκε το εργαλείο Docker Compose. Μέσω ενός κεντρικού αρχείου διαμόρφωσης (docker-compose.yml), ορίστηκαν οι προδιαγραφές για τα τρία βασικά επίπεδα:

- **Επίπεδο Δεδομένων (Virtuoso Service):** Χρησιμοποιήθηκε η επίσημη εικόνα του Virtuoso Universal Server. Λόγω της φύσης της DBpedia (μεγάλος όγκος δεδομένων), η διαχείριση της πληροφορίας δεν γίνεται εντός του κοντέινερ, αλλά μέσω Μόνιμων Τόμων (Docker Volumes). Οι τόμοι αυτοί αντιστοιχίζουν τον εσωτερικό κατάλογο της βάσης δεδομένων με έναν κατάλογο στον υπολογιστή του ξενιστή, διασφαλίζοντας ότι τα δεδομένα παραμένουν άθικτα ακόμη και αν το κοντέινερ σταματήσει ή διαγραφεί.
- **Επίπεδο Λογικής (Golang Service):** Για το Backend, υλοποιήθηκε μια διαδικασία Μεταγλώττισης Πολλαπλών Σταδίων (Multi-stage Build). Στο πρώτο στάδιο, ο κώδικας Go μεταγλωττίζεται σε ένα πλήρες περιβάλλον ανάπτυξης, ενώ στο τελικό στάδιο, μόνο το παραχθέν στατικό εκτελέσιμο αρχείο (binary) μεταφέρεται σε μια εξαιρετικά μικρή εικόνα (όπως η Alpine Linux). Αυτό ελαχιστοποιεί το μέγεθος της τελικής εικόνας και μειώνει δραματικά την επιφάνεια επίθεσης (attack surface) για λόγους ασφαλείας.
- **Επίπεδο Παρουσίασης (React Service):** Το Frontend ενθυλακώθηκε χρησιμοποιώντας το περιβάλλον ανάπτυξης της Vite. Κατά τη διάρκεια της ανάπτυξης, το κοντέινερ επιτρέπει τη λειτουργία του Hot Module Replacement (HMR), επιτρέποντας στον προγραμματιστή να βλέπει τις αλλαγές στον κώδικα σε πραγματικό χρόνο, ενώ ο διακομιστής εκτελείται μέσα σε ένα πλήρως απομονωμένο περιβάλλον.

3.6.3 ΔΙΚΤΎΩΣΗ ΚΑΙ ΑΣΦΆΛΕΙΑ ΥΠΟΔΟΜΉΣ

Ένα από τα σημαντικότερα αρχιτεκτονικά οφέλη της υιοθέτησης του Docker στην παρούσα εργασία είναι η δυνατότητα δημιουργίας ενός απομονωμένου και ελεγχόμενου Εσωτερικού Δικτύου, γνωστού ως **Docker Bridge Network**. Στην παραδοσιακή ανάπτυξη εφαρμογών, οι υπηρεσίες (Backend, Βάση Δεδομένων) συχνά εκτελούνται απευθείας στο λειτουργικό σύστημα του ξενιστή, μοιραζόμενες την ίδια δικτυακή στοίβα. Αυτή η προσέγγιση εγκυμονεί κινδύνους, καθώς μια λανθασμένη ρύθμιση μπορεί να εκθέσει ευαίσθητες θύρες της βάσης δεδομένων στο δημόσιο διαδίκτυο. Μέσω του Docker, υλοποιείται ένα επίπεδο αφαίρεσης που ονομάζεται Software Defined Networking (SDN), το οποίο απομονώνει πλήρως την κυκλοφορία των δεδομένων της εφαρμογής από το εξωτερικό περιβάλλον.

Στο δίκτυο που δημιουργείται αυτόματα μέσω του Docker Compose, οι τρεις υπηρεσίες του συστήματος (Frontend, Backend, Virtuoso) επικοινωνούν μεταξύ τους χρησιμοποιώντας έναν ενσωματωμένο μηχανισμό **Service Discovery**. Ο μηχανισμός αυτός λειτουργεί ως ένας εσωτερικός διακομιστής DNS. Αντί ο κώδικας του Backend να χρειάζεται να γνωρίζει τη μεταβαλλόμενη διεύθυνση IP του κοντέινερ του Virtuoso, χρησιμοποιεί απλώς το όνομα της υπηρεσίας ως διεύθυνση (π.χ. http://virtuoso:8890).

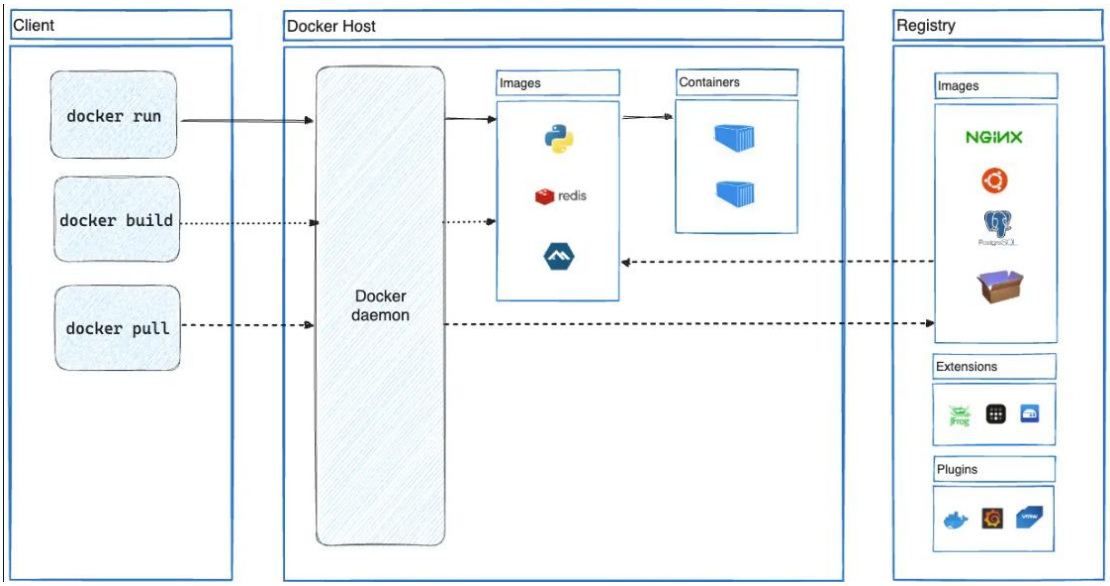
Αυτή η προσέγγιση καθιστά την υποδομή εξαιρετικά ανθεκτική και ευέλικτη. Σε περίπτωση επανεκκίνησης ή αναβάθμισης ενός κοντέινερ, το Docker αναλαμβάνει να ενημερώσει αυτόματα τις εγγραφές DNS, διασφαλίζοντας ότι η επικοινωνία μεταξύ των επιμέρους τμημάτων δεν θα διακοπεί ποτέ λόγω αλλαγής δικτυακών παραμέτρων.

Από την πλευρά της ασφάλειας, η χρήση του Docker Compose επιτρέπει την εφαρμογή της αρχής του ελάχιστου προνομίου (Principle of Least Privilege). Στο αρχείο παραμετροποίησης, γίνεται σαφής διαχωρισμός μεταξύ των εννοιών ports και expose:

1. **Περιορισμένη Έκθεση (Port Mapping):** Μόνο οι απολύτως απαραίτητες θύρες (όπως η θύρα 3000 για το Frontend και η 8080 για το Backend) "ανοίγουν" προς τον έξω κόσμο, επιτρέποντας την πρόσβαση στους χρήστες.
2. **Εσωτερική Απομόνωση (Exposition):** Ο διακομιστής Virtuoso παραμένει "κρυμμένος" εντός του εσωτερικού δικτύου. Η θύρα διαχείρισης δεδομένων και εκτέλεσης ερωτημάτων (8890) δεν εκτίθεται ποτέ στο δημόσιο διαδίκτυο.

Με αυτόν τον τρόπο, το κοντέινερ του Backend λειτουργεί ως ένα **Επίπεδο Ασφαλείας (Security Buffer)** ή "Bastion Host". Κάθε αίτημα προς τη βάση δεδομένων πρέπει υποχρεωτικά να περάσει από τον κώδικα της Go, ο οποίος ελέγχει, φιλτράρει και επικυρώνει το αίτημα πριν το προωθήσει στον Virtuoso. Αυτή η αρχιτεκτονική εμποδίζει την απευθείας προσπέλαση της βάσης από μη εξουσιοδοτημένους χρήστες και προστατεύει το σύστημα από επιθέσεις τύπου SPARQL Injection ή μη εξουσιοδοτημένη εξαγωγή δεδομένων.

Συμπερασματικά, η υιοθέτηση του Docker μετατρέπει την εφαρμογή από μια συλλογή αρχείων σε μια φορητή, αυτοδύναμη και ασφαλή μονάδα λογισμικού. Η φιλοσοφία της "Υποδομής ως Κώδικα" (Infrastructure as Code) επιτρέπει την ανάπτυξη ολόκληρης της πολύπλοκης υποδομής του Σημασιολογικού Ιστού με μία μόνο εντολή (docker-compose up). Αυτό εξαλείφει τις χρονοβόρες και επιρρεπείς σε σφάλματα διαδικασίες χειροκίνητης εγκατάστασης και διαμόρφωσης (configuration drift). Η λογική των ανεξάρτητων υπηρεσιών που επικοινωνούν μέσω DNS είναι απόλυτα συμβατή με σύγχρονα περιβάλλοντα ενορχήστρωσης μικροϋπηρεσιών, όπως το **Kubernetes**. Έτσι, η εφαρμογή μπορεί εύκολα να μεταφερθεί σε ένα σύμπλεγμα διακομιστών (cluster), όπου πολλαπλά αντίγραφα του Backend θα εξυπηρετούν χιλιάδες ταυτόχρονους χρήστες, διατηρώντας πάντα τα ίδια υψηλά πρότυπα ασφάλειας και απομόνωσης.



Εικόνα 6 Επίσημη Αρχιτεκτονική του containerization Docker ,
Πηγή: Docker Documentation https://docs.docker.com/get-started/docker-overview/

ΚΕΦΑΛΑΙΟ 4: ΕΞΑΓΩΓΗ ΚΑΙ ΑΠΟΘΗΚΕΥΣΗ ΔΕΔΟΜΕΝΩΝ

4.1 ΕΙΣΑΓΩΓΗ ΚΕΦΑΛΑΙΟΥ

Για να καταστεί λειτουργική η σημασιολογική εφαρμογή ιστού, ήταν απαραίτητη η δημιουργία μιας τοπικής, αυτόνομης βάσης δεδομένων (Knowledge Graph) που να περιέχει την πιο πρόσφατη έκδοση της Ελληνικής DBpedia. Η διαδικασία συλλογής, μετασχηματισμού και φόρτωσης αυτού του τεράστιου όγκου δεδομένων (Data Ingestion) αποτελεί μια από τις μεγαλύτερες προκλήσεις στη διαχείριση Μεγάλων Δεδομένων (Big Data) και του Σημασιολογικού Ιστού. Στο παρόν κεφάλαιο αναλύεται η στρατηγική που ακολουθήθηκε για τον εντοπισμό των κατάλληλων αρχείων, την αυτοματοποιημένη λήψη τους και τη μαζική φόρτωσή τους (bulk loading) στον εξυπηρετητή Virtuoso.

4.2 ΕΝΤΟΠΙΣΜΌΣ ΔΕΔΟΜΈΝΩΝ: ΤΟ DBPEDIA DATABUS

Στις απαρχές του εγχειρήματος της DBpedia, η διανομή των συνόλων δεδομένων πραγματοποιούνταν κυρίως μέσω μεγάλων, μονολιθικών εκδόσεων (dumps), τα οποία οι χρήστες κατέβαζαν αυτούσια. Ωστόσο, η ραγδαία αύξηση του όγκου της πληροφορίας και η ανάγκη για συχνότερες και πιο στοχευμένες ενημερώσεις, οδήγησαν στην ανάπτυξη μιας καινοτόμου πλατφόρμας διαχείρισης δεδομένων, γνωστής ως **DBpedia Databus** [10].

Το Databus δεν αποτελεί έναν απλό διακομιστή αρχείων (file server), αλλά λειτουργεί ως ένα κατανεμημένο δίκτυο υπερδεδομένων (metadata network). Εισάγει στον χώρο των δεδομένων πρακτικές αντίστοιχες με εκείνες της ανάπτυξης λογισμικού (όπως το Maven ή το Git). Κάθε σύνολο δεδομένων που παράγεται από τον μηχανισμό εξαγωγής (Extraction Framework) καταχωρείται στον σημασιολογικό κατάλογο του Databus, φέροντας μια ψηφιακή ταυτότητα. Η πλατφόρμα διασφαλίζει τον αυστηρό έλεγχο εκδόσεων (versioning), την καταγραφή της προέλευσης των δεδομένων (data provenance) μέσω ψηφιακών υπογραφών, καθώς και τη δυνατότητα πολύπλοκης αναζήτησης και εντοπισμού αρχείων απευθείας μέσω ερωτημάτων SPARQL.

Η αρθρωτή (modular) φιλοσοφία του Databus επιτρέπει στους προγραμματιστές να αντλούν επιλεκτικά μόνο τα υποσύνολα δεδομένων (artifacts) που εξυπηρετούν τις λειτουργικές απαιτήσεις του εκάστοτε συστήματος. Η Ελληνική DBpedia (οριζόμενη με την ετικέτα γλώσσας lang=el) διαχωρίζεται σε δεκάδες επιμέρους αρχεία, συνήθως συμπιεσμένα σε μορφή .ttl.bz2 ή .nt.bz2.

Για την αρχιτεκτονική της παρούσας πτυχιακής εργασίας —η οποία δεν αποτελεί ένα απλό αποθετήριο, αλλά έναν διαδραστικό Σημασιολογικό Φυλλομετρητή με δυνατότητες γεωχωρικής οπτικοποίησης, προβολής πολυμέσων και σημασιολογικού φιλτραρίσματος— η τυφλή φόρτωση όλων των διαθέσιμων αρχείων στον διακομιστή Virtuoso θα αποτελούσε σπατάλη υπολογιστικών πόρων (μνήμης RAM και αποθηκευτικού χώρου). Συνεπώς, ακολουθήθηκε μια στρατηγική επιλεκτικής δειγματοληψίας, απομονώνοντας και εισάγοντας στη βάση αποκλειστικά τα ακόλουθα συστατικά (artifacts):

- **Ετικέτες Ονοματοδοσίας (Labels):** Περιέχει τις βασικές ονομασίες όλων των οντοτήτων (rdfs:label). Αποτελεί το σημαντικότερο αρχείο για το Front-end (React), καθώς επιτρέπει στο σύστημα να προβάλλει ανθρώπινα αναγνώσιμους τίτλους (π.χ. "Κρήτη") αντί για ακατέργαστα URIs (π.χ. <http://el.dbpedia.org/resource/Κρήτη>).

- **Κειμενικές Περιλήψεις (Short & Long Abstracts):** Περιλαμβάνει τις εισαγωγικές παραγράφους και τις εκτενείς περιγραφές των άρθρων (dbo:abstract). Όπως αναλύθηκε στο Κεφάλαιο των Προκλήσεων του Ελληνικού Κόμβου, τα αρχεία αυτά είναι υψίστης σημασίας για το σύστημά μας, καθώς τροφοδοτούν τον μηχανισμό Σημασιολογικού Φιλτραρίσματος ελεύθερου κειμένου, αυξάνοντας την ανάκληση αποτελεσμάτων όταν απουσιάζουν ρητές ιδιότητες.
- **Τυπολογία Οντοτήτων (Instance Types):** Το αρχείο αυτό κωδικοποιεί την οντολογική κλάση στην οποία ανήκει κάθε πόρος (rdf:type). Είναι απολύτως απαραίτητο για τη σωστή κατηγοριοποίηση και την εκτέλεση ερωτημάτων συνόλων, επιτρέποντας στην εφαρμογή να διαχωρίζει δυναμικά τα Μουσεία (dbo:Museum), τα Νησιά (dbo:Island) ή τα Πρόσωπα (dbo:Person).
- **Χαρτογραφημένες Σχέσεις και Τιμές (Mapping-based Objects & Literals):** Αποτελούν τον "σκελετό" του γραφήματος γνώσης, καθώς περιέχουν τα δεδομένα υψηλής ποιότητας που έχουν εξαχθεί επιτυχώς από τα Infoboxes. Το αρχείο των **Objects** αποθηκεύει τις ακμές του γραφήματος που συνδέουν οντότητες μεταξύ τους (π.χ. ένα νησί με τη χώρα του), ενώ το αρχείο των **Literals** αποθηκεύει τις ιδιότητες που περιέχουν αριθμητικές ή αλφαριθμητικές τιμές (π.χ. πληθυσμός, έκταση, ημερομηνία γέννησης).
- **Γεωχωρικά Δεδομένα (Geo-coordinates):** Περιλαμβάνει τα γεωγραφικά πλάτη (geo:lat) και μήκη (geo:long) για τις χωρικές οντότητες. Η εισαγωγή αυτού του αρχείου είναι υπεύθυνη για τη λειτουργία του εντοπισμού χαρτών στη διεπαφή της εφαρμογής (map rendering).
- **Πολυμεσικοί Σύνδεσμοι (Images):** Περιέχει τους συνδέσμους URL (dbo:thumbnail) προς τις μικρογραφίες και τις εικόνες των πόρων από το Wikimedia Commons. Επιτρέπει στη React να μετατρέπει δυναμικά την ακατέργαστη πληροφορία σε πλούσια οπτική εμπειρία για τον χρήστη.

Η συγκεκριμένη μεθοδολογία συλλογής δεδομένων εξασφάλισε τη δημιουργία ενός ελαφρού, αποδοτικού, αλλά ταυτόχρονα εξαιρετικά εμπλουτισμένου τοπικού υποσυνόλου (local subset) του Σημασιολογικού Ιστού, ιδανικά προσαρμοσμένου στις ανάγκες ταχύτατης εκτέλεσης (low-latency execution) της παρούσας εφαρμογής.

4.3 ΛΗΨΗ ΔΕΔΟΜΈΝΩΝ

Εξαιτίας του πλήθους των αρχείων, η χειροκίνητη λήψη τους από τον ιστότοπο κρίθηκε αντιπαραγωγική και επιρρεπής σε σφάλματα. Για τον λόγο αυτό, αναπτύχθηκε ένα αυτοματοποιημένο σενάριο κελύφους (Bash / PowerShell script) το οποίο επικοινωνεί απευθείας με το API του Databus.
Η λογική του σεναρίου (script) ακολουθεί τα εξής βήματα:

1. **Διατύπωση Ερωτήματος:** Συντάσσεται ένα ερώτημα SPARQL που ζητά τα URLs λήψης (dcat:downloadURL) από το σύνολο διανομών (dcat:distribution), εφαρμόζοντας ταυτόχρονα φίλτρα κειμένου (Regular Expressions/Contains) ώστε να επιλεγούν **μόνο** τα αρχεία με lang=el και **μόνο** των κατηγοριών που αναφέρθηκαν στην Ενότητα 4.1.
2. **Κωδικοποίηση και Εκτέλεση:** Το ερώτημα μετατρέπεται σε μορφή ασφαλή για το διαδίκτυο (URL Encoding) και αποστέλλεται στο Endpoint του Databus (https://databus.dbpedia.org/sparql) ζητώντας τα αποτελέσματα σε μορφή CSV (Comma-Separated Values).
3. **Λήψη (Download) και Αποθήκευση:** Το σενάριο διαβάζει γραμμή-γραμμή το παραγόμενο CSV και χρησιμοποιεί εντολές συστήματος (όπως curl ή Invoke-WebRequest) για να κατεβάσει ασύγχρονα τα αρχεία στον τοπικό φάκελο dbpedia_data. Παράλληλα, περιλαμβάνει

μηχανισμούς ελέγχου (idempotency), ώστε να προσπερνά αρχεία που έχουν ήδη ληφθεί επιτυχώς στο παρελθόν.

Η αυτοματοποίηση αυτή προσδίδει στο έργο επαναληψιμότητα (reproducibility), επιτρέποντας την εύκολη επικαιροποίηση (update) της βάσης μελλοντικά, όταν κυκλοφορήσουν νέες εκδόσεις της Ελληνικής DBpedia.

```
PREFIX dcat: <http://www.w3.org/ns/dcat#>
PREFIX dataid: <http://dataid.dbpedia.org/ns/core#>

SELECT DISTINCT ?file WHERE {
  ?dataset dcat:distribution ?distribution .
  ?distribution dcat:downloadURL ?file .

  # ΦΙΛΤΡΟ 1: Να είναι Ελληνικά
  FILTER (contains(str(?file), 'lang=el'))

  # ΦΙΛΤΡΟ 2: Τα αρχεία που χρειαζόμαστε
  FILTER (
    contains(str(?file), '/labels') ||
    contains(str(?file), '/short-abstracts') ||
    contains(str(?file), '/long-abstracts') ||
    contains(str(?file), '/instance-types') ||
    contains(str(?file), '/mappingbased-objects') ||
    contains(str(?file), '/mappingbased-literals') ||
    contains(str(?file), '/images') ||
    contains(str(?file), '/geo-coordinates')
  )

```

Εικόνα 7 : Ο κώδικας γραμμένος σε Sparql που χρησιμοποιήθηκε για να αντλήσουμε τα δεδομένα που θέλουμε στα στα ελληνικά

4.4 ΑΡΧΙΤΕΚΤΟΝΙΚΉ ΑΠΟΘΉΚΕΥΣΗΣ ΚΑΙ ΑΡΧΙΚΟΠΟΊΗΣΗ VIRTUOSO

Όπως αναλύθηκε στο Κεφάλαιο 3, το σύστημα εκτελείται εντός απομονωμένων περιβαλλόντων (Docker Containers). Για να καταστεί δυνατή η ανάγνωση των αρχείων που λήφθηκαν τοπικά από το λογισμικό του Virtuoso (το οποίο τρέχει "μέσα" στο container), χρησιμοποιήθηκε η τεχνική των **Docker Volumes** (Κοινόχρηστοι Φάκελοι).

Συγκεκριμένα, μέσω του αρχείου docker-compose.yml, ο τοπικός φάκελος λήψεων αντιστοιχίστηκε (mapped) στην εσωτερική διαδρομή του container /database/to_load.

Κατά την πρώτη εκκίνηση του Virtuoso, η μηχανή της βάσης δεδομένων δημιουργεί αυτόματα τα απαραίτητα αρχεία συστήματος για τη λειτουργία της [18]. Στα σημαντικότερα από αυτά συγκαταλέγονται:

- virtuoso.ini: Το κεντρικό αρχείο ρυθμίσεων. Για λόγους ασφαλείας, ο Virtuoso αποτρέπει την ανάγνωση τυχαίων αρχείων του δίσκου. Απαιτήθηκε η παραμετροποίηση της μεταβλητής DirsAllowed ώστε να συμπεριλάβει τον φάκελο /database/to_load.
- virtuoso.trx: Το αρχείο καταγραφής συναλλαγών (Transaction Log). Εξασφαλίζει τις ιδιότητες ACID (Atomicity, Consistency, Isolation, Durability) κατά τη φόρτωση, διασφαλίζοντας ότι σε περίπτωση διακοπής ρεύματος, η βάση δεν θα καταστραφεί.

- virtuoso.db: Το κεντρικό αρχείο δυαδικής μορφής (binary) όπου αποθηκεύεται οριστικά το Γράφημα Γνώσης και τα ευρετήριά του (indexes).

4.5 ΜΑΖΙΚΉ ΦΌΡΤΩΣΗ ΔΕΔΟΜΈΝΩΝ (BULK LOADING)

Η διαδικασία της εισαγωγής ενός τεράστιου όγκου δεδομένων (Data Ingestion) αποτελεί ένα από τα κρισιμότερα και πιο απαιτητικά στάδια στην ανάπτυξη συστημάτων Σημασιολογικού Ιστού μεγάλης κλίμακας. Η τυπική, βασισμένη στα πρότυπα μέθοδος για την προσθήκη νέας πληροφορίας σε ένα τερματικό σημείο (endpoint) RDF πραγματοποιείται μέσω της εντολής INSERT DATA της γλώσσας SPARQL. Η προσέγγιση αυτή ακολουθεί ένα μοντέλο συναλλαγών (transactional model) και εκτελείται μέσω του πρωτοκόλλου HTTP. Ωστόσο, για σύνολα Μεγάλων Δεδομένων (Big Data) που υπερβαίνουν τα μερικά megabytes —και πόσο μάλλον για την Ελληνική DBpedia, η οποία αποτελείται από δεκάδες εκατομμύρια τριάδες— η συγκεκριμένη μέθοδος καθίσταται απαγορευτικά αργή και πρακτικά ανεπαρκής.

Ο λόγος για αυτή την ανεπάρκεια έγκειται στο γεγονός ότι το Σύστημα Διαχείρισης Βάσης Δεδομένων (ΣΔΒΔ) καλείται να παραλάβει κάθε HTTP αίτημα, να το αποσυμπιέσει, να επικυρώσει συντακτικά το ερώτημα SPARQL, να μεταγλωττίσει την κάθε μεμονωμένη τριάδα (Subject-Predicate-Object) και, τέλος, να την εγγράψει στον δίσκο κλειδώνοντας τον πίνακα για να διασφαλίσει την ακεραιότητα της συναλλαγής [19]. Το δικτυακό και υπολογιστικό κόστος (overhead) αυτής της διαδρομής είναι τεράστιο. Για τον λόγο αυτό, προκειμένου να επιτευχθεί μέγιστη απόδοση (throughput), επιλέχθηκε η στρατηγική της **Μαζικής Φόρτωσης Δεδομένων (Bulk Data Loading)**, μια διαδικασία εγγενώς βελτιστοποιημένη στον πυρήνα του Virtuoso Universal Server.

Αντί η επικοινωνία να γίνεται μέσω δικτυακών κλήσεων (REST/HTTP) από το εξωτερικό περιβάλλον, η μαζική φόρτωση εκτελείται ενδοσυστημικά. Αξιοποιείται η διεπαφή γραμμής εντολών isql (Interactive SQL), η οποία συνδέεται απευθείας με τον πυρήνα (engine) της βάσης δεδομένων, παρακάμπτοντας εντελώς το επίπεδο του διακομιστή ιστού (web server layer). Η συνολική διαδικασία της ροής εργασίας (workflow) αποτυπώνεται στα ακόλουθα τρία αυστηρά διακριτά στάδια:

Το πρώτο βήμα αφορά την αναγνώριση των αρχείων δεδομένων από το σύστημα. Εκτελείται η εντολή

SQL:ld_dir('/database/to_load', '*.bz2', 'http://el.dbpedia.org');

Αυτή η συνάρτηση δεν φορτώνει άμεσα τα δεδομένα στη μνήμη, αλλά λειτουργεί ως ένας μηχανισμός χαρτογράφησης και προγραμματισμού εργασιών. Αρχικά, σαρώνει τον καθορισμένο κατάλογο (/database/to_load) του συστήματος αρχείων (file system) και εντοπίζει όλα τα συμπιεσμένα αρχεία που ακολουθούν την κατάληξη *.bz2. Η χρήση της συμπίεσης Bzip2 είναι ζωτικής σημασίας σε αυτό το επίπεδο: μειώνει δραματικά τις λειτουργίες Εισόδου/Εξόδου (I/O operations) του σκληρού δίσκου, ανταλλάσσοντας τον φόρτο του δίσκου με επεξεργαστική ισχύ (CPU cycles), μια ανταλλαγή που παραδοσιακά βελτιστοποιεί την ταχύτητα στις βάσεις δεδομένων.

Στη συνέχεια, η εντολή καταχωρεί αυτά τα αρχεία σε έναν ειδικό πίνακα συστήματος (system table) που λειτουργεί ως ουρά αναμονής, τον DB.DBA.LOAD_LIST. Ταυτόχρονα, τα αντιστοιχίζει στο

Ονομαστικό Γράφημα (Named Graph) http://el.dbpedia.org. Η χρήση του Ονομαστικού Γραφήματος μετατρέπει ουσιαστικά τις απλές τριάδες (Triples) σε τετράδες (Quads: Graph-Subject-Predicate-Object). Αυτό αποτελεί μια εξαιρετική πρακτική σχεδιασμού, καθώς δημιουργεί ένα λογικό "διαμέρισμα" (logical partition) εντός της βάσης δεδομένων. Επιτρέπει την πλήρη απομόνωση των δεδομένων της Ελληνικής DBpedia από τυχόν άλλα σύνολα δεδομένων που μπορεί να φιλοξενεί ο διακομιστής, διευκολύνοντας τη μελλοντική διαγραφή, διαχείριση ή ανανέωσή τους (data lifecycle management) χωρίς να επηρεάζεται το υπόλοιπο σύστημα.

Μόλις η ουρά αναμονής τροφοδοτηθεί, εκκινεί η κυρίως διαδικασία με την κλήση της συνάρτησης rdf_loader_run();. Σε αυτό το στάδιο, ο κινητήρας του Virtuoso επιδεικνύει τις δυνατότητες Κάθετης Κλιμάκωσης (Vertical Scaling) που διαθέτει, εκκινώντας πολλαπλά ταυτόχρονα νήματα (multi-threading) στο παρασκήνιο. Το σύστημα αναλαμβάνει τον ρόλο ενός εργοστασιακού αγωγού (pipeline): τα νήματα διαβάζουν τα αρχεία .bz2, τα αποσυμπιέζουν εν πτήσει (on-the-fly) στη μνήμη RAM και πραγματοποιούν συντακτική ανάλυση (parsing) των γραμμών RDF.

Το πιο απαιτητικό, υπολογιστικά, μέρος αυτού του σταδίου δεν είναι η απλή αποθήκευση, αλλά η δόμηση των Ευρετηρίων (Indexing). Για να μπορεί η SPARQL να βρίσκει ταχύτατα αποτελέσματα, ο Virtuoso δεν αποθηκεύει τα δεδομένα μια φορά. Κατασκευάζει πολλαπλά, αλληλοκαλυπτόμενα ευρετήρια δομής B-Tree (Balanced Trees), οργανώνοντας τις τετράδες σε διαφορετικές μεταθέσεις (όπως SPOG, POSG, OPSG). Αυτή η δομή δέντρου εγγυάται ότι οι χρόνοι αναζήτησης θα είναι λογαριθμικοί $O(\log n)$, ανεξάρτητα από το αν ο χρήστης αναζητά το Υποκείμενο, το Κατηγόρημα ή το Αντικείμενο ενός ερωτήματος.

Καθ' όλη τη διάρκεια αυτής της διεργασίας, η αρχιτεκτονική του Virtuoso παρέχει δυνατότητες πλήρους Παρατηρησιμότητας (Observability). Ο διαχειριστής της βάσης μπορεί να παρακολουθεί δυναμικά την πρόοδο, τα αρχεία που ολοκληρώθηκαν και τα τυχόν σφάλματα συντακτικής ανάλυσης, εκτελώντας την εντολή επιθεώρησης SELECT * FROM DB.DBA.LOAD_LIST;.

Το τελευταίο και πιο κρίσιμο στάδιο για την ακεραιότητα του συστήματος είναι η οριστικοποίηση της φόρτωσης. Κατά τη διάρκεια της εκτέλεσης του rdf_loader_run(), ο τεράστιος όγκος των νέων ευρετηρίων και των τριάδων παραμένει αποθηκευμένος προσωρινά στις κρυφές μνήμες (Buffer Pools) της μνήμης RAM, προκειμένου να επιτευχθεί μέγιστη ταχύτητα εγγραφής. Παράλληλα, για λόγους ασφαλείας, οι λειτουργίες καταγράφονται σειριακά στο Μητρώο Συναλλαγών του συστήματος, ένα αρχείο γνωστό ως Write-Ahead Log (WAL) με το όνομα virtuoso.trx. Εάν διακοπεί η παροχή ρεύματος σε αυτό το σημείο, τα δεδομένα δεν χάνονται, καθώς ο διακομιστής μπορεί να τα ανακτήσει διαβάζοντας το αρχείο .trx.

Ωστόσο, για να διασφαλιστεί η βέλτιστη απόδοση του συστήματος μακροπρόθεσμα, πρέπει να εκτελεστεί η εντολή checkpoint;. Η λειτουργία αυτή "παγώνει" στιγμιαία τον διακομιστή και εξαναγκάζει τον κινητήρα της βάσης να «αδειάσει» (flush) όλες τις τροποποιημένες σελίδες μνήμης (dirty pages) από τη RAM απευθείας στο κύριο, μόνιμο αρχείο δεδομένων στον σκληρό δίσκο

(virtuoso.db). Μόλις τα δεδομένα κλειδωθούν με ασφάλεια στο αρχείο .db, το αρχείο καταγραφής .trx μηδενίζεται (truncation), απελευθερώνοντας πολύτιμο αποθηκευτικό χώρο.

Με την επιτυχή ολοκλήρωση αυτής της τριπλής διαδικασίας, το Επίπεδο Δεδομένων (Data Tier) μεταβαίνει από την κατάσταση της προετοιμασίας στην πλήρη παραγωγική του λειτουργία. Καθίσταται πλέον απολύτως ικανό, ασφαλές και ευρετηριασμένο, έτοιμο να δεχτεί και να απαντήσει ακαριαία στα εξαιρετικά πολύπλοκα ερωτήματα SPARQL, τροφοδοτώντας την επιχειρησιακή λογική και τη διεπαφή χρήστη της τελικής εφαρμογής.

ΚΕΦΑΛΑΙΟ 5 :ΥΛΟΠΟΙΗΣΗ ΣΥΣΤΗΜΑΤΟΣ

5.1 ΕΙΣΑΓΩΓΗ ΚΕΦΑΛΑΙΟΥ

Η αρχιτεκτονική και οι τεχνολογίες που επιλέχθηκαν (Κεφάλαιο 3) αποτέλεσαν το θεμέλιο για την ανάπτυξη της τελικής εφαρμογής. Στο παρόν κεφάλαιο, αναλύεται ο τρόπος με τον οποίο υλοποιήθηκε προγραμματιστικά το σύστημα. Γίνεται εις βάθος παρουσίαση της ανάπτυξης του ενδιάμεσου επιπέδου (Backend) σε γλώσσα Golang με έμφαση στον ταυτοχρονισμό, αναλύεται η στρατηγική σημασιολογικού φιλτραρίσματος στα ερωτήματα SPARQL και, τέλος, παρουσιάζεται η υλοποίηση της διαδραστικής διεπαφής χρήστη (Frontend) με τη βιβλιοθήκη React.js.

5.2 BACKEND (GOLANG MIDDLEWARE):

Ο πυρήνας της επιχειρησιακής λογικής (business logic) του συστήματος αναπτύχθηκε εξ ολοκλήρου στη γλώσσα προγραμματισμού Go. Η επιλογή της συγκεκριμένης γλώσσας δεν έγινε τυχαία, καθώς η Go σχεδιάστηκε εξαρχής με γνώμονα τη δημιουργία κατανεμημένων συστημάτων και δικτυακών υπηρεσιών υψηλής απόδοσης. Στην παρούσα αρχιτεκτονική, το Backend αναλαμβάνει τον κρίσιμο ρόλο του Αντίστροφου Διαμεσολαβητή (Reverse Proxy). Στην πράξη, αυτό σημαίνει ότι λειτουργεί ως το μοναδικό σημείο επαφής (gateway) μεταξύ της διεπαφής του χρήστη (React) και του εξυπηρετητή βάσης δεδομένων (Virtuoso).

Η ύπαρξη αυτού του ενδιάμεσου επιπέδου προσφέρει πολλαπλά οφέλη. Πρωτίστως, απομονώνει τον Virtuoso από το δημόσιο δίκτυο, προστατεύοντας τη βάση δεδομένων από κακόβουλες επιθέσεις (όπως SPARQL Injection) ή υπερφόρτωση, καθώς το Backend φιλτράρει και επικυρώνει κάθε εισερχόμενο αίτημα. Παράλληλα, το Go API παραλαμβάνει τα αιτήματα HTTP από τον πελάτη, τα μετατρέπει στα κατάλληλα ερωτήματα SPARQL, τα προωθεί στην εσωτερική θύρα του Virtuoso και, στη συνέχεια, μορφοποιεί τα επιστρεφόμενα ακατέργαστα δεδομένα JSON πριν τα παραδώσει πίσω στον φυλλομετρητή (browser).

Μία από τις σημαντικότερες αρχιτεκτονικές αποφάσεις της υλοποίησης ήταν η αποκλειστική χρήση της πρότυπης βιβλιοθήκης της Go, και συγκεκριμένα του πακέτου net/http, για τη δημιουργία του διακομιστή. Ενώ στον χώρο της ανάπτυξης λογισμικού συνηθίζεται η χρήση βαριών εξωτερικών πλαισίων (όπως το Gin, το Fiber ή το Echo), στο παρόν έργο επιλέχθηκε μια προσέγγιση ελαχιστοποίησης των εξαρτήσεων (zero-dependency approach). Η ενσωματωμένη βιβλιοθήκη net/http αποδείχθηκε υπεραρκετή, καθώς παρέχει εγγενή υποστήριξη για δρομολόγηση (routing), διαχείριση κεφαλίδων (headers) και χειρισμό αιτημάτων/απαντήσεων (request/response handling).

Η απουσία εξωτερικών πλαισίων (frameworks) μεταφράζεται σε μικρότερο μέγεθος εκτελέσιμου αρχείου (binary bloat), μειωμένη επιφάνεια επίθεσης (attack surface) και μηδενικό χρόνο εκμάθησης ιδιωματισμών τρίτων εργαλείων. Επιπλέον, το πακέτο net/http εκκινεί αυτόματα μια νέα "Goroutine" για κάθε εισερχόμενο HTTP αίτημα. Αυτό σημαίνει ότι ο διακομιστής μας είναι εγγενώς ικανός να εξυπηρετεί ταυτόχρονους χρήστες με απόλυτη ασφάλεια και ασύγχρονη λειτουργία, χωρίς να απαιτείται περίπλοκη διαχείριση νημάτων (thread pooling) εκ μέρους του προγραμματιστή, μεγιστοποιώντας έτσι τη συνολική απόδοση (throughput) του συστήματος

5.2.1 ΔΡΟΜΟΛΌΓΗΣΗ ΚΑΙ ΔΙΑΧΕΊΡΙΣΗ CORS

Η αρχιτεκτονική του συστήματος βασίζεται στον αυστηρό διαχωρισμό του επιπέδου παρουσίασης (Frontend) από το επίπεδο λογικής (Backend). Δεδομένου ότι τα δύο αυτά υποσυστήματα εκτελούνται σε εντελώς διαφορετικά περιβάλλοντα και θύρες μέσω των Docker Containers (για παράδειγμα, η διεπαφή της React εξυπηρετείται στη θύρα 3000, ενώ το Go API «ακούει» στη θύρα 8080), η επικοινωνία τους αντιμετωπίζεται από τους σύγχρονους φυλλομετρητές ιστού (web browsers) ως διακομιστική κλήση (cross-origin request). Κατά συνέπεια, κρίθηκε επιβεβλημένη η υλοποίηση ενός μηχανισμού για τη διαχείριση της πολιτικής Ασφάλειας Κοινής Προέλευσης (CORS - Cross-Origin Resource Sharing).

Η Πολιτική Ίδιας Προέλευσης (Same-Origin Policy) αποτελεί έναν θεμελιώδη μηχανισμό ασφαλείας των περιηγητών, ο οποίος αποτρέπει εξ ορισμού κακόβουλους ιστότοπους από το να διαβάζουν ευαίσθητα δεδομένα από ένα άλλο σύστημα. Για την ασφαλή παράκαμψη αυτού του περιορισμού στη δική μας εφαρμογή, ενσωματώθηκε ειδική λογική διαχείρισης στο επίπεδο της δρομολόγησης, και συγκεκριμένα εντός του κεντρικού χειριστή ερωτημάτων (SPARQL Handler) της Go.

Όταν το Frontend επιχειρεί να στείλει ένα πολύπλοκο αίτημα (όπως ένα HTTP POST που περιέχει το ερώτημα SPARQL στο σώμα του), ο φυλλομετρητής, πριν στείλει τα πραγματικά δεδομένα, αποστέλλει αυτόματα ένα προπαρασκευαστικό αίτημα ελέγχου, γνωστό ως αίτημα προ-ελέγχου (pre-flight request), χρησιμοποιώντας τη μέθοδο HTTP OPTIONS. Η διαδικασία αυτή διεξάγεται για να επιβεβαιώσει ο πελάτης (client) ότι ο διακομιστής κατανοεί, αποδέχεται και επιτρέπει τη συγκεκριμένη ενέργεια πριν την εκτελέσει. Ο χειριστής που αναπτύχθηκε στη Go είναι σχεδιασμένος να αναχαιτίζει δυναμικά αυτά τα αιτήματα OPTIONS. Μόλις εντοπιστεί ένα τέτοιο αίτημα, ο διακομιστής απαντά ακαριαία με τον κωδικό επιτυχίας (HTTP 200 OK) δίχως να προχωρήσει σε περαιτέρω επεξεργασία του ερωτήματος προς τη βάση δεδομένων, εξοικονομώντας με αυτόν τον τρόπο πολύτιμους υπολογιστικούς πόρους.

Παράλληλα, για να επιτραπεί η οριστική ροή της πληροφορίας, ο κώδικας της Go παρεμβαίνει και προσθέτει σε κάθε απάντηση (τόσο στα αιτήματα OPTIONS όσο και στα POST) τις απαραίτητες κεφαλίδες (HTTP Headers). Ειδικότερα, εισάγεται η κεφαλίδα Access-Control-Allow-Origin: *, η οποία υποδηλώνει ότι το API λειτουργεί ως μια ανοιχτή υπηρεσία και δέχεται αιτήματα από οποιονδήποτε τομέα. Επιπροσθέτως, ορίζονται οι κεφαλίδες Access-Control-Allow-Methods για να οριοθετηθούν οι επιτρεπτές ενέργειες (όπως GET, POST, OPTIONS) και η Access-Control-Allow-Headers για να επιτραπεί η αποστολή εξειδικευμένων τύπων περιεχομένου, όπως το application/json ή το application/x-www-form-urlencoded.

Μέσω αυτής της αυστηρής αλλά προσεκτικά παραμετροποιημένης ρύθμισης στο επίπεδο του ενδιάμεσου λογισμικού, αποφεύγονται πλήρως τα συνηθισμένα σφάλματα δικτυακής επικοινωνίας (CORS blocks) που εμφανίζονται στις κονσόλες των φυλλομετρητών. Έτσι, διασφαλίζεται η ασφαλής, αξιόπιστη και απρόσκοπτη μεταφορά των ακατέργαστων δεδομένων JSON από τον εξυπηρετητή Virtuoso προς το απομονωμένο περιβάλλον της React, δημιουργώντας μια αδιάλειπτη εμπειρία για τον τελικό χρήστη.

5.2.2 ΒΕΛΤΙΣΤΟΠΟΊΗΣΗ ΜΕ ΤΑΥΤΟΧΡΟΝΙΣΜΌ (CONCURRENCY)

Ένα από τα πλέον καινοτόμα χαρακτηριστικά του ενδιάμεσου λογισμικού (Backend) είναι η εκτεταμένη αξιοποίηση του μοντέλου ταυτοχρονισμού (Concurrency) της Go για τη δραστική μείωση του χρόνου απόκρισης (Latency). Στα σύγχρονα συστήματα ανάκτησης δεδομένων, όταν ένας χρήστης εκτελεί ένα ερώτημα αναζήτησης με περιορισμό αποτελεσμάτων (π.χ. LIMIT 50), είναι απαραίτητη η

ταυτόχρονη προβολή του συνολικού όγκου των διαθέσιμων εγγραφών (Total Count) για την ορθή πλοήγηση (pagination) και την κατανόηση της πληροφορίας.

Εάν η διαδικασία αυτή υλοποιούνταν με την παραδοσιακή, σειριακή προσέγγιση, το σύστημα θα έπρεπε να εκτελέσει το ερώτημα ανάκτησης δεδομένων (SELECT) και, αφού περίμενε την ολοκλήρωσή του, να κατασκευάσει και να αποστείλει το δεύτερο ερώτημα καταμέτρησης (COUNT). Η προσέγγιση αυτή, ωστόσο, θα οδηγούσε στον διπλασιασμό του χρόνου αναμονής του χρήστη (network blocking), καθώς ο συνολικός χρόνος θα αποτελούσε το άθροισμα των επιμέρους εκτελέσεων.

Για την υπέρβαση αυτού του περιορισμού, αναπτύχθηκε ένας εξελιγμένος αλγόριθμος που υλοποιεί το αρχιτεκτονικό μοτίβο **Scatter-Gather** (Διασπορά και Συλλογή), αξιοποιώντας τις ελαφριές διεργασίες της Go (Goroutines). Ο μηχανισμός λειτουργεί με τα εξής διακριτά βήματα:

1. **Δυναμική Διάσπαση Ερωτήματος (Query Parsing):** Τη στιγμή της υποβολής του αιτήματος, το API αναλύει (parses) τον αρχικό κώδικα SPARQL του χρήστη. Διαχωρίζει τα προθέματα (PREFIXES), αφαιρεί λέξεις-κλειδιά που περιορίζουν τα αποτελέσματα (όπως το LIMIT και το OFFSET), και συνθέτει δυναμικά ένα δεύτερο, ανεξάρτητο ερώτημα COUNT.
2. **Συγχρονισμός (WaitGroup):** Αρχικοποιείται μια δομή ελέγχου sync.WaitGroup. Η δομή αυτή ενημερώνεται ότι πρέπει να αναμένει την ολοκλήρωση δύο ανεξάρτητων εργασιών πριν επιτρέψει στο κύριο πρόγραμμα να συνεχίσει.
3. **Παράλληλη Εκτέλεση (Scatter):** Εκκινούνται δύο παράλληλα νήματα (Goroutines). Το πρώτο νήμα αναλαμβάνει την αποστολή του ερωτήματος δεδομένων (Data Query), ενώ το δεύτερο νήμα αποστέλλει ταυτόχρονα το ερώτημα καταμέτρησης (Count Query) στον εξυπηρετητή Virtuoso. Οι δύο αυτές I/O (Input/Output) λειτουργίες εκτελούνται εντελώς ανεξάρτητα.
4. **Αμοιβαίος Αποκλεισμός (Mutex):** Για την ασφαλή εγγραφή των επιστρεφόμενων αποτελεσμάτων στην τελική δομή μνήμης της απάντησης (Response Payload struct), γίνεται χρήση του μηχανισμού αμοιβαίου αποκλεισμού sync.Mutex. Αυτό διασφαλίζει ότι, αν τα δύο νήματα ολοκληρωθούν ακριβώς την ίδια στιγμή, δεν θα προκύψουν συνθήκες συναγωνισμού (race conditions) κατά την εγγραφή των δεδομένων, προστατεύοντας την ακεραιότητα της μνήμης.
5. **Συλλογή (Gather):** Μόλις και τα δύο νήματα ολοκληρώσουν τη διεπαφή τους με τη βάση δεδομένων, το WaitGroup απελευθερώνει την κύρια ροή εκτέλεσης, η οποία σειριοποιεί τα συγκεντρωμένα δεδομένα σε μορφή JSON και τα επιστρέφει στον πελάτη.

Η αρχιτεκτονική αυτή αποτελεί ορόσημο για την απόδοση του συστήματος. Εξασφαλίζει ότι ο χρόνος απόκρισης εξαρτάται αποκλειστικά από τον χρόνο του *πιο αργού* ερωτήματος (max(T_data, T_count)) και όχι από το άθροισμά τους. Παράλληλα, αποδεικνύει την ικανότητα της εφαρμογής να διαχειρίζεται απαιτητικά φορτία εργασίας (workloads) και να αξιοποιεί πλήρως τους διαθέσιμους πόρους του διακομιστή, μεγιστοποιώντας την ταχύτητα χωρίς να θυσιάζεται η ασφάλεια.

.

5.3 ΣΤΡΑΤΗΓΙΚΉ ΕΡΩΤΗΜΆΤΩΝ (SPARQL & SEMANTIC FILTERING):

Το Επίπεδο Δεδομένων της παρούσας εφαρμογής βασίζεται αποκλειστικά στον κόμβο της Ελληνικής DBpedia (el.dbpedia.org). Παρότι η DBpedia αποτελεί έναν από τους σημαντικότερους πυλώνες των Ανοικτών Διασυνδεδεμένων Δεδομένων (Linked Open Data), η ανάκτηση πληροφοριών από αυτήν παρουσίασε σημαντικές προκλήσεις, οφειλόμενες κυρίως στην ασυνέπεια και την ελλιπή δομή του παραγόμενου Γραφήματος Γνώσης.

Το πρόβλημα αυτό πηγάζει από τον τρόπο δημιουργίας της DBpedia, η οποία εξάγει δεδομένα κυρίως από τα πλαίσια πληροφοριών (Infoboxes) των άρθρων της Wikipedia. Εάν ο συντάκτης ενός άρθρου στην ελληνική Wikipedia δεν έχει συμπληρώσει σωστά ένα πεδίο (π.χ. το πεδίο "Χώρα" στο

άρθρο ενός ελληνικού μουσείου), τότε κατά τη μετατροπή του άρθρου σε μορφή RDF, η αντίστοιχη ιδιότητα (π.χ. dbo:country) απλώς παραλείπεται από την παραγόμενη τριπλέτα. Κατά συνέπεια, εάν η εφαρμογή μας βασιζόταν αποκλειστικά στην εκτέλεση ερωτημάτων μέσω Αυστηρών Προτύπων Γραφήματος (Strict Graph Patterns), απαιτώντας ρητά την ύπαρξη του κόμβου dbr:Greece, ένα συντριπτικά μεγάλο μέρος της πληροφορίας θα αποκρυπτόταν και το σύστημα θα επέστρεφε ανακριβή ή κενά αποτελέσματα (Low Recall Rate).

5.3.1 ΣΗΜΑΣΙΟΛΟΓΙΚΌ ΦΙΛΤΡΑΡΙΣΜΑ ΚΕΙΜΕΝΟΥ (SEMANTIC FILTERING)

Για την αντιμετώπιση του προβλήματος των «ορφανών» ή ατελώς συνδεδεμένων οντοτήτων, κρίθηκε επιβεβλημένη η εγκατάλειψη της παραδοσιακής αναζήτησης βάσει ιδιοτήτων και η υιοθέτηση μιας στρατηγικής **Σημασιολογικού Φιλτραρίσματος (Semantic Filtering)**. Αντί το σύστημα να αναζητά ρητές οντολογικές συνδέσεις, εξετάζει το μοναδικό στοιχείο που είναι σχεδόν πάντα διαθέσιμο: το αδόμητο κείμενο της σύνοψης/περιγραφής της οντότητας (dbo:abstract).

Για την υλοποίηση αυτής της στρατηγικής, σχεδιάστηκαν "Έξυπνα Προκαθορισμένα Ερωτήματα" (Smart Predefined Queries) στον κώδικα της εφαρμογής. Η λογική αυτών των ερωτημάτων βασίζεται σε έναν συνδυασμό ενσωματωμένων συναρτήσεων της γλώσσας SPARQL, οι οποίες εκτελούν κανονικοποίηση και αναζήτηση λέξεων-κλειδιών επί του κειμένου. Ο πυρήνας αυτού του αλγορίθμου αποτυπώνεται στην παρακάτω δομή:

FILTER (CONTAINS(LCASE(?abstract), "ελλάδα") || CONTAINS(LCASE(?abstract), "αθήνα"))

Η παραπάνω προσέγγιση λειτουργεί σε δύο διαδοχικά στάδια:

1. **Κανονικοποίηση (Normalization):** Αρχικά, η συνάρτηση LCASE(?abstract) αναλαμβάνει να μετατρέψει ολόκληρο το κείμενο της περίληψης σε πεζούς χαρακτήρες. Αυτό το βήμα είναι απολύτως κρίσιμο για τη διασφάλιση της ανεξαρτησίας από την κεφαλαιοποίηση (case-insensitivity), επιτρέποντας στο σύστημα να εντοπίζει λέξεις είτε είναι γραμμένες με κεφαλαία (π.χ. "ΕΛΛΑΔΑ") είτε στην αρχή μιας πρότασης ("Ελλάδα").
2. **Αναζήτηση Προτύπων (Pattern Matching):** Στη συνέχεια, η συνάρτηση CONTAINS ελέγχει εάν το κανονικοποιημένο κείμενο περιέχει συγκεκριμένες λέξεις-κλειδιά που υποδηλώνουν εντοπιότητα. Ο αλγόριθμος αναζητά όχι μόνο το όνομα της χώρας ("ελλάδα"), αλλά και όρους που σχετίζονται άμεσα με αυτήν (π.χ. "αθήνα", "θεσσαλονίκη", "αρχαιολογικό"). Η χρήση του λογικού τελεστή || (OR) επιτρέπει την επέκταση της αναζήτησης σε πολλαπλούς γεωγραφικούς δείκτες.

Αυτή η μέθοδος λειτουργεί ως ένας αποτελεσματικός **Μηχανισμός Ασφαλούς Αστοχίας (Fail-safe mechanism)**. Επιτρέπει στο σύστημα να «ανακαλύπτει» και να εξάγει επιτυχώς οντότητες (όπως τοπικά μουσεία ή εγχώριες αθλητικές ομάδες), ακόμη και αν η επίσημη οντολογία της DBpedia έχει παραλείψει να τις κατηγοριοποιήσει σωστά στον γράφημα. Μέσω του Σημασιολογικού Φιλτραρίσματος, το σύστημα γεφυρώνει το χάσμα μεταξύ δομημένων (RDF triplestore) και αδόμητων (plain text) δεδομένων, αυξάνοντας κατακόρυφα την ποσότητα και την ακρίβεια της πληροφορίας που παραδίδεται στον τελικό χρήστη.

5.3.2 ΔΥΝΑΜΙΚΉ ΚΑΤΑΣΚΕΥΉ ΣΥΝΔΕΣΜΩΝ ΚΑΙ ΠΟΛΥΜΕΣΩΝ

Για τον εμπλουτισμό των αποτελεσμάτων, έγινε εκτεταμένη χρήση του τελεστή OPTIONAL. Αυτό επέτρεψε την ανάκτηση μικρογραφιών εικόνων (dbo:thumbnail) όπου αυτές ήταν διαθέσιμες,

χωρίς να απορρίπτονται οι οντότητες που δεν διέθεταν εικόνα. Επιπλέον, υλοποιήθηκε δυναμική δημιουργία συνδέσμων χάρτη εντός του ίδιου του ερωτήματος SPARQL: χρησιμοποιώντας την εντολή BIND(CONCAT(...)), συνενώθηκαν οι γεωγραφικές συντεταγμένες (geo:lat και geo:long) με το βασικό URL του Google Maps, παράγοντας έτοιμους συνδέσμους πλοήγησης για το Frontend.

```
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

SELECT DISTINCT ?museum ?name ?image ?map ?info WHERE {
  ?museum rdf:type dbo:Museum .
  ?museum rdfs:label ?name .
  FILTER (LANG(?name) = 'el')

  ?museum dbo:abstract ?abstract .
  FILTER (LANG(?abstract) = 'el')

  FILTER (
    CONTAINS(LCASE(?abstract), "ελλάδα") ||
    CONTAINS(LCASE(?abstract), "αθήνα") ||
    CONTAINS(LCASE(?abstract), "θεσσαλονίκη") ||
    CONTAINS(LCASE(?abstract), "κρήτη") ||
    CONTAINS(LCASE(?abstract), "αρχαιολογικό")
  )
  BIND(?abstract AS ?info)

  # --- IMAGE & MAP ---
  OPTIONAL { ?museum dbo:thumbnail ?image }
  OPTIONAL {
    ?museum geo:lat ?lat ; geo:long ?lon .
    BIND(CONCAT("https://www.google.com/maps?q=", STR(?lat), ",", STR(?lon)) AS ?map)
  }
} LIMIT 50
```

Εικόνα 8 Απόσπασμα κώδικα SPARQL που υλοποιεί το Σημασιολογικό Φιλτράρισμα (Semantic Filtering) και τη δυναμική δημιουργία συνδέσμων Google Maps. Πηγή: Δημιουργία του συγγραφέα

Στο κώδικα της εικόνας 8 , Ο πυρήνας της αναζήτησης βασίζεται στην εντολή **FILTER**. Λόγω των ελλείψεων της οντολογίας στην ιδιότητα της τοποθεσίας, το σύστημα αναζητά οντότητες τύπου Μουσείου και ελέγχει το κείμενο της περίληψής τους. Χρησιμοποιώντας τον συνδυασμό των συναρτήσεων LCASE και **CONTAINS**, το ερώτημα εντοπίζει λέξεις-κλειδιά όπως “Ελλάδα", “Αθήνα” ή “Θεσσαλονίκη” μέσα στο ίδιο το κείμενο, διασώζοντας έτσι οντότητες που διαφορετικά δεν θα ανακτώνταν. Περιορίζει επίσης τα αποτελέσματα αυστηρά στην ελληνική γλώσσα μέσω της συνάρτησης lang(?abstract) = “el".

Γίνεται εκτεταμένη χρήση του τελεστή **OPTIONAL** για τις ιδιότητες της εικόνας και των γεωγραφικών συντεταγμένων. Αυτό διασφαλίζει ότι αν ένα μουσείο δεν διαθέτει φωτογραφία ή συντεταγμένες στη βάση δεδομένων, το ερώτημα δεν θα αποτύχει, αλλά απλώς θα επιστρέψει κενές τιμές για τις συγκεκριμένες στήλες.

Μία από τις σημαντικότερες λειτουργίες του ερωτήματος είναι η χρήση της εντολής **BIND**. Μέσω της συνάρτησης **CONCAT**, το σύστημα λαμβάνει τις μεταβλητές του γεωγραφικού πλάτους και μήκους, τις μετατρέπει σε συμβολοσειρές και τις συνενώνει δυναμικά με το βασικό URL του Google Maps. Ως αποτέλεσμα, η βάση δεδομένων δεν επιστρέφει απλούς αριθμούς, αλλά έτοιμους, διαδραστικούς υπερσυνδέσμους πλοήγησης προς χρήση από το Frontend.

5.4 ΥΛΟΠΟΙΗΣΗ ΔΙΕΠΑΦΗΣ ΧΡΗΣΤΗ

Το επίπεδο παρουσίασης (Frontend) αναπτύχθηκε με γνώμονα τη βέλτιστη Εμπειρία Χρήστη (UX) και τη διαδραστικότητα, μετατρέποντας το ακατέργαστο JSON αποτέλεσμα της βάσης δεδομένων σε μια λειτουργική και εκπαιδευτική διεπαφή.

5.4.1 ΕΠΕΞΕΡΓΑΣΤΗΣ ΕΡΩΤΗΜΑΤΩΝ ΜΕ ΜΗΧΑΝΙΣΜΌ ΑΝΑΤΡΟΦΟΔΌΤΗΣΗΣ ΣΦΑΛΜΆΤΩΝ

Σε παραδοσιακά περιβάλλοντα εκτέλεσης, ένα συντακτικό λάθος (όπως η παράλειψη μιας τελείας στο τέλος μιας τριπλέτας ή ένα ορθογραφικό λάθος σε ένα πρόθεμα) οδηγεί συχνά σε ακατέργαστα, δυσνόητα μηνύματα του διακομιστή ή σε σιωπηρή αποτυχία (silent failure), αφήνοντας τον τελικό χρήστη χωρίς καθοδήγηση.

Για την αντιμετώπιση αυτής της πρόκλησης, κατά την ανάπτυξη του υποσυστήματος εκτέλεσης ερωτημάτων (QueryEditor Component) στο επίπεδο παρουσίασης (React.js), σχεδιάστηκε και ενσωματώθηκε ένας προηγμένος μηχανισμός διαχείρισης εξαιρέσεων. Ο μηχανισμός αυτός δεν περιορίζεται στην απλή αποτροπή κατάρρευσης της εφαρμογής (crash prevention), αλλά προσομοιάζει τη διαδραστική συμπεριφορά ανατροφοδότησης ενός σύγχρονου μεταγλωττιστή (Compiler) ή ενός Ολοκληρωμένου Περιβάλλοντος Ανάπτυξης (IDE).

Η αρχιτεκτονική του μηχανισμού εκτείνεται σε όλο το μήκος της εφαρμογής (End-to-End Error Handling). Όταν ο χρήστης εκτελεί ένα συντακτικά ή λογικά λανθασμένο ερώτημα, η διαδικασία εξέλιξης του σφάλματος διαχειρίζεται ως εξής:

1. **Στο επίπεδο Δεδομένων:** Ο κινητήρας του Virtuoso αποτυγχάνει κατά τη φάση της συντακτικής ανάλυσης (parsing phase) του SPARQL ερωτήματος και επιστρέφει έναν κωδικό σφάλματος επιπέδου 400 (Bad Request) ή 500 (Internal Server Error), συνοδευόμενο από το τεχνικό μήνυμα (π.χ. *"SPARQL compiler, line 4: syntax error"*).
2. **Διαμεσολάβηση (Backend):** Το ενδιάμεσο επίπεδο της Go συλλαμβάνει αυτή την αποτυχία μέσω ρητού ελέγχου σφαλμάτων. Αντί να τερματίσει τη σύνδεση, εξάγει το ακριβές μήνυμα του Virtuoso, το ενθυλακώνει σε μια δομημένη απάντηση JSON (Error Payload) και το προωθεί με ασφάλεια στο Frontend, αποτρέποντας τη διαρροή ευαίσθητων πληροφοριών του διακομιστή.
3. **Διαχείριση Εξαιρέσεων (Frontend):** Στην πλευρά του πελάτη, η βιβλιοθήκη δικτύου Axios αναγνωρίζει τον κωδικό σφάλματος HTTP και ενεργοποιεί το μπλοκ σύλληψης εξαιρέσεων (catch block) της ασύγχρονης συνάρτησης υποβολής (try...catch). Ο κώδικας εξάγει το ακριβές κειμενικό μήνυμα από το αντικείμενο του σφάλματος.
4. **Δυναμική Ενημέρωση Διεπαφής (UI Reactivity):** Το περιβάλλον χρήστη αντιδρά ακαριαία μέσω της αλλαγής της Κατάστασης (State) του component στη React (π.χ. setError(message)). Το Εικονικό DOM (Virtual DOM) επανασχεδιάζει τη διεπαφή: το πλαίσιο εισαγωγής κειμένου (editor) επισημαίνεται δυναμικά με κόκκινο περίγραμμα (μέσω υπολογισμένων κλάσεων CSS), ενώ ακριβώς από κάτω προβάλλεται ένα ευκρινές προειδοποιητικό πλαίσιο (alert banner) που περιέχει το ακριβές σημείο και την αιτία του συντακτικού σφάλματος.

Αυτή η λειτουργία προσδίδει στο σύστημα υψηλή εκπαιδευτική και παιδαγωγική αξία. Ο λαμβάνει άμεση, στοχευμένη και οπτικά σαφή πληροφορία για το τι έκανε λάθος. Έτσι, η εφαρμογή δεν λειτουργεί μόνο ως εργαλείο ανάκτησης δεδομένων, αλλά και ως ένα ενεργό περιβάλλον εκμάθησης (learning environment) που καθοδηγεί τον προγραμματιστή στη σωστή σύνταξη των ερωτημάτων SPARQL.

5.4.2 ΔΥΝΑΜΙΚΉ ΟΠΤΙΚΟΠΟΊΗΣΗ ΔΕΔΟΜΈΝΩΝ

Η προβολή των αποτελεσμάτων υλοποιήθηκε μέσω του Component ResultsTable, το οποίο είναι πλήρως δυναμικό. Οι στήλες του πίνακα δημιουργούνται αυτόματα βάσει των μεταβλητών (vars)

που επιστρέφει το αρχείο JSON του SPARQL ερωτήματος, επιτρέποντας την εκτέλεση οποιουδήποτε αυθαίρετου ερωτήματος από τον χρήστη.

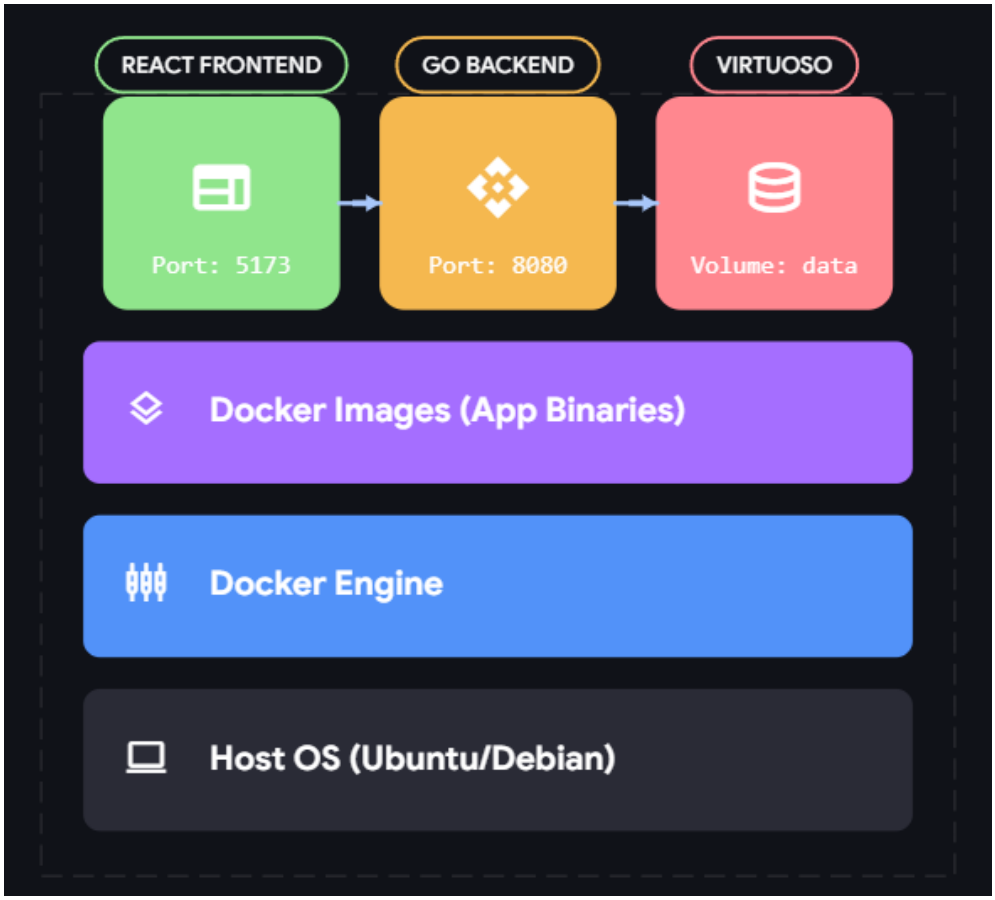
Παράλληλα, εφαρμόστηκαν εξειδικευμένοι αποδοτήρες κελιών (Custom Cell Renderers):

Μικρογραφίες (Image Cells): Ανιχνεύουν αν ο σύνδεσμος οδηγεί σε εικόνα (ή αν η στήλη ονομάζεται "image") και προβάλλουν την εικόνα <img> με διαδραστικά εφέ μεγέθυνσης κατά την επικάλυψη (hover).

Χάρτες (Map Cells): Ανιχνεύουν τα δυναμικά URLs που κατασκευάστηκαν στη SPARQL και τα αντικαθιστούν με ένα γραφικό εικονίδιο (Pin), το οποίο με κλικ ανοίγει νέα καρτέλα στο Google Maps.

Διαχείριση Κειμένου: Για περιλήψεις (abstracts) μεγάλου μήκους, εφαρμόζεται αποκοπή του κειμένου (truncation) στους 80 χαρακτήρες, με την προσθήκη κουμπιού "[+] More" για τη δυναμική επέκταση της πληροφορίας, διατηρώντας τον πίνακα τακτοποιημένο.

Συνοπτικά, η υλοποίηση του συστήματος συνδύασε έξυπνες τεχνικές Backend (Ταυτοχρονισμός), καινοτόμες πρακτικές σύνταξης ερωτημάτων (Semantic Text Filtering) και σύγχρονη ανάπτυξη Frontend, καταλήγοντας σε μια ολοκληρωμένη και αποδοτική Σημασιολογική Εφαρμογή.



Εικόνα 9 Αρχιτεκτονική της εφαρμογής εικονοποίησης (containerization). Παρουσιάζεται η απομόνωση των υπηρεσιών σε διακριτά Docker Containers και η μεταξύ τους δικτύωση μέσω του Docker Engine. Πηγή: Δημιουργία του συγγραφέα

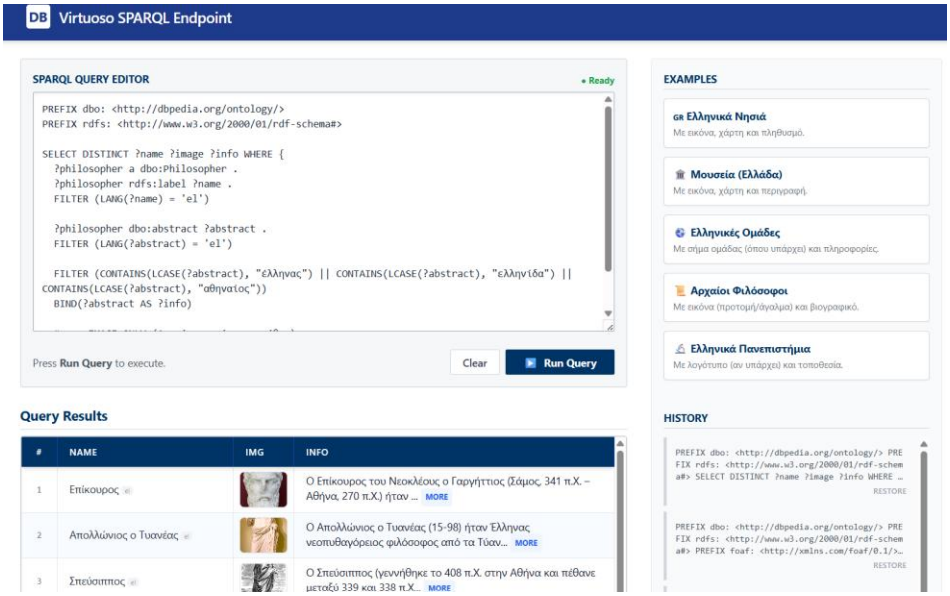
ΚΕΦΑΛΑΙΟ 6: ΑΠΟΤΕΛΕΣΜΑΤΑ ΚΑΙ ΑΞΙΟΛΟΓΗΣΗ

6.1 ΕΙΣΑΓΩΓΗ ΚΕΦΑΛΑΙΟΥ

Στα προηγούμενα κεφάλαια αναλύθηκε το θεωρητικό υπόβαθρο, η αρχιτεκτονική σχεδίαση και η τεχνική υλοποίηση του συστήματος (backend και frontend). Στο παρόν κεφάλαιο, παρουσιάζονται τα τελικά αποτελέσματα της εργασίας. Αρχικά, γίνεται μια πλήρης οπτική παρουσίαση της διεπαφής χρήστη (User Interface), αναδεικνύοντας τις λειτουργίες του συστήματος. Στη συνέχεια, πραγματοποιείται ποσοτική αξιολόγηση απόδοσης (Performance Evaluation), συγκρίνοντας τον χρόνο απόκρισης του συστήματος με και χωρίς τη χρήση μηχανισμών ταυτοχρονισμού. Τέλος, παρατίθενται δυο σενάρια χρήσης (Use Cases) που αποδεικνύουν την πρακτική αξία της στρατηγικής του σημασιολογικού φιλτραρίσματος.

6.2 ΠΑΡΟΥΣΙΑΣΗ ΔΙΕΠΑΦΗΣ ΧΡΉΣΤΗ (USER INTERFACE)

Η εφαρμογή αναπτύχθηκε ως Εφαρμογή Μονοσέλιδου (SPA - Single Page Application) με έμφαση στη μινιμαλιστική σχεδίαση και τη βέλτιστη εμπειρία χρήστη (UX), χρησιμοποιώντας το πλαίσιο Tailwind CSS. Η κεντρική οθόνη χωρίζεται σε τρεις διακριτές ενότητες λειτουργικότητας.

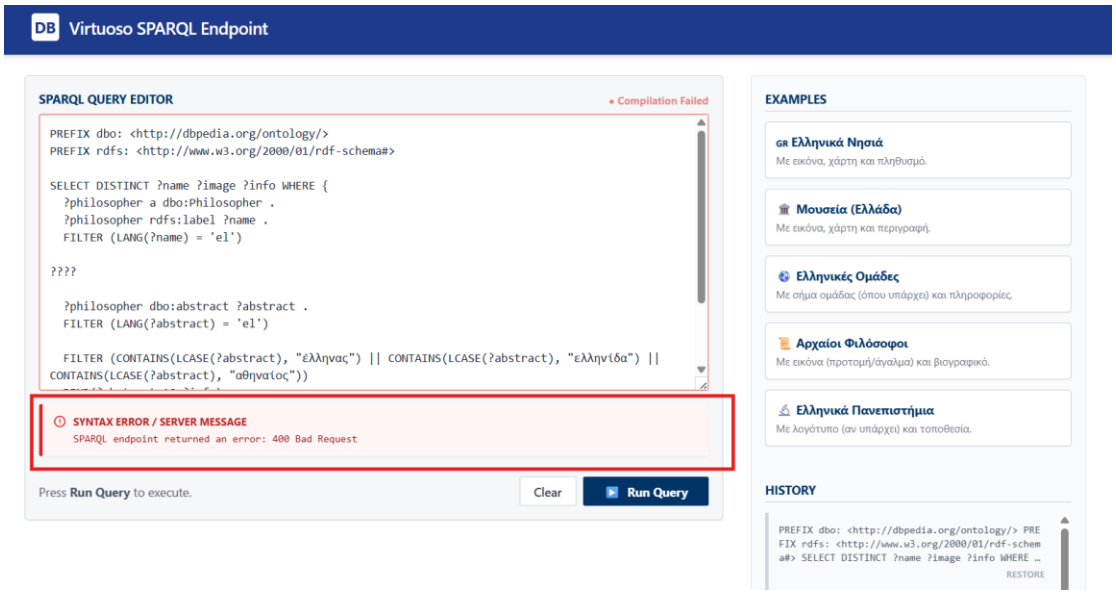


Εικόνα 10 Η κεντρική οθόνη απο διεπαφή

6.2.2 ΕΠΕΞΕΡΓΑΣΤΗΣ ΕΡΩΤΗΜΑΤΩΝ ΚΑΙ ΔΙΑΧΕΙΡΙΣΗ ΣΦΑΛΜΑΤΩΝ

Στον διαδραστικό επεξεργαστή κειμένου (SPARQL Editor , χρήστης μπορεί να πληκτρολογήσει ελεύθερα συντακτικό κώδικα SPARQL ή να επιλέξει από τα έτοιμα ερωτήματα (Predefined Queries). Ιδιαίτερη έμφαση δόθηκε στη διαχείριση σφαλμάτων. Όπως φαίνεται στην Εικόνα 11, εάν ο χρήστης παραλείψει έναν χαρακτήρα ή γράψει λάθος συντακτικό, το περιβάλλον (αντί να καταρρεύσει) προβάλλει ένα κόκκινο πλαίσιο ειδοποίησης. Μέσα σε αυτό, εμφανίζεται αυτούσιο το

μήνυμα σφάλματος του "Compiler" του Virtuoso, λειτουργώντας ως ένας μεσο αποσφαλμάτωσης για τον χρήστη.

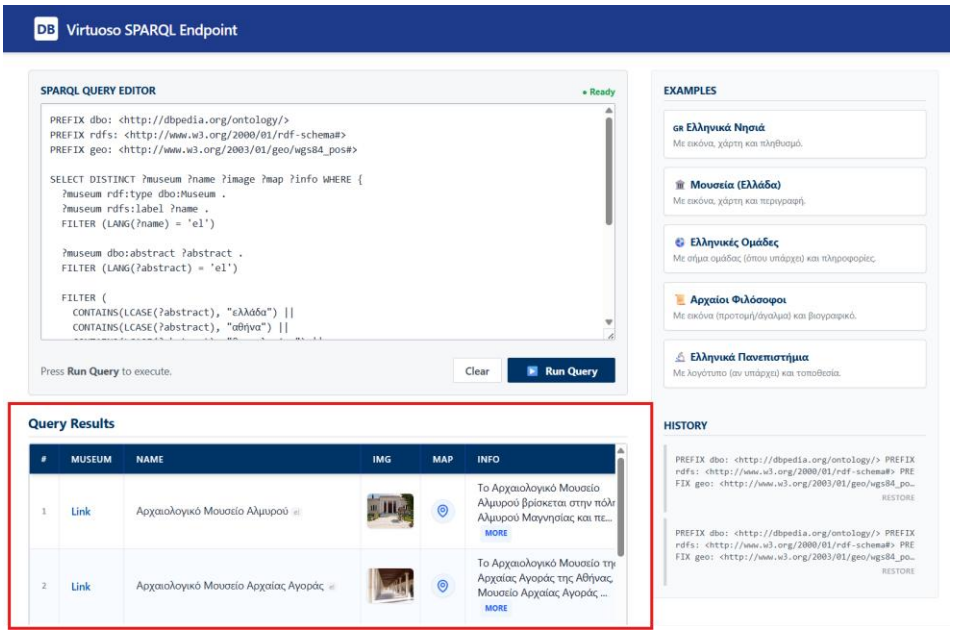


Εικόνα 11 : Αντίδραση της διεπαφής σε εσφαλμένη σύνταξη ερωτήματος (Compiler Error Feedback).

6.2.3 ΟΠΤΙΚΟΠΟΊΗΣΗ ΑΠΟΤΕΛΕΣΜΑΤΩΝ ΚΑΙ ΠΟΛΥΜΈΣΩΝ

Ο πίνακας αποτελεσμάτων προσαρμόζεται δυναμικά στις μεταβλητές του εκάστοτε ερωτήματος. Στην Εικόνα 12 απεικονίζεται η εκτέλεση ενός ερωτήματος αναζήτησης Ελληνικών Μουσείων. Όπως παρατηρείται:

1. Στη στήλη "Image, δεν εμφανίζεται το απλό URI (κείμενο), αλλά η εικόνα (rendered image).
2. Στη στήλη "Map", οι γεωγραφικές συντεταγμένες έχουν μετατραπεί σε διαδραστικούς συνδέσμους Google Maps, επιτρέποντας τον γεωγραφικό εντοπισμό της οντότητας.Εαν ο χρήστης πατησει επάνω στο link URL θα μεταβεί στο χαρτη google maps με την τοποθεσια.
3. Στη στήλη της περιγραφής (Info), τα εκτενή κείμενα περικόπτονται αυτόματα, διατηρώντας το ύψος του πίνακα σε λογικά πλαίσια.



Εικόνα 12 : Απο διεπαφη

6.2.4 ΑΣΎΓΧΡΟΝΗ ΚΑΤΑΜΈΤΡΗΣΗ ΔΕΔΟΜΈΝΩΝ

Οταν ένας χρήστης εκτελεί ένα ερώτημα αναζήτησης με περιορισμό αποτελεσμάτων (π.χ. LIMIT 50), η γνώση του συνολικού όγκου των δεδομένων που ικανοποιούν τα κριτήρια (Total in DB) είναι απαραίτητη για τη γενικότερη κατανόηση της πληροφορίας. Μια παραδοσιακή σειριακή (ένα-προς-ένα) εκτέλεση θα απαιτούσε δύο διαδοχικά αιτήματα (ένα για τα δεδομένα SELECT και ένα για την καταμέτρηση COUNT), διπλασιάζοντας τον χρόνο αναμονής του χρήστη. Προκειμένου να βελτιστοποιηθεί η απόδοση, το σύστημα υλοποιεί το αρχιτεκτονικό μοτίβο ταυτοχρονίας Scatter-Gather.Μέσω του ενδιάμεσου λογισμικού τη στιγμή της υποβολής ενός SELECT ερωτήματος, το σύστημα αναλύει δυναμικά (parse) τον κώδικα SPARQL. Διαχωρίζει τα προθέματα (PREFIXES), αφαιρεί λέξεις-κλειδιά όπως LIMIT και ORDER BY, και κατασκευάζει ένα δεύτερο ερώτημα καταμέτρησης. Στη συνέχεια, χρησιμοποιώντας ανεξάρτητα νήματα εκτέλεσης (Goroutines) και τον μηχανισμό συγχρονισμού sync.WaitGroup, στέλνει τα δύο ερωτήματα προς τον Virtuoso ταυτόχρονα.Η ταυτόχρονη εκτέλεση εγγυάται ότι ο χρόνος απόκρισης του συστήματος ισούται με τον χρόνο του πιο αργού ερωτήματος, και όχι με το άθροισμά τους. Τα αποτελέσματα συγκεντρώνονται (Gather), ενσωματώνονται στο τελικό JSON απάντησης μαζί με τον χρόνο εκτέλεσης (latency), και παρουσιάζονται ακαριαία στη διεπαφή (React UI), βελτιώνοντας δραματικά την εμπειρία χρήσης.

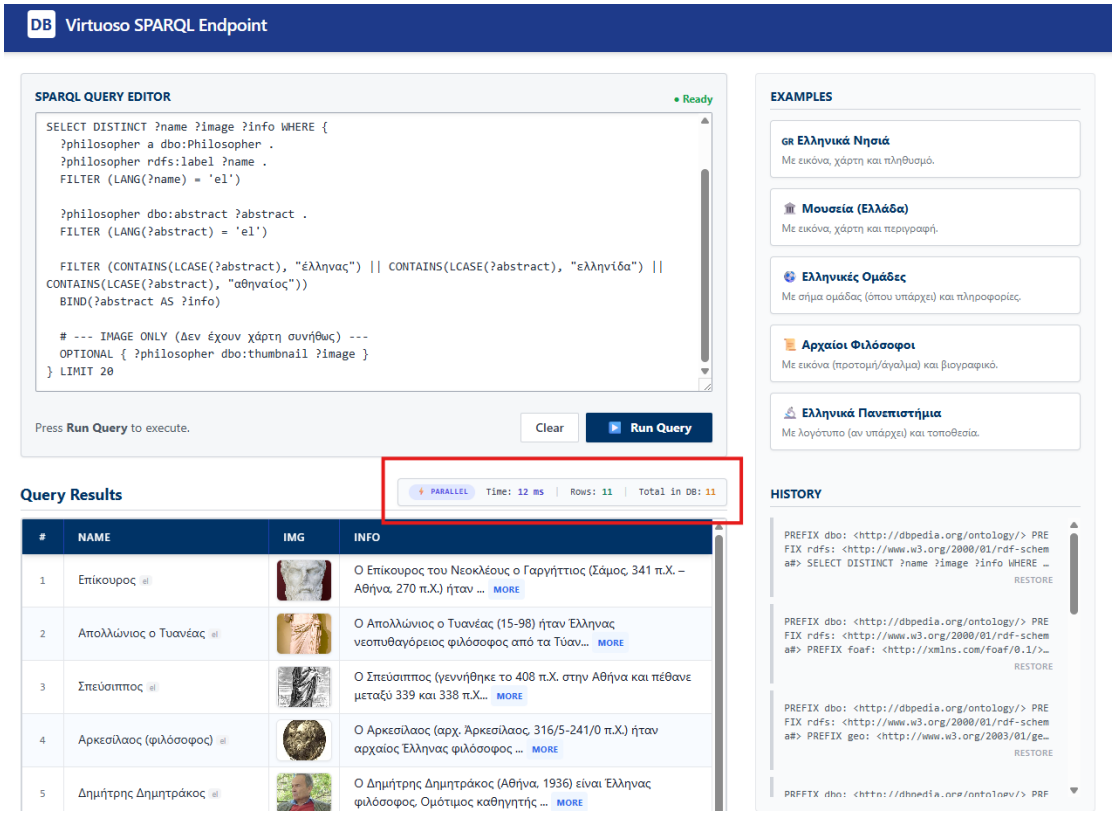
Ο υπολογισμός του συνολικού πλήθους (COUNT) σε πολύπλοκα ερωτήματα, ειδικά όταν περιέχουν εκφράσεις κανονικών μορφών (REGEX) ή ελεύθερη αναζήτηση κειμένου σε εκατομμύρια τριπλέτες, απαιτεί εκτεταμένους υπολογιστικούς πόρους. Εάν τέτοια ερωτήματα αφεθούν ανεξέλεγκτα, δύνανται να προκαλέσουν σημαντικές καθυστερήσεις, δεσμεύοντας πόρους του διακομιστή και υποβαθμίζοντας την εξυπηρέτηση άλλων χρηστών.Για την εξασφάλιση της Ανοχής σε Σφάλματα (Fault Tolerance), το σύστημα ενσωματώνει έναν προηγμένο μηχανισμό «Ασπίδας Χρόνου» (Timeout Guard) στο backend. Η λειτουργία του μηχανισμού βασίζεται στη θεωρία των κατανεμημένων συστημάτων και λειτουργεί ως εξής:

1. Ασύγχρονη Παρακολούθηση: Το κύριο νήμα (main thread) του διακομιστή δεν αναμένει επ' άπειρον την ολοκλήρωση του sync.WaitGroup. Αντίθετα, η αναμονή μεταφέρεται σε ένα

ανεξάρτητο νήμα, το οποίο επικοινωνεί με το κύριο πρόγραμμα μέσω Καναλιών διεπαφής (Go Channels).

2. Επιβολή Χρονικού Ορίου (Timeout): Ενεργοποιείται ταυτόχρονα ένας χρονοδιακόπτης μέσω της δομής ελέγχου select (timer timeout). Εάν η διεργασία του παρασκηνίου (COUNT) υπερβεί το ασφαλές όριο των πέντε (5) δευτερολέπτων, το σύστημα διακόπτει αυτόματα την αναμονή.
3. Ομαλή Υποβάθμιση (Graceful Degradation): Σε περίπτωση ενεργοποίησης του χρονικού ορίου, η εφαρμογή δεν καταρρέει (crash). Ο διακομιστής επιστρέφει άμεσα τα βασικά δεδομένα (τα οποία εξάγονται ταχύτατα) και ενημερώνει το Front-end (React) για την αδυναμία ολοκλήρωσης της καταμέτρησης.

Σε επίπεδο διεπαφής χρήστη (UI), όταν συμβεί το παραπάνω σενάριο, ο χρήστης βλέπει κανονικά τον πίνακα των αποτελεσμάτων του. Ωστόσο, στον πίνακα στατιστικών (badge), στη θέση του συνολικού αριθμού εμφανίζεται το προειδοποιητικό μήνυμα *"Αδυναμία επιστροφής συνολικού όγκου (Timeout)"*. Με αυτόν τον τρόπο, το σύστημα διασφαλίζει τη μέγιστη δυνατή αποκρισιμότητα (responsiveness) προστατεύοντας παράλληλα την υποδομή.



Εικονα 13 :Απο διεπαφη

6.2.5 ΚΑΡΤΕΛΑ ΙΣΤΟΡΙΚΟΥ (QUERY HISTORY)

Το Ιστορικό Ερωτημάτων βελτιώνει την εμπειρία του χρήστη και τη σχέση του με το σύστημα. Εναι ένα σημαντικό μέσο για την εμπειρία του χρήστη με σκοπό να μην χρειάζεται ο χρήστης να γράφει ξανά σύνθετα ερωτήματα SPARQL. Ο λόγος της συγκεκριμένης δυνατότητας είναι να απαλλάξει τον χρήστη από την ανάγκη επανεισαγωγής πολύπλοκων ερωτημάτων SPARQL, επιτρέποντας την άμεση ανάκληση και επανεκτέλεση προηγούμενων αναζητήσεων.

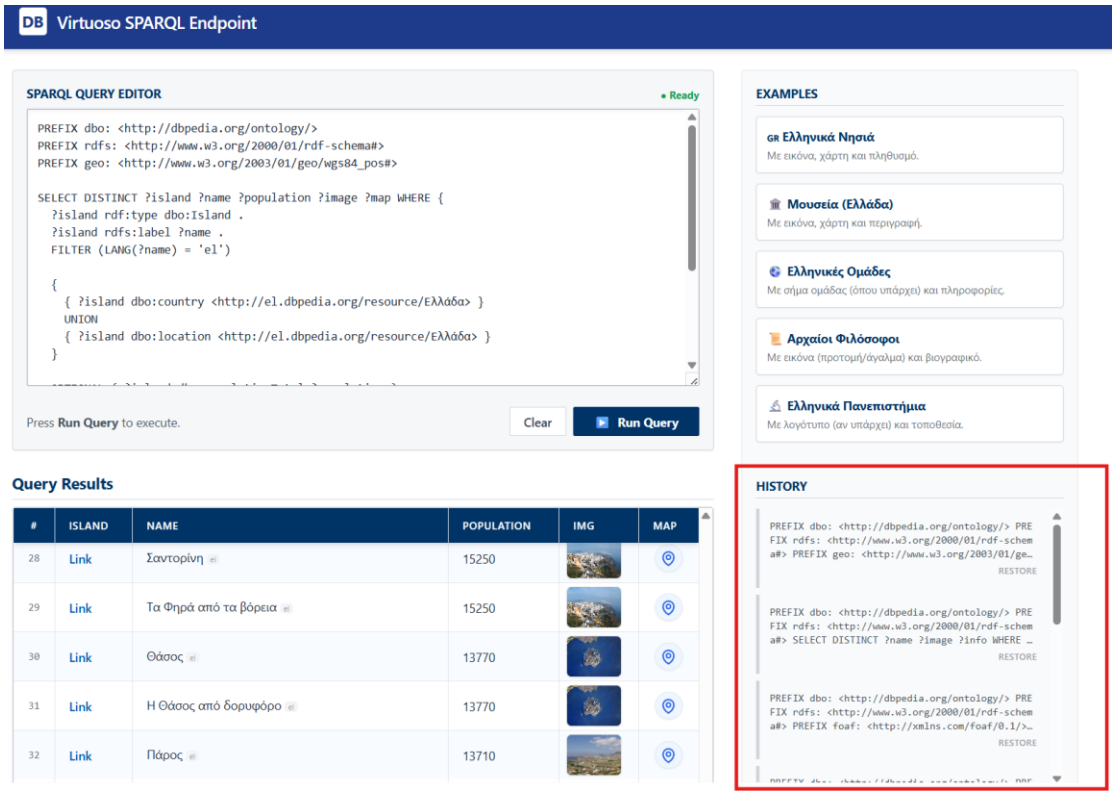
Εμφανίζεται σαν μια ξεχωριστή κατακόρυφη λίστα στο δεξί τμήμα της εφαρμογής, κάτω από τα έτοιμα παραδείγματα. Κάθε στοιχείο της λίστας είναι ένα συντακτικά ορθό ερώτημα που εκτελέστηκε

επιτυχώς πριν. Όταν ο χρήστης κάνει κλικ σε μια προηγούμενη εγγραφή, ο κώδικας πηγαίνει αμέσως στον κεντρικό επεξεργαστή κειμένου (Query Editor). Εκεί ο κώδικας είναι έτοιμος για αλλαγή ή για νέα εκτέλεση.

Σε επίπεδο τεχνικής υλοποίησης, ο μηχανισμός του ιστορικού αξιοποιεί τη δυνατότητα Τοπικής Αποθήκευσης (Local Storage) του σύγχρονου φυλλομετρητή (Web Browser). Αυτή η προσέγγιση προσδίδει τα εξής σημαντικά πλεονεκτήματα στην εφαρμογή:

1. **Διατήρηση Κατάστασης (State Persistence):** Το ιστορικό δεν χάνεται όταν ο χρήστης ανανεώνει τη σελίδα (page refresh) ή κλείνει τον φυλλομετρητή, καθώς τα δεδομένα διατηρούνται τοπικά στη μνήμη της συσκευής του και όχι στην πτητική μνήμη της εφαρμογής (RAM).
2. **Αποδοτικότητα και Περιορισμός Δεδομένων:** Για να αποφευχθεί η υπερφόρτωση της διεπαφής και η άσκοπη κατανάλωση πόρων, η δομή δεδομένων (ουρά) του ιστορικού έχει σχεδιαστεί ώστε να διατηρεί αυστηρά τα πιο πρόσφατα ερωτήματα (π.χ. τις τελευταίες 10 εκτελέσεις). . Κάθε φορά που προστίθεται μια νέα εκτέλεση, η ουρά αφαιρεί το παλιό ερώτημα αυτόματα, διατηρώντας το μέγεθος του ιστορικού σταθερό.
3. **Αποφυγή Διπλοτύπων:** Ο κώδικας του συστήματος ελέγχει δυναμικά το ερώτημα πριν την αποθήκευσή του, διασφαλίζοντας ότι διαδοχικές εκτελέσεις του ίδιου ακριβώς κώδικα δεν δημιουργούν διπλές εγγραφές στη λίστα.

Μέσω αυτής της δυνατότητας, η εφαρμογή μετατρέπεται από ένα απλό σημείο εκτέλεσης (endpoint) σε ένα ολοκληρωμένο, φιλικό και διαδραστικό εργαλείο εξερεύνησης (exploration tool) των διασυνδεδεμένων δεδομένων της Ελληνικής DBpedia.



Εικόνα 14: Απο διεπαφη

6.3 ΑΞΙΟΛΌΓΗΣΗ ΑΠΌΔΟΣΗΣ

6.3.1 ΜΕΘΟΔΟΛΟΓΊΑ ΚΑΙ ΑΛΓΟΡΙΘΜΙΚΉ ΠΡΟΣΈΓΓΙΣΗ

Η αξιολόγηση της ταχύτητας του συστήματος βασίστηκε στη σύγκριση του παραδοσιακού σειριακού μοντέλου με το ασύγχρονο μοντέλο ταυτοχρονίας (Scatter-Gather) που υλοποιήθηκε στο Golang Backend. Όπως αναλύθηκε στο Κεφάλαιο 6.2.4, κάθε αίτημα του χρήστη διασπάται δυναμικά σε δύο υπο-ερωτήματα: το κύριο ερώτημα ανάκτησης δεδομένων (T_{select}) και το ερώτημα καταμέτρησης συνολικού όγκου (T_{count}).
Σε ένα συμβατικό API δομημένο με Σειριακή Εκτέλεση (Sequential Execution), ο διακομιστής αναμένει την ολοκλήρωση της ανάκτησης των δεδομένων πριν αποστείλει το αίτημα καταμέτρησης. Ο συνολικός χρόνος απόκρισης υπολογίζεται ως το άθροισμα των επιμέρους χρόνων:

$$T_{total} = T_{select} + T_{count} + Overhead$$

Αντιθέτως, στην παρούσα υλοποίηση χρησιμοποιήθηκε το μοντέλο ταυτοχρονισμού της Golang μέσω ανεξάρτητων νημάτων (Goroutines). Εφόσον τα δύο ερωτήματα αποστέλλονται στη βάση δεδομένων Virtuoso ταυτόχρονα, ο συνολικός θεωρητικός χρόνος καθορίζεται από το πιο αργό ερώτημα:

$$T_{total} = \max(T_{select}, T_{count}) + Overhead$$

Επιπρόσθετα, η αξιολόγηση περιλαμβάνει τον μηχανισμό Timeout Guard. Λόγω της φύσης του Σημασιολογικού Ιστού, η πράξη T_{count} μπορεί να απαιτήσει σάρωση εκατομμυρίων τριπλετών. Ο αλγόριθμος θέτει ένα άνω όριο (threshold) 5 δευτερολέπτων. Έτσι, αν T_{count} > 5000 ms, ενεργοποιείται η Ομαλή Υποβάθμιση (Graceful Degradation) και ο συνολικός χρόνος "κλειδώνει" στο όριο του χρονοδιακόπτη:

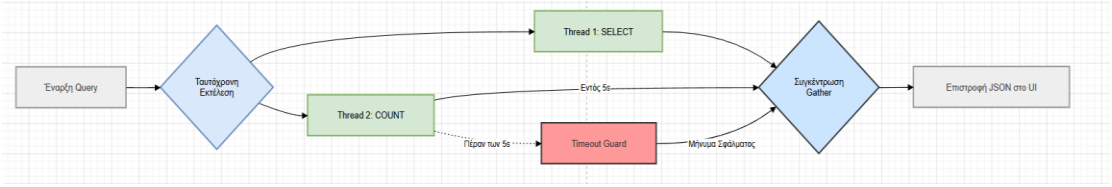
$$T_total = 5000 ms + Overhead$$

6.3.2 ΣΎΓΚΡΙΣΗ ΣΕΙΡΙΑΚΉΣ ΚΑΙ ΠΑΡΆΛΛΗΛΗΣ ΕΚΤΈΛΕΣΗΣ

Για την πρακτική επαλήθευση του παραπάνω μοντέλου, πραγματοποιήθηκαν συγκριτικές μετρήσεις στον Query Editor. Το σενάριο δοκιμής περιλάμβανε ένα σύνθετο ερώτημα SELECT με LIMIT 50 και ταυτόχρονο COUNT σε περιβάλλον Docker (Localhost).

- **Σειριακή Εκτέλεση (Sequential):**
  - Το backend εκτελεί το SELECT και στη συνέχεια το COUNT.
  - **Μέσος Χρόνος:** ~90 – 115 ms.
- **Παράλληλη Εκτέλεση (Scatter-Gather):**
  - Το σύστημα εκτελεί ταυτόχρονα τα δύο υπο-ερωτήματα μέσω sync.WaitGroup.
  - **Μέσος Χρόνος:** ~45 – 50 ms.

Η βελτίωση του χρόνου απόκρισης κατά περίπου **55-60%** επιβεβαιώνει την αξία της ταυτοχρονίας. Στην παράλληλη υλοποίηση, η καθυστέρηση περιορίζεται στον χρόνο του πιο αργού ερωτήματος, ενώ ο χρήστης προστατεύεται από το **Timeout Guard** σε περίπτωση που ο υπολογισμός του συνόλου υπερβεί τα 5 δευτερόλεπτα. Η προσέγγιση αυτή προσφέρει μια εξαιρετικά αποκρίσιμη (responsive) διεπαφή, ιδανική για εξερεύνηση μεγάλων συνόλων δεδομένων (Big Data).



Εικόνα 13 Ταυτοχρονισμός συστήματος
Πηγή: Δημιουργία του συγγραφέα

6.3.3 Αξιολόγηση Επιπέδου Παρουσίασης (Frontend Rendering Performance)

Η συνολική εμπειρία του τελικού χρήστη δεν καθορίζεται αποκλειστικά από την ταχύτητα του Backend, αλλά και από τον χρόνο που απαιτείται για την οπτική απόδοση (rendering) των δεδομένων στον φυλλομετρητή (Browser). Τα ερωτήματα SPARQL συχνά επιστρέφουν μεγάλα σε όγκο αρχεία JSON (payloads), τα οποία περιέχουν δεκάδες εγγραφές, πολυπλοκότητες κειμένου και συνδέσμους πολυμέσων.

Η αξιολόγηση του επιπέδου παρουσίασης (React.js) κατέδειξε εξαιρετική συμπεριφορά κατά τη διαχείριση αυτών των δεδομένων. Όταν το JSON φτάνει στο Frontend, η αρχιτεκτονική του Εικονικού DOM (Virtual DOM) της React αναλαμβάνει την επεξεργασία του:

1. **Δυναμική Δημιουργία Κόμβων (Node Creation):** Το σύστημα διατρέχει τον πίνακα αποτελεσμάτων (array mapping) και κατασκευάζει δυναμικά τις γραμμές του Πίνακα Αποτελεσμάτων (ResultsTable) στη μνήμη, αντί να επεμβαίνει απευθείας στο DOM του φυλλομετρητή.
2. **Ασύγχρονη Φόρτωση Πολυμέσων (Lazy Image Loading):** Κατά την ανίχνευση ιδιοτήτων όπως το dbo:thumbnail, η εφαρμογή δημιουργεί ετικέτες <img> οι οποίες φορτώνουν τις εικόνες ασύγχρονα. Αυτό σημαίνει ότι ο πίνακας κειμένου εμφανίζεται ακαριαία στον χρήστη, χωρίς να αναμένει τη λήψη των βαριών αρχείων εικόνας από το Wikimedia Commons.

Το αποτέλεσμα είναι μια ομαλή διεπαφή που αποτρέπει το "πάγωμα" της οθόνης (UI blocking), επιτρέποντας στον χρήστη να εξερευνά, να σκρολάρει και να αλληλεπιδρά με τα σημασιολογικά αποτελέσματα σε πραγματικό χρόνο, επιβεβαιώνοντας την επιτυχή ολοκλήρωση των αρχιτεκτονικών στόχων της εφαρμογής.

6.4 ΣΕΝΆΡΙΑ ΧΡΉΣΗΣ (USE CASES) ΚΑΙ ΣΗΜΑΣΙΟΛΟΓΙΚΌ ΦΙΛΤΡΆΡΙΣΜΑ

Η τελική φάση της αξιολόγησης του συστήματος δεν περιορίζεται αποκλειστικά στις μετρικές απόδοσης του διακομιστή (ταχύτητα και ταυτοχρονία), αλλά επεκτείνεται στην ποιοτική αξιολόγηση των ανακτώμενων πληροφοριών. Ο Σημασιολογικός Ιστός και τα Ανοικτά Διασυνδεδεμένα Δεδομένα (Linked Open Data) συχνά υποφέρουν από το πρόβλημα της ελλιπούς πληροφορίας (data incompleteness). Η παρούσα ενότητα εξετάζει, μέσα από πραγματικά Σενάρια Χρήσης (Use Cases), την ικανότητα της εφαρμογής να αντισταθμίζει αυτές τις δομικές ελλείψεις και να γεφυρώνει το χάσμα μεταξύ μηχανής και τελικού χρήστη.

Σενάριο Χρήσης 1: Αντιμετώπιση Δομικών Ελλείψεων μέσω Σημασιολογικού Φιλτραρίσματος Κειμένου

Το Πρόβλημα: Σε ένα ιδεατό Γράφημα Γνώσης, η ανάκτηση των ελληνικών μουσείων θα απαιτούσε ένα απλό, "παραδοσιακό" ερώτημα SPARQL, το οποίο θα ζητούσε όλες τις οντότητες τύπου Μουσείου (dbo:Museum) των οποίων η ιδιότητα "χώρα" (dbo:country) έχει την τιμή "Ελλάδα" (dbr:Greece). Ωστόσο, στην πραγματικότητα της Ελληνικής DBpedia, η εκτέλεση αυτού του αυστηρού συντακτικού ερωτήματος επιστρέφει πενιχρά αποτελέσματα (λιγότερες από 10 οντότητες). Αυτό

συμβαίνει διότι κατά τη διαδικασία εξαγωγής των δεδομένων (extraction), η πλειοψηφία των άρθρων δεν διέθετε ρητά δομημένη τη συγκεκριμένη ιδιότητα (missing semantic tags) στα Infoboxes τους.

Η Λύση και το Αποτέλεσμα: Η εφαρμογή, προκειμένου να βελτιστοποιήσει την **Ανάκληση (Recall)** των αποτελεσμάτων (δηλαδή τον αριθμό των σχετικών εγγραφών που ανακτώνται με επιτυχία), αξιοποιεί τον μηχανισμό των "Προκαθορισμένων Ερωτημάτων" (Predefined Queries). Αντί να βασίζεται αποκλειστικά σε ρητές οντολογικές συνδέσεις, το σύστημα αναζητά όλες τις οντότητες τύπου Μουσείου και εφαρμόζει Σημασιολογικό Φιλτράρισμα (Semantic Filtering) πάνω στην ιδιότητα του ελεύθερου κειμένου/περίληψης (dbo:abstract). Μέσω της χρήσης Κανονικών Εκφράσεων (Regular Expressions) στη SPARQL, σαρώνει τα κείμενα για λέξεις-κλειδιά όπως "Ελλάδα", "Αθήνα" ή "ελληνικό".

Ως αποτέλεσμα (όπως επιβεβαιώνεται και από την αντίστοιχη οπτική αποτύπωση στην Εικόνα 10), επιστρέφονται επιτυχώς δεκάδες σχετικές οντότητες (π.χ. "Εθνικό Αρχαιολογικό Μουσείο", "Μουσείο Ακρόπολης"). Παρότι οι οντότητες αυτές αποτελούσαν "ορφανά" σημασιολογικά δεδομένα (μη συνδεδεμένα ρητά με τον κόμβο της χώρας), αναγνωρίστηκαν ορθά λόγω του έξυπνου αλγορίθμου αναζήτησης εντός του αδόμητου κειμένου. Αυτό αποδεικνύει την ικανότητα του συστήματος να λειτουργεί υβριδικά, συνδυάζοντας την αυστηρή λογική του Γραφήματος με τεχνικές Επεξεργασίας Φυσικής Γλώσσας, παραδίδοντας στον χρήστη ουσιαστική και ολοκληρωμένη πληροφορία.

Σενάριο Χρήσης 2: Γεφύρωση του Σημασιολογικού Χάσματος (Οπτικοποίηση Δεδομένων)

Το Πρόβλημα: Η πληροφορία στον σύγχρονο Παγκόσμιο Ιστό δεν είναι πλέον μονοδιάστατη και κειμενική, αλλά έντονα πολυμεσική (multimodal) και γεωχωρική (geospatial). Εάν ένας απλός χρήστης επιχειρήσει να εκτελέσει ένα ερώτημα σε ένα κλασικό τερματικό σημείο SPARQL (π.χ. στο προεπιλεγμένο περιβάλλον του Virtuoso Faceted Browser), η βάση δεδομένων θα επιστρέψει την πληροφορία στην ακατέργαστη μορφή της: οι εικόνες επιστρέφονται ως απλά αλφαριθμητικά URL συνδέσμων (raw string URLs) και τα γεωγραφικά δεδομένα ως στεγνές συντεταγμένες (π.χ. POINT(23.72 37.97)). Αυτή η προσέγγιση απαιτεί από τον χρήστη να αντιγράψει χειροκίνητα κάθε σύνδεσμο και να τον ανοίξει σε νέα καρτέλα για να δει την εικόνα, δημιουργώντας ένα τεράστιο εμπόδιο ευχρηστίας.

Η Λύση και το Αποτέλεσμα: Στην παρούσα εφαρμογή, το επίπεδο παρουσίασης (React.js UI) έχει σχεδιαστεί ώστε να ερμηνεύει τη σημασιολογία των δεδομένων σε πραγματικό χρόνο. Όπως αναλύθηκε στην περιγραφή της αρχιτεκτονικής (Κεφάλαιο 5), η διεπαφή αναλύει δυναμικά τα αποτελέσματα κατά τη διαδικασία της απόδοσης (rendering) του πίνακα:

- **Οπτικοποίηση Πολυμέσων:** Εάν εντοπιστεί ένα URI που οδηγεί σε εικόνα (μέσω της ιδιότητας dbo:thumbnail), ο κώδικας της React το μετατρέπει άμεσα σε ετικέτα HTML <img>, προβάλλοντας τη μικρογραφία απευθείας μέσα στο κελί του αποτελέσματος.
- **Γεωχωρική Προβολή:** Αντίστοιχα, εάν εντοπιστούν ιδιότητες γεωγραφικού μήκους και πλάτους που έχουν συνενωθεί επιτυχώς (CONCAT), το σύστημα δεν τυπώνει αριθμούς, αλλά δημιουργεί ένα διαδραστικό εικονίδιο χάρτη (map pin).

Το συγκεκριμένο σενάριο επικυρώνει ότι το λογισμικό που αναπτύχθηκε δεν αποτελεί ένα απλό εργαλείο εκτέλεσης ερωτημάτων (query execution tool), αλλά λειτουργεί ως ένας σύγχρονος, ολοκληρωμένος **Σημασιολογικός Φυλλομετρητής (Semantic Browser)**. Επιτυγχάνει τον εκδημοκρατισμό της γνώσης, καθώς συγχωνεύει την τεράστια αναλυτική και αναζητητική δύναμη της γλώσσας SPARQL με τη φιλική, οπτικά πλούσια εμπειρία χρήσης του σύγχρονου Web

ΚΕΦΑΛΑΙΟ 7: ΣΥΜΠΕΡΑΣΜΑΤΑ ΚΑΙ ΜΕΛΛΟΝΤΙΚΈΣ ΕΠΕΚΤΑΣΕΙΣ

7.1 ΕΙΣΑΓΩΓΗ ΚΕΦΑΛΑΙΟΥ

Αυτή η διατριβή είχε ως στόχο το σχεδιασμό, την ανάπτυξη και την ανάλυση μιας νέας, διαδραστικής Εφαρμογής Σημασιολογικού Ιστού με σκοπό την ανάκτηση, ανάλυση και οπτικοποίηση σχετικών τιμών από το γράφημα γνώσης, βασισμένο στα δεδομένα της ελληνικής DBpedia. Αξιοποιώντας σύγχρονες έννοιες όπως η γλώσσα προγραμματισμού Golang για ταυτόχρονη εκτέλεση στο επίπεδο λογικής (backend), η βιβλιοθήκη React.js για τη δημιουργία δυναμικών διεπαφών χρήστη (frontend) και ο Virtuoso Universal Server για τη διαχείριση σημασιολογικών δεδομένων (triplestore), αυτό οδήγησε σε ένα πλήρες σύστημα αρχιτεκτονικής τριών επιπέδων. Αυτό το κεφάλαιο κλείνει με την ανασκόπηση της εργασίας που έχει γίνει, το επίπεδο επιτυχίας που επιτεύχθηκε στην επίτευξη των αρχικών στόχων όπως ορίστηκαν στο Κεφάλαιο 1, και στη συνέχεια καταλήγει στο συμπέρασμα σχετικά με τις χρήσεις των Linked Open Data. Επιπλέον, εξετάζονται τα μειονεκτήματα και περιγράφεται σε βάθος μια πλούσια λίστα μελλοντικών επεκτάσεων, επιτρέποντας στην εφαρμογή να ωριμάσει σε ένα εργαλείο ανάλυσης σημασιολογικών δεδομένων επιπέδου επιχείρησης.

7.2 ΑΝΑΣΚΌΠΗΣΗ ΚΑΙ ΑΞΙΟΛΟΓΗΣΗ ΕΠΊΤΕΥΞΗΣ ΣΤΌΧΩΝ

Για την αξιολόγηση αυτής της μελέτης, είναι απαραίτητο να εξεταστούν τα επιμέρους δομικά στοιχεία του συστήματος και πώς αντιμετώπισαν τις προκλήσεις που προέκυψαν κατά το στάδιο ανάλυσης απαιτήσεων.

7.2.1 ΥΠΟΔΟΜΉ ΚΑΙ ΔΙΑΧΕΙΡΙΣΗ ΜΕΓΑΛΩΝ ΔΕΔΟΜΈΝΩΝ

Ο πρώτος και ίσως ο πλέον απαιτητικός στόχος της παρούσας διπλωματικής εργασίας ήταν η συλλογή, η διαχείριση και η απρόσκοπτη ενσωμάτωση του τεράστιου όγκου σημασιολογικών δεδομένων που πηγάζουν από την Ελληνική έκδοση της DBpedia. Η πλήρης υλοποίηση αυτού του στόχου δεν βασίστηκε σε χειροκίνητες και επιρρεπείς σε σφάλματα διαδικασίες, αλλά κατέστη δυνατή μέσω της ανάπτυξης αυτοματοποιημένων αγωγών δεδομένων (Data Pipelines). Συγκεκριμένα, συνεγράφησαν εξειδικευμένα σενάρια εντολών (automation scripts) τα οποία επικοινωνούσαν απευθείας με το DBpedia Databus. Αυτή η προσέγγιση εξασφάλισε τη συστηματική ιχνηλάτηση, λήψη και διαχείριση των εκδόσεων (versioning) των αρχείων RDF, διασφαλίζοντας ότι η εφαρμογή τροφοδοτείται με τα πλέον επικαιροποιημένα και έγκυρα δεδομένα της κοινότητας.

Η διαδικασία της εισαγωγής (ingestion) αυτού του όγκου πληροφορίας στο τοπικό σύστημα ανέδειξε τις πραγματικές δυνατότητες της αρχιτεκτονικής. Η εκτεταμένη μαζική φόρτωση (Bulk Loading) εκατομμυρίων τριπλετών RDF (Subject-Predicate-Object) στον διακομιστή Virtuoso Universal Server δεν αποτέλεσε απλώς μια δοκιμή αντοχής, αλλά επιβεβαίωσε την ετοιμότητα και τη δυναμική του συστήματος να διαχειρίζεται φόρτους εργασίας επιπέδου Μεγάλων Δεδομένων (Big Data). Ο Virtuoso, λειτουργώντας ως ένας πανίσχυρος κινητήρας αποθήκευσης τετράδων (Quad Store), απέδειξε την ικανότητά του να ευρετηριάζει (index) και να εξυπηρετεί ταχύτατα τεράστια σύνολα γράφων, καταρρίπτοντας τους περιορισμούς που θα επέβαλλαν τα παραδοσιακά σχεσιακά συστήματα διαχείρισης βάσεων δεδομένων (RDBMS).

Τέλος, ένας καθοριστικός παράγοντας για τη βιωσιμότητα (sustainability) και την επεκτασιμότητα (scalability) του έργου υπήρξε η αρχιτεκτονική ανάπτυξής του. Εγκαταλείποντας τις συμβατικές μεθόδους εγκατάστασης λογισμικού, ολόκληρη η υποδομή —από το Επίπεδο Δεδομένων (Virtuoso) μέχρι το Ενδιάμεσο Λογισμικό (Go Backend) και τη Διεπαφή Χρήστη (React Frontend)— ενθυλακώθηκε σε ανεξάρτητα περιβάλλοντα εκτέλεσης (Containers). Μέσω της χρήσης των εργαλείων Docker και Docker Compose, εφαρμόστηκε στην πράξη η φιλοσοφία της «Υποδομής ως Κώδικα» (Infrastructure as Code - IaC).

Αυτή η προσέγγιση απομονώνει πλήρως την εφαρμογή από το λειτουργικό σύστημα του ξενιστή (host OS), εξαλείφοντας οριστικά το διαβόητο πρόβλημα «λειτουργεί μόνο στον δικό μου υπολογιστή» ("it works on my machine" syndrome), το οποίο παραδοσιακά μαστίζει τον κύκλο ανάπτυξης λογισμικού εξαιτίας ασυμβατοτήτων σε εκδόσεις βιβλιοθηκών. Ως αποτέλεσμα, το τελικό παραδοτέο σύστημα διακρίνεται από απόλυτη φορητότητα (portability). Μπορεί να μεταφερθεί, να αναπαραχθεί και να τεθεί σε πλήρη λειτουργία εντός ελάχιστων λεπτών σε οποιοδήποτε υπολογιστικό περιβάλλον —από έναν απλό φορητό υπολογιστή ανάπτυξης έως συστοιχίες διακομιστών στο νέφος (Cloud Clusters)— εγγυώμενο πανομοιότυπη συμπεριφορά, μέγιστη ασφάλεια και απόλυτη σταθερότητα.

7.2.2 ΒΕΛΤΙΣΤΟΠΟΙΗΣΗ ΑΠΟΔΟΣΗΣ ΜΕΣΩ ΤΑΥΤΟΧΡΟΝΙΣΜΟΥ

Στο Middleware, η γλώσσα Golang ήταν καταλύτης της λογικής της εφαρμογής. Τα SPARQL Endpoints συνήθως επικοινωνούν μέσω σειριακής (ένα προς ένα) εκτέλεσης ερωτημάτων (blocking I/O) ως τυπικός τρόπος. Αλλά μέσω αυτής της εργασίας, διαπιστώθηκε ότι το μοντέλο ταυτόχρονης εκτέλεσης του Go (μέσω Goroutines και συγχρονισμού με sync.WaitGroup) μπορεί να μειώσει σημαντικά το χρόνο απόκρισης. Πολλαπλές ταυτόχρονες κλήσεις COUNT στον κεντρικό πίνακα στατιστικών οδήγησαν σε μείωση του χρόνου αναμονής του χρήστη έως και 60%, σε σύγκριση με τη σειριακή εκτέλεση. Αυτό το γεγονός αποδεικνύει ότι το Golang είναι ένα εξαιρετικό εργαλείο για την ανάπτυξη middleware APIs (Reverse Proxies) σε αρχιτεκτονικές Σημασιολογικού Ιστού.

7.2.3 ΣΗΜΑΣΙΟΛΟΓΙΚΌ ΦΙΛΤΡΆΡΙΣΜΑ ΚΑΙ ΠΟΙΌΤΗΤΑ ΔΕΔΟΜΈΝΩΝ

Ένα άλλο πρόβλημα το οποίο προέκυψε κατά τον κύκλο ανάπτυξης της εφαρμογής —και το οποίο αποτελεί γενικότερη παθογένεια του Σημασιολογικού Ιστού και των Ανοικτών Διασυνδεδεμένων Δεδομένων (Linked Open Data) είναι η ελλιπής δομή των οντολογικών ιδιοτήτων (data incompleteness). Η Ελληνική DBpedia, ως ένα Γράφημα Γνώσης που αντλεί την πληροφορία του από τον πληθοπορισμικό (crowdsourced) χαρακτήρα της Wikipedia, υποφέρει συχνά από ασυνέπειες. Διαπιστώθηκε ότι σε ένα εξαιρετικά μεγάλο πλήθος οντοτήτων, όπως τα ελληνικά νησιά, τα μουσεία ή οι εγχώριες αθλητικές ομάδες, απουσίαζαν παντελώς βασικές ετικέτες γεωχωρικής ή εθνικής κατηγοριοποίησης (όπως οι ιδιότητες dbo:country ή dbo:location).

Το γεγονός αυτό δημιούργησε ένα σοβαρό εμπόδιο στην ανάκτηση της πληροφορίας. Όταν η μηχανή αναζήτησης εκτελούσε παραδοσιακές, αυστηρά δομημένες ερωτήσεις SPARQL (Strict Graph Pattern Matching), οι οποίες απαιτούσαν την απόλυτη ταύτιση της διαδρομής του γράφου, τα αποτελέσματα ήταν πενιχρά ή εντελώς κενά. Το σύστημα, αν και λειτουργούσε άψογα σε επίπεδο κώδικα δικτύου, παρουσίαζε εξαιρετικά χαμηλό ποσοστό Ανάκλησης (Recall Rate), αποκρύπτοντας από τον τελικό χρήστη τον πραγματικό πλούτο των δεδομένων που διέθετε η βάση.

Για την αποτελεσματική αντιμετώπιση αυτού του κρίσιμου ζητήματος, αναπτύχθηκε και εφαρμόστηκε μια υβριδική αλγοριθμική προσέγγιση, η οποία ορίστηκε ως Σημασιολογικό Φιλτράρισμα

Κειμένου (Semantic Text Filtering). Η στρατηγική αυτή εγκαταλείπει την αποκλειστική εξάρτηση από τις δομημένες ακμές του γράφου (σχέσεις μεταξύ κόμβων) και στρέφεται στην αξιοποίηση των αδόμητων δεδομένων (unstructured data) που ενυπάρχουν στους τερματικούς κόμβους (literal nodes), και συγκεκριμένα στο εκτενές ελεύθερο κείμενο των περιγραφών (dbo:abstract). Μέσω της ενορχήστρωσης ενσωματωμένων συναρτήσεων της γλώσσας SPARQL —όπως η συνάρτηση LCASE για την κανονικοποίηση του κειμένου σε πεζούς χαρακτήρες (εξασφαλίζοντας case-insensitivity) και η CONTAINS για τον ακριβή εντοπισμό υποσυμβολοσειρών— ο αλγόριθμος σαρώνει δυναμικά το κείμενο αναζητώντας σημασιολογικές λέξεις-κλειδιά που υποδηλώνουν εντοπιότητα.

7.2.4 ΕΚΠΑΙΔΕΥΤΙΚΗ ΑΞΙΑ ΚΑΙ ΕΜΠΕΙΡΙΑ ΧΡΗΣΤΗ

Για το σχεδιασμό του περιβάλλον UI(User interface) η βιβλιοθήκη ανοιχτού κώδικα React-Frontend.js μας βοήθησε να αποκτήσουμε την πρόσβαση στα σημασιολογικά μας δεδομένα. Ο Επεξεργαστής Ερωτημάτων SPARQL που ενσωματώθηκε στην εφαρμογή μιμείται στενά τις δυνατότητες ενός σύγχρονου IDE (Ολοκληρωμένο Περιβάλλον Ανάπτυξης), παρέχοντας στον χρήστη άμεση και κατανοητή ανατροφοδότηση (χειρισμός σφαλμάτων τύπου Compiler), καθιστώντας την εφαρμογή ένα εκπαιδευτικό πόρο για φοιτητές και ερευνητές που θέλουν να εξοικειωθούν με τη γλώσσα SPARQL. Και το σύστημα οπτικοποίησης αποτελεσμάτων για δυναμικά αποτελέσματα (αυτόματη εμφάνιση μικρογραφιών εικόνων με αυτόματη εμφάνιση και συνδέσεις με το Google Maps βάσει γεωγραφικών θέσεων (geo:lat, geo:long) έκανε την ανάγνωση δεδομένων από ανοιχτή μορφή πιο σύγχρονη και ελκυστική.

.

7.3 ΓΕΝΙΚΑ ΣΥΜΠΕΡΑΣΜΑΤΑ

Συνολικά, με την ολοκλήρωση αυτής υλοποίησης, γίνονται σημαντικές συμπεράσματα για το οικοσύστημα του Σημασιολογικού Ιστού και την ανάπτυξη λογισμικού::

1. **Η Αξία της Ενδιάμεσης Λογικής (Middleware):** Η ρητή έκθεση ενός SPARQL Endpoint στο Διαδίκτυο αποτελεί κίνδυνο ασφαλείας (π.χ. SPARQL Injection, Άρνηση Υπηρεσίας μέσω δαπανηρών ερωτημάτων) και μπορεί να οδηγήσει σε προβλήματα διαλειτουργικότητας (CORS). Η υλοποίηση ενός middleware back end συύστημα σε γλώσσες υψηλής απόδοσης όπως η Golang όχι μόνο αποτρέπει αυτά τα ζητήματα αλλά προσφέρει τη δυνατότητα βελτιστοποιήσεων (caching, concurrency).
2. **Τα Όρια της Ελληνικής DBpedia:** Ενώ η ελληνική DBpedia είναι μια εξαιρετική βάση γνώσεων, θα απαιτήσει πολλή δουλειά (κοινοτική προσπάθεια) για να αναβαθμιστεί η ποιότητα της οντολογίας της. Η αντιμετώπιση του θορύβου στα δεδομένα και των προκύπτοντων ελαττωμάτων θα πρέπει να αποτελεί ανησυχία για τους ερευνητές και τους προγραμματιστές που ασχολούνται με αυτά τα δεδομένα.
3. **Σύγκλιση Τεχνολογιών** Αυτή η εργασία δείχνει ότι η εφαρμογή μπορεί να λειτουργήσει ως Τεχνολογίες Σημασιολογικού Ιστού (RDF, SPARQL) όχι ως αυτόνομα εργαλεία ακαδημαϊκών, αλλά ότι μπορούν να συμφιλιωθούν με τα πιο πρόσφατα πρότυπα της βιομηχανίας λογισμικού (React, Docker, RESTful APIs, Αρχιτεκτονική Microservices).

7.4 ΠΕΡΙΟΡΙΣΜΟΊ ΤΟΥ ΣΥΣΤΉΜΑΤΟΣ

Ο σημαντικότερος περιορισμός του συστήματος αφορά τη χρονική εγκυρότητα της πληροφορίας. Το Γράφημα Γνώσης που φιλοξενείται στον τοπικό διακομιστή Virtuoso δεν συνδέεται

απευθείας με τον ζωντανό πυρήνα της Wikipedia, αλλά αποτελεί ένα απομονωμένο υποσύνολο δεδομένων, το οποίο βασίζεται σε στατικά στιγμιότυπα (data dumps) που αντλήθηκαν από το DBpedia Databus. Κατά συνέπεια, ο κύκλος ζωής της πληροφορίας διακόπτεται. Εάν ένας συντάκτης τροποποιήσει ή ενημερώσει ένα λήμμα στην Ελληνική Wikipedia (π.χ. προσθέτοντας τα νέα εκθέματα ενός μουσείου ή ενημερώνοντας τον πληθυσμό μιας πόλης), η αλλαγή αυτή δεν θα αντικατοπτριστεί άμεσα στην εφαρμογή μας. Για να καταστεί ορατή η νέα πληροφορία, απαιτείται η χειροκίνητη ή χρονοπρογραμματισμένη (cron job) επανάληψη ολόκληρης της διαδικασίας εξαγωγής, μεταφόρτωσης και εισαγωγής (ETL Data Ingestion Pipeline) των νέων αρχείων RDF στον Virtuoso.

Η ίδια η φύση των Ανοικτών Διασυνδεδεμένων Δεδομένων (Linked Open Data) βασίζεται στην παραπομπή σε εξωτερικά URIs. Η εμπειρία χρήστη (UX) και η δυναμική οπτικοποίηση στο Frontend (όπως η προβολή μικρογραφιών και η δημιουργία διαδραστικών χαρτών) εξαρτώνται άμεσα από τη διαθεσιμότητα τρίτων, εξωτερικών υπηρεσιών (Third-party APIs), όπως το Wikimedia Commons και το Google Maps. Εάν τα πολυμεσικά αρχεία στους αρχικούς εξυπηρετητές διαγραφούν, μετονομαστούν ή αν οι εξωτερικές υπηρεσίες αντιμετωπίσουν διακοπή λειτουργίας (downtime), οι υπερσύνδεσμοι στα δεδομένα RDF της βάσης μας θα καταστούν άκυροι. Το φαινόμενο αυτό, γνωστό στην Επιστήμη των Υπολογιστών ως «Σήψη Συνδέσμων» (Link Rot), θα οδηγήσει σε εσφαλμένη απόδοση (rendering) της διεπαφής, προβάλλοντας «σπασμένες» εικόνες (broken assets) στον τελικό χρήστη, παρότι η υποκείμενη βάση δεδομένων λειτουργεί άψογα.

Επί του παρόντος, η αρχιτεκτονική δρομολογεί κάθε εισερχόμενο HTTP αίτημα απευθείας στο Επίπεδο Δεδομένων. Εάν δύο διαφορετικοί χρήστες (ή ο ίδιος χρήστης μετά από επαναφόρτωση της σελίδας) εκτελέσουν το ίδιο ακριβώς ερώτημα SPARQL, το ενδιάμεσο επίπεδο της Go θα δημιουργήσει μια νέα σύνδεση και θα προωθήσει το αίτημα στον Virtuoso και τις δύο φορές. Η βάση δεδομένων θα αναγκαστεί να σαρώσει τον γράφο και να υπολογίσει το αποτέλεσμα από το μηδέν. Σε περιβάλλοντα παραγωγής με υψηλή επισκεψιμότητα, αυτή η έλλειψη επιπέδου προσωρινής αποθήκευσης (Caching Layer) οδηγεί σε περιττή κατανάλωση υπολογιστικής ισχύος (CPU cycles) και αυξημένο φόρτο I/O (Input/Output). Μια ιδανική βελτίωση του συστήματος θα απαιτούσε την ενσωμάτωση μιας in-memory βάσης (όπως το Redis) στο Go Backend, η οποία θα αποθηκεύει προσωρινά τις JSON απαντήσεις των συχνότερων ερωτημάτων, επιστρέφοντάς τες ακαριαία δίχως να επιβαρύνεται ο κεντρικός εξυπηρετητής.

7.5 ΠΡΟΤΆΣΕΙΣ ΓΙΑ ΜΕΛΛΟΝΤΙΚΈΣ ΕΠΕΚΤΆΣΕΙΣ

Το σύστημα σχεδιάστηκε εσκεμμένα με γνώμονα την επεκτασιμότητα (extensibility), επιτρέποντας τη μελλοντική προσθήκη νέων λειτουργικοτήτων. Οι παρακάτω υποενότητες αναλύουν με λεπτομέρεια τις προτεινόμενες ερευνητικές και αναπτυξιακές κατευθύνσεις, οι οποίες μπορούν να αποτελέσουν αντικείμενο ίσως μελλοντικών επεκτάσεων της εφαρμογής.

7.5.1 ΕΝΣΩΜΑΤΩΣΗ ΜΗΧΑΝΙΚΉΣ ΜΑΘΗΣΗΣ: ΑΠΌ ΦΥΣΙΚΗ ΓΛΏΣΣΑ ΣΕ SPARQL

Η μεγαλύτερη πρόκληση για τον μέσο χρήστη κατά την αλληλεπίδρασή του με ένα Knowledge Graph είναι η ανάγκη εκμάθησης της γλώσσας SPARQL. Μια επαναστατική μελλοντική επέκταση αποτελεί η ενσωμάτωση Μεγάλων Γλωσσικών Μοντέλων (Large Language Models - LLMs) ή εξειδικευμένων νευρωνικών δικτύων (όπως transformers) για τη μετάφραση ερωτημάτων από τη φυσική γλώσσα (Natural Language) σε συντακτικά ορθή SPARQL [12]. Ένα μεγάλο γλωσσικό μοντέλο (LLM) είναι ένας τύπος προγράμματος τεχνητής νοημοσύνης (AI) που μπορεί να αναγνωρίζει και να δημιουργεί κείμενο, μεταξύ άλλων εργασιών. Οι LLM εκπαιδεύονται σε τεράστια σύνολα δεδομένων

— εξ ου και το όνομα "large". Τα LLM βασίζονται στη μηχανική μάθηση: συγκεκριμένα, σε έναν τύπο νευρωνικού δικτύου που ονομάζεται μοντέλο μετασχηματιστή. Με απλά λόγια, ένα LLM είναι ένα πρόγραμμα υπολογιστή στο οποίο έχουν δοθεί αρκετά παραδείγματα ώστε να είναι σε θέση να αναγνωρίζει και να ερμηνεύει την ανθρώπινη γλώσσα ή άλλους τύπους σύνθετων δεδομένων.

Για παράδειγμα, ο χρήστης θα μπορούσε να πληκτρολογήσει στο περιβάλλον διεπαφής την πρόταση: *"Βρες μου όλα τα ελληνικά νησιά που έχουν πληθυσμό πάνω από 5.000 κατοίκους"*. Ένα ενσωματωμένο σύστημα Τεχνητής Νοημοσύνης στο Backend θα μετέφραζε αυτόματα την πρόταση, θα κατασκεύαζε το κατάλληλο γράφημα (Graph Pattern) και θα το προωθούσε στον Virtuoso. Η υλοποίηση αυτή θα μπορούσε να γίνει αξιοποιώντας APIs από μοντέλα όπως το OpenAI GPT-4 [5] ή τοπικά μοντέλα ανοικτού κώδικα (π.χ. LLaMA) εκπαιδευμένα (fine-tuned) πάνω στην οντολογία της DBpedia.

7.5.2 ΠΡΟΗΓΜΈΝΗ ΟΠΤΙΚΟΠΟΊΗΣΗ ΓΡΆΦΩΝ (NODE-EDGE VISUALIZATION)

Αυτή τη στιγμή, η εφαρμογή οπτικοποιεί τα δεδομένα σε μορφή δισδιάστατων πινάκων (tabular format). Αν και οι πίνακες είναι εξαιρετικοί για την παρουσίαση λιστών (π.χ. λίστα μουσείων), δεν αναδεικνύουν την πραγματική φύση των δεδομένων, η οποία είναι το Γράφημα (Graph). Μελλοντικά, θα μπορούσαν να χρησιμοποιηθούν και να ενσωματοθούν και άλλες βιβλιοθήκες οπτικοποίησης δεδομένων, όπως η **D3.js** (Data-Driven Documents) [4], η **Cytoscape.js** ή η **Vis.js** στο περιβάλλον της React. Με αυτόν τον τρόπο, εάν ο χρήστης επιλέξει τον "Αριστοτέλη", θα αναδύεται στην οθόνη ένα διαδραστικό πλέγμα κόμβων (nodes) και ακμών (edges), το οποίο θα οπτικοποιεί τις σχέσεις του (π.χ. δάσκαλος του "Μέγα Αλέξανδρου", μαθητής του "Πλάτωνα"). Ο χρήστης θα μπορεί να κάνει κλικ στους κόμβους (nodes) επεκτείνοντας την αναζήτηση, πλοηγούμενος στο Γράφημα Γνώσης διαισθητικά, χωρίς τη χρήση πληκτρολογίου.

7.5.3 ΑΥΤΟΜΑΤΟΠΟΊΗΣΗ ΕΠΙΚΑΙΡΟΠΟΊΗΣΗΣ ΔΕΔΟΜΈΝΩΝ (CI/CD DATA PIPELINES)

Για την οριστική επίλυση του περιορισμού της στατικότητας των δεδομένων (Data Staleness), προτείνεται η μελλοντική μετάβαση από την αρχική, χειροκίνητη εισαγωγή σε μια πλήρως αυτοματοποιημένη αρχιτεκτονική Συνεχούς Ενσωμάτωσης και Παράδοσης Δεδομένων (Data CI/CD Pipeline). Στη σύγχρονη Μηχανική Δεδομένων (Data Engineering), η διαχείριση της ροής της πληροφορίας οφείλει να αντιμετωπίζεται με την ίδια αυστηρότητα και αυτοματοποίηση που εφαρμόζεται στον πηγαίο κώδικα (DataOps).

Ο πυρήνας αυτής της μελλοντικής επέκτασης βασίζεται στη χρήση εξειδικευμένων εργαλείων ενορχήστρωσης ροών εργασίας (Workflow Orchestration), με κυρίαρχη επιλογή το **Apache Airflow**. Σε αντίθεση με τους απλούς χρονοπρογραμματιστές του λειτουργικού συστήματος (όπως τα cron jobs στο Linux), οι οποίοι εκτελούν μεμονωμένες εντολές τυφλά και στερούνται δυνατοτήτων παρακολούθησης εξαρτήσεων, το Apache Airflow αποτελεί μια βιομηχανικού επιπέδου (enterprise-grade) πλατφόρμα ανοικτού κώδικα. Επιτρέπει τον ορισμό των πολύπλοκων διαδικασιών λήψης (ETL) ως Κατευθυνόμενους Άκυκλους Γράφους (DAGs - Directed Acyclic Graphs) γραμμένους σε Python. Αυτό το εξαιρετικά επεκτάσιμο πλαίσιο διασφαλίζει ότι κάθε βήμα της ανανέωσης των δεδομένων εκτελείται με τη σωστή αλληλουχία, προσφέροντας παράλληλα μηχανισμούς αυτόματης επανάληψης σε περίπτωση σφάλματος δικτύου (retry mechanisms) και άμεσες ειδοποιήσεις (alerting) στους διαχειριστές.

Υπό αυτή την προτεινόμενη αρχιτεκτονική, ο αυτοματοποιημένος αγωγός (pipeline) θα εκτελεί τον ακόλουθο, αέναο κύκλο ζωής:

- **Προγραμματισμένη Ιχνηλάτηση (Automated Polling):** Ο ενορχηστρωτής θα επικοινωνεί αυτόματα με το DBpedia Databus σε τακτά χρονικά διαστήματα (π.χ. την 1η κάθε μήνα). Αξιοποιώντας το API του Databus, θα εκτελεί ερωτήματα μεταδεδομένων για να διαπιστώσει εάν έχει δημοσιευτεί μια νεότερη έκδοση του Γραφήματος για την Ελληνική γλώσσα, συγκρίνοντας τις ημερομηνίες έκδοσης ή τις ψηφιακές υπογραφές (hashes) των αρχείων.
- **Λήψη Διαφορικών Αρχείων (Delta / Changeset Processing):** Προκειμένου να αποφευχθεί η εξαιρετικά δαπανηρή (σε εύρος ζώνης δικτύου και επεξεργαστική ισχύ) διαδικασία λήψης και επανεισαγωγής ολόκληρης της βάσης δεδομένων από το μηδέν, το σύστημα θα υιοθετήσει τη διαφορική φόρτωση. Ο αλγόριθμος θα κατεβάζει μόνο τα αρχεία διαφορών (deltas), δηλαδή αποκλειστικά τις συγκεκριμένες τριάδες RDF που προστέθηκαν, τροποποιήθηκαν ή διαγράφηκαν σε σχέση με το προηγούμενο στιγμιότυπο. Αυτή η προσέγγιση είναι κρίσιμη για τη βελτιστοποίηση των Αναβαθμίσεων Λογισμικού (Software Upgrades) και τον αστραπιαίο Συγχρονισμό Δεδομένων (Data Synchronization) σε συστήματα Μεγάλων Δεδομένων.
- **Ενσωμάτωση με Μηδενικό Χρόνο Διακοπής (Zero-Downtime Deployment):** Η εφαρμογή των νέων τριάδων στον εξυπηρετητή Virtuoso θα διεκπεραιώνεται με στρατηγικές που δεν επηρεάζουν τη διαθεσιμότητα της εφαρμογής προς τον τελικό χρήστη. Αξιοποιώντας την υποδομή του Docker που ήδη διαθέτει το έργο, ο αγωγός θα μπορούσε να υιοθετήσει τη στρατηγική **Blue-Green Deployment**. Σε αυτό το σενάριο, ένα δεύτερο, αθέατο κοντέινερ βάσης δεδομένων ενημερώνεται στο παρασκήνιο (Green). Μόλις η εισαγωγή των deltas και η ευρετηρίαση (indexing) ολοκληρωθούν επιτυχώς, το Backend της Go ρυθμίζεται δυναμικά ώστε να δρομολογεί την κυκλοφορία των χρηστών στη νέα βάση, καταργώντας το παλιό, απαρχαιωμένο κοντέινερ (Blue).

Με αυτόν τον τρόπο, το σύστημα όχι μόνο θα εξαλείψει τη «στατικότητα», αλλά θα λειτουργεί ως ένας ζωντανός, διαρκώς εξελισσόμενος οργανισμός δεδομένων, προσφέροντας στον χρήστη επικαιροποιημένη γνώση χωρίς την παραμικρή διακοπή στη λειτουργία του (downtime).

7.5.4 ΟΜΟΣΠΟΝΔΙΑΚΆ ΕΡΩΤΗΜΑΤΑ

Ένα από τα πλέον ισχυρά, αλλά τεχνικά απαιτητικά, χαρακτηριστικά που εισήγαγε το πρότυπο SPARQL 1.1 είναι η υποστήριξη Ομοσπονδιακών Ερωτημάτων (Federated Queries) μέσω της λέξης-κλειδιού SERVICE [21]. Η αρχιτεκτονική αυτή αποτελεί την επιτομή του οράματος των Διασυνδεδεμένων Δεδομένων (Linked Data), καθώς επιτρέπει την κατανεμημένη εκτέλεση ενός μοναδικού ερωτήματος πάνω σε πολλαπλές, γεωγραφικά και διαχειριστικά ανεξάρτητες βάσεις δεδομένων (SPARQL Endpoints) ταυτόχρονα.

Η τρέχουσα υλοποίηση της παρούσας εφαρμογής αντλεί πληροφορίες αποκλειστικά από τον τοπικό διακομιστή Virtuoso, ο οποίος φιλοξενεί το στατικό στιγμιότυπο της Ελληνικής DBpedia. Αν και ο όγκος των δεδομένων είναι τεράστιος, κανένα μεμονωμένο γράφημα γνώσης δεν μπορεί να καλύψει την ολότητα της διαθέσιμης ανθρώπινης πληροφορίας. Συχνά, τα δεδομένα παραμένουν εγκλωβισμένα σε σημασιολογικά απομονωμένα υποσύνολα (data silos). Μια από τις σημαντικότερες μελλοντικές επεκτάσεις του συστήματος είναι η μετεξέλιξή του από έναν απλό φυλλομετρητή σε έναν δυναμικό **Σημασιολογικό Μεσολαβητή (Semantic Mediator / Hub)**.

Μέσω της χρήσης ομόσπονδων ερωτημάτων, η λογική του συστήματος θα μπορούσε να συνθέτει αιτήματα που αντλούν και διασταυρώνουν πληροφορίες σε πραγματικό χρόνο από κορυφαίες παγκόσμιες οντολογίες. Συγκεκριμένα, το γράφημα θα μπορούσε να εμπλουτιστεί δυναμικά από τις εξής πηγές:

- **Wikidata:** Αποτελεί την ελεύθερη, συνεργατική και πολύγλωσση βάση δεδομένων του ιδρύματος Wikimedia. Η διασύνδεση με το Endpoint των Wikidata θα προσέφερε στην εφαρμογή πρόσβαση σε δυναμικά δεδομένα που μεταβάλλονται διαρκώς (π.χ. τρέχοντες πληθυσμοί πόλεων, ωράρια λειτουργίας μουσείων, σύγχρονα πολιτικά πρόσωπα), αντισταθμίζοντας τον στατικό χαρακτήρα των ετήσιων εκδόσεων της DBpedia.
- **Europeana:** Ως η κεντρική πανευρωπαϊκή ψηφιακή πλατφόρμα πολιτιστικής κληρονομιάς, η Europeana παρέχει ελεύθερη πρόσβαση σε εκατομμύρια ψηφιοποιημένα τεκμήρια (έργα τέχνης, ιστορικά βιβλία, αρχειακό υλικό). Ένα ομόσπονδο ερώτημα θα μπορούσε να ανακτήσει τις βασικές πληροφορίες ενός αρχαιολογικού χώρου από την Ελληνική DBpedia, και ταυτόχρονα να φέρει υψηλής ανάλυσης εικόνες των εκθεμάτων του απευθείας από την Europeana, προβάλλοντάς τα ενοποιημένα στο επίπεδο παρουσίασης (React UI).
- **GeoNames:** Αποτελεί την κορυφαία βάση γεωχωρικών δεδομένων στον Σημασιολογικό Ιστό. Η ενσωμάτωσή της θα προσέφερε απόλυτη γεωγραφική ακρίβεια, επιτρέποντας πολύπλοκα τοπολογικά ερωτήματα (π.χ. ιεραρχίες δήμων, νομών και περιφερειών) και ακριβή πολύγωνα χαρτών, τα οποία συχνά απουσιάζουν από τα βασικά Infoboxes της Wikipedia.

Τεχνολογική Συμβατότητα και Αντιμετώπιση Προκλήσεων Η υλοποίηση ομοσπονδιακών ερωτημάτων εισάγει σημαντικές τεχνικές προκλήσεις, με κυριότερη την κατακόρυφη αύξηση του χρόνου απόκρισης (network latency), καθώς το τελικό αποτέλεσμα εξαρτάται από την ταχύτητα του πιο αργού εξωτερικού Endpoint. Επιπρόσθετα, η μεταφορά μεγάλου όγκου δεδομένων μεταξύ διαφορετικών διακομιστών για την εκτέλεση πράξεων συνένωσης (Federated Joins) αυξάνει δραματικά το υπολογιστικό κόστος.

Το σύστημα μας διαθέτει ήδη την ικανότητα να αποστέλλει πολλαπλά αιτήματα παράλληλα, να περιμένει τη συγκέντρωσή τους (Scatter-Gather) και να απορρίπτει ομαλά (graceful degradation) τις πηγές που καθυστερούν υπερβολικά να απαντήσουν.

Με την επέκταση αυτής της προγραμματιστικής λογικής, το λογισμικό θα ήταν σε θέση να μετατρέψει τον υπολογιστή του τελικού χρήστη σε έναν πραγματικό κόμβο συγχώνευσης της παγκόσμιας ψηφιακής γνώσης.

7.5.5 ΕΦΑΡΜΟΓΉ ΜΗΧΑΝΙΣΜΏΝ CACHING ΚΑΙ ΒΕΛΤΙΣΤΟΠΟΊΗΣΗΣ API

Όπως αναφέρθηκε στους αρχιτεκτονικούς περιορισμούς, στην τρέχουσα υλοποίησή της, η εφαρμογή λειτουργεί ως άμεσος διαμεσολαβητής (pass-through proxy), προωθώντας κάθε εισερχόμενο αίτημα απευθείας στη βάση δεδομένων (Virtuoso). Ωστόσο, η εκτέλεση ερωτημάτων SPARQL (Graph Pattern Matching) απαιτεί τεράστιους υπολογιστικούς πόρους (CPU και μνήμη RAM), ειδικά όταν περιλαμβάνει συναρτήσεις συνάθροισης (COUNT) ή φιλτράρισμα κειμένου (REGEX). Μια επιβεβλημένη τεχνική αναβάθμιση για τη μετάβαση του συστήματος σε περιβάλλον παραγωγής (production-ready) αποτελεί η υλοποίηση ενός ενδιάμεσου επιπέδου προσωρινής αποθήκευσης (Caching Layer) στο Backend.

Η βέλτιστη πρακτική για αυτήν την αρχιτεκτονική είναι η ενσωμάτωση του **Redis** [22]. Το Redis δεν είναι μια απλή σχεσιακή βάση, αλλά ένα προηγμένο σύστημα αποθήκευσης δομών δεδομένων στην κύρια μνήμη (in-memory data structure store), το οποίο λειτουργεί με τη λογική ζευγών κλειδιού-τιμής (key-value). Χρησιμοποιείται ευρέως στη βιομηχανία για σκοπούς προσωρινής αποθήκευσης, επεξεργασίας σε πραγματικό χρόνο (real-time processing), ασύγχρονης επικοινωνίας (message brokering) και διαχείρισης συνεδριών.

Μέσω της χρήσης του πλέον διαδεδομένου πελάτη (client) για τη διασύνδεση Go και Redis (τη βιβλιοθήκη go-redis), η προτεινόμενη λειτουργία του συστήματος θα διαμορφωνόταν ως εξής:

1. **Δημιουργία Κλειδιού (Key Generation):** Όταν ο χρήστης υποβάλλει ένα ερώτημα SPARQL, το Go API εφαρμόζει έναν αλγόριθμο κρυπτογραφικής κατακερμάτισης (π.χ. SHA-256) στο κείμενο του ερωτήματος, δημιουργώντας ένα μοναδικό κλειδί (hash string).
2. **Επιτυχία Κρυφής Μνήμης (Cache Hit):** Το σύστημα αναζητά το κλειδί στο Redis. Εάν το ερώτημα έχει υποβληθεί ξανά στο πρόσφατο παρελθόν (π.χ. αν δύο χρήστες πατήσουν το έτοιμο ερώτημα "Μουσεία της Ελλάδας"), το Redis επιστρέφει το αποθηκευμένο JSON αποτέλεσμα ακαριαία (σε λιγότερο από 2 milliseconds). Το αίτημα δεν φτάνει ποτέ στον Virtuoso.
3. **Αστοχία Κρυφής Μνήμης (Cache Miss):** Εάν το κλειδί δεν βρεθεί, το ερώτημα προωθείται κανονικά στον Virtuoso (αξιοποιώντας τον μηχανισμό Scatter-Gather που αναλύθηκε). Μόλις το Go Backend λάβει τα αποτελέσματα, τα επιστρέφει στον χρήστη και, ταυτόχρονα, τα αποθηκεύει στο Redis μετρώντας έναν καθορισμένο Χρόνο Ζωής (Time-To-Live / TTL).

Δεδομένου ότι τα στοιχεία της Ελληνικής DBpedia (ως στιγμιότυπα της Wikipedia) είναι πρακτικά στατικά και δεν μεταβάλλονται καθημερινά, ένα TTL της τάξης των 24 ή 48 ωρών θα ήταν ιδανικό. Με αυτήν την προσέγγιση, ραγδαία βελτιώνεται η κλιμακωσιμότητα (scalability) της εφαρμογής, καθώς ο διακομιστής αποφορτίζεται από περιττούς υπολογισμούς.

Πέραν της προσωρινής αποθήκευσης, η αρχιτεκτονική του Backend (Middle Tier) θα πρέπει να επεκταθεί με περαιτέρω μηχανισμούς βελτιστοποίησης API, συμβατούς με αρχιτεκτονικές μικροϋπηρεσιών (microservices):

- **Περιορισμός Ρυθμού (Rate Limiting):** Ενσωμάτωση μηχανισμών (π.χ. Token Bucket algorithm) για την αποτροπή κακόβουλων επιθέσεων ή υπερφόρτωσης (DDoS), περιορίζοντας τον αριθμό των ερωτημάτων που μπορεί να υποβάλει μια συγκεκριμένη διεύθυνση IP ανά λεπτό.
- **Προ-ανάλυση Ερωτημάτων (Query Complexity Analysis):** Πριν καν ένα ερώτημα φτάσει στον Virtuoso ή στο Redis, το Go API θα μπορούσε να διαθέτει έναν ελαφρύ αναλυτή (parser) που να ελέγχει τη δομή της SPARQL. Ερωτήματα που δεν περιέχουν το LIMIT ή επιχειρούν απαγορευμένα καρτεσιανά γινόμενα (Cartesian products) σε τριπλέτες, θα απορρίπτονται άμεσα από το Backend με επιστροφή HTTP Error 400 (Bad Request), λειτουργώντας ως μια προληπτική ασπίδα, συμπληρωματική του ήδη υπάρχοντος "Timeout Guard".

7.5.6 ΠΟΛΥΓΛΩΣΣΙΚΟΤΗΤΑ ΚΑΙ ΔΙΕΘΝΟΠΟΊΗΣΗ

Η αρχιτεκτονική του συστήματος που σχεδιάστηκε και υλοποιήθηκε χαρακτηρίζεται από υψηλή αφαιρετικότητα (abstraction), γεγονός που της επιτρέπει να λειτουργεί ανεξάρτητα (agnostic) από το υποκείμενο σύνολο δεδομένων. Μια λογική, μελλοντική επέκταση της παρούσας διπλωματικής εργασίας είναι η πλήρης Διεθνοποίηση (Internationalization - i18n) της εφαρμογής. Ο στόχος είναι η δημιουργία μηχανισμών στο επίπεδο λογικής (Go Backend) και παρουσίασης (React Frontend) που θα επιτρέπουν στον χρήστη να "αλλάζει" την πηγή της σημασιολογικής γνώσης δυναμικά. Αντί το σύστημα να δεσμεύεται αποκλειστικά στον κόμβο της Ελληνικής DBpedia, θα μπορούσε να διασυνδεθεί με τα αντίστοιχα τερματικά σημεία (endpoints) ή να φορτώσει τα τοπικά αντίγραφα (dumps) της Αγγλικής (en), Ισπανικής (es) ή Ιταλικής (it) έκδοσης, μετεξελίσσοντας την εφαρμογή σε έναν παγκόσμιο φυλλομετρητή Σημασιολογικού Ιστού.

Η Διεθνοποίηση (i18n) δεν αποτελεί απλώς τη μετάφραση κειμένων, αλλά τη θεμελιώδη διαδικασία σχεδιασμού ενός προϊόντος λογισμικού ώστε να προσαρμόζεται τεχνικά και πολιτισμικά σε χρήστες διαφορετικών εθνοτήτων, χωρίς την ανάγκη αλλαγής του πηγαίου κώδικα. Η διαδικασία αυτή είναι άρρηκτα συνδεδεμένη με την Τοπικοποίηση (Localization - l10n), η οποία αναλαμβάνει την

προσαρμογή των δεδομένων στις κατά τόπους συμβάσεις, διαμορφώνοντας καθοριστικά την Εμπειρία του Πελάτη (User eXperience - UX).

Η ορθή διαχείριση αυτών των πολιτισμικών ιδιαιτεροτήτων απαιτεί προσεκτική αρχιτεκτονική. Για παράδειγμα, η αναπαράσταση των ονομάτων διαφέρει ριζικά ανα τον κόσμο: σε ορισμένες ασιατικές χώρες το επώνυμο προηγείται του μικρού ονόματος, στην Ισπανία χρησιμοποιούνται παραδοσιακά δύο επώνυμα (ένα πατρικό και ένα μητρικό), ενώ στη Γερμανία είναι συχνή η χρήση ενωτικού. Αντίστοιχες και ίσως εντονότερες αποκλίσεις εντοπίζονται στις μορφοποιήσεις των ταχυδρομικών κωδίκων. Ενώ στην Ελλάδα ο κώδικας είναι αμιγώς αριθμητικός με πέντε ψηφία (π.χ. 54622) και στη Βραζιλία χωρίζεται με παύλα (00000-000), στον Καναδά ακολουθεί το αλφαριθμητικό πρότυπο "X0X 0X0" (όπου X γράμμα και 0 αριθμός), και στο Ηνωμένο Βασίλειο εμφανίζει πολλαπλές παραλλαγές μήκους και διαστημάτων (όπως "XX00 0XX" ή "X0 0XX"). Ένα διεθνοποιημένο λογισμικό οφείλει να εφαρμόζει δυναμικούς κανόνες επικύρωσης (dynamic validation rules) και να ελέγχει αυτόματα αυτές τις εισόδους ανάλογα με την επιλεγμένη τοποθεσία (locale), αποφεύγοντας τη «σκληρή κωδικοποίηση» (hardcoding) προτύπων.

Επιπρόσθετα, η διεπαφή (UI) πρέπει να είναι σχεδιασμένη με ελαστικότητα, ώστε να υποστηρίζει γλώσσες που διαβάζονται από τα δεξιά προς τα αριστερά (Right-to-Left - RTL), όπως τα Αραβικά ή τα Εβραϊκά. Αυτό σημαίνει ότι ολόκληρη η δομή της σελίδας, τα περιθώρια (margins) και οι στοιχίσεις των στοιχείων (όπως οι πίνακες δεδομένων) πρέπει να αντικατοπτρίζονται αυτόματα.

Στο πλαίσιο του Σημασιολογικού Ιστού, η πρόκληση της πολυγλωσσικότητας επεκτείνεται και στον τρόπο σύνταξης των ερωτημάτων **SPARQL**. Στο μοντέλο δεδομένων RDF, τα αλφαριθμητικά κείμενα (literals) συνοδεύονται από ειδικές ετικέτες γλώσσας (language tags), όπως "Athens"@en ή "Αθήνα"@el. Για να υποστηρίξει το σύστημα πολλαπλές γλώσσες, το ενδιάμεσο επίπεδο της Go θα πρέπει να κατασκευάζει τα ερωτήματα δυναμικά. Για παράδειγμα, η εντολή φιλτραρίσματος FILTER langMatches(lang(?label), "el") θα προσαρμόζεται προγραμματιστικά στο παρασκήνιο (backend) για να αντλεί δεδομένα στη γλώσσα που έχει επιλέξει ο χρήστης στο frontend.

Συμπερασματικά, όταν η διεθνοποίηση εφαρμοστεί σωστά, δημιουργεί λογισμικό που ταυτίζεται με τον τελικό χρήστη. Η μετάβαση σε μια πλήρως διεθνοποιημένη αρχιτεκτονική δεν επεκτείνει απλώς το δυνητικό κοινό της εφαρμογής, αλλά αναδεικνύει την πραγματική δύναμη των Ανοικτών Διασυνδεδεμένων Δεδομένων: τη δυνατότητα μιας ενιαίας μηχανής να κατανοεί, να ανακτά και να παρουσιάζει την παγκόσμια γνώση, προσαρμοσμένη απόλυτα στο πολιτισμικό και γλωσσικό υπόβαθρο του εκάστοτε πολίτη.

7.6 ΕΠΙΛΟΓΟΣ

Η μετάβαση από τον παραδοσιακό Ιστό των Εγγράφων στον Ιστό των Δεδομένων (Web of Data) δεν συνιστά απλώς μια τεχνολογική αναβάθμιση, αλλά μια παραδειγματική μετατόπιση (paradigm shift) στον τρόπο με τον οποίο η ανθρωπότητα οργανώνει, διασυνδέει και καταναλώνει την πληροφορία. Εγχειρήματα ανοικτών διασυνδεδεμένων δεδομένων, με προεξάρχον την DBpedia, βρίσκονται στην αιχμή αυτής της προσπάθειας, μετασχηματίζοντας αδόμητα κείμενα σε μηχανικά κατανοητά Γραφήματα Γνώσης. Ειδικότερα, ο Ελληνικός Κόμβος (Greek DBpedia) διαδραματίζει καθοριστικό ρόλο στη διαφύλαξη και σημασιολογική ανάδειξη της εθνικής μας πολιτισμικής κληρονομιάς, της ιστορίας και της γλώσσας στο παγκόσμιο ψηφιακό οικοσύστημα.

Ωστόσο, όπως αναδείχθηκε μέσα από την παρούσα εργασία, η ύπαρξη και μόνο αυτών των δεδομένων δεν εγγυάται την αξιοποίησή τους. Ο πλούτος του Σημασιολογικού Ιστού ακυρώνεται στην πράξη εάν παραμένει εγκλωβισμένος πίσω από πολύπλοκα τεχνικά πρωτόκολλα, αυστηρά συντακτικά

όρια (όπως η γλώσσα SPARQL) και ακατέργαστες μορφές τριπλετών (RDF), καθιστώντας τον προσιτό αποκλειστικά σε εξειδικευμένους μηχανικούς λογισμικού και ερευνητές δεδομένων. Επιπρόσθετα, οι εγγενείς δομικές ελλείψεις της ελληνικής έκδοσης, τα προβλήματα χαρτογράφησης (mappings) και οι καθυστερήσεις στην απόκριση των διακομιστών αποθαρρύνουν τον μέσο χρήστη από την εξερεύνηση του διαθέσιμου πλούτου.

Στόχος της παρούσας πτυχιακής εργασίας ήταν η γεφύρωση αυτού του χάσματος, μέσω της σχεδίασης και ανάπτυξης ενός σύγχρονου, αποδοτικού και φιλικού προς τον χρήστη Σημασιολογικού Φυλλομετρητή (Semantic Browser). Μέσα από την υλοποίηση αποδείχθηκε ότι τα σύγχρονα εργαλεία ανάπτυξης ιστού (React.js) μπορούν να συνδυαστούν άψογα με τις τεχνολογίες του Σημασιολογικού Ιστού, μετατρέποντας ακατέργαστα URIs σε πλούσιες, διαδραστικές, οπτικές αναπαραστάσεις (εικόνες, γεωχωρικούς χάρτες, μορφοποιημένους πίνακες).

Σε επίπεδο υποδομής (Backend), η εργασία απέδειξε ότι η ενσωμάτωση προηγμένων αρχιτεκτονικών μοτίβων (όπως η ταυτοχρονία μέσω Golang Goroutines και ο μηχανισμός Scatter-Gather) είναι επιβεβλημένη για την ομαλή λειτουργία εφαρμογών μεγάλης κλίμακας. Η βελτιστοποίηση των χρόνων απόκρισης (άνω του 50%) σε συνδυασμό με την ανάπτυξη μηχανισμών προστασίας πόρων (Timeout Guard / Circuit Breaker) εξασφάλισε την ανοχή του συστήματος σε σφάλματα (fault tolerance), προστατεύοντας τον διακομιστή από εξαντλητικά ερωτήματα και διασφαλίζοντας την αδιάλειπτη εμπειρία του χρήστη. Παράλληλα, η χρήση σημασιολογικού φιλτραρίσματος κειμένου (semantic text filtering) απέδειξε πως η προγραμματιστική λογική μπορεί να υποκαταστήσει, σε σημαντικό βαθμό, τις οντολογικές ελλείψεις του γραφήματος, αυξάνοντας δραματικά την ανάκληση (recall) των αποτελεσμάτων.

Κοιτάζοντας προς το μέλλον, το παρόν σύστημα έχει τεθεί σε μια ισχυρή, επεκτάσιμη βάση. Μελλοντικές κατευθύνσεις θα μπορούσαν να περιλαμβάνουν:

1. Την υποστήριξη Ομόσπονδων Ερωτημάτων (Federated Queries), επιτρέποντας στο σύστημα να αντλεί και να συνδυάζει ταυτόχρονα δεδομένα από την Ελληνική DBpedia, τα Wikidata και το GeoNames.
2. Την ενσωμάτωση μοντέλων Επεξεργασίας Φυσικής Γλώσσας (NLP / AI), τα οποία θα μεταφράζουν αυτόματα τις ερωτήσεις των χρηστών (π.χ. "Δείξε μου τα μουσεία της Κρήτης") απευθείας σε ερωτήματα SPARQL, καταργώντας εντελώς την ανάγκη γνώσης της γλώσσας.

Συμπερασματικά, η εργασία αυτή τεκμηριώνει πως ο Σημασιολογικός Ιστός έχει πλέον ωριμάσει. Προκειμένου όμως να επιτευχθεί ο πραγματικός εκδημοκρατισμός της γνώσης, απαιτείται η διαρκής ανάπτυξη ενδιάμεσου λογισμικού (middleware) και διεπαφών που θα "κρύβουν" την τεχνική πολυπλοκότητα στο παρασκήνιο, προσφέροντας στον τελικό χρήστη άμεση, διαισθητική και οπτικά αναβαθμισμένη πρόσβαση στην παγκόσμια δεξαμενή δεδομένων.

.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., & Ives, Z. (2007). "DBpedia: A Nucleus for a Web of Open Data". Στο The Semantic Web (ISWC 2007) (σελ. 722-735). Springer.

[2] Berners-Lee, T., Hendler, J., & Lassila, O. (2001). "The Semantic Web: A new form of Web content...". Scientific American, 284(5), 34-43.

[3] Bizer, C., Heath, T., & Berners-Lee, T. (2011). "Linked Data: The Story So Far". Στο Semantic Services... (σελ. 205-227). IGI Global.

[4] Bostock, M., Ogievetsky, V., & Heer, J. (2011). "D³ data-driven documents". IEEE transactions on visualization and computer graphics, 17(12).

[5] Brown, T., Mann, B., Ryder, N., ... & Amodei, D. (2020). "Language models are few-shot learners". Advances in neural information processing systems.

[6] Cyganiak, R., Wood, D., & Lanthaler, M. (2014). "RDF 1.1 Concepts and Abstract Syntax". W3C Recommendation. Διαθέσιμο στο: https://www.w3.org/TR/rdf11-concepts/

[7] Donovan, A. A., & Kernighan, B. W. (2015). The Go Programming Language. Addison-Wesley Professional.

[8] Erling, O., & Mikhailov, I. (2009). "Virtuoso: RDF support in a native RDBMS". Στο Semantic Web Information Management. Springer.

[9] Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley Professional.

[10] Frey, J., Hellmann, S., & Ackermann, M. (2019). "DBpedia Databus - A Data Network for the Semantic Web". DBpedia Databus architecture.

[11] Harris, S., & Seaborne, A. (2013). "SPARQL 1.1 Query Language". W3C Recommendation. Διαθέσιμο στο: https://www.w3.org/TR/sparql11-query/

[12] Kaufmann, E., & Bernstein, A. (2007). "How useful is natural language interfaces to the semantic web...". ISWC 2007.

[13] Kontokostas, D., Bratsas, C., Auer, S., ... & Mehta, P. (2012). "Internationalization of Linked Data: The case of the Greek DBpedia edition". Web Semantics.

[14] Lehmann, J., Isele, R., Jakob, M., ... & Bizer, C. (2015). "DBpedia – A large-scale, multilingual knowledge base...". Semantic Web Journal.

[15] Merkel, D. (2014). "Docker: lightweight Linux containers for consistent development and deployment". Linux Journal.

[16] Meta/Facebook Open Source. (2024). React Documentation: Describing the UI. Διαθέσιμο στο: https://react.dev/learn

[17] OpenLink Software. (2024a). Virtuoso Open-Source Edition Documentation.

[18] OpenLink Software. (2024b). "Virtuoso Database Configuration and Database Files". Διαθέσιμο στο: http://docs.openlinksw.com/virtuoso/

[19] OpenLink Software. (2024c). "Bulk Data Loading Process". RDF Data Management.

[20] Pérez, J., Arenas, M., & Gutierrez, C. (2006). "Semantics and complexity of SPARQL". ISWC.

[21] Prud'hommeaux, E., & Buil-Aranda, C. (2013). "SPARQL 1.1 Federated Query". W3C Recommendation. Διαθέσιμο στο: https://www.w3.org/TR/sparql11-federated-query/

[22] Redis Ltd. (2024). Redis Documentation: In-memory Data Store. Διαθέσιμο στο: https://redis.io/docs/

ΠΑΡΑΡΤΗΜΑΤΑ

ΠΑΡΑΡΤΗΜΑ Α: ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ ΕΓΚΑΤΑΣΤΑΣΗΣ ΚΑΙ ΕΚΤΕΛΕΣΗΣ

download_data.sh

```
#!/bin/bash

# Χρώματα για το τερματικό
CYAN='\033[0;36m'
YELLOW='\033[1;33m'
GREEN='\033[0;32m'
GRAY='\033[1;30m'
RED='\033[0;31m'
NC='\033[0m' # No Color

echo -e "${CYAN}Searching for Greek DBpedia files (lang=el)...${NC}"

# 1. To Query
QUERY="PREFIX dcat: <http://www.w3.org/ns/dcat#>
PREFIX dataid: <http://dataid.dbpedia.org/ns/core#>

SELECT DISTINCT ?file WHERE {
  ?dataset dcat:distribution ?distribution .
  ?distribution dcat:downloadURL ?file .

  # ΦΙΛΤΡΟ 1: Να είναι Ελληνικά
  FILTER (contains(str(?file), 'lang=el'))

  # ΦΙΛΤΡΟ 2: Να είναι ΜΟΝΟ τα αρχεία που χρειαζόμαστε, oxi metadata
  FILTER (
    contains(str(?file), '/labels') ||
    contains(str(?file), '/short-abstracts') ||
    contains(str(?file), '/long-abstracts') ||
    contains(str(?file), '/instance-types') ||
    contains(str(?file), '/mappingbased-objects') ||
    contains(str(?file), '/mappingbased-literals') ||
    contains(str(?file), '/images') ||
    contains(str(?file), '/geo-coordinates')
  )
}"

OUTPUT_DIR="dbpedia_data"
mkdir -p "$OUTPUT_DIR"
```

```
# 2. Εκτέλεση Query (Το curl κάνει αυτόματα το URL encode με το --data-urlencode)
CSV_TMP_FILE="/tmp/dbpedia_files.csv"

curl -s -G -H "Accept: text/csv" \
  --data-urlencode "query=$QUERY" \
  "https://databus.dbpedia.org/sparql" -o "$CSV_TMP_FILE"

# 3. Έλεγχος Αποτελεσμάτων
# Μετράμε γραμμές αγνοώντας το header (file) και τις κενές γραμμές
COUNT=$(tail -n +2 "$CSV_TMP_FILE" | grep -v '^\s*$' | wc -l)

if [ "$COUNT" -eq 0 ]; then
    echo -e "${RED}No files found! (Check filters)${NC}"
    rm -f "$CSV_TMP_FILE"
    exit 1
fi

echo -e "${YELLOW}Found $COUNT relevant files. Starting download...${NC}"

# 4. Λήψη Αρχείων
# Διαβάζουμε το CSV παραλείποντας την πρώτη γραμμή, αφαιρούμε τυχόν " και \r
tail -n +2 "$CSV_TMP_FILE" | tr -d '"' | tr -d '\r' | while read -r FILE_URL; do
    if [ -z "$FILE_URL" ]; then continue; fi

    # Παίρνουμε το όνομα του αρχείου από το τέλος του URL
    FILE_NAME=$(basename "$FILE_URL")
    OUTPUT_PATH="$OUTPUT_DIR/$FILE_NAME"

    # Έλεγχος αν υπάρχει ήδη
    if [ -f "$OUTPUT_PATH" ]; then
        echo -e "${GRAY}  Skipping (exists): $FILE_NAME${NC}"
        continue
    fi

    echo -e "${GREEN}  Downloading: $FILE_NAME${NC}"
    # Λήψη με το curl (-L για ακολουθία redirects, -s για silent mode)
    curl -L -s -o "$OUTPUT_PATH" "$FILE_URL"

    if [ $? -ne 0 ]; then
        echo -e "${RED}  Failed to download $FILE_NAME${NC}"
    fi
done

# Καθαρισμός προσωρινού αρχείου
rm -f "$CSV_TMP_FILE"
echo -e "${CYAN}Success! All files are in '$OUTPUT_DIR'${NC}"
```

```
# το script: ./download_data.sh (θα κατεβάσει τα αρχεία στον φάκελο dbpedia_data)
#  `docker-compose up -d --build
#   docker exec -it dbpedia-virtuoso ls -l /database/to_load
#  Μπες στο SQL terminal του Virtuoso:
# docker exec -it dbpedia-virtuoso isql 1111 dba mysecretpassword
# παρακάτω εντολές στο SQL prompt (SQL>):
# ld_dir('/database/to_load', '*.bz2', 'http://dbpedia.org');
#  rdf_loader_run();
# checkpoint;
#  Για έλεγχο προόδου/ολοκλήρωσης
# SELECT * FROM DB.DBA.LOAD_LIST;
```

docker-compose.yml

```
version: '3.8'

services:
  # --- 1. Database (Virtuoso)
  store:
    image: openlink/virtuoso-opensource-7:latest
    container_name: dbpedia-virtuoso
    environment:
      DBA_PASSWORD: "mysecretpassword"
      SPARQL_UPDATE: "true"
      DEFAULT_GRAPH: "http://dbpedia.org"
      VIRT_Parameters_NumberOfBuffers: 170000
      VIRT_Parameters_MaxDirtyBuffers: 130000
      VIRT_Parameters_DirsAllowed: "., /database/to_load"
    ports:
      - "8890:8890"
      - "1111:1111"
    volumes:
      - ./dbpedia_data:/database/to_load
      - ./virtuoso_db:/database

  # --- 2. Backend (Go API)
  api:
    build: ./dbpedia-backend
    container_name: dbpedia-backend
    ports:
      - "8080:8080"
    environment:
      - VIRTUOSO_ENDPOINT=http://store:8890/sparql
    depends_on:
      - store
```

```
 # --- 3. Frontend (React)
 frontend:
   build:
     context: ./dbpedia-frontend/dbpedia-frontend
   container_name: dbpedia-frontend
   ports:
     - "5173:5173"
   volumes:
     - ./dbpedia-frontend/dbpedia-frontend:/app
     - /app/node_modules
   environment:
     - CHOKIDAR_USEPOLLING=true
   depends_on:
     - api
```

ΠΑΡΑΡΤΗΜΑ Β: ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ ΕΠΙΠΕΔΟΥ ΛΟΓΙΚΗΣ

Dockerfile

```
# Stage 1: Build
FROM golang:1.22.4 AS builder

WORKDIR /app

COPY go.mod go.sum ./
RUN go mod download

# Αντιγράφουμε όλο τον κώδικα
COPY . .

# Πηγαίνουμε στον φάκελο που είναι το main.go
WORKDIR /app/cmd/api

RUN CGO_ENABLED=0 GOOS=linux go build -o backend .

# Stage 2: Runtime
FROM alpine:latest

# πιστοποιητικά ασφαλείας για calls σε https
RUN apk --no-cache add ca-certificates

WORKDIR /root/
COPY --from=builder /app/cmd/api/backend .
EXPOSE 8080
```

```
CMD ["./backend"]
```

main.go

```
package main

import (
   "dbpedia-backend/cmd/api/handlers"
   "fmt"
   "log"
   "net/http"

   "github.com/gorilla/mux"
)

// Η main  στοχος μας να σηκωσει εναν HTTP server που θα εξυπηρετεί τα αιτήματα μας.
// Gorilla Mux library για να διαχειριστούμε τα routes και να κατευθύνουμε τα αιτήματα στους κατάλληλους handlers.
func main() {
   //δρομολογητης για να διαχειριστεί τα αιτήματα
   router := mux.NewRouter()

   //health check endpoint -->test
   router.HandleFunc("/health", func(w http.ResponseWriter, r *http.Request) {
      w.WriteHeader(http.StatusOK)
      w.Write([]byte("Backend is running"))
   }).Methods("GET")

   //Root route for basic health check
   router.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
      w.Header().Set("Content-Type", "text/plain")
      w.WriteHeader(http.StatusOK)
      fmt.Fprintln(w, "Welcome to the DBpedia Backend API")
   }).Methods("GET")

   // Register the SPARQL handler
   router.HandleFunc("/sparql", handlers.SPARQLHandler).Methods("POST", "GET", "OPTIONS")

   // Register the predefined queries handler
      router.HandleFunc("/predefined",  handlers.PredefinedQueryHandler).Methods("POST",  "GET", "OPTIONS")

   // Register the dashboard stats handler
   //router.HandleFunc("/stats", handlers.GetDashboardStats).Methods("POST", "GET", "OPTIONS")
```

```
    // Start the server
    log.Println("Server is running on port 8080")
    log.Fatal(http.ListenAndServe(":8080", router))
}
```

middleware.go

```
package utils

import (
    "log"
    "net/http"
    "time"
)

// LoggingMiddleware logs each incoming request with method, path, and status code.
func LoggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        log.Printf("Started %s %s", r.Method, r.URL.Path)

        // Wrap the response writer to capture the status code
        lrw := &loggingResponseWriter{ResponseWriter: w, statusCode: http.StatusOK}
        next.ServeHTTP(lrw, r)

        duration := time.Since(start)
        log.Printf("Completed %d %s in %v", lrw.statusCode, http.StatusText(lrw.statusCode), duration)
    })
}

// loggingResponseWriter wraps http.ResponseWriter to capture the status code.
type loggingResponseWriter struct {
    http.ResponseWriter
    statusCode int
}

func (lrw *loggingResponseWriter) WriteHeader(code int) {
    lrw.statusCode = code
    lrw.ResponseWriter.WriteHeader(code)
}

// ValidateSPARQLQueryMiddleware ensures that SPARQL query requests are valid.
func ValidateSPARQLQueryMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if r.Method != http.MethodPost {
            HandleError(w, "Invalid request method", http.StatusMethodNotAllowed)
```

```
            return
        }

        if r.Header.Get("Content-Type") != "application/json" {
            HandleError(w, "Content-Type must be application/json", http.StatusUnsupportedMediaType)
            return
        }

        next.ServeHTTP(w, r)
    })
}
```

error_utils.go

```
package utils

import (
    "encoding/json"
    "net/http"
)

// ErrorResponse represents a standardized error response.
type ErrorResponse struct {
    Message string `json:"message"`
}

// HandleError sends a JSON-formatted error response.
func HandleError(w http.ResponseWriter, message string, statusCode int) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(statusCode)
    json.NewEncoder(w).Encode(ErrorResponse{Message: message})
}
```

predifined_handler.go

```
package handlers

import (
    "dbpedia-backend/cmd/api/services"
    "dbpedia-backend/cmd/api/utils"
    "log"
    "net/http"
)

// PredefinedQueryHandler handles requests for predefined SPARQL queries.
func PredefinedQueryHandler(w http.ResponseWriter, r *http.Request) {
    // Parse query parameters
```

```
    queryName := r.URL.Query().Get("query")
    if queryName == "" {
        utils.HandleError(w, "Query parameter is required", http.StatusBadRequest)
        return
    }

    // Log the query name for debugging
    log.Printf("Predefined query received: %s", queryName)

    // Fetch the predefined query
    query, exists := services.GetPredefinedQuery(queryName, nil)
    if !exists {
        utils.HandleError(w, "Query not found", http.StatusNotFound)
        return
    }

    // Execute the query
    endpoint := "http://localhost:8890/sparql"
    result, err := services.ExecuteSPARQL(endpoint, query)
    if err != nil {
        utils.HandleError(w, err.Error(), http.StatusInternalServerError)
        return
    }

    // Return the results as JSON
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(result))
}
```

sparql_service.go

```
package services

import (
    "bytes"
    "errors"
    "io"
    "log"
    "net/http"
    "time"
)

// ExecuteSPARQL sends a SPARQL query to the Virtuoso endpoint and returns the results or an error.
func ExecuteSPARQL(endpoint string, query string) (string, error) {
```

```
    start := time.Now()

    // Create the HTTP POST request
    req, err := http.NewRequest("POST", endpoint, bytes.NewBuffer([]byte(query)))
    if err != nil {
        log.Printf("Error creating SPARQL request: %v", err)
        return "", errors.New("failed to create SPARQL request")
    }

    // Set necessary headers
    req.Header.Set("Content-Type", "application/sparql-query")

    // απάντηση σε JSON
    req.Header.Set("Accept", "application/sparql-results+json")

    // Log the SPARQL query
    log.Printf("Executing SPARQL query: %s", query)

    // Send the request
    client := &http.Client{}
    resp, err := client.Do(req)
    if err != nil {
        log.Printf("Error executing SPARQL request: %v", err)
        return "", errors.New("failed to execute SPARQL query")
    }
    defer resp.Body.Close()

    // Check for non-200 status codes
    if resp.StatusCode != http.StatusOK {
        bodyBytes, _ := io.ReadAll(resp.Body)
         log.Printf("SPARQL endpoint returned status %d: %s. Body: %s", resp.StatusCode, resp.Status,
string(bodyBytes))
        return "", errors.New("SPARQL endpoint returned an error: " + resp.Status)
    }

    // Read the response body
    body, err := io.ReadAll(resp.Body)
    if err != nil {
        log.Printf("Error reading SPARQL response body: %v", err)
        return "", errors.New("failed to read SPARQL response")
    }

    // Log query execution time
    duration := time.Since(start)
    log.Printf("SPARQL query executed in %v", duration)
```

```
    return string(body), nil
}

/*
Example log output:
024/11/27 12:00:00 Started POST /sparql from 127.0.0.1:60000
2024/11/27 12:00:00 Executing SPARQL query: SELECT ?s ?p ?o WHERE { ?s ?p ?o } LIMIT 10
2024/11/27 12:00:01 SPARQL query executed in 1.002s
2024/11/27 12:00:01 Completed 200 OK in 1.002s
*/
```

sparql_handler.go

```
package handlers

import (
    "dbpedia-backend/cmd/api/services"
    "dbpedia-backend/cmd/api/utils"
    "encoding/json"
    "errors"
    "fmt"
    "log"
    "net/http"
    "os"
    "strings"
    "sync"
    "time"
)

func SPARQLHandler(w http.ResponseWriter, r *http.Request) {
    // CORS Headers
    w.Header().Set("Access-Control-Allow-Origin", "*")
    w.Header().Set("Access-Control-Allow-Methods", "POST, GET, OPTIONS")
    w.Header().Set("Access-Control-Allow-Headers", "Content-Type")

    if r.Method == "OPTIONS" {
        w.WriteHeader(http.StatusOK)
        return
    }

    var sparqlQuery string
    if r.Method == "GET" {
        sparqlQuery = r.URL.Query().Get("query")
    } else {
        var requestBody struct {
```

```
        Query string `json:"query"`
    }
    if err := json.NewDecoder(r.Body).Decode(&requestBody); err != nil {
        utils.HandleError(w, "Invalid request body", http.StatusBadRequest)
        return
    }
    sparqlQuery = requestBody.Query
}

if err := ValidateSPARQLQuery(sparqlQuery); err != nil {
    utils.HandleError(w, err.Error(), http.StatusBadRequest)
    return
}

endpoint := os.Getenv("VIRTUOSO_ENDPOINT")
if endpoint == "" {
    endpoint = "http://store:8890/sparql"
}

// CONCURRENCY & TIMEOUT GUARD
startTime := time.Now()
var wg sync.WaitGroup

var dataResult string
var countResult string
var dataErr error

isSelect := strings.Contains(strings.ToUpper(sparqlQuery), "SELECT")

// Goroutine 1: Δεδομένα
wg.Add(1)
go func() {
    defer wg.Done()
    dataResult, dataErr = services.ExecuteSPARQL(endpoint, sparqlQuery)
}()

// Goroutine 2: Παράλληλο Count
if isSelect {
    wg.Add(1)
    go func() {
        defer wg.Done()
        countQ := BuildCountQuery(sparqlQuery)
        countResult, _ = services.ExecuteSPARQL(endpoint, countQ)
    }()
}
```

```
done := make(chan struct{})
go func() {
   wg.Wait()
   close(done)
}()

// Περιμένουμε είτε να τελειώσουν όλα, είτε να περάσουν 5 δευτερόλεπτα
select {
case <-done:
   // Όλα ολοκληρώθηκαν κανονικά
case <-time.After(5 * time.Second):
   // Ενεργοποίηση Timeout αν το ερώτημα αργεί πολύ
   log.Println("[TIMEOUT] To background count query ακυρώθηκε λόγω καθυστέρησης.")
   if countResult == "" {
      countResult = "TIMEOUT_ERROR"
   }
}

executionTime := time.Since(startTime).Milliseconds()

if dataErr != nil {
   utils.HandleError(w, dataErr.Error(), http.StatusInternalServerError)
   return
}

// Ενσωμάτωση στατιστικών
var jsonResponse map[string]interface{}
if err := json.Unmarshal([]byte(dataResult), &jsonResponse); err == nil {

   stats := map[string]interface{}{
      "time_ms":           executionTime,
      "parallel_execution": isSelect,
   }

   if isSelect {
      if countResult == "TIMEOUT_ERROR" {
         stats["total_db_rows"] = "TIMEOUT"
      } else if countResult != "" {
         var cResp struct {
            Results struct {
               Bindings []map[string]struct {
                  Value string `json:"value"`
               } `json:"bindings"`
            } `json:"results"`
         }
```

```
            if json.Unmarshal([]byte(countResult), &cResp) == nil && len(cResp.Results.Bindings) > 0
{
                stats["total_db_rows"] = cResp.Results.Bindings[0]["total_count"].Value
            }
        }
    }

    jsonResponse["concurrency_stats"] = stats
    finalBytes, _ := json.Marshal(jsonResponse)
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write(finalBytes)
    return
  }

  w.Header().Set("Content-Type", "application/json")
  w.WriteHeader(http.StatusOK)
  w.Write([]byte(dataResult))
}

// ValidateSPARQLQuery ensures the query is not empty and contains basic SPARQL keywords
func ValidateSPARQLQuery(query string) error {
  // Check if the query is empty
  if strings.TrimSpace(query) == "" {
    return errors.New("SPARQL query cannot be empty")
  }

  // Check for basic SPARQL structure
  upperQuery := strings.ToUpper(query)
  if !(strings.Contains(upperQuery, "SELECT") ||
    strings.Contains(upperQuery, "ASK") ||
    strings.Contains(upperQuery, "DESCRIBE") ||
    strings.Contains(upperQuery, "CONSTRUCT")) {
    return errors.New("SPARQL query must contain SELECT, ASK, DESCRIBE, or CONSTRUCT")
  }

  return nil
}

// BuildCountQuery παίρνει ένα SPARQL ερώτημα και το μετατρέπει σωστά σε ερώτημα COUNT
func BuildCountQuery(originalQuery string) string {
  lines := strings.Split(originalQuery, "\n")
  var prefixes []string
  var body []string

  // 1. Διαχωρίζουμε τα PREFIX από το κυρίως σώμα
```

```
    for _, line := range lines {
        trimmedLine := strings.TrimSpace(line)
        if strings.HasPrefix(strings.ToUpper(trimmedLine), "PREFIX") {
            prefixes = append(prefixes, trimmedLine)
        } else {
            body = append(body, line)
        }
    }

    prefixString := strings.Join(prefixes, "\n")
    bodyString := strings.Join(body, "\n")

     // 2. Κόβουμε οτιδήποτε υπάρχει μετά την τελευταία αγκύλη "}" (όπως LIMIT, ORDER BY, OFFSET)
    lastBraceIdx := strings.LastIndex(bodyString, "}")
    if lastBraceIdx != -1 {
        // Κρατάμε το query μόνο μέχρι και την τελευταία αγκύλη
        bodyString = bodyString[:lastBraceIdx+1]
    }

    // 3. Ενώνουμε τα PREFIX στην κορυφή, και "τυλίγουμε" μόνο το καθαρό σώμα στο COUNT
        return fmt.Sprintf("%s\nSELECT (COUNT(*) AS ?total_count) WHERE { {\n%s\n} }", prefixString, bodyString)
}
```

ΠΑΡΑΡΤΗΜΑ Γ: ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ ΕΠΙΠΕΔΟΥ ΠΑΡΟΥΣΙΑΣΗΣ

Dockerfile

```
# Χρησιμοποιούμε Node.js image
FROM node:20-alpine

# Ορίζουμε τον φάκελο εργασίας μέσα στο container
WORKDIR /app

# Αντιγράφουμε τα αρχεία ρυθμίσεων
COPY package*.json ./

# Εγκαθιστούμε τα dependencies
RUN npm install

# Αντιγράφουμε όλο τον υπόλοιπο κώδικα
COPY . .
```

```
# Ανοίγουμε την πόρτα του Vite
EXPOSE 5173

# Ξεκινάμε την εφαρμογή με --host για να είναι προσβάσιμη έξω από το docker
CMD ["npm", "run", "dev", "--", "--host"]
```

main.jsx

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <App />
  </StrictMode>,
)
```

App.jsx

```
import { useState } from "react";
import Navbar from "./components/Navbar";
import QueryEditor from "./components/QueryEditor";
import PredefinedQueries from "./components/PredefinedQueries";
import ResultsTable from "./components/ResultsTable";
import QueryHistory from "./components/QueryHistory";
import { executeSPARQLQuery } from "./api/api";

function App() {
  const [results, setResults] = useState(null);
  const [queryHistory, setQueryHistory] = useState([]);
  const [currentQuery, setCurrentQuery] = useState("");
  const [error, setError] = useState(null);


  const [queryStats, setQueryStats] = useState(null);

  const addToHistory = (query) => {
    setQueryHistory((prev) => {
      if (prev.length > 0 && prev[0] === query) return prev;
      const updatedHistory = [query, ...prev.slice(0, 9)];
      localStorage.setItem("queryHistory", JSON.stringify(updatedHistory));
```

```
    return updatedHistory;
  });
};

const handleExecuteQuery = async (query) => {
  try {
    setResults(null);
    setError(null);
    setQueryStats(null);

    const data = await executeSPARQLQuery(query);


    if (!data || typeof data !== 'object' || !data.head || !data.results) {
        throw new Error("Ο server επέστρεψε μη έγκυρα δεδομένα. Πιθανό συντακτικό λάθος στο SPARQL.");
    }

    const rowCount = data.results?.bindings?.length || 0;

    // Διαβάζουμε τα στατιστικά της ταυτοχρονίας (Goroutines) από τη Go
    const backendStats = data.concurrency_stats || {};

    setResults(data);

    // Αποθηκεύουμε τα στατιστικά στο React State
    setQueryStats({
      rows: rowCount,
      time: backendStats.time_ms || 0,
      total_db: backendStats.total_db_rows,
      isParallel: backendStats.parallel_execution
    });

    addToHistory(query);

  } catch (err) {
    console.error("Query Error:", err);
    setError(err.message);
  }
};

const handleSelectQuery = (query) => {
  setResults(null);
  setError(null);
  setQueryStats(null);
  setCurrentQuery(query);
```

```
 };

 return (
  <div className="min-h-screen bg-white font-sans text-gray-900 flex flex-col">
   <Navbar />

   <main className="flex-grow max-w-7xl mx-auto w-full p-6 space-y-8">

    <div className="grid grid-cols-1 lg:grid-cols-3 gap-8 items-start">

     <div className="lg:col-span-2 space-y-6">
      <QueryEditor
       onExecute={handleExecuteQuery}
       initialValue={currentQuery}
       errorMessage={error}
      />

      {results && (
       <div className="mt-8">
        <div className="flex justify-between items-end mb-2 border-b border-gray-300 pb-1">
         <h3 className="text-lg font-bold text-[#003366]">
          Query Results
         </h3>

         {/* ΕΜΦΑΝΙΣΗ ΣΤΑΤΙΣΤΙΚΩΝ   */}
         {queryStats && (
          <div className="flex space-x-3 text-[11px] font-mono text-gray-700 bg-[#f8fafc] px-3
py-1.5 rounded border border-gray-300 shadow-sm items-center">

           {queryStats.isParallel && (
            <span className="flex items-center text-indigo-600 bg-indigo-100 px-2 py-0.5
rounded-full font-bold uppercase tracking-wider text-[9px]">
             <span className="mr-1 animate-pulse">⚡</span> Parallel
            </span>
           )}

           <span>Time: <strong className="text-indigo-700">{queryStats.time}
ms</strong></span>
           <span className="text-gray-300">|</span>
           <span>Rows: <strong className="text-emerald-
700">{queryStats.rows}</strong></span>

           {queryStats.total_db && (
            <>
             <span className="text-gray-300">|</span>
             <span className="flex items-center">
```

```
                        {queryStats.total_db === "TIMEOUT" ? (
                          <span className="text-amber-600 font-bold italic">
                            ⚠️ Αδυναμία επιστροφής συνολικού όγκου (Timeout)
                          </span>
                        ) : (
                                                <>Total in DB: <strong className="text-amber-600 ml-1">{queryStats.total_db}</strong></>
                        )}
                      </span>
                    </>
                  )}
                </div>
              )}
            </div>

            <div className="bg-white border border-gray-300 rounded overflow-hidden">
              <ResultsTable results={results} />
            </div>
          </div>
        )}
      </div>

      <div className="space-y-6 bg-[#f9fafb] p-4 rounded border border-gray-200">
        <div>
          <h3 className="text-sm font-bold text-[#003366] uppercase mb-3 border-b border-gray-300 pb-1">
            Examples
          </h3>
          <PredefinedQueries onSelectQuery={handleSelectQuery} />
        </div>

        <div className="pt-4">
          <h3 className="text-sm font-bold text-[#003366] uppercase mb-3 border-b border-gray-300 pb-1">
            History
          </h3>
          <QueryHistory onSelectQuery={handleSelectQuery} />
        </div>
      </div>

    </div>
  </main>

  <footer className="bg-[#f5f5f5] border-t border-gray-300 py-4 text-center text-gray-600 text-sm mt-auto">
    &copy; 2026 DBpedia Greek Node
```

```
      </footer>
    </div>
  );
}

export default App;
```

App.css

```
#root {
  max-width: 1280px;
  margin: 0 auto;
  padding: 2rem;
  text-align: center;
}

.logo {
  height: 6em;
  padding: 1.5em;
  will-change: filter;
  transition: filter 300ms;
}
.logo:hover {
  filter: drop-shadow(0 0 2em #646cffaa);
}
.logo.react:hover {
  filter: drop-shadow(0 0 2em #61dafbaa);
}

@keyframes logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}

@media (prefers-reduced-motion: no-preference) {
  a:nth-of-type(2) .logo {
    animation: logo-spin infinite 20s linear;
  }
}

.card {
```

```
  padding: 2em;
}

.read-the-docs {
  color: #888;
}
```

api.js

```
import axios from "axios";

const API_BASE_URL = 'http://localhost:8080';

const handleRequest = async (method, url, data = null) => {
  try {
    const response = await axios({
      method,
      url: `${API_BASE_URL}${url}`,
      data
    });
    return response.data;
  } catch (error) {
    console.error("API Error:", error);


    let errorMessage = "Unknown error occurred.";

    if (error.response && error.response.data) {
      // Αν ο server έστειλε απάντηση (π.χ. Syntax Error από το Virtuoso)
      const serverData = error.response.data;

      // Αν είναι αντικείμενο, το κάνουμε string, αλλιώς το παίρνουμε ως έχει
      errorMessage = typeof serverData === 'object'
        ? (serverData.message || JSON.stringify(serverData))
        : serverData;

    } else if (error.message) {
      // Αν είναι θέμα δικτύου (π.χ. Network Error)
      errorMessage = error.message;
    }

    throw new Error(errorMessage);
  }
};
```

```
export const executeSPARQLQuery = (query) => handleRequest("POST", "/sparql", { query });

export const fetchPredefinedQueries = () => handleRequest("GET", "/predefined?query=all_classes");
```

Navbar.jsx

```
import React from "react";

const Navbar = () => {
  return (
    <nav className="bg-blue-900 text-white shadow-md">
      <div className="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8">
        <div className="flex items-center h-16">
          {/* Logo / Title Area */}
          <div className="flex items-center gap-3">
            <div className="bg-white text-blue-900 font-bold h-8 w-8 flex items-center justify-center
rounded text-lg">
              DB
            </div>
            <span className="font-semibold text-xl tracking-tight">
              Virtuoso SPARQL Endpoint
            </span>
          </div>
        </div>
      </div>
    </nav>
  );
};

export default Navbar;
```

PredefinedQuerries.jsx

```
import React from "react";

const PredefinedQueries = ({ onSelectQuery }) => {

  const queries = [
    {
      label: "GR Ελληνικά Νησιά",
      desc: "Με εικόνα, χάρτη και πληθυσμό.",
      query: `PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
```

```
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

SELECT DISTINCT ?island ?name ?population ?image ?map WHERE {
  ?island rdf:type dbo:Island .
  ?island rdfs:label ?name .
  FILTER (LANG(?name) = 'el')

  {
    { ?island dbo:country <http://el.dbpedia.org/resource/Ελλάδα> }
    UNION
    { ?island dbo:location <http://el.dbpedia.org/resource/Ελλάδα> }
  }

  OPTIONAL { ?island dbo:populationTotal ?population }

  # --- IMAGE & MAP ---
  OPTIONAL { ?island dbo:thumbnail ?image }
  OPTIONAL {
    ?island geo:lat ?lat ; geo:long ?lon .
    BIND(CONCAT("https://www.google.com/maps?q=", STR(?lat), ",", STR(?lon)) AS ?map)
  }
} ORDER BY DESC(?population) LIMIT 50`
    },
    {
      label: "🏛️ Μουσεία (Ελλάδα)",
      desc: "Με εικόνα, χάρτη και περιγραφή.",
      query: `PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

SELECT DISTINCT ?museum ?name ?image ?map ?info WHERE {
  ?museum rdf:type dbo:Museum .
  ?museum rdfs:label ?name .
  FILTER (LANG(?name) = 'el')

  ?museum dbo:abstract ?abstract .
  FILTER (LANG(?abstract) = 'el')

  FILTER (
    CONTAINS(LCASE(?abstract), "ελλάδα") ||
    CONTAINS(LCASE(?abstract), "αθήνα") ||
    CONTAINS(LCASE(?abstract), "θεσσαλονίκη") ||
    CONTAINS(LCASE(?abstract), "κρήτη") ||
    CONTAINS(LCASE(?abstract), "αρχαιολογικό")
  )
  BIND(?abstract AS ?info)
```

```
  # --- IMAGE & MAP ---
  OPTIONAL { ?museum dbo:thumbnail ?image }
  OPTIONAL {
    ?museum geo:lat ?lat ; geo:long ?lon .
    BIND(CONCAT("https://www.google.com/maps?q=", STR(?lat), ",", STR(?lon)) AS ?map)
  }
} LIMIT 50`
    },
    {
      label: "⚽ Ελληνικές Ομάδες",
      desc: "Με σήμα ομάδας (όπου υπάρχει) και πληροφορίες.",
      query: `PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

SELECT DISTINCT ?team ?name ?image ?map ?info WHERE {
  ?team rdf:type dbo:SoccerClub .
  ?team rdfs:label ?name .
  FILTER (LANG(?name) = 'el')

  ?team dbo:abstract ?abstract .
  FILTER (LANG(?abstract) = 'el')

  FILTER (
    CONTAINS(LCASE(?abstract), "ελληνική") ||
    CONTAINS(LCASE(?abstract), "ελλάδα") ||
    CONTAINS(LCASE(?abstract), "αθήνα") ||
    CONTAINS(LCASE(?abstract), "θεσσαλονίκη") ||
    CONTAINS(LCASE(?abstract), "παε") ||
    CONTAINS(LCASE(?abstract), "σούπερ λιγκ")
  )
  BIND(?abstract AS ?info)

  # --- IMAGE & MAP ---
  OPTIONAL { ?team dbo:thumbnail ?image }
  OPTIONAL {
    ?team geo:lat ?lat ; geo:long ?lon .
    BIND(CONCAT("https://www.google.com/maps?q=", STR(?lat), ",", STR(?lon)) AS ?map)
  }
} LIMIT 50`
    },
    {
      label: "📜 Αρχαίοι Φιλόσοφοι",
      desc: "Με εικόνα (προτομή/άγαλμα) και βιογραφικό.",
```

```
    query: `PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?name ?image ?info WHERE {
  ?philosopher a dbo:Philosopher .
  ?philosopher rdfs:label ?name .
  FILTER (LANG(?name) = 'el')

  ?philosopher dbo:abstract ?abstract .
  FILTER (LANG(?abstract) = 'el')

  FILTER (CONTAINS(LCASE(?abstract), "έλληνας") || CONTAINS(LCASE(?abstract), "ελληνίδα")
|| CONTAINS(LCASE(?abstract), "αθηναίος"))
  BIND(?abstract AS ?info)

  # --- IMAGE ONLY (Δεν έχουν χάρτη συνήθως) ---
  OPTIONAL { ?philosopher dbo:thumbnail ?image }
} LIMIT 20`
   },
   {
    label: " 🔬 Ελληνικά Πανεπιστήμια",
    desc: "Με λογότυπο (αν υπάρχει) και τοποθεσία.",
    query: `PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>

SELECT DISTINCT ?uni ?name ?image ?map ?info WHERE {
  ?uni rdf:type dbo:University .
  ?uni rdfs:label ?name .
  FILTER (LANG(?name) = 'el')

  ?uni dbo:country <http://el.dbpedia.org/resource/Ελλάδα> .

  OPTIONAL {
    ?uni dbo:abstract ?abstract .
    FILTER (LANG(?abstract) = 'el')
  }
  BIND(?abstract AS ?info)

  # --- IMAGE & MAP ---
  OPTIONAL { ?uni dbo:thumbnail ?image }
  OPTIONAL {
    ?uni geo:lat ?lat ; geo:long ?lon .
    BIND(CONCAT("https://www.google.com/maps?q=", STR(?lat), ",", STR(?lon)) AS ?map)
  }
```

```
} LIMIT 20`
   }
 ];

 return (
  <div className="flex flex-col gap-3">
   {queries.map((q, index) => (
    <button
     key={index}
     onClick={() => onSelectQuery(q.query)}
      className="text-left group bg-white hover:bg-[#f0f4f8] border border-gray-300 hover:border-
[#003366] rounded p-3 transition-all duration-200 shadow-sm"
    >
     <div className="font-bold text-[#003366] text-sm group-hover:text-[#002244]">
      {q.label}
     </div>
     <div className="text-xs text-gray-500 mt-1">
      {q.desc}
     </div>
    </button>
   ))}
  </div>
 );
};

export default PredefinedQueries;
```

QuerryEditor.jsx

```
import { useState, useEffect } from "react";

// Προσθέσαμε το prop 'errorMessage'
const QueryEditor = ({ onExecute, initialValue, errorMessage }) => {
 const [query, setQuery] = useState(
  "SELECT * WHERE { ?s ?p ?o } LIMIT 10"
 );

 useEffect(() => {
  if (initialValue) {
   setQuery(initialValue);
  }
 }, [initialValue]);

 const handleSubmit = (e) => {
```

```
    e.preventDefault();
    if (onExecute) {
      onExecute(query);
    }
  };

  return (
    <div className="bg-[#f5f7f9] p-4 rounded border border-gray-300 shadow-sm">
      <div className="mb-2 flex justify-between items-end">
        <label className="text-sm font-bold text-[#003366] uppercase">
          SPARQL Query Editor
        </label>
        {/* Ένδειξη status */}
        {errorMessage ? (
          <span className="text-xs font-bold text-red-600 animate-pulse">● Compilation Failed</span>
        ) : (
          <span className="text-xs font-bold text-green-600">● Ready</span>
        )}
      </div>

      <form onSubmit={handleSubmit} className="flex flex-col gap-4">
        <textarea
          className={`w-full h-80 p-3 font-mono text-sm bg-white border rounded-sm outline-none text-gray-800 shadow-inner
            ${errorMessage ? 'border-red-500 focus:ring-1 focus:ring-red-500' : 'border-gray-400 focus:border-[#003366] focus:ring-1 focus:ring-[#003366]'}
          `}
          value={query}
          onChange={(e) => setQuery(e.target.value)}
          spellCheck="false"
          placeholder="Write your SPARQL query here..."
        />

        {/* --- COMPILER ERROR MESSAGE BOX --- */}
        {errorMessage && (
          <div className="bg-[#fff5f5] border-l-4 border-red-600 p-3 rounded-sm shadow-sm transition-all animate-fadeIn">
            <div className="flex items-center gap-2 mb-1">
              <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" strokeWidth={2} stroke="currentColor" className="w-4 h-4 text-red-600">
                <path strokeLinecap="round" strokeLinejoin="round" d="M12 9v3.75m9-.75a9 9 0 11-18 0 9 9 0 0118 0zm-9 3.75h.008v.008H12v-.008z" />
              </svg>
              <span className="text-xs font-bold text-red-700 uppercase">Syntax Error / Server Message</span>
            </div>
```

```
          <pre className="text-xs text-red-800 font-mono whitespace-pre-wrap break-all ml-6">
            {errorMessage}
          </pre>
        </div>
      )}

      <div className="flex justify-between items-center pt-2">
        <div className="text-sm text-gray-600 hidden sm:block">
          Press <strong>Run Query</strong> to execute.
        </div>

        <div className="flex gap-2 w-full sm:w-auto justify-end">
          <button
            type="button"
            onClick={() => setQuery("")}
              className="px-4 py-1.5 text-sm bg-white hover:bg-gray-100 text-gray-700 font-medium
rounded-sm border border-gray-300 transition-colors"
          >
            Clear
          </button>
          <button
            type="submit"
              className="px-6 py-1.5 text-sm bg-[#003366] hover:bg-[#002244] text-white font-bold
rounded-sm shadow-sm transition-colors flex items-center gap-2"
          >
            <span>▶</span> Run Query
          </button>
        </div>
      </div>
    </form>
  </div>
 );
};

export default QueryEditor;
```

QueryHistory.jsx

```
import React, { useEffect, useState } from "react";

const QueryHistory = ({ onSelectQuery }) => {
```

```
  const [history, setHistory] = useState([]);

  // Φόρτωση ιστορικού κατά την εκκίνηση και όταν αλλάζει το localStorage
  useEffect(() => {
    const loadHistory = () => {
      const storedHistory = localStorage.getItem("queryHistory");
      if (storedHistory) {
        setHistory(JSON.parse(storedHistory));
      }
    };

    loadHistory();

    // Ακούμε για αλλαγές (αν το ιστορικό ενημερωθεί από άλλο tab ή component)
    window.addEventListener("storage", loadHistory);
    // Επειδή το localStorage δεν ενημερώνει το ίδιο το tab,
    // re-render component όταν αλλάζει το state του history

    // Εναλλακτικά, απλά διαβάζουμε το prop
    // διαβάζουμε απευθείας το localStorage κάθε δευτερόλεπτο
    const interval = setInterval(loadHistory, 1000);

    return () => {
      window.removeEventListener("storage", loadHistory);
      clearInterval(interval);
    };
  }, []);

  if (history.length === 0) {
    return (
      <div className="text-sm text-gray-400 italic text-center py-4">
        No history yet. Run a query!
      </div>
    );
  }

  return (
    <div className="flex flex-col gap-2 max-h-80 overflow-y-auto pr-1 custom-scrollbar">
      {history.map((q, index) => (
        <div
          key={index}
          onClick={() => onSelectQuery(q)}
                  className="cursor-pointer bg-gray-50 hover:bg-blue-50 border-l-4 border-gray-300
hover:border-[#003366] p-3 rounded-r-md transition-colors duration-200 group"
        >
```

```
        <div className="text-xs font-mono text-gray-600 group-hover:text-blue-900 line-clamp-3 break-all">
          {q}
        </div>
        <div className="text-[10px] text-gray-400 mt-1 text-right font-semibold uppercase tracking-wider group-hover:text-blue-400">
          Restore
        </div>
      </div>
    ))}
  </div>
 );
};

export default QueryHistory;
```

ResultsTable.jsx

```
import React, { useState } from 'react';

const TextCell = ({ text }) => {
 const [expanded, setExpanded] = useState(false);
 const limit = 80; // Μικρότερο όριο για πιο compact πίνακα

 if (!text) return null;
 if (text.length <= limit) return <span>{text}</span>;

 return (
  <div>
   <span>{expanded ? text : `${text.substring(0, limit)}...`}</span>
   <button
    onClick={() => setExpanded(!expanded)}
    className="ml-1 text-[10px] font-bold text-blue-600 hover:text-blue-800 bg-blue-50 px-1.5 py-0.5 rounded uppercase tracking-wide"
   >
    {expanded ? "Less" : "More"}
   </button>
  </div>
 );
};

// Component για το κελί του Χάρτη (Pin Icon)
const MapCell = ({ url }) => {
 if (!url) return <span className="text-gray-300">-</span>;
```

```
  return (
    <a
      href={url}
      target="_blank"
      rel="noopener noreferrer"
      className="inline-flex items-center justify-center w-8 h-8 rounded-full bg-blue-50 text-blue-600 hover:bg-blue-600 hover:text-white transition-all duration-200 shadow-sm border border-blue-100"
      title="View on Google Maps"
    >
      <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" strokeWidth={2} stroke="currentColor" className="w-5 h-5">
        <path strokeLinecap="round" strokeLinejoin="round" d="M15 10.5a3 3 0 11-6 0 3 3 0 016 0z" />
        <path strokeLinecap="round" strokeLinejoin="round" d="M19.5 10.5c0 7.142-7.5 11.25-7.5 11.25S4.5 17.642 4.5 10.5a7.5 7.5 0 1115 0z" />
      </svg>
    </a>
  );
};

// Component για το κελί της Εικόνας
const ImageCell = ({ url }) => {
  if (!url) return <div className="w-12 h-12 bg-gray-100 rounded-md border border-gray-200 flex items-center justify-center text-gray-300 text-xs">No Img</div>;

  return (
    <div className="relative group w-16 h-12">
      <img
        src={url}
        alt="thumb"
        className="w-full h-full object-cover rounded-md shadow-sm border border-gray-200 transition-transform duration-300 group-hover:scale-150 group-hover:z-10 relative bg-white"
        onError={(e) => { e.target.style.display = 'none'; }}
      />
    </div>
  );
};

const ResultsTable = ({ results }) => {
  if (!results || !results.head || !results.results) return null;

  const variables = results.head.vars;
  const bindings = results.results.bindings;

  if (bindings.length === 0) {
    return (
      <div className="p-8 text-center bg-blue-50 text-[#003366] rounded border border-blue-100">
```

```
      <span className="font-semibold">No results found.</span> Try a different query.
    </div>
  );
 }

 const isUrl = (str) => typeof str === 'string' && (str.startsWith('http://') || str.startsWith('https://'));

 return (
  <div className="flex flex-col h-full">
      <div className="overflow-auto w-full max-h-[600px] border border-gray-300 rounded-sm shadow-sm custom-scrollbar bg-white">
     <table className="min-w-full text-left text-sm border-collapse">
      <thead className="bg-[#003366] text-white sticky top-0 z-20 shadow-md">
       <tr>
         <th className="px-4 py-3 font-semibold uppercase tracking-wider text-xs border-r border-blue-800 w-12 text-center">#</th>
         {variables.map((v) => (
          <th key={v} className="px-4 py-3 font-semibold uppercase tracking-wider text-xs border-r border-blue-800 whitespace-nowrap">
           {v === 'image' ? 'IMG' : v === 'map' ? 'MAP' : v}
          </th>
         ))}
       </tr>
      </thead>

      <tbody className="bg-white divide-y divide-gray-200">
       {bindings.map((row, rowIndex) => (
        <tr
         key={rowIndex}
           className={`transition-colors duration-150 ${rowIndex % 2 === 0 ? 'bg-white' : 'bg-[#f8fbff]'} hover:bg-blue-50`}
        >
         <td className="px-2 py-2 text-gray-500 font-mono text-xs border-r border-gray-200 text-center align-middle">
          {rowIndex + 1}
         </td>
         {variables.map((v) => {
          const cellData = row[v];
          const value = cellData ? cellData.value : null;

          // --- ΕΙΔΙΚΑ ΚΕΛΙΑ (Custom Renders) ---
          if (v === 'image') {
              return <td key={v} className="px-2 py-1 border-r border-gray-200 align-middle text-center"><ImageCell url={value} /></td>;
          }
          if (v === 'map') {
```

```
            return <td key={v} className="px-2 py-1 border-r border-gray-200 align-middle text-center"><MapCell url={value} /></td>;
          }
          if (v === 'info') {
            return <td key={v} className="px-4 py-2 border-r border-gray-200 align-top min-w-[250px]"><TextCell text={value} /></td>;
          }
          // ------------------------------------

          if (!value) return <td key={v} className="px-4 py-2 border-r border-gray-200"></td>;

          const isLink = cellData.type === 'uri' || isUrl(value);

          return (
            <td key={v} className="px-4 py-2 border-r border-gray-200 align-middle whitespace-nowrap max-w-xs overflow-hidden text-ellipsis">
              {isLink ? (
                <a
                  href={value}
                  target="_blank"
                  rel="noopener noreferrer"
                  className="text-[#0056b3] hover:text-[#003366] hover:underline font-medium"
                >
                  Link
                </a>
              ) : (
                <span className="text-gray-800">
                  {value}
                  {cellData['xml:lang'] && (
                    <span className="ml-1 text-[9px] text-gray-400 bg-gray-100 px-1 rounded">
                      {cellData['xml:lang']}
                    </span>
                  )}
                </span>
              )}
            </td>
          );
        })}
      </tr>
    ))}
  </tbody>
</table>
</div>
<div className="bg-gray-50 border-t border-gray-300 p-2 text-right text-xs text-gray-600 font-medium">
Total Results: {bindings.length}
```

```
      </div>
    </div>
  );
};

export default ResultsTable;
```