
# ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών


**Σχεδιασμός και υλοποίηση πρότυπου διαχειριστή ταυτόχρονων συνδιαλλαγών βασισμένο στον έλεγχο πολλαπλών εκδόσεων δεδομένων και υλοποίηση κατανεμημένου συλλέκτη μη έγκυρων δεδομένων για τη μη σχεσιακή βάση MongoDB**

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του


ΓΙΑΛΟΥΣΗ ΜΙΛΤΙΑΔΗ

**Επιβλέπουσα**: Θεοδώρα Βαρβαρίγου
Καθηγήτρια Ε.Μ.Π.



Αθήνα, Ιούλιος 2017

# ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ


ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Τομέας Τεχνολογίας Πληροφορικής & Υπολογιστών


## Σχεδιασμός και υλοποίηση πρότυπου διαχειριστή ταυτόχρονων συνδιαλλαγών βασισμένο στον έλεγχο πολλαπλών εκδόσεων δεδομένων και υλοποίηση κατανεμημένου συλλέκτη μη έγκυρων δεδομένων για τη μη σχεσιακή βάση MongoDB

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του


ΓΙΑΛΟΥΣΗ ΜΙΛΤΙΑΔΗ

Επιβλέπουσα: Θεοδώρα Βαρβαρίγου
Καθηγήτρια Ε.Μ.Π.


Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25η Ιουλίου 2017.

| | | |
|---|---|---|
| …… | …… | …… |
| Θ. Βαρβαρίγου | Σ. Παπαβασιλείου | Δ. Ασκούνης |
| Καθηγήτρια Ε.Μ.Π. | Καθηγητής Ε.Μ.Π. | Καθηγητής Ε.Μ.Π. |

Αθήνα, Ιούλιος 2017

……..……

ΓΙΑΛΟΥΣΗΣ ΜΙΛΤΙΑΔΗΣ
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

# Περιεχόμενα

# Κατάλογος σχημάτων

# ΠΕΡΙΛΗΨΗ

Στις μέρες μας η ραγδαία ανάπτυξη του cloud computing παράλληλα με την κατακόρυφη αύξηση του αριθμού των χρηστών του διαδικτύου αλλά και των υπηρεσιών που παρέχονται μέσω αυτού έχει οδηγήσει στην εξέλιξη των συστημάτων διαχείρισης βάσεων δεδομένων. Μία προφανής ένδειξη του γεγονότος αυτού είναι και η εμφάνιση των μη σχεσιακών βάσεων, όπως η MongoDb, η δημοτικότητα των οποίων πυροδοτήθηκε από τις ανάγκες των εταιριών κολοσσών στο χώρο του Web 2.0, όπως η Google, η Facebook και η Amazon και γενικότερα της ανάπτυξης του τομέα των εφαρμογών μεγάλων δεδομένων (Big Data) και πραγματικού χρόνου (real-time web).

Στόχος της διπλωματικής αυτής είναι η υλοποίηση πρότυπου διαχειριστή ταυτόχρονων συνδιαλλαγών βασισμένο στον έλεγχο πολλαπλών εκδόσεων δεδομένων. Το σύστημα το οποίο υλοποιήσαμε έχει σχεδιαστεί με πλαίσιο το CoherentPaas, πάνω από το οποίο ουσιαστικά δημιουργήσαμε ένα αφαιρετικό επίπεδο (level of abstraction), που μας προσφέρει επιπλέον ευελιξία και ευκολία τροποποίησης της συμπεριφοράς του. Επιπλέον στη προσπάθεια βελτίωσης και επέκτασης του υπάρχοντος συστήματος, σχεδιάσαμε και υλοποιήσαμε έναν κατανεμημένο συλλέκτη μη έγκυρων δεδομένων για τη μη σχεσιακή βάση MongoDB , αναπτύσσοντας μία αυτόνομη Java εφαρμογή η οποία επικοινωνεί με τις διάφορες εφαρμογές μέσω διεύρυνσης που πραγματοποιήσαμε στην υπάρχουσα διεπαφή και με την βάση δεδομένων μέσω του MongoDB API που προσφέρει η Java.

**ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ**

Μη σχεσιακές βάσεις δεδομένων, συλλογή απορριμμάτων, MongoDB, Java, ACID, απομόνωση στιγμιότυπου, έλεγχος συγχρονικότητας πολλαπλών εκδόσεων, διαχειριστής συναλλαγών

# ABSTRACT

Nowadays the rapid development of cloud computing alongside the rise in the number of the internet users, but also the services provided by it, has led to the evolution of the database management system (DBMS). A clear indication of this fact is the appearance of the NoSQL databases, like MongoDB, whose popularity was triggered by the needs of Web 2.0 companies such as Amazon, Google and Facebook and generally the advancements in the Big Data and real-time web applications.

The purpose of this thesis is the design and implementation of a concurrent transactional manager based in the multiversion concurrency control (MVCC or MCC). The system which we implemented has been designed using the existing structure provided by CoherentPass, atop of who we created a level of abstraction providing us additional flexibility and the easy of manipulating the existing behavior. Furthermore, in the process of improving and extending the existing system we designed and realized a distributed garbage collector of invalid data versions for the MongoDB database, by developing a standalone Java application. This standalone Java application communicates with the various user applications through the expanded interface we provided and with the database through the MongoDB API provided by Java.

**KEYWORDS**

NoSQL Databases, MongoDB, Garbage Collector, ACID, Snapshot Isolation, Multiversion Concurrency Control, Transactional Management, Apache Avro, Java

# Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά την επιβλέποντα καθηγήτρια της διπλωματικής μου εργασίας Θεοδώρα Βαρβαρίγου για τη δυνατότητα που μου έδωσε να ασχοληθώ με ένα τόσο ενδιαφέρον αντικείμενο, με πολύ μεγάλο ερευνητικό και πρακτικό ενδιαφέρον. Επίσης, ευχαριστώ θερμά τον υποψήφιο διδάκτορα Παύλο Κρανά για την συμμετοχή στην επίβλεψη της διπλωματικής εργασίας και για το χρόνο που αφιέρωσε συνολικά.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου, για την υποστήριξη τους προς εμένα καθ' όλη τη διάρκεια των σπουδών μου και τη συμβολή τους στην ολοκλήρωση αυτών.

Γιαλούσης Μιλτιάδης

# ΚΕΦΑΛΑΙΟ 1
# Εισαγωγή

Η άνθιση του οικοσυστήματος του υπολογιστικού νέφους οδήγησε πλήθος εταιριών να μεταφέρουν τις υπηρεσίες τους σε αυτό, διαμορφώνοντας απαιτήσεις με μεγάλο βαθμό ποικιλομορφίας όσο αφορά τη διαχείριση δεδομένων. Οι απλές εφαρμογές στο παρελθόν βασίζονταν κυρίως σε OLTP (Online transaction processing) φορτία, όπου τα κλασσικά σχεσιακά συστήματα βάσεων δεδομένων κυριαρχούσαν. Αυτές οι βάσεις δεδομένων μπορούν να προσπελαστούν από γνωστές SQL γλώσσες και να εξασφαλίσουν ACID ιδιότητες βάσης δεδομένων κατά την εκτέλεση παράλληλων συναλλαγών, παρέχοντας με αυτό το τρόπο την απαραίτητο εννοιολογικό φάσμα των συναλλαγών. Ωστόσο, οι εφαρμογές του νέφους έχουν αυξημένες ανάγκες με επιπλέον απαιτήσεις, όπως η υψηλή διαθεσιμότητα και την ικανότητα να κλιμακώνονται ελαστικά σε πλήθος *εκδοχών (instances)* έτσι ώστε να ανταπεξέλθουν ποικίλα φορτία. Από την άλλη πλευρά δεν απαιτούν ιδιαίτερα υψηλό επίπεδο συνοχής δεδομένων. Επιπροσθέτως, η ανάδειξη IoT (Internet of Things) εφαρμογών οδήγησε στην έξαρση του κλάδου του big data όπου και εκεί η συνοχή δεν είναι πρώτη προτεραιότητα. Αντίθετα πρωταρχική έννοια για τις Big Data εφαρμογές είναι η δυνατότητα κλιμάκωσης της βάσης δεδομένων έτσι ώστε να εξασφαλίζει χαμηλή καθυστέρηση και υψηλή ικανότητα διεκπεραίωσης κάτω από έντονα φορτία.

Για να ανταποκριθεί στις νέες αυτές απαιτήσεις, μία νέα κατηγορία αποθήκευσης δεδομένων εμφανίστηκε τα τελευταία χρόνια, κοινώς γνωστή ως NoSQL συστήματα βάσεων δεδομένων. Εκτός από την πολύ απλούστερο μοντέλο δεδομένων, μία από τις κύριες διαφορές μεταξύ άλλων είναι η έλλειψη υποστήριξης του εννοιολογικού φάσματος συναλλαγών. Οι NoSQL βάσεις δεδομένων έχουν αναπτυχθεί, έχοντας εξαρχής ως σχεδιαστικό στόχο να έχουν ανοχή στο διαχωρισμό σε μικρότερα κομμάτια και επικεντρώνονται κυρίως στην υψηλή διαθεσιμότητα αντί για τη συνοχή, η οποία τελικά επιτυγχάνεται από τους επιμέρους κόμβους. Το γεγονός ότι μπορεί να χωριστεί σε μικρότερα κομμάτια εύκολα επιτρέπει στη NoSQL βάση δεδομένων να κλιμακώνεται πολύ εύκολα σε μεγάλο αριθμό κόμβων, προσφέροντας με αυτό το τρόπο χαμηλή καθυστέρηση (latency) και υψηλή διεκπεραιωτική ικανότητα (throughput).

Το εμφανές μειονέκτημα χρήση NoSQL βάσεων δεδομένων είναι η έλλειψη του εννοιολογικού φάσματος συναλλαγών. Για τις περισσότερες βάσεις κλειδιού-τιμής ( key-value stores) αυτός είναι ένας αποδεκτός συμβιβασμός καθώς η πλειοψηφία των δεδομένων που αποθηκεύονται δεν είναι κρίσιμα και αντιστοιχούν σε big data δεδομένα παραγόμενα από IoT αισθητήρες ή άλλα δεδομένα παρακολούθησης, όπου μόνο εξεταστικές λειτουργίες πραγματοποιούνται και δεν υπάρχουν συνήθως

τροποποιήσεις. Ωστόσο, οι βάσεις εγγράφων (document-store) παρέχουν ένα πολύ πιο περίπλοκο μοντέλο δεδομένων, επιτρέποντας στον προγραμματιστή της εφαρμογής να το εκμεταλλευτεί για να αποθηκεύσει το μοντέλο δεδομένων της εφαρμογής του. Αυτό οδηγεί στην προσπέλαση των εγγραφών κάτω από OLTP φορτία, όπου συχνές τροποποιήσεις δεδομένων πραγματοποιούνται καθιστώντας επιτακτική την παροχή του εννοιολογικού φάσματος των συναλλαγών από το επίπεδο πρόσβασης δεδομένων (data access layer). Καθώς οι NoSQL γλώσσες δεν μπορούν να διασφαλίσουν τις ACID ιδιότητες, αυτό διευθετείται συνήθως στο επίπεδο εφαρμογών (application level). Ο προγραμματιστής της εφαρμογής οφείλει να υλοποιήσει όλη την πολυπλοκότητα η οποία παρέχεται από τις παραδοσιακές βάσεις δεδομένων, γεγονός το οποίο καθιστά τον κώδικα επιρρεπή σε λάθη και αρκετά πιο δύσκολο στη συντήρηση. Καθώς η δημοτικότητα των document-store βάσεων δεδομένων αυξάνεται, και μεταξύ άλλων η MongoDB χρησιμοποιείται όλο και περισσότερο από τις εφαρμογές νέφους, η ανάγκη για την παροχή του εννοιολογικού φάσματος συναλλαγών από τις γλώσσες αυτές γίνεται ολοένα και μεγαλύτερη.

Το CoherentPaas, ένα έργο το οποίο αναπτύσσεται από το Ινστιτούτο Συστημάτων Επικοινωνιών και Πληροφορικής σε συνεργασία με 11 εταιρίες και ακαδημαϊκά ιδρύματα, αποτελεί μια πλατφόρμα η οποία θέλει να καλύψει την ανάγκη αυτή παροχής εννοιολογικού φάσματος συναλλαγών στα NoSQL συστήματα βάσεων δεδομένων.

## 1.1 Αντικείμενο της διπλωματικής εργασίας

Στόχος της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός και υλοποίηση ενός επιπλέον αφαιρετικού επιπέδου στα επιμέρους συστατικά με τα οποία επικοινωνεί ο διαχειριστής συνδιαλλαγών του συστήματος CoherentPaas έτσι ώστε να δημιουργηθούν διεπαφές και αφηρημένες κλάσεις οι οποίες θα κληρονομούνται και θα επεκτείνονται από τα ήδη υπάρχοντα συστατικά του CoherentPaas τόσο για χρήση με το δικό της διαχειριστή συναλλαγών, αλλά δίνοντας με αυτό το τρόπο και την δυνατότητα λειτουργίας του συστήματος αυτού με οποιαδήποτε άλλο διαχειριστή συναλλαγών, ο οποίος όμως να προσφέρει κάποιες ελάχιστες απαραίτητες λειτουργίες όπως αυτές θα παρουσιαστούν στη συνέχεια.

Επιπροσθέτως αντικείμενο της εργασίας αυτής αποτέλεσε και η επέκταση του υπάρχοντος συστήματος με την υλοποίηση κατανεμημένου συλλέκτη μη έγκυρων δεδομένων για τη μη σχεσιακή βάση MongoDB. Στόχος του είναι η ταυτοποίηση και απόρριψη δεδομένων τα οποία δεν χρειάζονται πλέον για τη σωστή λειτουργία του διαχειριστή ταυτόχρονων συνδιαλλαγών, έτσι ώστε να ελευθερωθούν οι πόροι του συστήματος ( στη προκειμένη περίπτωση η μνήμη που καταλαμβάνουν οι άχρηστες εγγραφές) και να επαναχρησιμοποιηθούν.

Στο πλαίσιο της διπλωματικής εργασίας, η γλώσσα που χρησιμοποιήθηκε για την ανάπτυξη του συλλέκτη απορριμμάτων αλλά και για του αφαιρετικού επιπέδου – διεπαφής του CoherentPaas είναι η Java έκδοση 1.7. Η έκδοση της βάσης δεδομένων MongoDB είναι η 3.4. Τέλος η ανάπτυξη, δοκιμή και αποσφαλμάτωση του συστήματος αυτού πραγματοποιήθηκε με τη βοήθεια του ολοκληρωμένου περιβάλλοντος ανάπτυξης (IDE) Eclipse Mars (4.5) στο λειτουργικό σύστημα Windows 7.

## 1.2 Οργάνωση κειμένου

Τα υπόλοιπα κεφάλαια της εργασίας οργανώνονται ως εξής:

Στο κεφάλαιο 2 παρουσιάζεται το απαραίτητο θεωρητικό υπόβαθρο για τη συνέχεια της διπλωματικής, και αναλύονται βασικές έννοιες, κυρίως όσον αφορά τις βάσεις δεδομένων, που θα χρησιμοποιηθούν παρακάτω.

Το κεφάλαιο 3 είναι αφιερωμένο στη περιγραφή του Coherent Paas, της πλατφόρμας η οποία αποτελεί τη βάση της εφαρμογής μας την οποία αναλάβαμε και να επεκτείνουμε .

Στο κεφάλαιο 4 περιγράφεται αναλυτικά ο σχεδιασμός και υλοποίηση πρότυπου διαχειριστή ταυτόχρονων συνδιαλλαγών βασισμένο στον έλεγχο πολλαπλών εκδόσεων δεδομένων (transactional manager).

Στο κεφάλαιο 5 περιέχει τη περιγραφή του σχεδιασμού και της υλοποίησης κατανεμημένου συλλέκτη μη έγκυρων δεδομένων για τη μη σχεσιακή βάση MongoDB.

Στο κεφάλαιο 6 παρατίθενται τα κυριότερα εργαλεία που χρησιμοποιήθηκαν για την εκπόνηση της διπλωματικής εργασίας.

Στο κεφάλαιο 7 συγκεντρώνονται τα συμπεράσματα που εξήχθησαν κατά τη διεξαγωγή της διπλωματικής εργασίας, καθώς και κάποιες προτάσεις για μελλοντική υλοποίηση.

Στο κεφάλαιο 8 τέλος, παρατίθεται η βιβλιογραφία που χρησιμοποιήθηκε για την εκπόνηση της διπλωματικής εργασίας.

# ΚΕΦΑΛΑΙΟ 2
# Βάσεις Δεδομένων

## Ορισμός

Με τον όρο **βάση δεδομένων** εννοείται μία συλλογή από συστηματικά μορφοποιημένα σχετιζόμενα δε μέσω αναζήτησης κατ' απαίτηση δοσμένα στα οποία είναι δυνατή η ανάκτηση δεδομένων [1]. Μια βάση δεδομένων μπορεί να έχει οποιοδήποτε μέγεθος και κυμαινόμενη πολυπλοκότητα. Για παράδειγμα η λίστα ονομάτων και διευθύνσεων που αναφέρθηκε προηγουμένως μπορεί να αποτελείται μόνο από λίγες εκατοντάδες εγγραφές, που κάθε μία τους έχει κάποια απλή δομή. Από την άλλη πλευρά, ο ηλεκτρονικός κατάλογος μιας μεγάλης βιβλιοθήκης μπορεί να περιέχει μισό εκατομμύριο καταχωρήσεις οργανωμένες υπό διαφορετικές κατηγορίες –ως προς το όνομα του βασικού συγγραφέα, ως προς το θέμα, ως προς τίτλο βιβλίου- με κάθε κατηγορία οργανωμένη κατά αλφαβητική σειρά.

Ένα **σύστημα διαχείρισης βάσεων δεδομένων** (database-management system – DBMS) είναι ένα σύνολο από σχετιζόμενα δεδομένα και ένα σύνολο από προγράμματα που χρησιμοποιούνται για πρόσβαση σε αυτά τα δεδομένα [2]. Η συλλογή των δεδομένων, που συνήθως αναφέρεται ως βάση δεδομένων, περιέχει πληροφορίες σχετικές μ' ένα οργανισμό. Ο βασικός στόχος ενός DBMS είναι να παρέχει έναν τρόπο να αποθηκεύονται και να ανακαλούνται πληροφορίες από τις βάσεις δεδομένων, που να είναι βολικός και αποτελεσματικός.

Τα συστήματα βάσεων δεδομένων σχεδιάζονται με τρόπο τέτοιο, ώστε να χειρίζονται μεγάλη ποσότητα πληροφοριών. Η διαχείριση των δεδομένων περιλαμβάνει τόσο τον ορισμό των δομών για τη αποθήκευση των πληροφοριών, όσο και την παροχή μηχανισμών για τον χειρισμό των πληροφοριών, Επιπλέον τα συστήματα βάσεων δεδομένων πρέπει να εξασφαλίζουν την ασφάλεια των πληροφοριών που αποθηκεύονται, ανεξάρτητα από τα προβλήματα του συστήματος ή τις προσπάθειες μη πιστοποιημένης πρόσβασης. Αν τα δεδομένα είναι κοινόχρηστα μεταξύ διαφόρων χρηστών, το σύστημα θα πρέπει να αποφεύγει πιθανά λανθασμένα αποτελέσματα.

Οι βάσεις δεδομένων και τα συστήματα των βάσεων δεδομένων αποτελούν ένα σημαντικό στοιχείο της καθημερινής ζωής στη σύγχρονη κοινωνία: κατά τη διάρκεια μιας ημέρας οι περισσότεροι από εμάς συμμετέχουμε σε κάποια δραστηριότητα που περιλαμβάνει κάποια διαπροσωπεία με μια βάση δεδομένων. Για παράδειγμα αν πάμε στη τράπεζα για κατάθεση ή ανάληψη χρημάτων, αν κάνουμε κράτηση ξενοδοχείου ή αεροπορικού ταξιδιού, αν ψάχνουμε βιβλιογραφικά στοιχεία από έναν κατάλογο βιβλιοθήκης ή αν πραγματοποιήσουμε μια αγορά από το διαδίκτυο, είναι βέβαιο πως οι δραστηριότητες μας περιλαμβάνουν κάποιο πρόγραμμα με προσπέλαση σε βάση δεδομένων.

Αυτές οι διαπροσωπείες είναι παραδείγματα αυτού που αποκαλούμε **παραδοσιακές εφαρμογές των βάσεων δεδομένων** όπου οι περισσότερες αποθηκευμένες πληροφορίες είναι ή σε μορφή κειμένου ή σε μορφή αριθμών. Υπάρχουν όμως και βάσεις δεδομένων πολυμέσων που μπορούν να αποθηκεύσουν εικόνες, video, και μηνύματα ήχου. Τα γεωγραφικά πληροφοριακά συστήματα μπορούν να αποθηκεύσουν και να αναλύσουν δεδομένα καιρού και δορυφορικές εικόνες. Οι αποθήκες δεδομένων και τα online συστήματα αναλυτικής επεξεργασίας χρησιμοποιούνται σε πολλές εταιρίες για την εξαγωγή και ανάλυση χρήσιμων πληροφοριών από μεγάλες βάσεις δεδομένων για λήψη αποφάσεων. Οι τεχνικές αναζήτησης των βάσεων δεδομένων έχουν εφαρμοσθεί στο διαδίκτυο για τη βελτίωση της αναζήτησης πληροφοριών που χρειάζονται οι χρήστες που περιηγούνται στο διαδίκτυο.

## 2.1 Διαχείριση συναλλαγών

Ο όρος συναλλαγή (transaction) αναφέρεται σε ένα σύνολο από λειτουργίες που αποτελούν μια λογική μονάδα. Για παράδειγμα, η μεταφορά χρημάτων από ένα λογαριασμό σε έναν άλλον είναι μια συναλλαγή, που αποτελείται από δύο ενημερώσεις, μία για κάθε λογαριασμό.

Είναι σημαντικό είτε να εκτελεσθούν πλήρως όλες οι ενέργειες μίας συναλλαγής είτε στην περίπτωση κάποιου προβλήματος, να ακυρωθεί όποιο μέρος της συναλλαγής έχει εκτελεσθεί εν μέρει. Αυτή η ιδιότητα ονομάζεται ***ατομικότητα***. Επιπλέον, αφού εκτελεστεί με επιτυχία μία συναλλαγή. Το αποτέλεσμα της πρέπει να παραμείνει στη βάση δεδομένων, δηλαδή ένα πρόβλημα στο σύστημα δεν πρέπει να κάνει την βάση δεδομένων να ξεχάσει μία συναλλαγή που ολοκληρώθηκε με επιτυχία. Αυτή η ιδιότητα ονομάζεται ***ανθεκτικότητα***.

Σε ένα σύστημα βάσης δεδομένων όπου εκτελούνται πολλαπλές συναλλαγές ταυτόχρονα, αν δεν ελέγχονται οι ενημερώσεις σε κοινόχρηστα δεδομένα, υπάρχει πιθανότητα οι συναλλαγές να χρησιμοποιούν δεδομένα που είναι σε ασυνεπείς ενδιάμεσες καταστάσεις, εξ αιτίας των ενημερώσεων άλλων συναλλαγών. Μία τέτοια κατάσταση μπορεί να καταλήξει σε λανθασμένες ενημερώσεις των δεδομένων που αποθηκεύονται στη βάση δεδομένων. Έτσι, τα συστήματα βάσεων δεδομένων πρέπει να παρέχουν μηχανισμούς για απομόνωση των συναλλαγών από τις επιδράσεις άλλων συναλλαγών που εκτελούνται ταυτόχρονα. Αυτή η ιδιότητα ονομάζεται ***απομόνωση***.

Λόγω των ανωτέρω τριών ιδιοτήτων, οι συναλλαγές αποτελούν έναν ιδανικό τρόπο της δόμησης της αλληλεπίδρασης με μία βάση δεδομένων. Αυτό μας οδηγεί να επιβάλουμε μία απαίτηση στις ίδιες τις συναλλαγές. Μία συναλλαγή πρέπει να διατηρεί τη ***συνέπεια*** της βάσης δεδομένων – εάν μία συναλλαγή εκτελείται ατομικά σε απομόνωση και ξεκινά με μία συνεπή βάση δεδομένων, η βάση δεδομένων θα πρέπει να είναι συνεπής στο τέλος της συναλλαγής. Ο τρόπος που θα γίνει αυτό είναι ευθύνη του προγραμματιστή που κωδικοποιεί μία συναλλαγή.

### 2.1.1 Συναλλαγές

Συναλλαγή είναι μια σειρά από λειτουργείες οι οποίες πραγματοποιούνται σαν μία μοναδική λογική μονάδα εκτέλεσης. Η λογική αυτή μονάδα πρέπει να διακρίνεται από τις τέσσερις ιδιότητες που αναφέραμε παραπάνω : ατομικότητα, συνέπεια, απομόνωση και ανθεκτικότητα (ACID) για να θεωρηθεί *συναλλαγή* [3]. Η *συναλλαγή* στις βάσεις δεδομένων έχουν δύο κύριους σκοπούς :

- Να παρέχουν αξιόπιστες μονάδες εργασίας που επιτρέπουν την ορθή ανάκαμψη από σφάλματα και κρατούν την βάση δεδομένων σε *συνεπή* κατάσταση ακόμα και σε περιπτώσεις σφάλματος του συστήματος, όταν η εκτέλεση σταματά ( μερικώς ή πλήρως) και πολλά ερωτήματα – λειτουργίες προς την βάση εκκρεμούν, σε ασαφή κατάσταση.
- Να παρέχουν απομόνωση μεταξύ των προγραμμάτων που έχουν, παράλληλα , πρόσβαση στη βάση δεδομένων. Αν αυτή η απομόνωση δεν εξασφαλίζεται, το αποτέλεσμα των προγραμμάτων δεν είναι ντετερμινιστικό και πιθανών καταλήγει σε κάποια εσφαλμένη κατάσταση.

Οι συναλλαγές υιοθετούν μία «όλα ή τίποτα» στάση, όπου κάθε ***λογική μονάδα εκτέλεσης*** η οποία πραγματοποιείται στη βάση δεδομένων πρέπει είτε να ολοκληρωθεί στο σύνολό της είτε να μην έχει καμία τελείως επίδραση. Ακόμα, το σύστημα πρέπει να απομονώνει κάθε συναλλαγή από τις υπόλοιπες, τα αποτελέσματα να συμμορφώνονται στους υπάρχοντες περιορισμούς της βάσης και οι συναλλαγές που ολοκληρώνονται με επιτυχία να γράφονται σε κάποιο «μη πτητικό» μέσο αποθήκευσης.

### 2.1.2 Απομόνωση Στιγμιότυπου (Snapshot Isolation)

Στις βάσεις δεδομένων η απομόνωση στιγμιότυπου μας εξασφαλίζει πως όλες οι αναγνώσεις οι οποίες πραγματοποιούνται σε μια συναλλαγή θα “δουν” μια συνεπή απεικόνιση της βάσης [4]. Δηλαδή διαβάζει τις τελευταίες τιμές των δεδομένων στη βάση, τη χρονική στιγμή που δημιουργήθηκε. Όλες οι επιμέρους ενημερώσεις της συναλλαγής εφαρμόζονται στο στιγμιότυπο το οποίο της αντιστοιχεί, το οποίο δεν είναι αόρατο στις υπόλοιπες τρέχουσες συναλλαγές. Μόλις ολοκληρωθεί η συναλλαγή αυτή θα καταχωρίσει τα δεδομένα επιτυχώς, μόνο αν δεν έχουν γίνει παράλληλα ενημερώσεις, που έρχονται σε σύγκρουση με τα δεδομένα προς καταχώριση, στο διάστημα που μεσολάβησε από τη στιγμή δημιουργίας του στιγμιότυπου. Η τεχνική αυτή έχει υιοθετηθεί τόσο από πλήθος βάσεων δεδομένων όπως οι Oracle, PostgreSQL, MongoDB, Microsoft SQL Server και άλλες, χάρη στις πολύ καλές επιδόσεις της. Η απομόνωση στιγμιότυπου προϋποθέτει τη χρήση ενός συστήματος ελέγχου πολλαπλών εκδόσεων.

### 2.1.3 Έλεγχος Ταυτοχρονισμού Πολλαπλών Εκδόσεων

Ο Έλεγχος Ταυτοχρονισμού Πολλαπλών Εκδόσεων (Multi Version Concurrency Control-MVCC) είναι μια μέθοδος που χρησιμοποιείται από τα συστήματα διαχείρισης βάσεων δεδομένων για τον έλεγχο των ταυτόχρονων συναλλαγών. Στόχος της συγκεκριμένης έννοιας είναι να παρέχει ταυτόχρονη πρόσβαση στην βάση , που δεν θα βασίζεται στα κλειδώματα και συνεπώς δεν θα προκαλεί τα προβλήματα που προέρχονται από τα κλειδώματα, όπως η παρεμπόδιση και η αναμονή των συναλλαγών, στο να συνεχίσουν την εκτέλεση τους, από άλλες ταυτόχρονες συναλλαγές.

Όπως υποδηλώνει και το όνομα της η συγκεκριμένη μέθοδος βασίζεται σε πολλαπλές εκδόσεις των δεδομένων για να επιτύχει το σκοπό της. Πιο συγκεκριμένα, όταν μια MVCC βάση δεδομένων θέλει να ενημερώσει κάποιο πεδίο σε ένα έγγραφο, δεν θα αντικαταστήσει το παλιό έγγραφο με το ενημερωμένο-καινούργιο αλλά θα μαρκάρει το παλιό ως «απαρχαιωμένο» και θα προσθέσει τη καινούργια έκδοση κάπου αλλού. Οπότε θα υπάρχουν πολλαπλές εκδόσεις αποθηκευμένες, αλλά μόνο μία θα είναι η πιο πρόσφατη. Αυτό επιτρέπει στους χρήστες-εφαρμογές να έχουν πρόσβαση στα δεδομένα που ήταν εκεί όταν αρχίσαν την ανάγνωση, ακόμα και έχουν τροποποιηθεί ή διαγραφεί στο διάστημα που έχει μεσολαβήσει, από κάποιον άλλο. Επιτρέπει ακόμα στη βάση δεδομένων να αποφύγει το κόστος που φέρει ο κατακερματισμός της μνήμης και των δομών αποθήκευσης του δίσκου υπό κανονικές συνθήκες εξαιτίας των διαγραφών, αλλά προϋποθέτει ότι το σύστημα θα καθαρίζει περιοδικά τη βάση από απαρχαιωμένα δεδομένα.

Εύκολα μπορούμε να καταλάβουμε πως για κάθε συναλλαγή που εκτελείται είναι απαραίτητο να γνωρίζουμε τη χρονική στιγμή που εκτελείται, αλλά και να γνωρίζουμε για κάθε έκδοση εγγραφής που υπάρχει στη βάση ποια χρονική στιγμή εισήχθη. Οι MVCC βάσεις δεδομένων ικανοποιούν την ανάγκη αυτή κάνοντας χρήση ενός συστήματος απόδοσης χρονοσφραγίδων (timestamps). Με τον όρο χρονοσφραγίδα αναφερόμαστε σε μία ακολουθία χαρακτήρων που μας δίνει την ακριβή χρονική στιγμή που συνέβη ένα γεγονός. Όποτε μία συναλλαγή ξεκινά, δέχεται μια χρονοσφραγίδα, η οποία υποδεικνύει τη σειρά με την οποία πρέπει να εκτελεστεί, σε σχέση με τις άλλες συναλλαγές. Επομένως, δεδομένου δύο συναλλαγών που επηρεάζουν το ίδιο αντικείμενο, πρώτη πρέπει να εκτελεστεί η συναλλαγή με την μικρότερη (πιο νωρίς) χρονοσφραγίδα. Σε αντίθετη περίπτωση, αν δηλαδή η λάθος συναλλαγή εκτελεστεί πρώτα, η διαδικασία αποτυγχάνει και η συναλλαγή πρέπει να «επανεκκινήσει».

Κάθε συναλλαγή έχει μία χρονοσφραγίδα εκκίνησης και μία χρονοσφραγίδα καταχώρησης/δέσμευσης (commit timestamp). Η χρονοσφραγίδα εκκίνησης μιας συναλλαγής υποδηλώνει ουσιαστικά ποιες προϋπάρχουσες εκδόσεις δεδομένων είναι

ορατές πριν από αυτήν, ενώ η χρονοσφραγίδα καταχώρησης καθορίζει τη χρονική στιγμή στην οποία οι εγγραφές που πραγματοποιεί γίνονται ορατές στις άλλες συναλλαγές. Όταν μια νέα εγγραφή πρόκειται εισαχθεί στη βάση δεδομένων, τότε σε αυτή προστίθεται και η τιμή της χρονοσφραγίδας καταχώρησης της συναλλαγής που πραγματοποίησε την τροποποίηση αυτή. Με βάση το πλαίσιο αυτό των χρονοσφραγίδων μπορούμε να ορίσουμε τη διαδικασία εκτέλεσης μιας συναλλαγής ως εξής :

1. Στην αρχή μιας συναλλαγής της ανατίθεται η χρονοσφραγίδα εκκίνησης (ισούται με την χρονοσφραγίδα της τελευταίας καταχώρισης)
2. Όταν μια συναλλαγή, έστω Α, θέλει να πραγματοποιήσει μια εγγραφή, έστω στο αρχείο Χ, τότε εκτελείται πρώτα ένας έλεγχος για σύγκρουση από τον διαχειριστή συναλλαγών. Σύγκρουση συμβαίνει όταν υπάρχει μια ταυτόχρονη συναλλαγή, έστω Β, η οποία δεν έχει καταχωρηθεί ακόμα ή η χρονοσφραγίδα καταχώρισης της είναι μεγαλύτερη από την χρονοσφραγίδα εκκίνησης της συναλλαγής Α και η συναλλαγή Β έχει γράψει στο αρχείο Χ.
3. i. Αν δεν υπάρχει κάποια σύγκρουση η εγγραφή μπορεί να προχωρήσει, δημιουργώντας μια καινούργια ιδιωτική έκδοση του αρχείου Χ, ορατή αρχικά μόνο από τη συναλλαγή Α.
   ii. Αν υπάρχει σύγκρουση συναλλαγή Α πρέπει να ακυρωθεί για να εξασφαλιστούν οι ιδιότητες της απομόνωσης στιγμιότυπου. Αυτό συνεπάγεται την καταστροφή των ιδιωτικών εκδόσεων δεδομένων που έχει δημιουργήσει η συναλλαγή Α.
4. Για την ανάγνωση του αρχείου Χ από μια συναλλαγή, έστω Α, η βάση πρέπει να παρέχει το αρχείο που δημιουργήθηκε από την συναλλαγή Β με χρονοσφραγίδα καταχώρισης, τέτοια ώστε να είναι μικρότερη από την χρονοσφραγίδα έναρξης της συναλλαγής Α και να μην υπάρχει έκδοση του αρχείου σε οποιαδήποτε ταυτόχρονη συναλλαγή της οποίας η χρονοσφραγίδα καταχώρισης να είναι μεγαλύτερη από αυτή της συναλλαγής Β και μικρότερη της χρονοσφραγίδας εκκίνησης της συναλλαγής Α. Αυτό με τη σειρά του μας εξασφαλίζει ότι τηρούνται οι ιδιότητες της απομόνωσης στιγμιότυπου.

Βασικό μειονέκτημα της μεθόδους αυτής είναι η ανάγκη για περισσότερο αποθηκευτικό χώρο αφού το πλήθος των δεδομένων (εγγραφών) είναι πολύ μεγαλύτερο εξαιτίας των διαφορετικών εκδόσεων για κάθε συναλλαγή. Παρόλα αυτά, το συγκεκριμένο γεγονός αφενός είναι λιγότερο σημαντικός παράγοντας συγκριτικά με την ταχύτητα και την απόδοση του συστήματος, αφετέρου μπορεί να αντιμετωπιστεί, όπως θα δούμε και στη συνέχεια, με τη χρήση ενός προγράμματος συλλέκτη σκουπιδιών (garbage collector).

## 2.2 Βάση Δεδομένων Συναλλαγών

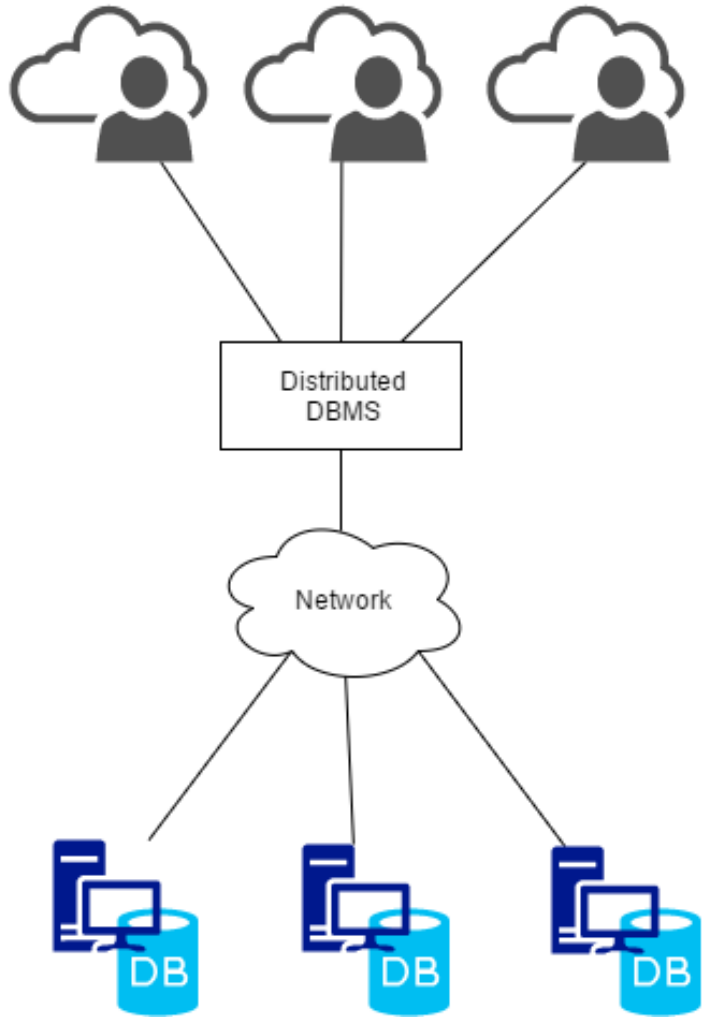
Μία βάση δεδομένων συναλλαγών ένα σύστημα διαχείρισης βάσεων δεδομένων οπού οι *συναλλαγές* εγγραφής στη βάση είναι δυνατόν να αναιρεθούν εφόσον δεν ολοκληρωθούν επιτυχώς. Σε ένα τέτοιο σύστημα βάσης δεδομένων μία συναλλαγή

μπορεί να αποτελείται από περισσότερα από ένα ερωτήματα – δηλώσεις , καθένα από τα οποία διαβάζει ή/και γράφει πληροφορίες στη βάση. Οι χρήστες των συστημάτων αυτών θεωρούν την συνέπεια και την ακεραιότητα των δεδομένων υψίστης σημασίας. Μία απλή συναλλαγή έχει το παρακάτω μοτίβο:

- Αρχή συναλλαγής
- Εκτέλεση ενός συνόλου «ερωτημάτων »
- Αν δεν προκύψουν λάθη τότε πραγματοποιείται «δέσμευση» (commit) της συναλλαγής και ολοκληρώνεται η διαδικασία
- Αν προκύψουν λάθη τότε αναιρούνται όλες οι ενέργειες που πραγματοποιήθηκαν

## 2.3 Κατανεμημένη Βάση Δεδομένων

Σε κατανεμημένα συστήματα βάσεων δεδομένων, η βάση δεδομένων αποθηκεύεται σε διάφορους υπολογιστές. Οι υπολογιστές αυτοί μπορούν να βρίσκονται είτε στην ίδια τοποθεσία είτε να επικοινωνούν μεταξύ τους μέσω διαφόρων μέσων επικοινωνίας, συνήθως με ιδιωτικά δίκτυα υψηλής ταχύτητας.




Εικόνα 1: Σύστημα κατανεμημένης βάσης δεδομένων

Κύρια πλεονεκτήματα [5] αυτής της αρχιτεκτονικής των βάσεων δεδομένων είναι :

- **Κοινή χρήση δεδομένων.** Για παράδειγμα, σε ένα κατανεμημένο σύστημα Πανεπιστημίου, όπου κάθε τμήμα του Πανεπιστημίου αποθηκεύει δεδομένα σχετικά με αυτό το τμήμα, είναι δυνατόν ένας χρήστης σε ένα τμήμα να έχει πρόσβαση στα δεδομένα ενός άλλου τμήματος. Χωρίς αυτή τη δυνατότητα, ένας χρήστης που θέλει να μεταφέρει ένα σπουδαστή από ένα τμήμα σε ένα άλλο, θα πρέπει να καταφύγει σε κάποιους εξωτερικούς μηχανισμούς που συνδέουν τα υπάρχοντα συστήματα.

- **Αυτονομία.** Το βασικό πλεονέκτημα της κοινής χρήσης δεδομένων σε σχέση με την κατανομή των δεδομένων είναι ότι κάθε θέση μπορεί να διατηρεί έλεγχο πάνω στα δεδομένα που είναι αποθηκευμένα τοπικά. Σε ένα κεντρικό σύστημα, ο διαχειριστής της βάσης δεδομένων της κεντρικής θέσης ελέγχει τη βάση δεδομένων. Σε ένα κατανεμημένο σύστημα, υπάρχει ένας καθολικός διαχειριστής που είναι υπεύθυνος για ολόκληρο το σύστημα. Ένα μέρος αυτών των ευθυνών ανατίθεται σε ένα διαχειριστή της βάσης δεδομένων κάθε θέσης. Ανάλογα με τη σχεδίαση της κατανεμημένης βάσης δεδομένων, κάθε διαχειριστής μπορεί να έχει διαφορετικό βαθμό τοπικής αυτονομίας. Η δυνατότητα τοπικής αυτονομίας είναι συνήθως ένα μεγάλο πλεονέκτημα των κατανεμημένων βάσεων δεδομένων.

- **Διαθεσιμότητα.** Αν χαλάσει μία θέση σε ένα κατανεμημένο σύστημα, οι υπόλοιπες θέσεις μπορούν να συνεχίσουν να λειτουργούν. Ιδιαίτερα, αν κάποια στοιχεία είναι αντιγραμμένα σε διάφορες θέσεις. Μία συναλλαγή που χρειάζεται συγκεκριμένα δεδομένα μπορεί να τα δει από τις άλλες θέσεις. Έτσι, το πρόβλημα μίας θέσης δεν συνεπάγεται απαραίτητα το κλείσιμο του συστήματος. Το πρόβλημα μίας θέσης μπορεί να εντοπιστεί από το σύστημα και να γίνουν κατάλληλες ενέργειες ώστε να ανακάμψει από την καταστροφή, Το σύστημα δεν πρέπει να χρησιμοποιεί πλέον τις υπηρεσίες της χαλασμένης θέσης. Τέλος, όταν η χαλασμένη θέση επανέλθει ή διορθωθεί, πρέπει να υπάρχουν διαθέσιμοι μηχανισμοί για να ενοποιηθεί ξανά ομαλά στο σύστημα.

Φυσικά τα οφέλη αυτά έρχονται με κάποιο κόστος:

- **Κόστος ανάπτυξης λογισμικού.** Είναι πιο δύσκολο να χειριστείτε ένα κατανεμημένο σύστημα βάσης δεδομένων. Συνεπώς, κοστίζει περισσότερο.

- **Μεγαλύτερη πιθανότητα για λάθη.** Αφού οι θέσει, που αποτελούν το κατανεμημένο σύστημα λειτουργούν παράλληλα, είναι δυσκολότερο να εξασφαλίσετε ότι είναι σωστοί οι αλγόριθμοι, ειδικά για λειτουργίες στην διάρκεια προβλημάτων μέρους του συστήματος κ αι η αποκατάσταση από προβλήματα. Υπάρχει πιθανότητα να περάσουν πολύ λεπτά λάθη.

- **Αυξημένο κόστος επεξεργασίας.** Η ανταλλαγή μηνυμάτων και οι επιπλέον υπολογισμοί που απαιτούνται ώστε να επιτευχθεί ο συντονισμός μεταξύ των θέσεων είναι μία μορφή κόστους που δεν υπάρχει στα κεντρικά συστήματα.

## 2.4 Μη σχεσιακή Βάση Δεδομένων

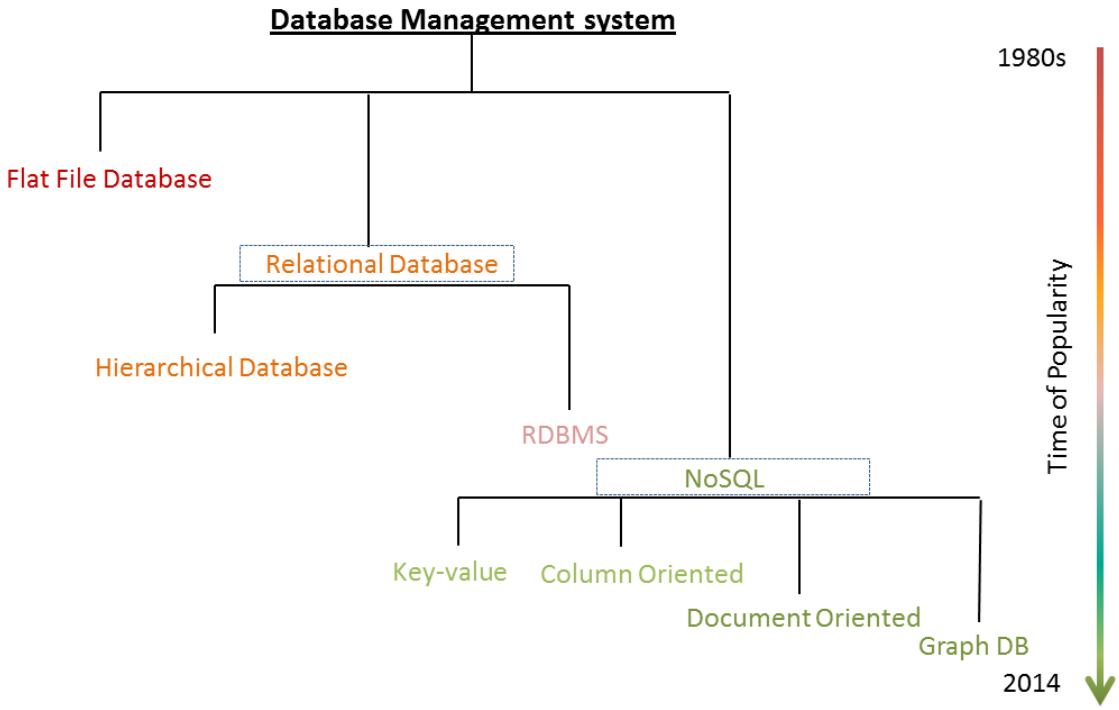
Μία μη σχεσιακή βάση δεδομένων είναι οποιαδήποτε βάση δεδομένων η οποία δεν ακολουθεί το σχεσιακό μοντέλο που συναντάται στα παραδοσιακά σχεσιακά συστήματα βάσεων δεδομένων [6]. Η κατηγορία αυτών των βάσεων , οι οποίες καλούνται και NoSQL Databases, έχει δει μια σταθερή ανάπτυξη τα τελευταία χρόνια με την έξαρση των Big Data εφαρμογών. Οι NoSQL βάσεις είναι πιο ευέλικτες καθώς και ευκολότερα επεκτάσιμες σε σχέση με τις σχεσιακές, από τις οποίες διαφοροποιούνται κυρίως στα παρακάτω σημεία:

- Μοντέλα Δεδομένων. Σε αντίθεση με της σχεσιακές που απαιτούν προκαθορισμένο σχήμα (schema), οι NoSQL βάσεις προσφέρουν ευέλικτο σχεδιασμό του σχήματος το οποίο διευκολύνει σημαντικά την βάση στο να ανταποκρίνεται σε αλλαγές των απαιτήσεων της εφαρμογής.

- Δομή Δεδομένων. Οι μη σχεσιακές βάσεις είναι σχεδιασμένες να χειρίζονται δεδομένα χωρίς καθορισμένη δομή τα οποία δεν μπορούν να οργανωθούν σε γραμμές και στήλες . Αυτό είναι πολύ σημαντικό αν σκεφτεί κανείς ότι τα δεδομένα που παράγονται σήμερα είναι , ως κατά το πλείστο, αδόμητα.

- Επεκτασιμότητα. Το σύστημα μπορεί εύκολα να επεκταθεί οριζόντια εκμεταλλευόμενο τους φθηνούς “commodity servers”

- Μοντέλο ανάπτυξης. Οι NoSQL βάσεις είναι συνήθως ανοικτού κώδικα (open source) αφαιρώντας με αυτό το τρόπο το αρκετά μεγάλο κόστος των αδειοδότησης λογισμικού.




Εικόνα 2: Δημοτικότητα Βάσεων Δεδομένων

## 2.4.1 Ταξινόμηση NoSQL βάσεων δεδομένων

Οι NoSQL βάσεις δεδομένων χωρίζονται σε διάφορες κατηγορίες ανάλογα με το τρόπο με τον οποία τα δεδομένα απεικονίζονται και αποθηκεύονται σε αυτές [7]. Οι κυριότερες από αυτές είναι :

**Key-Value Store**

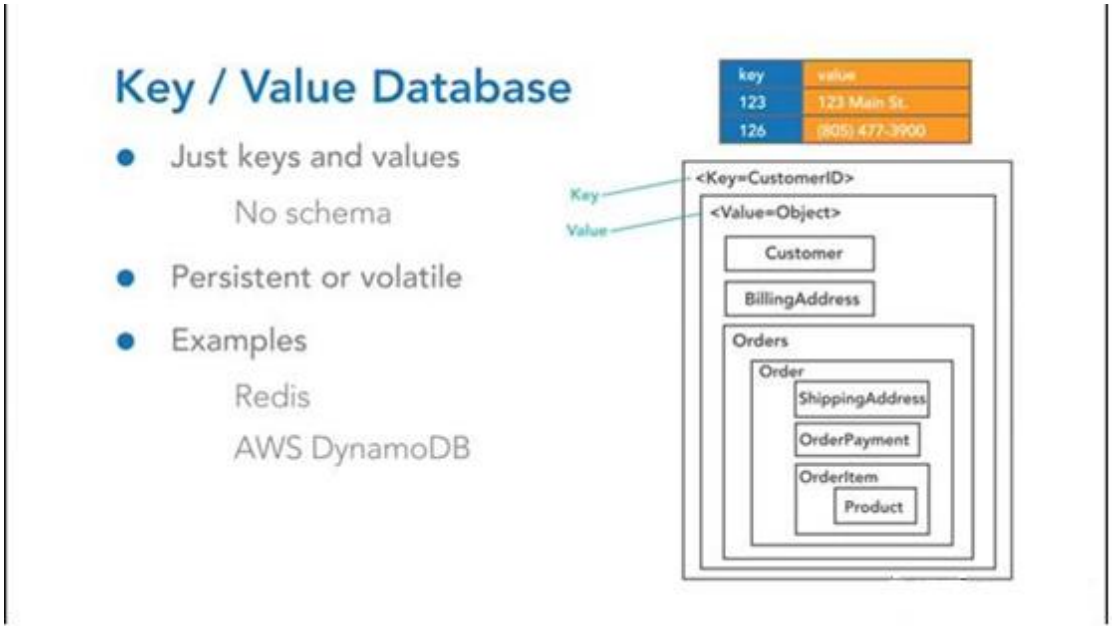
- **Key-Value Cache** : Coherence, eXtreme Scale, GigaSpaces, GemFire, Hazelcast, Infinispan, JBoss Cache, Memcached, Repcached, Terracotta, Velocity
- **Key-Value Store** (Eventually-Consistent) : DovetailDB, Oracle NoSQL Database, Dynamo, Riak, Dynomite, MotionDb, Voldemort, SubRecord
- **Key-Value Store** (Ordered) : Actord, FoundationDB, InfinityDB, Lightcloud, LMDB, Luxio, MemcacheDB, NMDB, Scalaris, TokyoTyrant
- **Key-Value Store** (Hiearchical) : Cache, GT.m

- **Document-based Store :** ArangoDB, Clusterpoint, Couchbase, CouchDB, DocumentDB, IBM Domino, MarkLogic, MongoDB, Qizx, RethinkDB, XML-databases
- **Column-based Store :** Accumulo, Cassandra, Druid, HBase, Vertica, SAP HANA
- **Graph-based :** AllegroGraph, ArangoDB, InfiniteGraph, Apache Giraph, MarkLogic, Neo4J, OrientDB, Virtuoso

#### 2.4.1.1 Key Value Store

Στις key value store βάσεις το βασικό μοντέλο δεδομένων αποτελεί ένας πίνακας συσχετισμών (associative array) όπου κάθε κλειδί αντιστοιχίζεται με μία και μόνο μία τιμή στη συλλογή (collection). Η σχέση αυτή είναι γνωστή ως key-value pair (ζευγάρι). Σε κάθε τέτοιο ζευγάρι το κλειδί αναπαρίσταται από ένα αυθαίρετο string όπως για παράδειγμα ένα όνομα αρχείου ένα URI ή ένα hash. Η τιμή μπορεί να είναι οποιοδήποτε τύπος δεδομένου όπως μία εικόνα, ένα αρχείο ήχου, ένα αρχείο κειμένου. Η τιμή αυτή αποθηκεύεται σαν ένα blob χωρίς την ανάγκη για κάποιον εκ των προτέρων καθορισμό σχήματος ή μοντέλου δεδομένων. Η αποθήκευση αυτή της τιμής ως blob αφαιρεί την ανάγκη για την εισαγωγή του σε ευρετήριο ώστε να πετύχει καλύτερη επίδοση.

Από την άλλη πλευρά, δεν υπάρχει η δυνατότητα φιλτραρίσματος και ελέγχου του αντικειμένου που μας επιστρέφει μία αναζήτηση με βάση τη τιμή αφού η τιμή είναι αδιαφανής. Ένα ακόμα πρόβλημα των key value store βάσεων δεδομένων είναι ότι δεν προσφέρουν κάποιες από τις βασικές δυνατότητες των παραδοσιακών βάσεων όπως την ατομικότητα των συναλλαγών και την συνοχή όταν πολλαπλές συναλλαγές εκτελούνται παράλληλα. Ενώ καθώς ο όγκος δεδομένων αυξάνει η τήρηση της μοναδικότητας των κλειδιών αποδεικνύεται αρκετά δύσκολη με αποτέλεσμα την εισαγωγή πολυπλοκότητας στην εφαρμογή για την παραγωγή string που θα μας εξασφαλίζουν την μοναδικότητα τους μέσα σε ένα εξαιρετικά μεγάλο σύνολο κλειδιών.




Εικόνα 3: Παράδειγμα Key/Value Βάσης Δεδομένων

### 2.4.1.2 Document Store

Η κεντρική ιδέα μίας document-based βάσης δεδομένων είναι η έννοια του εγγράφου (document). Ενώ σε κάθε ξεχωριστή document-based βάση η υλοποίηση διαφέρει στις λεπτομέρειες του εκάστοτε ορισμού, σε γενικές γραμμές όλες υποθέτουν πως τα αντικείμενα εμπεριέχουν και κωδικοποιούν δεδομένα ( ή γενικά πληροφορία) σε κάποιο γνωστό format ή encoding. Κάποια από τα πιο ευρέως γνωστά encodings είναι τα XML, YAML, JSON και BSON ενώ κάποια δυαδικά formats είναι τα PDF και τα Microsoft Office documents.

Τα έγγραφα (documents) σε μία document store βάση είναι το ανάλογο ενός αντικειμένου στο περιβάλλον του προγραμματισμού. Δεν είναι υποχρεωμένα να υπακούν σε κάποιο συγκεκριμένο «σχήμα» (schema) ούτε έχουν τις ίδιες θέσεις , κλειδιά ή κομμάτια. Όπως ακριβώς και στα προγράμματα υπάρχουν πολλοί διαφορετικοί τύποι από αντικείμενα, και τα αντικείμενα αυτά συχνά έχουν πολλά προαιρετικά πεδία. Κάθε αντικείμενο, ακόμα και εκείνα που ανήκουν στην ίδια κλάση μπορούν να είναι πολύ διαφορετικά. Οι Document-based βάσεις δεδομένων μοιάζουν στο ότι επιτρέπουν διαφορετικούς τύπου εγγράφων σε ένα store, επιτρέπουν τα πεδία τους να είναι προαιρετικά και συχνά μπορούν να κωδικοποιηθούν χρησιμοποιώντας διαφορετικά συστήματα κωδικοποίησης.

Για παράδειγμα το ακόλουθο έγγραφο είναι κωδικοποιημένο σε JSON :

```
{
        “LastName”: “Gialousis”,
        “FirstName”: “Miltiadis”,
        “CityOfBirth”: “Pireas”
}
```

Ένα δεύτερο έγγραφο μπορεί να είναι κωδικοποιημένο σε XML :

```
<student>
    <lastname>Kontzilas</lastname>
    <firstname>Panagiotis</firstname>
    <phone type=”Mobile”>699 999 9999</phone>
    <phone type=”Home”>210 9999999</phone>
    <address>
        <city>Nafplio</city>
        <street>Laodikias</street>
        <number>12</number>
    </address>
</student>
```

Τα δύο έγγραφα μοιράζονται κάποιες κοινές δομές μεταξύ τους αλλά έχουν και αρκετά μοναδικά στοιχεία. Η δομή, το κείμενο και τα άλλα δεδομένα μέσα σε ένα έγγραφο αναφέρονται ως τα περιεχόμενα του εγγράφου. Σε αντίθεση με τις σχεσιακές βάσεις δεδομένων όπου κάθε εγγραφή περιέχει και τα ίδια πεδία, αφήνοντας τα μη χρησιμοποιημένα πεδία άδεια, δεν υπάρχουν κενά πεδία στις document—based βάσεις, όπως βλέπουμε και στο παραπάνω παράδειγμα. Η προσέγγιση αυτή δίνει τη δυνατότητα στο χρήστη να προσθέσει επιπλέον δεδομένα σε ορισμένες εγγραφές χωρίς την απαίτηση να μοιράζονται την ίδια δομή όλες οι εγγραφές της βάσης.

### CRUD λειτουργίες

Οι κύριες λειτουργίες που υποστηρίζει μία document-based βάση δεδομένων είναι παρόμοιες με αυτές των άλλων βάσεων, και ενώ η ορολογία δεν είναι τελείως θεσμοθετημένη, οι περισσότεροι επαγγελματίες χρήστες αναφέρονται σε αυτές ως CRUD:

- Creation (Δημιουργία ή εισαγωγή)
- Retrieval (Ερώτημα-query, αναζήτηση)
- Update (Ενημέρωση ή τροποποίηση)

- Deletion (Διαγραφή)

**Κλειδιά**

Τα έγγραφα προσπελαύνονται στη βάση μέσω ενός μοναδικού κλειδιού το οποίο και αντιπροσωπεύει το εκάστοτε έγγραφο. Το κλειδί λειτουργεί πρόκειται για ένα απλό id, το οποίο είναι συνήθως string, URI ή ένα μονοπάτι (path). Το κλειδί μπορεί να χρησιμοποιηθεί για την ανάκτηση ενός εγγράφου από τη βάση δεδομένων. Τυπικά η βάση δεδομένων διατηρεί ένα δείκτη στο κλειδί για να αυξήσει την ταχύτητα ανάκτησης του αντικειμένου ενώ σε μερικές περιπτώσεις το κλειδί είναι αναγκαίο για τη δημιουργία ή την εισαγωγή ενός εγγράφου στη βάση.

### 2.4.1.3 Column-based Store

Οι column-based βάσεις δεδομένων αποθηκεύουν τους πίνακες δεδομένων κατά στήλη, αντίθεση με τις κλασσικές σχεσιακές βάσεις δεδομένων στις οποίες τα δεδομένα αποθηκεύονται κατά σειρά. Η πρακτική διαφορά μεταξύ της αποθήκευσης κατά στήλης και αποθήκευσης κατά σειράς είναι πολύ μικρή στο περιβάλλον των σχεσιακών συστημάτων διαχείρισης βάσεων δεδομένων. Τόσο οι κατά σειρά όσο και οι κατά στήλη οργανωμένες βάσεις μπορούν να χρησιμοποιήσουν τις παραδοσιακές γλώσσες ερωτημάτων (query languages) όπως η SQL για να φορτώσουν δεδομένα και να πραγματοποιήσουν αναζητήσεις (queries). Επιπροσθέτως και οι δύο αυτοί τρόποι οργάνωσης της βάσης έχουν τη δυνατότητα να καθιστούν η ραχοκοκαλιά ενός συστήματος παρέχοντας δεδομένα για κοινά εργαλεία εξόρυξης, μετασχηματισμού, φόρτωσης και απεικόνισης δεδομένων. Ωστόσο, αποθηκεύοντας τα δεδομένα σε στήλες αντί για σειρές. Η βάση δεδομένων μπορεί με ακρίβεια να εντοπίσει τα δεδομένα που χρειάζεται για να απαντήσει στο εκάστοτε ερώτημα (query) αντί να προσπελαύνει ανεπιθύμητα δεδομένα σε σειρές. Αυτό έχει προφανώς ως αποτέλεσμα μια αυξημένη συνήθως επίδοση στα ερωτήματα, κυρίως σε μεγάλα σύνολα δεδομένων.

Ακολουθεί ένας απλός πίνακας μία σχεσιακής βάσης δεδομένων τον οποία θα μετατρέψουμε σε αναπαράσταση column-based βάσης.

| Id | Lastname | FirstName | City of Birth |
|---|---|---|---|
| 400 | Gialousis | Miltiadis | Pireas |
| 450 | Sxoinas | Antreas | Xania |
| 500 | Kranas | Pavlos | Athens |
| 600 | Kontzilas | Panagiotis | Nafplio |

Εικόνα 4: Πίνακας σχεσιακής βάσης δεδομένων

Η αναπαράσταση των δεδομένων σε μία column-based βάση δεδομένων προκύπτει από τη σειριοποίηση όλων των τιμών μιας στήλης μαζί, εν συνεχεία των τιμών τις επόμενης στήλης και ούτε καθεξής. Με τον τρόπο αυτό , ο παραπάνω πίνακας σε μία column based βάση δεδομένων θα είχε την ακόλουθη μορφή :

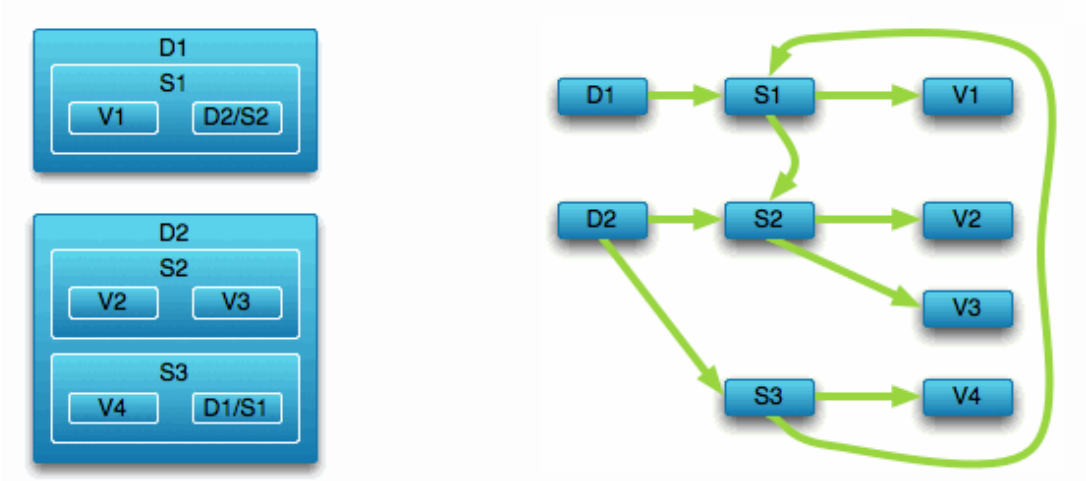
400:001, 450:002, 500:003, 600:004; Gialousis:001, Sxoinas:002, Kranas:003, Kontzilas:004; Miltiadis:001, Antreas:002, Pavlos:003, Panagiotis:004; Pireas:001, Xania:002, Athens:003, Nafplio:004;

Στη πράξη οι column-based βάσεις δεδομένων ενδείκνυται για OLAP είδους εργασίες (Online analytical processing) , όπως οι αποθήκες δεδομένων, στις οποίες υπάρχει συνήθως υψηλή πολυπλοκότητα ερωτημάτων (query) που πραγματοποιούνται με στόχο το σύνολο των δεδομένων, τα οποία κυμαίνονται στη τάξη μεγέθους των petabytes. Ωστόσο κάποιες επιπλέον εργασία χρειάζεται για το γράψιμο των δεδομένων σε μία column based βάση. Οι συναλλαγές οφείλουν να χωριστούν σε κολώνες και να συμπιεστούν καθώς αποθηκεύονται, γεγονός που έρχεται σε σύγκρουση με τις ανάγκες των OLTP (Online transaction processing) εργασιών. Στα προβλήματα αυτής της κατηγορίες οι row-based βάσεις δεδομένων ταιριάζουν καλύτερα καθώς τα συστήματα αυτά εμπεριέχουν μεγάλο αριθμό από «διαδραστικές» (interactive) συναλλαγές

### 2.4.1.4 Graph-based Store

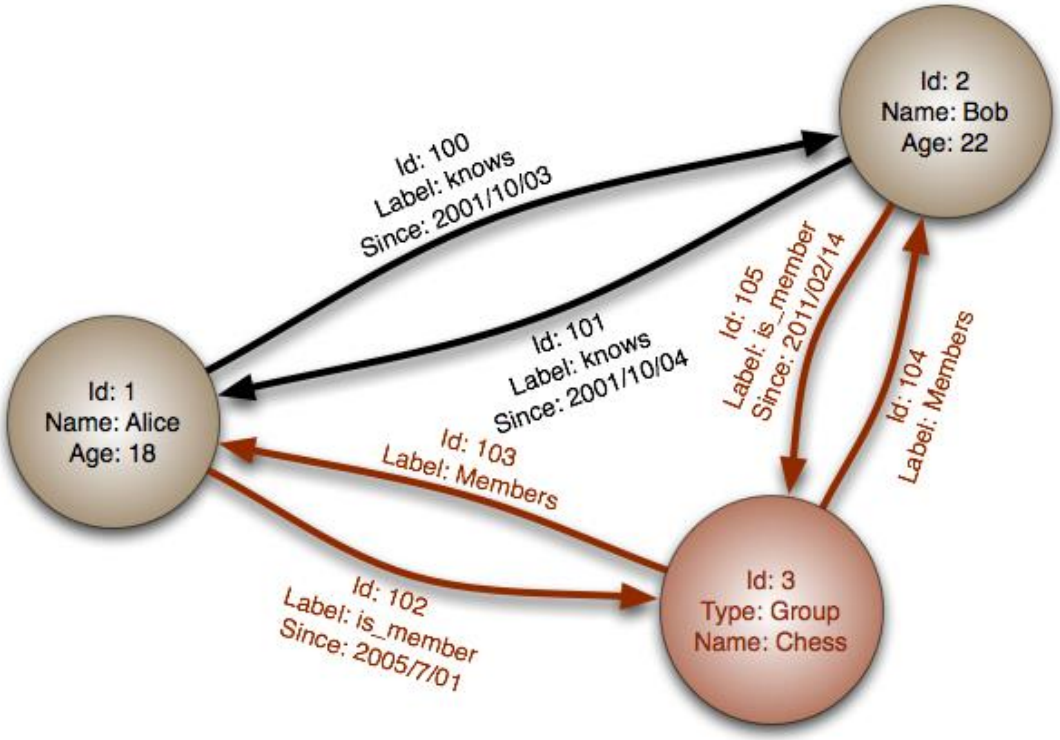
Οι Graph-based βάσεις δεδομένων σχεδιάστηκαν για δεδομένα που απεικονίζονται καλύτερα σε γραφήματα, όπως οι χάρτες, τοπολογίες δικτύου, μεταφορές, η αναπαράσταση των κοινωνικών σχέσεων. Σε μία Graph Base NoSQL βάση δεδομένων δεν βρίσκουμε τη συνηθισμένη άκαμπτη μορφή της SQL ή την

αναπαράσταση σε πίνακες και κολώνες, αλλά μία ευέλικτη γραφική αναπαράσταση η οποία είναι ιδανική για να αντιμετωπίσει τις ανάγκες επεκτασιμότητας (scalability) που υπάρχουν. Τα γραφήματα σχεδιάζονται με ακμές, κόμβους και ιδιότητες οι οποίες παρέχουν γειτνίαση χωρίς τη χρήση ευρετηρίου. Τα δεδομένα μπορούν πολύ εύκολα να μετατραπούν από ένα μοντέλο στο άλλο κάνοντας χρήση μιας Graph Based NoSQL βάσης δεδομένων.




Εικόνα 5: Μία Graph βάση διασχίζει ένα Document Store

Τα γραφήματα περιέχουν κόμβους οι οποίοι ***σημειώνονται*** κατάλληλα με ορισμένες ιδιότητες και αυτοί οι κόμβοι έχουν κάποια ***σχέση*** μεταξύ τους η οποία υποδεικνύεται με τις ***ακμές.*** Οι ακμές αυτές έχουν πάντα κατεύθυνση, τύπο, αρχικό και τελικό κόμβο. Όπως οι κόμβοι έτσι και οι ακμές μπορούν να έχουν ιδιότητες. Για παράδειγμα, όπως φαίνεται στο σχήμα που ακολουθεί, το «Η Alice γνωρίζει τον Bob» απεικονίζεται με μία ακμή η οποία και έχει κάποιες ***ιδιότητες.***



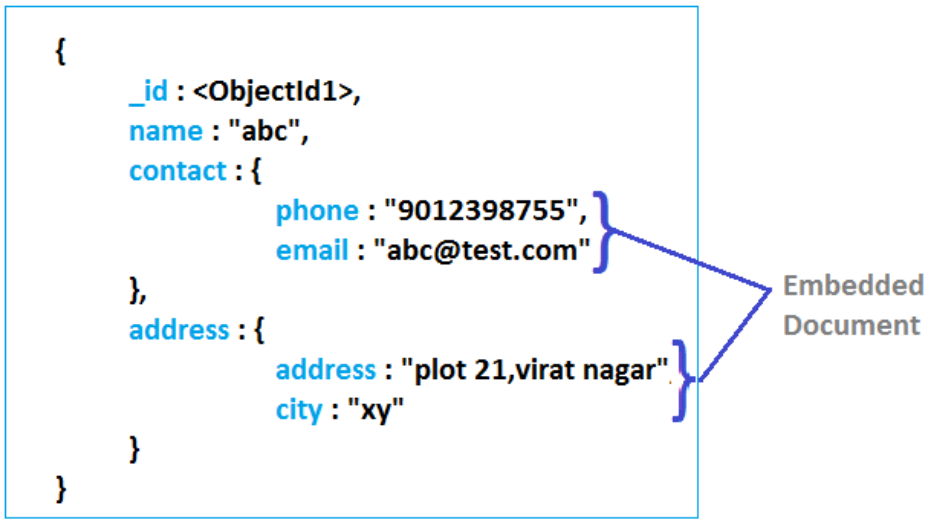

Εικόνα 6: Γραφική αναπαράσταση παραδείγματος Graph Store βάσης δεδομένων

Ενώ τα σχεσιακά μοντέλα βάσεων δεδομένων μπορούν να αντιγράψουν τις graph-based βάσεις, κάθε ακμή απαιτεί ένα join κάτι το οποίο έχει πάρα πολύ μεγάλο κόστος ως προς τον χρόνο. Οι graph-based βάσεις δεδομένων σε σχέση με τις κλασσικές σχεσιακές βάσεις έχουν σαφώς καλύτερη επίδοση για δεδομένα τα οποία έχουν κάποια συσχέτιση μεταξύ τους ενώ προσφέρονται για προβλήματα που επιλύονται με αντικειμενοστρέφεια. Να τονίσουμε ακόμα την ευελιξία που προσφέρουν καθώς η δομή του γραφήματος μπορεί να αλλάζει παράλληλα με τις τρέχουσες απαιτήσεις χωρίς την ανάγκη για επένδυση χρόνου σε αρχικό σχεδιασμό και χωρίς να διακινδυνεύσει η υπάρχουσα λειτουργικότητα. Ενώ ταιριάζει απόλυτα με τον agile μέθοδο ανάπτυξης εφαρμογών που χρησιμοποιείται ευρέως, επιτρέποντας στη βάση δεδομένων να εξελίσσεται συνεχώς μαζί με την ανάπτυξη της εφαρμογής αφού παρέχει τα κατάλληλα εργαλεία και ιδιότητες για «ανεπηρέαστη» ανάπτυξη (frictionless development) και συντήρηση συστημάτων. Αν ακόμα υπάρχουν αμφιβολίες για την υπεροχή των graph based βάσεων δεδομένων , στους τομείς που αναφέραμε παραπάνω, αρκεί να αναφέρουμε πως χρησιμοποιείται κατά κόρων από τους τεχνολογικούς γίγαντες όπως το Facebook , Google, PayPal και το LinkedIn.

## 2.5 Mongo DB

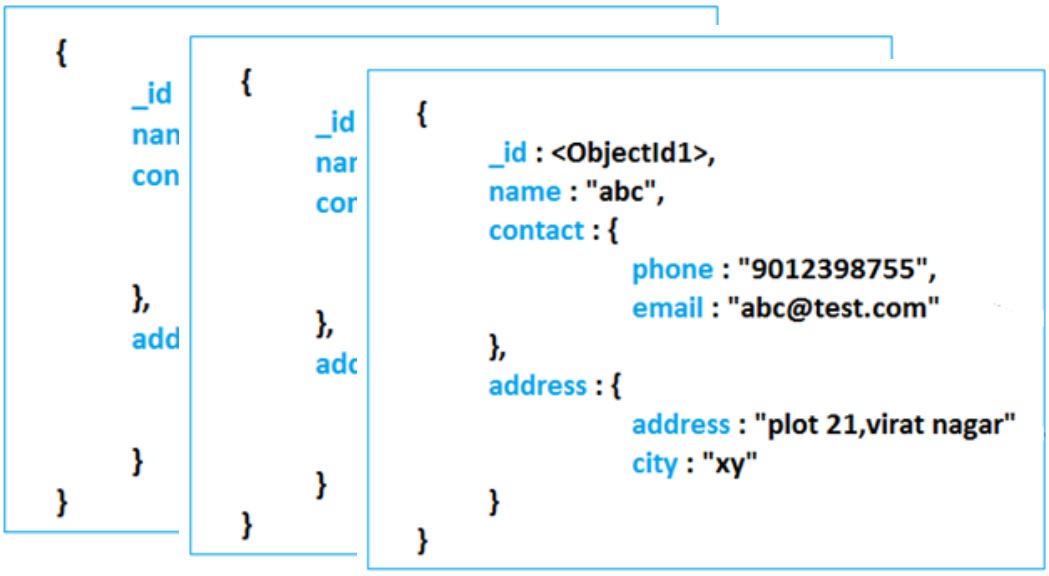
Η Mongo DB είναι μια ανοιχτού κώδικα βάση δεδομένων «εγγράφων» (document database) και μια από τις πιο δημοφιλής μη σχεσιακές βάσεις [7]. Είναι σχεδιασμένη ώστε να επεκτείνεται οριζόντια, μπορώντας να διαχειριστεί αυξήσεις στα μεγέθη των δεδομένων. Είναι απλή στην εκμάθηση και τη χρήση, ενώ συνοδεύεται από πολύ καλή υποστήριξη καθώς και από ένα αχανές πλήθος από εργαλεία που βελτιώνουν την παραγωγικότητα τόσο του προγραμματιστή όσο και του διαχειριστή.

Το αντίστοιχο της εγγραφής της SQL ονομάζεται έγγραφο στη MongoDB και περιέχει πεδία που απαρτίζονται από ζεύγη κλειδιών-τιμών. Τα έγγραφα της MongoDB είναι παρόμοια με τα αντικείμενα JSON, αλλά είναι τυπικά BSON έγγραφα, μια δυαδική αναπαράσταση του JSON με επιπλέον πληροφορίες. Στα έγγραφα, η τιμή ενός πεδίου μπορεί να είναι οποιοσδήποτε τύπος δεδομένου BSON, συμπεριλαμβανομένου άλλων εγγράφων, πινάκων, πινάκων από έγγραφα και άλλα.




Εικόνα 7: Παράδειγμα εγγράφου στη MongoDB

Η MongoDB αποθηκεύει όλα τα έγγραφα σε *συλλογές*. Δηλαδή μια σειρά από συσχετιζόμενα έγγραφα τα οποία έχουν ένα κοινό σύνολο μοιραζόμενων δεικτών. Θα μπορούσαμε να πούμε πως είναι ανάλογο με τους πίνακες στις σχεσιακές βάσεις δεδομένων.




Εικόνα 8: Παράδειγμα συλλογής στη MongoDB

## 2.5.1 Κύρια Χαρακτηριστικά

**Ad hoc ερωτήματα**

Η MongoDB υποστηρίζει ερωτήματα με ονόματα πεδίων, εύρος τιμών, regular expressions και μπορεί να διαμορφωθεί ώστε να επιστρέφει ένα . Ακόμα περιέχει δείκτες οι οποίοι μπορούν να δεικτοδοτήσουν κλειδιά και σε ενσωματωμένα έγγραφα (documents).

**Υπηρεσία Λειτουργίας Αντιγράφων (Replication)**

Η MongoDB εξασφαλίζει υψηλή διαθεσιμότητα με τα λεγόμενα replica sets (συλλογή αντιγράφων). Ένα replica set αποτελείται από δύο ή περισσότερα αντίγραφα των δεδομένων. Καθένα από αυτά μπορεί να δρα ως η κύρια ή δευτερεύουσα replica ανά πάσα στιγμή. Όλες οι εγγραφές και οι αναγνώσεις γίνονται στη κύρια replica ενώ η δευτερεύουσα διατηρεί αντίγραφο της κύριας. Όταν υπάρξει κάποια αστοχία στη κύρια replica, τότε άμεσα μέσα από μία εσωτερική διαδικασία «εκλογής» αποφασίζεται ποια από τις δευτερεύουσες replicas θα είναι η νέα κύρια.

**Εξισορρόπηση Φόρτου ( Load Balancing)**

Η MongoDB προσφέρει οριζόντια κλιμάκωση ως βασική της υπηρεσία και μας δίνει τη δυνατότητα κατακερματισμού των δεδομένων(sharding) σε ένα σύνολο(cluster) υπολογιστών. Ο χρήστης διαλέγει ένα κλειδί κατακερματισμού (shard key) και αποφασίζει πώς θα οργανωθούν τα δεδομένα σε μία συλλογή (collection). Τα δεδομένα χωρίζονται σε εύρη (ανάλογα με το κλειδί κατακερματισμού) και μοιράζονται σε πολλαπλά shards (βάσεις σε διαφορετική μηχανήματα).

### Ομαδοποίηση δεδομένων (Aggregation)

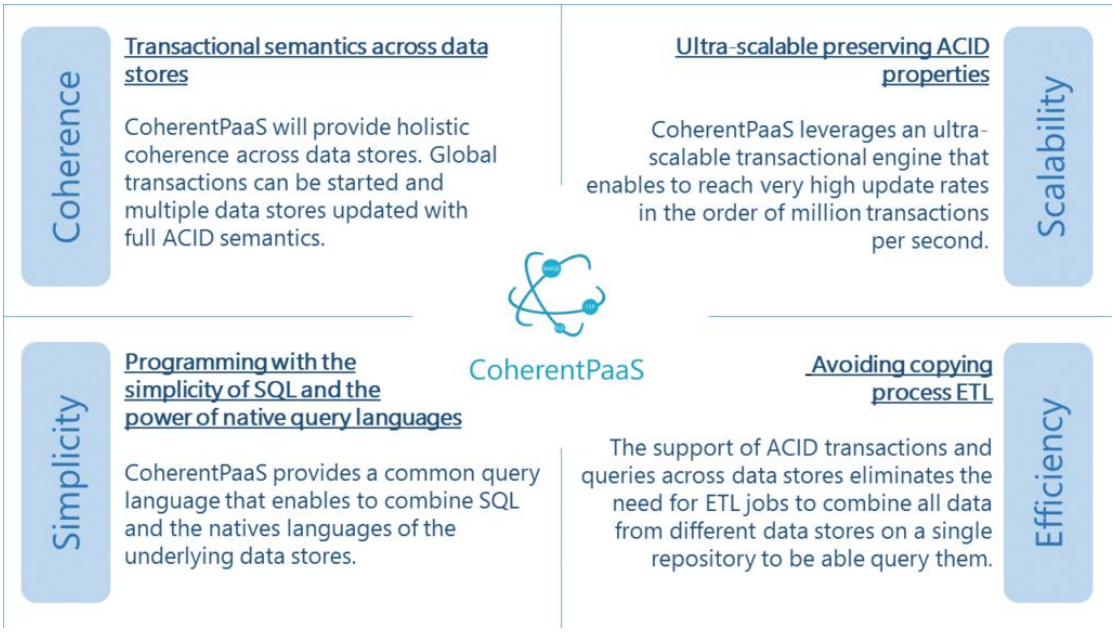
Ο χρήστης έχει τη δυνατότητα να αποκτήσει αποτελέσματα ερωτημάτων στην ίδια μορφή με αυτή που επιστρέφουν οι σχεσιακές γλώσσες με τη χρήση του GROUP BY τελεστή. Το πλαίσιο (framework) που μας παρέχει η Mongo περιλαμβάνει τον τελεστή $lookup ο οποίος μπορεί να πραγματοποιήσει “join” εγγράφων με πολλαπλά έγγραφα, καθώς και στατιστικούς τελεστές όπως η τυπική απόκλιση. Οι τελεστές ομαδοποίησης (ή και συνάθροισης όπως αλλιώς ονομάζονται) μπορούν να χρησιμοποιηθούν σειριακά σε μορφή pipeline , παρόμοια με τα Unix pipes.

### Sharding

Sharding ονομάζεται η διαδικασία «σπασίματος» των δεδομένων σε πολλά μικρότερα κομμάτια. Η κυριολεκτική μετάφρασή της λέξης shard στα ελληνικά είναι «θραύσμα». Πρόκειται ουσιαστικά για μία μέθοδο η οποία μας δίνει τη δυνατότητα να αποθηκεύουμε τα δεδομένα μας σε πολλούς servers. Το sharding αξιοποιείται από τη MongoDB για την υποστήριξη εφαρμογών οι οποίες εμπλέκονται με τη διαχείριση μεγάλου όγκου δεδομένων και υψηλό ρυθμό μεταφοράς αυτών (high data transfer rate).

# ΚΕΦΑΛΑΙΟ 3
# Coherent Paas

Το project Coherent Paas αναπτύσσεται από το Ινστιτούτο Συστημάτων Επικοινωνιών και Πληροφορικής σε συνεργασία με 11 εταιρίες και ακαδημαϊκά ιδρύματα [8]. Έχει ως στόχο να ανταποκριθεί στις σύγχρονες απαιτήσεις για το χτίσιμο κλιμακούμενων εφαρμογών νέφους (scalable cloud applications). Οι προγραμματιστές εφαρμογών στις μέρες μας πρέπει να χρησιμοποιήσουν μια σειρά από διαφορετικές τεχνολογίες διαχείρισης δεδομένων νέφους οι οποίες είναι εντελώς ασύνδετες μεταξύ τους. Για παράδειγμα, μία βάση δεδομένων γραφημάτων, μαζί με μία αποθήκευση κλειδιού-τιμής σε μία σχεσιακή βάση δεδομένων. Αυτή η εξ' ολοκλήρου έλλειψη συντονισμού μεταξύ των δομών αποθήκευσης νέφους δημιουργεί πολλά προβλήματα. Τα κυριότερα τρία από αυτά είναι η έλλειψη συνοχής των δεδομένων (data coherence), η μεγάλη προσπάθεια που καταβάλλεται για την ανάπτυξη ερωτημάτων (queries) σε διαφορετικές δομές αποθήκευσης που βρίσκονται σε όλη την έκταση του νέφους και πρέπει να προγραμματιστούν χειροκίνητα. Τέλος η δυσκολία πραγματοποίησης αποσφαλμάτωσης επίδοσης (performance debugging) σε εφαρμογές που εκτελούνται σε διάφορα μέρη του νέφους.




**Εικόνα 9: Θεμελιώδεις βάσεις του CoherentPaas**

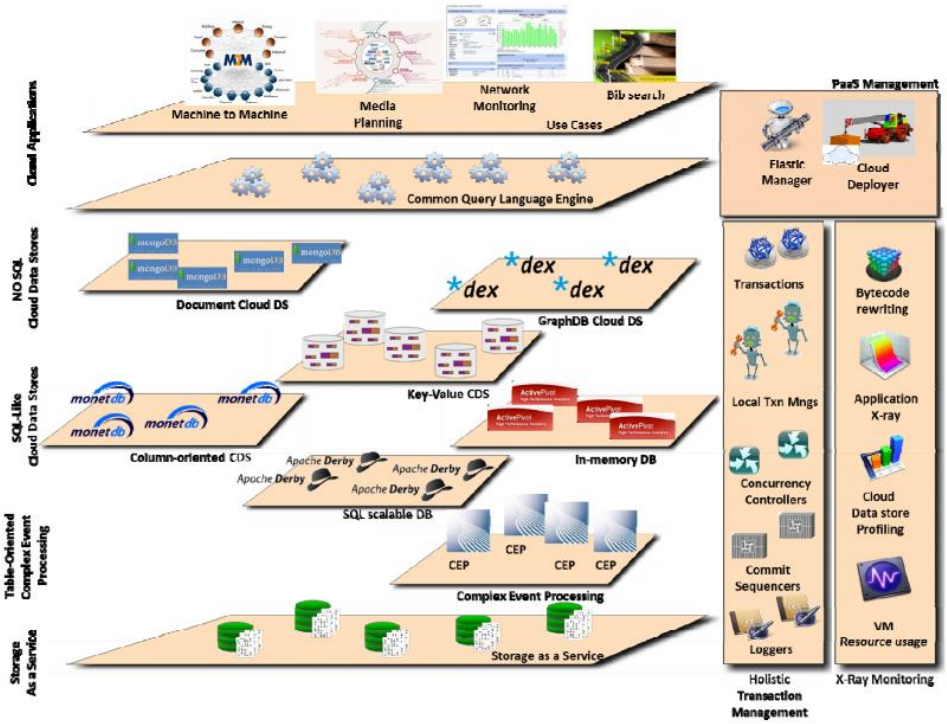
Το CoherentPaas απευθύνεται στα προβλήματα αυτά πραγματοποιώντας τρείς συνεισφορές. Το πρόβλημα συνοχής δεδομένων αντιμετωπίζεται παρέχοντας ολιστικές (holistic) συναλλαγές για όλες τις ενέργειες αποθήκευσης δεδομένων. Με αποτέλεσμα έναν διαχειριστή συναλλαγών ο οποίος επιτρέπει συναλλαγές με οποιοδήποτε συνδυασμό δεδομένων αποθήκευσης παρέχοντας άρτια εφαρμογή της ACID σημασιολογίας.

Με την ανάπτυξη μίας κοινής γλώσσας ερωτημάτων και της αντίστοιχης «μηχανής» της η οποία μπορεί να εκτελεί ερωτήματα γραμμένα σε SQL σε κάθε τύπο δομών αποθήκευσης δεδομένων το CoherentPaas εξαφανίζεται η ανάγκη χειροκίνητης ανάπτυξης ερωτημάτων. Εφόσον κάποιες δομές αποθήκευσης έχουν κάποια εξειδικευμένη διεπαφή επικοινωνίας ή γλώσσα ερωτημάτων η οποία είναι σημαντικό να επιτυγχάνει υψηλή επίδοση, όπως η βάσης γραφημάτων, η γλώσσα ερωτημάτων που αναπτύχθηκε μπορεί να ενσωματώσει υποερωτήματα τα οποία είναι εξ' ολοκλήρου γραμμένα στη native γλώσσα αυτών. Με αυτό τον τρόπο είναι δυνατό να συνδυάσει τη πλήρη δύναμη των native γλωσσών ερωτημάτων μαζί με την εκφραστικότητα της SQL ώστε να επιτύχει το καλύτερο δυνατό αποτέλεσμα. Η κοινή αυτή μηχανή της γλώσσας ερωτημάτων είναι ενσωματωμένη με τις ολιστικές συναλλαγές για να εγγυηθούν πλήρη συνοχή συναλλαγών για τις συναλλαγές αυτές που περιλαμβάνουν καθολικά ερωτήματα (global queries).

Η δυσκολία πραγματοποίησης αποσφαλμάτωσης σε πολύπλοκες εφαρμογές η οποίες περιλαμβάνουν πολλαπλές δομές αποθήκευσης νέφους αντιμετωπίζεται με έναν συνδυασμό ενεργειών από μέρους της πλατφόρμας. Αρχικά παρέχεται ένα πλαίσιο το οποίο οδηγεί διαφανώς της εφαρμογές σε fine grain παρακολούθηση του χρόνου εκτέλεσης κάθε μεθόδου της εφαρμογής και κάθε κλήσης σε δομή αποθήκευση νέφους. Με τον τρόπο αυτό γίνεται δυνατό να βρούμε που συμβαίνουν bottleneck επίδοσης στο επίπεδο εφαρμογής. Επιπροσθέτως, οι ενέργειες αποθήκευσης, αυτές καθ' αυτές, θα παρακολουθούνται σε fine grain επίπεδο και αυτές με τη σειρά τους, επιτρέποντας μας να κάνουμε συσχετισμό με την επίδοση της εφαρμογής. Η παρακολούθηση αυτή στο σύνολό της καθιστά ικανό το προγραμματιστή να πραγματοποιήσει λεπτές ενέργειες συντονισμού στων δομών αποθήκευσης του νέφους και να βρει ποιες ρυθμίσεις είναι επαρκής για το φόρτο εργασίας τους.

Τα διάφορα υποσυστήματα παρουσιάζονται στη παρακάτω εικόνα. Ο ολιστικός διαχειριστής συναλλαγών συμπεριλαμβάνει όλα τα συστατικά για την υπερ-κλιμακούμενη (ultra-scalable) επεξεργασία συναλλαγών, ενώ συναναστρέφεται με όλες τις δομές αποθήκευσης νέφους, NoSQL, Sql και CEP. Η κοινή μηχανή γλώσσας ερωτημάτων αναπαρίσταται κοντά στη κορυφή μεταξύ των εφαρμογών και των δομών αποθήκευσης νέφους. Έρχεται σε επαφή με όλες τις δομές αποθήκευσης νέφους και την ολιστικό διαχειριστή συναλλαγών. Υπάρχει ένα κομμάτι που είναι υπεύθυνο για την διαχείριση της πλατφόρμας φροντίζοντας τις αποφάσεις που αφορούν την επανεξέταση των παραμέτρων ρυθμίσεων καθώς και τη διαδικασία

εγκατάστασης της πλατφόρμας στο περιβάλλον του νέφους. Τέλος στη κορυφή του σχήματος βλέπουμε κάποιες περιπτώσεις χρήσης. Οι συγκεκριμένες έχουν επιλεχθεί διότι αποτελούν εφαρμογές που μπορούν να εκμεταλλευτούν άμεσα την ύπαρξη πολλαπλών τεχνολογιών διαχείρισης δεδομένων νέφους που είναι ενσωματωμένες στο project αυτό.




Εικόνα 10: Η πλατφόρμα του Coherent Paas

## 3.1 Κύρια Χαρακτηριστικά

Στο σημείο αυτό, και πριν αναφερθούμε στο κομμάτι το οποίο είμασταν υπεύθυνοι για την ανάπτυξη του, κρίνεται απαραίτητη η περιγραφή των βασικών συστατικών του CoherentPaas έτσι ώστε να καταλάβουμε ακριβώς πως λειτουργεί το σύστημα στο σύνολο του. Τα κυριότερα από αυτά είναι η Εφαρμογή χρήστη, ο dsClient, το αντικείμενο MongoDB, ο LTMClient και ο Transcational Manager.

Αρχίζοντας από τη πλευρά του χρήστης ο οποίο καλείται να χρησιμοποιήσει την εφαρμογή αυτή, έχουμε την **Εφαρμογή χρήστη** (client application), με την οποία διεκπεραιώνονται και όλες οι συναλλαγές.

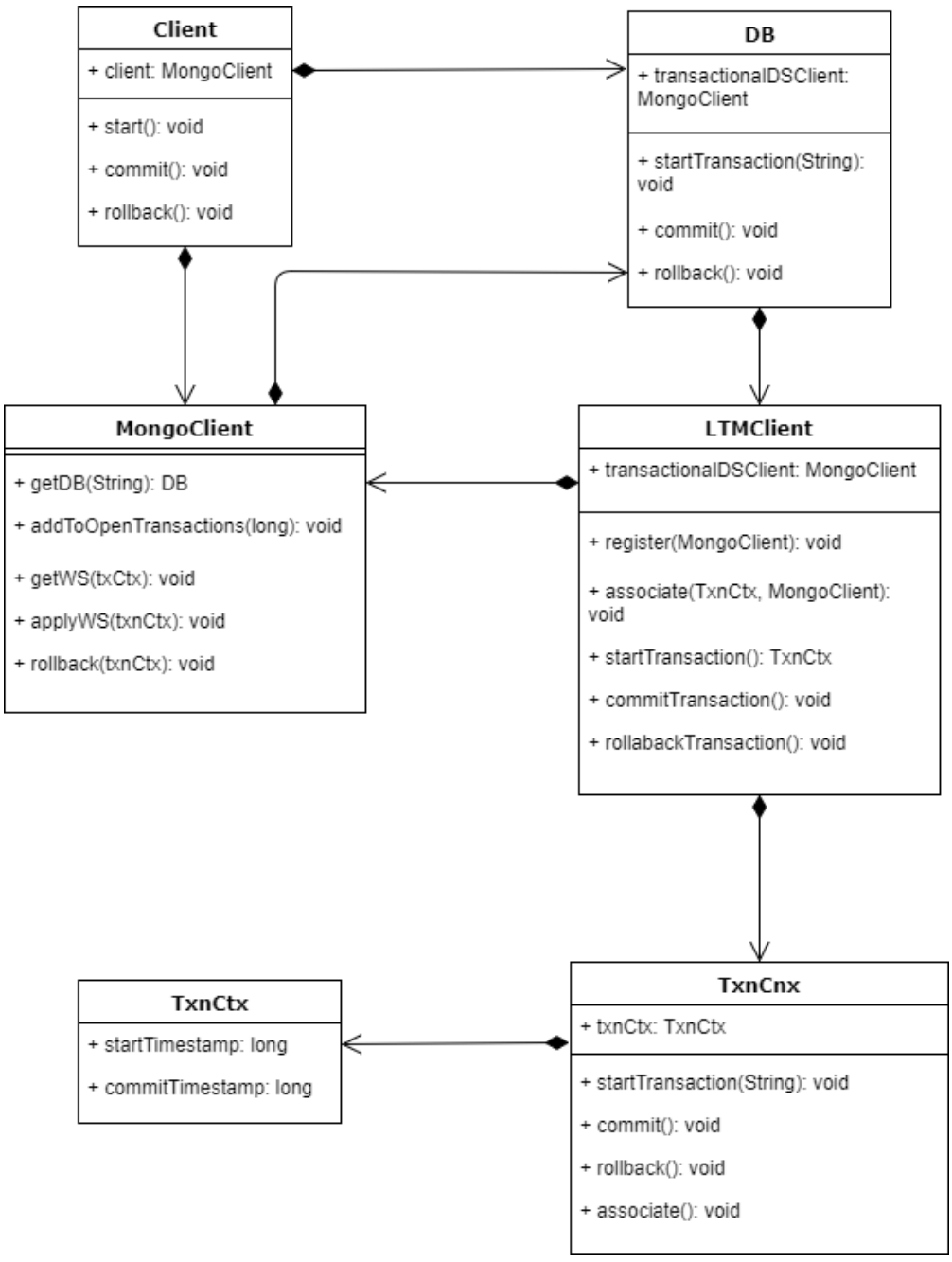
Στη συνέχεια συναντάμε το singleton αντικείμενο **dsClient** το οποίο κάνει χρήση η Εφαρμογή χρήστη για να δημιουργήσει κάποιο αντικείμενο τύπου MongoDB και μόνο γι' αυτό. Αφού σε όλες τις άλλες περιπτώσεις προσπελαύνεται από άλλα εσωτερικά αντικείμενα και μεθόδους του συστήματος και όχι από την εφαρμογή χρήστη.

Το **αντικείμενο MongoDB**, το οποίο αναφέραμε παραπάνω, πρόκειται για τη διεπαφή η οποία επιτρέπει στην εφαρμογή χρήστη να πραγματοποιεί όλες τις λειτουργίες κάποιας συναλλαγής. Είτε πρόκειται για δημιουργία , για ολοκλήρωση αλλά και για ματαίωση. Το αντικείμενο αυτό το οποίο δημιουργείται συνεχίζει να υπάρχει και να επικοινωνεί με τις εσωτερικές δομές του συστήματος με τρόπο αδιαφανή προς το χρήστη. Όπως ο dsClient είναι singleton και υπάρχει ένα προς ένα αντιστοιχία του με την εφαρμογή χρήστη με τον ίδιο τρόπο αντιστοιχεί μόνο ένα αντικείμενο MongoDB στην εφαρμογή το οποίο και δημιουργεί ο ίδιος ο dsClient και το παρέχει στον χρήστη.

Εν συνεχεία ένα από τα βασικά στοιχεία του συστήματος, επιφορτισμένο για τη παρακολούθηση ορισμένων λειτουργιών των συναλλαγών σε τοπικό επίπεδο το οποίο και βρίσκεται σε συνεχή επικοινωνία με τον απομακρυσμένο διαχειριστή συναλλαγών, αποτελεί ο **LTM Client**. Κατά την εκκίνηση μίας συναλλαγής, από την εφαρμογή χρήστη , έχουμε την εγγραφή στον LTM Client. Επιπροσθέτως, κατά τη διαδικασία ολοκλήρωσής της, ο LTM Client είναι υπεύθυνος να διεξάγει ελέγχους στη δομή με τις ενεργές συναλλαγές που αντιστοιχούν στην εφαρμογή χρήστη για να εξακριβωθεί αν υπάρχουν συγκρούσεις (conflicts) με άλλες συναλλαγές. Ενώ σε περίπτωση ματαίωσης ή ολοκλήρωσης της συναλλαγής έχουμε τη διαγραφεί της από την προαναφερθείσα δομή με τις ενεργές συναλλαγές. Με τον ίδιο τρόπο με τα προηγούμενα συστατικά, ο LTM Client έχει μοναδική αντιστοίχιση με το αντικείμενο τύπου MongoDB. Συμπεραίνουμε δηλαδή πως και για τα τρία αυτά αντικείμενα οποιαδήποτε αλληλεπίδραση πραγματοποιηθεί, συμβαίνει στα 3 μοναδικά instances που έχουν δημιουργηθεί, ένα για κάθε τύπο.

Τέλος, το τελευταίο και πιο σημαντικό συστατικό του συστήματος αποτελεί ο **Transactional Manager** (Διαχειριστής Συναλλαγών). Πρόκειται για μία μονάδα η οποία βρίσκεται απομακρυσμένη πάνω σε κάποια υποδομή και είναι μοναδική για το σύνολο του συστήματος. Ο ρόλος του είναι η διαχείριση των συναλλαγών οι οποίες πηγάζουν από το σύνολο των εφαρμογών χρήστη που είναι ενεργές. Για την επίτευξη του σκοπού αυτού κάνει χρήση μιας δομής στην οποία διατηρεί όλες τις ενεργές συναλλαγές στις οποίες και απονέμει χρονοσφραγίδες τόσο τη στιγμή εισαγωγής τους (εκκίνηση) αλλά και κατά την ολοκλήρωσή τους.

Αφού περιγράψαμε επιγραμματικά καθένα από τα κύρια συστατικά του συστήματος CoherentPaas με τη βοήθεια ενός απλοποιημένου διαγράμματος κλάσης θα γίνει πιο κατανοητό πως συνδέονται και επικοινωνούν μεταξύ τους.




Εικόνα 11: Η αλληλεπίδραση των κύριων κλάσεων του Transactional Manager

## 3.2 Περιγραφή του αλγόριθμου διαχείρισης συναλλαγών του CoherentPaas

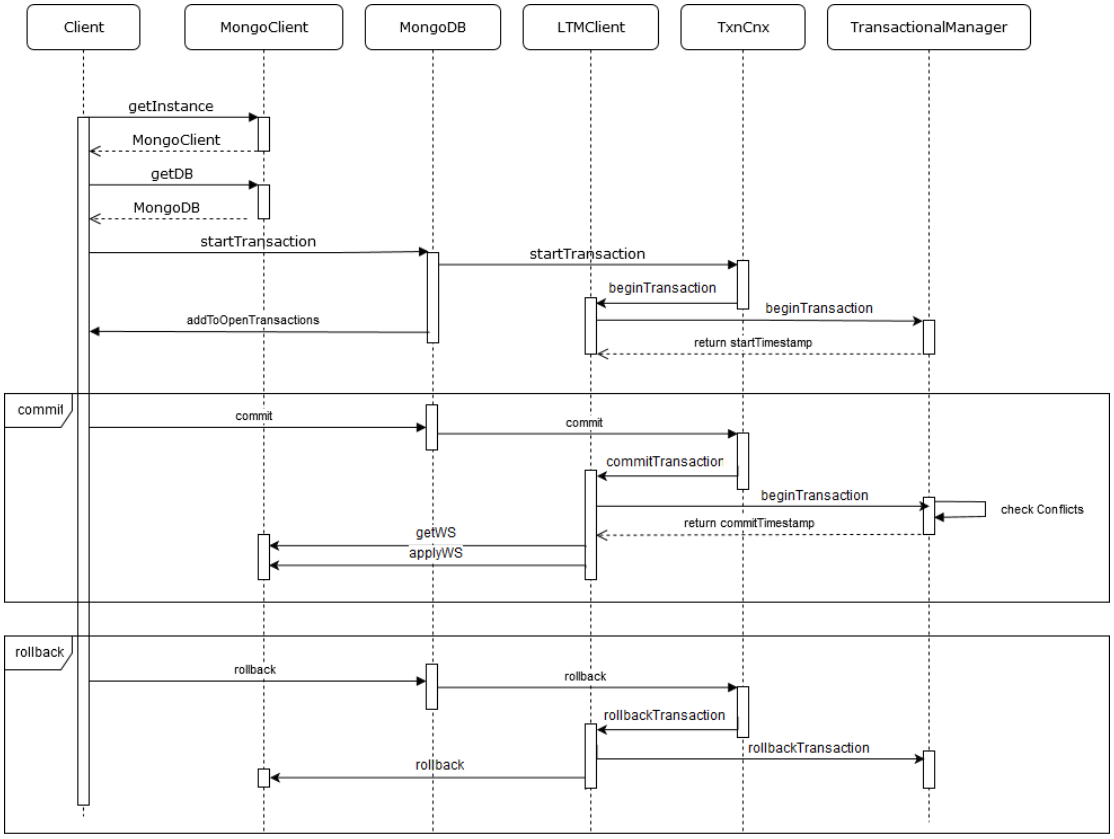
Στο προηγούμενο κεφάλαιο παρουσιάσαμε τα κύρια συστατικά του CoherentPaas στο και στο σημείο αυτό θα εξετάσουμε πως αυτά επικοινωνούν και γενικότερα αλληλεπιδρούν μεταξύ τους ώστε να ανταποκριθούν στις απαιτήσεις, όπως αυτές έχουν οριστεί από το στάδιο σχεδίασης, και να παρέχουν τα επιθυμητά αποτελέσματα.

Το έναυσμα για την εκκίνηση των λειτουργιών του συστήματος δίνεται, όπως πάντα, από την εφαρμογή χρήστη (client) η οποία δημιουργεί ένα singleton (μοναδικό) αντικείμενο MongoClient, με τη βοήθεια του οποίο παράγεται και ένα singleton αντικείμενο DB. Ο χρήστης χρησιμοποιεί το DB αυτό αντικείμενο για να εκτελέσει όλες τις λειτουργίες που αφορούν μία συναλλαγή - δηλαδή δημιουργία, ολοκλήρωση ή ματαίωση – με πλήρη αδιαφάνεια ως προς το πώς υλοποιούνται οι λειτουργίες αυτές.

Επιπροσθέτως το αντικείμενο DB δημιουργεί ένα αντικείμενο LTMClient το οποίο σχετίζεται παράλληλα με το ίδιο το DB αντικείμενο αλλά και το MongoClient αντικείμενο. Η σχέση τους, λόγω και της singleton ιδιότητας των αντικειμένων, είναι «ένα προς ένα», γεγονός που συνεπάγεται πως όταν το ένα θελήσει να χρησιμοποιήσει τη λειτουργικότητα του άλλου τότε θα επικοινωνήσει με το μοναδικό αντικείμενο, με το οποίο και έχει συσχετιστεί.

Όσον αφορά τον LTMClient, εκείνος αναλαμβάνει τον ρόλο μεσολαβητή μεταξύ της εφαρμογής χρήστη και του διαχειριστή συναλλαγών καθώς και την υποστήριξη, σε τοπικό επίπεδο ορισμένων λειτουργιών του διαχειριστή συναλλαγών όπως ο έλεγχος για «σύγκρουσης» (conflicts) μεταξύ των συναλλαγών. Συμπερασματικά όταν μια συναλλαγή εκκινήσει από την εφαρμογή χρήστη αυτή καταγράφεται στον LTMClient ενώ όταν έρθει η στιγμή της ολοκλήρωσης της ο LTMClient είναι υπεύθυνος για την εξέταση της δομής με τις ενεργές συναλλαγές που αφορούν την συγκεκριμένη εφαρμογή χρήστη ώστε να εντοπίσει πιθανές συγκρούσεις με άλλες συναλλαγές. Τέλος η ολοκλήρωση ή ματαίωση μιας συναλλαγής ακολουθείται από την αφαίρεση της από την δομή με τις ενεργές συναλλαγές που μόλις αναφέραμε.

Η παραπάνω περιγραφή του αλγορίθμου διαχείρισηq συναλλαγών CoherentPaas αναπαρίσταται και στο σχήμα που ακολουθεί για την καλύτερη κατανόηση.




**Εικόνα 12: Αλγόριθμος διαχείρισης συναλλαγών του CoherentPaas**

# ΚΕΦΑΛΑΙΟ 4
# Ntua Transactional Manager

Ο ρόλος του NtuaTranscationalManager, δηλαδή του διαχειριστή συναλλαγών, είναι να δέχεται ερωτήματα από τις εφαρμογές χρήστη και να τα χειρίζεται κατάλληλα. Ο διαχειριστής συναλλαγών εκτελείται, μοναδικά, σε ένα απομακρυσμένο μηχάνημα. Κάποιες από τις κύριες λειτουργίες του είναι η λήψη απόφασης για το αν μια συναλλαγή μπορεί να εκκινήσει ή σε περίπτωση «σύγκρουσης» (conflict) αν μπορεί να ολοκληρωθεί. Επιπροσθέτως είναι υπεύθυνη για την αναίρεση των συναλλαγών καθώς και για την παροχή των εφαρμογών χρήστη των απαραίτητων πληροφοριών σχετικά με τις ανοιχτές συναλλαγές που υπάρχουν.

## 4.1 Βασικά πεδία Ntua Transactional Manager

*trId*: AtomicLong
Μία thread safe long μεταβλητή που χρησιμοποιείται ως αναγνωριστικό της συναλλαγής, αλλά και ως επόμενο χρονικό αναγνωριστικό (χρονοσφραγίδα) καθώς είναι μοναδικό και αυξάνεται με τον χρόνο.

*minOpenTransactionId*: AtomicLong
Μία επίσης thread safe long μεταβλητή που περιέχει το αναγνωριστικό της συναλλαγής με το μέγιστο αναγνωριστικό για την οποία όλες οι προηγούμενες συναλλαγές είναι ολοκληρωμένες.

*addressSocketMap*: ConcurrentHashMap<InetAddress,Integer>
Ένα thread safe HashMap στο οποίο αποθηκεύονται οι εφαρμογές χρήστη οι οποίες είναι ενεργές και σε επικοινωνία με τον διαχειριστή συναλλαγών. Στο HashMap αυτό διατηρούνται τα ζεύγη τιμών IPs και πόρτας επικοινωνίας κάθε εφαρμογής, στα οποία ο διαχειριστής συναλλαγών ενημερώνει ανά τακτά χρονικά διαστήματα τις εφαρμογές χρηστών για το minOpenTransactionId.

*openTrId:* ConcurrentHashMap<Long,TransactionContext>
Ακόμα ένα thread safe HashMap στο οποίο αποθηκεύονται οι ενεργές συναλλαγές από τις εφαρμογές χρηστών που είναι σε επικοινωνία με τον διαχειριστή συναλλαγών. Σαν κλειδί της εκάστοτε εγγραφής στην συγκεκριμένη δομή ορίζεται το μοναδικό αναγνωριστικό της εφαρμογής, ενώ η τιμή της εγγραφής είναι το περιεχόμενο της συναλλαγής.

*openTrCtx*: ConcurrentHashMap<TransactionContext,Long>

## 4.2 Βασικές μέθοδοι Ntua Transactional Manager

Για την υλοποίηση της ζητούμενης λειτουργικότητας του διαχειριστή συναλλαγών, δημιουργήθηκαν οι απαραίτητες μέθοδοι μέσα από τις οποίες επικοινωνούν οι εφαρμογές χρηστών με τον διαχειριστή συναλλαγών.

**register** (RegisterRequest request) : **boolean**

Η μέθοδος register καλείται κατά την αρχικοποίηση μιας εφαρμογής χρήστη και χρησιμοποιείται για την εκκίνηση της επικοινωνίας της εφαρμογής με τον διαχειριστή συναλλαγών. Σαν όρισμα έχει ένα αντικείμενο τύπου RegisterRequest (το οποίο θα αναλύσουμε περισσότερο στην παράγραφο 5.2.3) στο οποίο περιλαμβάνονται τα δικτυακά στοιχεία της εφαρμογής χρήστη (δηλαδή η IP και η προκαθορισμένη πόρτα επικοινωνίας). Όταν η εφαρμογή χρήστη επικοινωνεί σε αυτή την μέθοδο, τότε ο διαχειριστής συναλλαγών προσθέτει μία εγγραφή για την εφαρμογή χρήστη στην δομή με τις συνδεδεμένες εφαρμογές (addressSocketMap) και επιστρέφει την τιμή *true* αν ολοκληρωθεί επιτυχώς.

**beginTransaction**(CharSequence writeSet) : **TransactionContext**
Η μέθοδος beginTransaction καλείται από τον NtuaLTMClient κατά την αρχή μιας συναλλαγής. Έπειτα από τους τοπικούς ελέγχους της εφαρμογής χρήστη για τυχόν διενέξεις στο εσωτερικό της, καλείται ο διαχειριστής συναλλαγών στην μέθοδο αυτή, ώστε να προχωρήσει σε έλεγχο για τυχόν διενέξεις και με τις υπόλοιπες εφαρμογές χρηστών που είναι συνδεδεμένες σε αυτόν. Μέσα στην παράμετρο writeSet περιέχεται η πληροφορία για τις συλλογές τις οποίες θέλει να επεξεργαστεί η συναλλαγή. Έτσι ελέγχεται εάν κάποια άλλη συναλλαγή πραγματοποιεί αυτή τη στιγμή αλλαγές στην συγκεκριμένη συλλογή και σε περίπτωση που ισχύει κάτι τέτοιο επιστρέφεται η κατάλληλη απόκριση και η συναλλαγή δεν επιτρέπεται να εκκινήσει. Αντίθετα, αν δεν υπάρχει άλλη ενεργή συναλλαγή που να επεξεργάζεται κάποια κοινή συλλογή, δημιουργείται μία νέα εγγραφή της συναλλαγής στην δομή όπου αποθηκεύονται οι ενεργές συναλλαγές, της ανατίθεται ένα μοναδικό αναγνωριστικό και μία χρονοσφραγίδα εκκίνησης και επιστρέφεται στην εφαρμογή χρήστη.

**commitTransaction**(TransactionContext ctx):**TransactionContext**
Η commitTransaction καλείται από τον NTUALTMClient μιας εφαρμογής χρήστη κατά την ολοκλήρωση κάποιας συναλλαγής. Περνάει σαν παράμετρο το TransactionContext που δημιουργήθηκε κατά την beginTransaction και με βάση αυτό γίνεται έλεγχος για τυχόν διενέξεις με άλλες συναλλαγές. Αν δεν υπάρχουν τέτοιες, τότε η συναλλαγή επιτρέπεται να συνεχίσει και να ολοκληρωθεί. Επομένως της ανατίθεται μια χρονοσφραγίδα ολοκλήρωσης από τον διαχειριστή συναλλαγών και η αναφορά της αφαιρείται από το σύνολο των ανοιχτών συναλλαγών.

Τέλος δημιουργήθηκε ένα ξεχωριστό νήμα (thread) το οποίο μέσω ενός scheduler εκτελεί μια συγκεκριμένη διεργασία ανά καθορισμένο χρονικό διάστημα, μέσω της

οποίας ενημερώνει όλες, τις συνδεδεμένες με τον διαχειριστή συναλλαγών, εφαρμογές χρηστών για την τιμή του minOpenTransactionId.

```
final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
final Runnable clientNotifier= new Runnable() {
        public void run() {
                long temp = minOpenTransactionId.get();
                while (!openTransactionId.containsKey(temp++));
                minOpenTransactionId.set(--temp);

                for (Map.Entry<InetAddress, Integer> entry :
addressSocketMap.entrySet()) {
                        InetAddress address = entry.getKey();
                        Integer port = entry.getValue();
                        try {
                                notifyClient(address, port);
                        }catch(IOException ioe) {
                                ioe.printStackTrace();
                        }
                }
        }
```

*Ο scheduler που τρέχει την διεργασία για την ενημέρωση των εφαρμογών χρήστη*

**Επικοινωνία εφαρμογών χρήστη με τον διαχειριστή συναλλαγών**

Για την επικοινωνία των εφαρμογών χρηστών με τον απομακρυσμένο διαχειριστή συναλλαγών αποφασίστηκε να γίνει χρήση του Avro της Apache. Κάνοντας χρήση της RPC λειτουργικότητας του, οριστήκαν τα κατάλληλα σημεία επικοινωνίας μέσω ενός αρχείου avpr που φαίνεται παρακάτω. Σε αυτό σύμφωνα με το πρότυπο του Avro περιγράφηκαν σε δομή JSON οι υπογραφές των μεθόδων επικοινωνίας καθώς και τα είδη των σειριοποιήσιμων αντικειμένων – παραμέτρων τους.

```
{
 "namespace": "eu.coherentpaas.mongovcc.ntua.avro",
 "protocol": "MongoBroker",

 "types": [
  {"name": "TransactionContext", "type": "record",
   "fields": [
    {"name": "id",   "type": "long"},
    {"name": "startTimestamp", "type": "long"},
    {"name": "commitTimestamp", "type": "long"}
   ]
  },
  {"name": "RegisterRequest", "type" : "record",
   "fields": [
    {"name": "clientIp", "type": "string"},
    {"name": "port",   "type": "int"}
```

```
    ]
   }
  ],

  "messages" : {
   "register" :{
    "request" :[{"name" : "request", "type" : "RegisterRequest"}],
    "response" :"boolean"
   },

   "beginTransaction" : {
    "request" :[{"name" : " writeSet ", "type" : "string"}],
    "response" : "TransactionContext"
   },

   "commitTransaction": {
    "request" : [{"name" : "transCtx", "type" : "TransactionContext"}],
    "response" : "TransactionContext"
   },

   "rollbackTransaction": {
    "request" : [{"name" : "transCtx", "type" : "TransactionContext"}],
    "response" : "boolean"
   },

   "checkConflicts": {
    "request" : [{"name" : "transCtx", "type" : "TransactionContext"}],
    "response" : "boolean"
   }
  }
}
```

*Το avpr αρχείο για τον ορισμό των σχημάτων και των μεθόδων επικοινωνίας των εφαρμογών χρηστών με τον διαχειριστή συναλλαγών*

Έτσι, οριστήκαν οι υπογραφές των μεθόδων επικοινωνίας με τον διαχειριστή συναλλαγών, καθώς και τα σύνθετα αντικείμενα τα οποία χρησιμοποιούνται ως παράμετροι ή ως επιστρεφόμενες τιμές για τις αυτές (όπως περιγράφηκαν στην προηγούμενη παράγραφο - 5.2.2). Ας εξετάσουμε ως παράδειγμα την μέθοδο beginTransaction.

```
"beginTransaction" : {
 "request" :[{"name" : " writeSet ", "type" : "string"}],
 "response" : "TransactionContext"
}
```

*Περιγραφή της μεθόδου "beginTransaction"*

Όπως φαίνεται, η μέθοδος αυτή παίρνει ως όρισμα μια παράμετρο τύπου String με την ονομασία writeSet. Σαν απόκριση της μεθόδου επιστρέφεται ένα αντικείμενο τύπου TransactionContext, το οποίο έχει οριστεί όπως φαίνεται παρακάτω.

```
{"name": "TransactionContext", "type": "record",
 "fields": [
  {"name": "id",  "type": "long"},
  {"name": "startTimestamp", "type": "long"},
  {"name": "commitTimestamp", "type": "long"}
 ]
}
```

*Περιγραφή του αντικειμένου "TransactionContext"*

Το TransactionContext ορίστηκε, λοιπόν, ως νέο αντικείμενο τύπου record, πράγμα που στην γλώσσα του avro μεταφράζεται σε σύνθετη δομή, που περιέχει επιπλέον πεδία στο εσωτερικό της. Στην συγκεκριμένη περίπτωση, το TransactionContext περιέχει ένα πεδίο id τύπου long το οποίο περιέχει το μοναδικό αναγνωριστικό της συναλλαγής, το οποίο της δίνει ο διαχειριστής συναλλαγών. Και επίσης, περιέχει δύο επιπλέον long πεδία τα startTimestamp και commitTimestamp τα οποία περιέχουν τις χρονοσφραγίδες εκκίνησης και ολοκλήρωσης, αντίστοιχα, όπως επίσης τις αποδίδει ο διαχειριστής συναλλαγών.

Κατά αντίστοιχο τρόπο και με βάση τις απαιτήσεις για παραμέτρους και επιστρεφόμενες τιμές των διαφόρων δημόσιων μεθόδων του διαχειριστή συναλλαγών, δημιουργήθηκαν και οι υπόλοιπες μέθοδοι και σύνθετα αντικείμενα. Για την δημιουργία όμως των ίδιων των κλάσεων των αντικειμένων, αλλά και της κλάσης MongoBroker, η οποία περιλαμβάνει όλες τις μεθόδους που περιγράψαμε στο Avro χρειάζεται να τρέξει η διαδικασία generate-sources του Avro.

Επιλέξαμε να ορίσουμε η διαδικασία αυτή να πραγματοποιείται αυτόματα κατά την διαδικασία του χτισίματος της εφαρμογής μέσω του Maven. Έτσι αναθέσαμε την διαδικασία στο avro-maven-plugin. Για να χρησιμοποιήσουμε το plugin αυτό, αλλά και γενικότερα την βιβλιοθήκη του Avro, καθώς και την ζητούμενη rpc λειτουργικότητα της, χρειάστηκε να φορτώσουμε μέσω του Maven τις απαραίτητες βιβλιοθήκες από το default maven repository:

```
<dependency>
  <groupId>org.apache.avro</groupId>
   <artifactId>avro</artifactId>
   <version>1.8.1</version>
</dependency>
<dependency>
   <groupId>org.apache.avro</groupId>
   <artifactId>avro-ipc</artifactId>
   <version>1.8.1</version>
</dependency>
```

*Φόρτωση των κατάλληλων βιβλιοθηκών του Avro*

Στην συνέχεια ορίσαμε κατά το χτίσιμο (build) της εφαρμογής να διαβάζεται το avpr αρχείο που δημιουργήσαμε και να δημιουργούνται οι κατάλληλες κλάσεις. Αυτό έγινε με την ρύθμιση (configuration) που εμφανίζεται παρακάτω, όπου δηλώσαμε στην maven ρύθμιση να βρίσκει όλα τα αρχεία Avro (avsc, avpr, κτλ) στον φάκελο που βρίσκεται στην διαδρομή: «${project.basedir}/src/main/avro/» (όπου ${project.basedir} είναι ο φάκελος στον οποίο βρίσκεται η εφαρμογή) και τον πηγαίο κώδικα των java κλάσεων που θα παράγει να τον αποθηκεύει στην διαδρομή: «${project.basedir}/src/main/java/» από όπου θα μπορεί ο κώδικας μας να έχει πρόσβαση στα αρχεία αυτά.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.avro</groupId>
      <artifactId>avro-maven-plugin</artifactId>
      <version>1.8.1</version>
      <executions>
        <execution>
          <phase>generate-sources</phase>
          <goals>
            <goal>protocol</goal>
          </goals>
          <configuration>
            <sourceDirectory>
              ${project.basedir}/src/main/avro/
            </sourceDirectory>
            <outputDirectory>
              ${project.basedir}/src/main/java/
            </outputDirectory>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

*Χρήση του Maven για δημιουργία κλάσεων από το avpr αρχείο*

Έτσι κατά την εκτέλεση της διαδικασίας χτισίματος της εφαρμογής, δημιουργήθηκαν τρεις νέες Java κλάσεις όπως τις περιγράψαμε στο avpr αρχείο. Οι κλάσεις αυτές είναι οι TransactionContext.java, RegisterRequest.java και MongoBroker.java. Κάθε μία από τις δύο πρώτες περιγράφει ένα αρχείο που χρησιμοποιείτε είτε σαν είσοδος, είτε σαν έξοδος σε κάποια από τις μεθόδους επικοινωνίας με τον διαχειριστή

συναλλαγών. Επομένως είναι σειριποιήσιμα αντικείμενα με τα κατάλληλα πεδία, όπως τα ορίσαμε, με τους απαραίτητους constructors, καθώς και με τις απαραίτητες μεθόδους ανάκτησης των δεδομένων τους (getters) και ορισμού των δεδομένων τους (setters).

Τέλος στο τρίτο αρχείο ορίστηκε μία διεπαφή, με το όνομα MongoBroker στην οποία ορίζονται οι υπογραφές των καταλλήλων μεθόδων επικοινωνίας. Για την χρήση της διεπαφής αυτής, δημιουργήθηκε μία νέα κλάση BrokerImpl, η οποία χρησιμοποιεί την διεπαφή και επεκτείνει τις προκαθορισμένες μεθόδους προσθέτοντας τους την απαραίτητη λογική που περιγράφηκε στην παράγραφο 5.2.3. Επίσης, χρειάστηκε να στηθεί ένας Netty Server σε ένα ξεχωριστό νήμα του διαχειριστή συναλλαγών, ο οποίος μόνιμα θα άκουγε για νέα αιτήματα επικοινωνίας από κάποια εφαρμογή χρήστη και στην συνέχεια της ανέθετε ένα αντικείμενο τύπου BrokerImpl πάνω στο οποίο θα μπορούσαν να καλέσουν τις ορισμένες μεθόδους, οι οποίες μέσω του Avro θα εκτελούνταν στον διαχειριστή συναλλαγών.

# ΚΕΦΑΛΑΙΟ 5
# Συλλέκτης Μη Έγκυρων Δεδομένων (Garbage Collector)

Με τον όρο συλλέκτη απορριμμάτων, ή αλλιώς garbage collector (GC) αναφερόμαστε σε μια μορφή αυτόματης διαχείρισης της μνήμης. Ο GC προσπαθεί να ανακτήσει θέσεις μνήμης οι οποίες είναι κατειλημμένες από δεδομένα που δεν χρησιμοποιούνται πλέον από την τρέχουσα διεργασία. Είναι ουσιαστικά το αντίθετο από την χειροκίνητη διαχείριση μνήμης, που συναντάμε για παράδειγμα στην C, και στην οποία απαιτείται από τον χρήστη να προσδιορίσει τα αντικείμενα που μπορούν να διαγραφούν από την μνήμη.

Η *συλλογή απορριμμάτων* είναι μία τεχνική που χρησιμοποιείται από γλώσσες προγραμματισμού υψηλού επιπέδου όμως οι βασικές αρχές της μπορούν να επεκταθούν και σε άλλους τομείς. Αυτές είναι:

1. Εντοπισμός δεδομένων που δεν πρόκειται να προσπελαστούν στο μέλλον .
2. Παραχώρηση των πόρων που δέσμευαν τα δεδομένα αυτά πίσω στο σύστημα.

Στην δικιά μας περίπτωση, όπου εφαρμόζουμε τη τεχνική αυτή σε μία βάση δεδομένων, αναφερόμαστε στην απομάκρυνση από τη μνήμη δεδομένων τα οποία είναι πλέον πεπαλαιωμένα. Πρόκειται ουσιαστικά για εκδόσεις δεδομένων οι οποίες δεν είναι προσπελάσιμες από καμία συναλλαγή.

## 5.1 Ανάλυση Προβλήματος

Όπως ήδη έχουμε αναφέρει στο σύστημα μας υλοποιείται η απομόνωση στιγμιότυπου (snapshot isolation) με έλεγχο ταυτοχρονισμού πολλαπλών εκδόσεων (mvcc). Γεγονός που σημαίνει πως πρέπει για κάθε οντότητα να κρατάμε πολλαπλές εκδόσεις. Οι εκδόσεις αυτές οφείλουν να παραμείνουν στη μνήμη όχι μόνο μέχρι μια καινούργια έκδοση να εισαχθεί αλλά έως ότου δεν υπάρχει άλλη εκκρεμής συναλλαγή η οποία να κάνει χρήση παλαιότερων εκδόσεων. Το πλήθος των εκδόσεων μεγαλώνει αναλογικά με τις διάφορες εφαρμογές που εξυπηρετούνται από τη βάση και πολύ γρήγορα την γεμίζουν με άκυρα έγγραφα τα οποία όχι μόνο καταλαμβάνουν πολύτιμο χώρο αποθήκευσης του συστήματος αλλά καθυστερούν σε κάποιο βαθμό και τις βασικές λειτουργίες της βάσης.

Επομένως γίνεται εύκολα εμφανές πως είναι επιτακτική η ανάγκη κατάστρωσης μια στρατηγικής για τον περιορισμό των άχρηστων δεδομένων που υπάρχουν στη βάση.

## 5.2 Σχεδιαστικές Αποφάσεις

Ένας από τους πρωταρχικούς στόχους στη σχεδίαση ενός εργαλείου όπως ο GC (Garbage Collector) είναι να μην επιβαρύνει το σύστημα ή να το επιβαρύνει όσο λιγότερο γίνεται , για να δικαιολογεί πρακτικά και την εφαρμογή του. Αυτό μας οδήγησε στην ανάπτυξη του GC ως μίας standalone εφαρμογής η οποία θα βρίσκεται μαζί με τη βάση δεδομένων(κατά προτίμηση στο ίδιο φυσικό μηχάνημα ή στο ίδιο δίκτυο). Με αυτόν τον τρόπο επιτυγχάνουμε :

- Να μην επιβαρύνουμε τον Transactional Manager με επιπλέον «λογική» προς υλοποίηση.

- Την κατά βούληση παραχώρηση διαθέσιμων πόρων για την εκτέλεση της εφαρμογής.

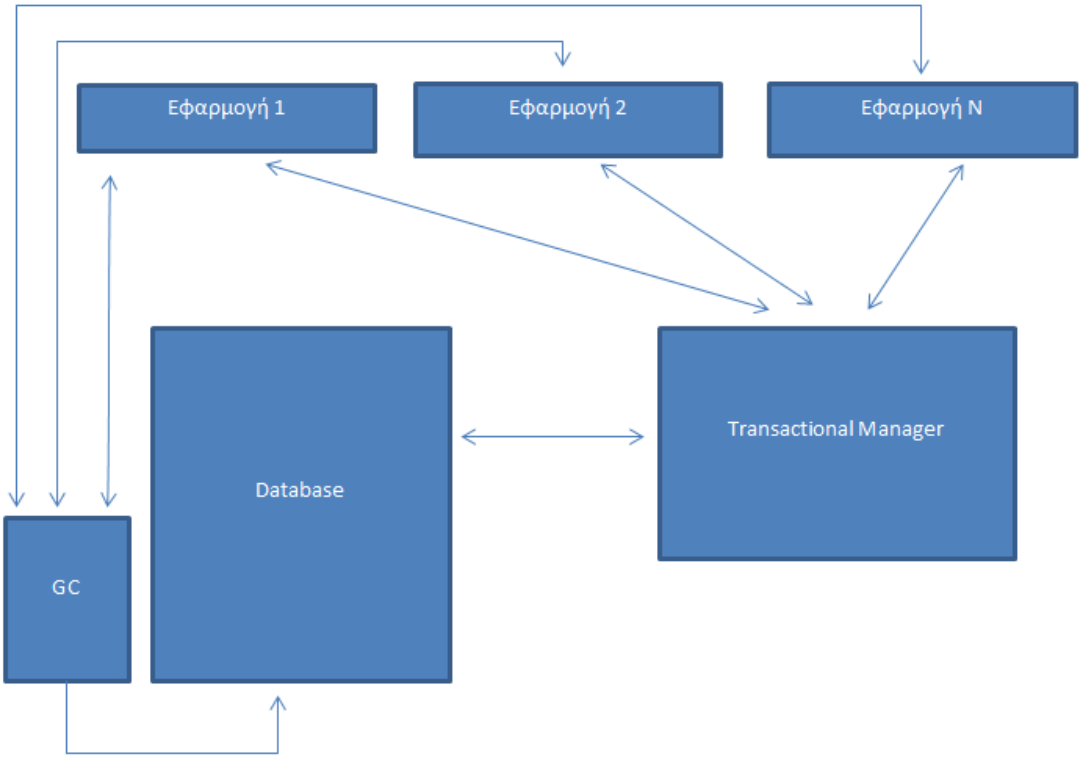
- Την ευκολία στη διαδικασία δοκιμών(testing), αφού προσφέρεται για την εκτέλεση σε junit περιβάλλον.

Μία ακόμα παράμετρος που επηρεάζει εξίσου την απόδοση του συστήματος και του Garbage Collector είναι η συχνότητα εκτέλεσης του. Είναι προφανές πως δεν είναι αποδοτικό τα δεδομένα να διαγράφονται την ίδια στιγμή την οποία καθίστανται απαρχαιωμένα, αφού με αυτόν τον τρόπο θα θυσιάζαμε υπολογιστική ισχύ για τον συνεχή έλεγχο απαρχαιωμένων εκδόσεων καθώς θα είχαμε τόσες κλήσεις του GC όσες και οι διαφορετικές εκδόσεις των δεδομένων. Η αναλογία του χώρου αποθήκευσης που θα εξοικονομήσουμε σε σχέση με το κόστος στην απόδοση του συστήματος που επιφέρει η κλήση του GC, είναι το κριτήριο στην επιλογή της συχνότητας εκτέλεσης του.

Οι δύο στρατηγικές που επιχειρήσαμε να εφαρμόσουμε για να επιτύχουμε το βέλτιστο αποτέλεσμα ήταν , αρχικά η εκτέλεση του Garbage Collector συναρτήσει του πλήθος των δεδομένων της βάσης και εν συνεχεία σε προκαθορισμένα χρονικά διαστήματα.
Στην πρώτη περίπτωση, δοκιμάσαμε να παρακολουθούμε το σύνολο των εφαρμογών-client που αλληλοεπιδρούσαν με τη βάση και ανάλογα με τον αριθμό των εκδόσεων δεδομένων που προκύπταν από τις συναλλαγές προσαρμόζαμε τη συχνότητα εκτέλεσης του Garbage Collector. Έμπνευση αυτής της τεχνικής αποτέλεσε ο Garbage Collector της Java , ο οποίος έχει κοινά χαρακτηριστικά και με αυτόν των άλλων γλωσσών προγραμματισμού, όπου ο JVM αποφασίζει πότε είναι απαραίτητος να κληθεί όταν το σύστημα βρίσκεται σε idle κατάσταση ή όταν υπάρχει ανάγκη για μνήμη. Στο δικό μας σύστημα όμως ήρθαμε στο συμπέρασμα πως η διαδικασία αυτή για την επιλογή του χρόνου εκκίνησης του Garbage Collector είναι πλεονάζουσα αν όχι αρνητικής επίπτωσης για την συνολική επίδοση. Αντίθετα η επιλογή ενός προκαθορισμένου χρονικού διαστήματος, κατά το οποίο θα πραγματοποιείται η συλλογή απορριμμάτων, είναι αρκετή ώστε να μας εξασφαλίσει το επιθυμητό

αποτέλεσμα χωρίς την ανάγκη επιπλέον υπολογισμών από τη πλευρά της εφαρμογής μας. Επιπλέον επιτρέποντας την αντικατάσταση της προκαθορισμένης τιμής του χρονικού διαστήματος αυτού μέσω ενός configuration file, παρέχουμε στο διαχειριστή του συστήματος τη δυνατότητα fine tuning της συμπεριφοράς του Garbage Collector, στο ενδεχόμενο που έχει πληροφορία, εκ των προτέρων, σχετικά με το φόρτο της βάσης δεδομένων.

## 5.3 Υλοποίηση




Εικόνα 13: Απλοποιημένη παρουσίαση του συστήματος

Παραπάνω βλέπουμε ένα απλοποιημένο σχήμα του συστήματος, στο οποίο παρουσιάζουμε τα κύρια κομμάτια και βήματα της συλλογής απορριμμάτων. Αρχικά η εφαρμογή , κάνοντας πάντα χρήση του mongoClient API, αφού στείλει αίτημα εγγραφής στον Transcactional Manager πληροφορείται για το IP και την πόρτα της standalone εφαρμογής υπεύθυνη για τη συλλογή απορριμμάτων. Εν συνεχεία ο Transactional Manager είναι υπεύθυνος να ενημερώσει τις εφαρμογές, μιλώντας στον κάθε LTM client ξεχωριστά, με τη τελευταία «έγκυρη χρονοσφραγίδα»(valid timestamp). Με τον όρο αυτό αναφερόμαστε στη χρονοσφραγίδα που έχει η πιο πρόσφατη ολοκληρωμένη συναλλαγή. Εφόσον η ζωντανή ενημέρωση των επί μέρους client με την τελευταία χρονοσφραγίδα, όπως υπογραμμίσαμε και στο

κεφάλαιο σχεδιαστικών αποφάσεων, κρίθηκε μη απαραίτητη αλλά σε αρκετές περιπτώσεις και μη αποδοτική, υλοποιήσαμε ένα schedule το οποίο θα καλεί την notifyClient (InetAddress address, Integer port) για την αποστολή της χρονοσφραγίδας στους εγγεγραμμένους (registered) clients. Έπειτα ο LTM client αναλαμβάνει να μεταφέρει τη χρονοσφραγίδα αυτή στην εφαρμογή Garbage Collector καλώντας την μέθοδο garbageCollectMongo (long). Τέλος η εφαρμογή συλλογής απορριμμάτων εξυπηρετεί κάθε εφαρμογή ξεχωριστά και παράλληλα, δημιουργώντας ένα νήμα για κάθε κλήση. Όπως θα περιγράψουμε αναλυτικότερα και στο επόμενο κεφάλαιο, το ScheduledExecutorService είναι υπεύθυνο για τον καθαρισμό, σε συγκεκριμένα χρονικά διαστήματα, των απαρχαιωμένων εγγραφών στη βάση δεδομένων, σύμφωνα με την χρονοσφραγίδα που έχει μεταδώσει η κάθε εφαρμογή.

**ScheduledExecutorService**

Για να υλοποιήσουμε την περιοδική εκτέλεση της συλλογής απορριμμάτων , πιο συγκεκριμένα του ερωτήματος στη βάση το οποίο είναι υπεύθυνο για το σβήσιμο των απαρχαιωμένων τιμών, χρησιμοποιήσαμε την διεπαφή **ScheduledExecutorService** της Java [9].

Με την μέθοδο *Schedule* μπορούμε να δημιουργήσουμε εργασίες (tasks ) με διαφορετικές χρονικές καθυστερήσεις και μας επιστρέφει ένα αντικείμενο το οποίο χρησιμοποιούμε για να ακυρώσουμε ή να ελέγξουμε την εκτέλεση. Εμείς διαλέξαμε να χρησιμοποιήσουμε την **scheduleAtFixedRate** με την οποία δημιουργήσαμε την εργασία (task) της συλλογής απορριμμάτων και η οποία θα εκτελείται μέχρι να ακυρωθεί (Στη δική μας υλοποίηση μόνο σε περίπτωση τερματισμού της εφαρμογής).

```
scheduler.scheduleAtFixedRate(garbageCollectorTask, 0,
SCHEDULED_INTERVAL,
                    TimeUnit.SECONDS);
```

Όπως έχουμε ήδη αναφέρει το χρονικό διάστημα , ***SCHEDULED_INTERVAL*** , έχει κάποια προκαθορισμένη τιμή η οποία όμως μπορεί να αντικατασταθεί από το χρήστη μέσω ενός configuration αρχείου. Δεν υπάρχει όμως η δυνατότητα αλλαγής (hot swapping) εφόσον έχει γίνει εκκίνηση της εφαρμογής.

Στο σημείο αυτό να αναφέρουμε ότι επιλέξαμε να κατασκευάσουμε μία «δεξαμενή νημάτων» μεγέθους 1 εφόσον στις δοκιμές που κάναμε η συχνότητα εκτέλεσης του task δεν δικαιολογούσε τη χρήση παραπάνω πυρήνων αφού στο διάστημα που μεσολαβούσε, για να τρέξει η επόμενη εκτέλεση του task, είχε ολοκληρωθεί το προηγούμενο.

```
final ScheduledExecutorService scheduler =
        Executors.newScheduledThreadPool(1);
```

## ShutDownHook

Είναι προφανές πως όταν μία μονονηματική(single threaded) εφαρμογή τερματίζει εξαιτίας κάποιας εξαίρεσης(exception) που δεν έπιασε κανένας χειριστής (handler) το πρόγραμμα σταματά και παράγει ένα ίχνος στοίβας, το οποίο είναι πολύ διαφορετικό από το τυπική και προβλεπόμενη έξοδο του προγράμματος. Σε περίπτωση όμως αποτυχίας ενός νήματος σε μία «παράλληλη» (concurrent) εφαρμογή ούτε η παρατήρηση και αναγνώριση της αποτυχίας είναι προφανής αλλά ούτε και το αποτέλεσμα. Το ίχνος μπορεί να εμφανιστεί στη κονσόλα , αλλά κανείς να μην την κοιτάζει, ενώ όταν ένα νήμα αποτύχει η εφαρμογή μπορεί να φαίνεται ότι συνεχίζει να λειτουργεί κανονικά κάνοντας το χρήστη να παραβλέψει την αποτυχία. Για το σκοπό αυτό η Java έχει φροντίσει την υλοποίηση κατάλληλων μηχανισμών και εργαλείων που επιτρέπουν τόσο τον εντοπισμό όσο και την πρόληψη νημάτων που «διαρρέουν» (thread leak) από μία εφαρμογή. Ένα από αυτά είναι και το ShutDownHook.

Η εικονική μηχανή της Java ( Java Virtual Machine) μπορεί να τερματίσει είτε ομαλά είτε «βίαια». Ένας ομαλός τερματισμός ξεκινά όταν και το τελευταίο (μη-δαίμονας) νήμα τερματίζει , κάποιος καλεί System.exe ή με κάποιο άλλο ειδικό μέσο που παρέχει η πλατφόρμα, όπως το σήμα SIGINT ή ο συνδυασμός πλήκτρων Ctrl-C. Ενώ αυτός είναι ο κανονικός και προτεινόμενος τρόπος τερματισμού του JVM μπορεί εναλλακτικά να τερματίσει βίαια καλώντας Runtime.halt ή «σκοτώνοντας» την διεργασία JVM μέσω του λειτουργικού συστήματος.

Σε ένα κανονικό τερματισμό λειτουργίας, το JVM εκτελεί όλα τα shutdown hooks που έχουν εγγραφεί σε αυτό. Πρόκειται ουσιαστικά για νήματα τα οποία έχουν προστεθεί με την εντολή Runtime.addShutdownHook. Σύμφωνα με τη βιβλιογραφία, το JVM δεν εγγυάται τη σειρά με την οποία θα ξεκινήσουν, ενώ αν άλλα νήματα της εφαρμογής είναι ενεργά κατά τον τερματισμό, συνεχίζουν να τρέχουν παράλληλα με τη διαδικασία τερματισμού. Σε περίπτωση που κάποιο shutdown hook δεν ολοκληρωθεί, τότε η διαδικασία τερματισμού μπλοκάρει και ο JVM πρέπει να τερματιστεί «βίαια». Να τονίσουμε πως σε έναν «βίαιο» τερματισμό του JVM τα shutdown hooks δεν εκτελούνται. Εμείς φροντίσαμε ώστε η standalone εφαρμογή του garbage collector να εγκαθιδρύσει ένα shutdown hook με το οποία σε περίπτωση εξόδου θα αποδέσμευση όλους τους πόρους του συστήματος που χρησιμοποιεί ενώ θα τερματίσει όλες τις ενεργές και εν αναμονή συνδέσεις προς τη βάση δεδομένων επιτρέποντας την εύκολη εκτέλεση ενός άλλου στιγμιότυπου της αμέσως μετά τον τερματισμό της.

### MongoDB API

Για την επικοινωνία της εφαρμογής με τη βάση δεδομένων χρησιμοποιήσαμε το κατάλληλο API που αντιστοιχεί στη Java και μας παρέχει η ίδια η Mongo [10]. Η εγκατάσταση του είναι τόσο απλή όσο η συμπλήρωση μερικών γραμμών στο pom

αρχείο του Maven Project. Ο driver που παρέχει η Mongo μας επιτρέπει την εκτέλεση όλων των λειτουργειών διαθέσιμων στη βάση αφού πρώτα εγκαθιδρύσουμε σύνδεση σε αυτήν, πολύ εύκολα με τον παρακάτω τρόπο :

```
MongoClient = new MongoClient( "localhost" , 27017 );
```

Η εντολή αυτή μας γυρνάει ένα αντικείμενο με τη βοήθεια του οποίου εκτελούμε όλες τις λειτουργείες που επιθυμούμε στη βάση δεδομένων.

### 5.3.1 Design Patterns

Τα Design Patterns πρόκειται για καλά ορισμένες λύσεις σε κοινά προγραμματιστικά προβλήματα [11]. Είναι πολύ δημοφιλή μεταξύ των προγραμματιστών, ιδιαίτερα αυτών που χρησιμοποιούν αντικειμενοστραφής γλώσσες όπως είναι η Java. Μερικά από τα πλεονεκτήματα που προσφέρουν είναι :

- Η χρήση τους ενισχύει την επαναχρησιμοποίηση κώδικα η οποία οδηγεί σε πιο άρτιο και εύκολα συντηρήσιμο κώδικα. Με τον τρόπο αυτό συμβάλει στη μείωση του συνολικού κόστους ιδιοκτησίας (TCO – total cost of ownership) του λογισμικού.

- Εφόσον τα design pattern είναι ήδη ορισμένα και παρέχουν μία προσέγγιση που υπαγορεύεται από τη βιομηχανία για τη λύση επαναλαμβανόμενων προβλημάτων, εξοικονομούμε πολύ χρόνο χρησιμοποιώντας μηχανικά κάποιο από τα patterns, αυτό που ταιριάζει κάθε φορά.

- Για τον ίδιο λόγο, το ότι είναι ήδη ορισμένα τα design patterns, είναι πολύ εύκολο να κατανοήσουμε και να πραγματοποιήσουμε debug στον κώδικα. Αυτό συνεπάγεται και γρηγορότερη ανάπτυξη λογισμικού ενώ παρέχει ένα κοινό λεξιλόγιο στους προγραμματιστές το οποίο μπορούν να χρησιμοποιούν για να επικοινωνήσουν ευκολότερα σχετικά με τα προγραμματιστικά προβλήματα και τις λύσεις τους.

#### Singleton Pattern

Το Singleton Pattern εξασφαλίζει πως μία κλάση έχει μόνο μία *εκδοχή* της (instance), δηλαδή ανά πάσα στιγμή υπάρχει μόνο ένα αντικείμενο της κλάσης αυτής στον JVM και παρέχεται πρόσβαση σε αυτό από οποιοδήποτε σημείο του κώδικα.

Το Pattern αυτό το χρησιμοποιήσαμε, όπως φαίνεται και στο κομμάτι κώδικα παρακάτω, για να διαχειριστούμε τη σύνδεση που πραγματοποιούμε με την βάση δεδομένων Mongo. Με τον τρόπο αυτό εξασφαλίζουμε ότι θα υπάρχει μόνο μία σύνδεση ανά πάσα στιγμή και η χρήση αυτής θα γίνεται με τη βοήθεια του μοναδικού αυτού αντικειμένου που δημιουργούμε.

```
private static GarbageCollectorMain uniqueInstance;
static {
    try {
         uniqueInstance = new GarbageCollectorMain();
    } catch (UnknownHostException e) {
        throw new ExceptionInInitializerError(e);
    }
}
private GarbageCollectorMain() throws UnknownHostException {
    MongoClient mongo = new MongoClient("localhost", 27017);
    DB db = mongo.getDB(DATABASE_NAME);
    collection = db.getCollection(COLLECTION_NAME);
}

public static GarbageCollectorMain getInstance() {
    return uniqueInstance;
}
```

Η ανατομία της singleton κλάσης είναι πολύ απλό να τη κατανοήσουμε. Η κλάση έχει στη κλασσική της περίπτωση έναν private constructor ο οποίος με αυτό το τρόπο απαγορεύει στον προγραμματιστή να φτιάξει *εκδοχές* (instances) της singleton κλάσης. Αντίθετα παρέχει πρόσβαση σε μία static συνάρτηση, τύπου ίδιου με αυτού της singleton κλάσης, η οποία δίνει πρόσβαση στο *δείκτη* (reference) του ήδη κατασκευασμένου μοναδικού αντικειμένου. Οι μέθοδοι αυτοί εξασφαλίζουν πως δεν θα υπάρξει παρά μόνο μία έκδοση της singleton κλάσης καθ' όλη τη διάρκεια ζωής της εφαρμογής.

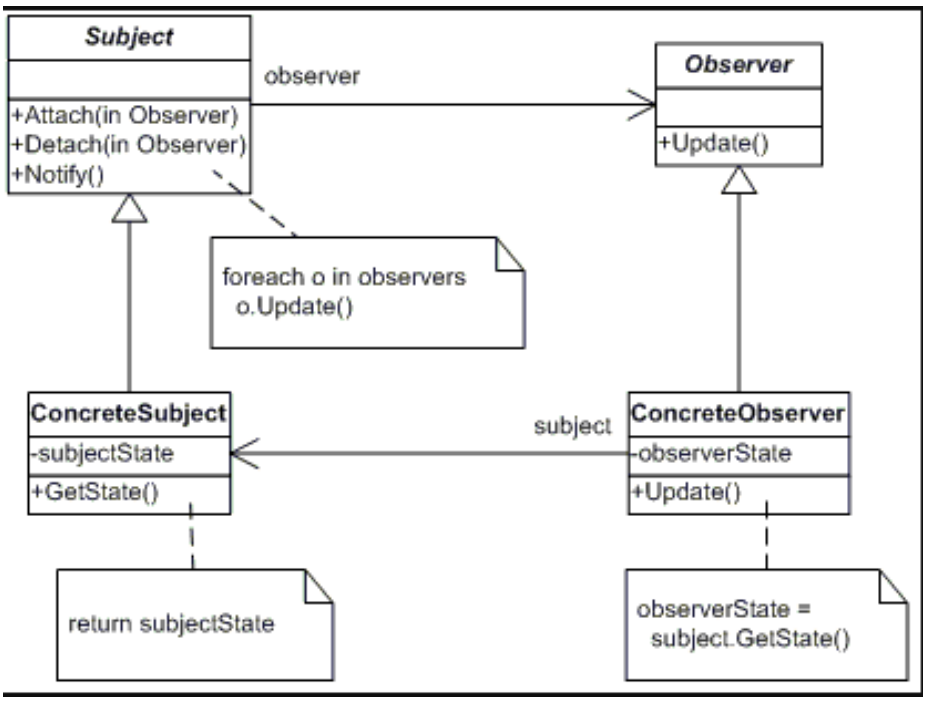
Για την αρχικοποίηση του singleton αντικειμένου προτιμήσαμε static initialization έναντι του lazy initialization, καθώς όχι μόνο προτιμούσαμε το overhead της δημιουργίας του αντικειμένου να τεθεί στην εκκίνηση της εφαρμογής ώστε να μην επηρεαστεί η επίδοση όταν κληθεί για πρώτη φορά η διαδικασία συλλογής απορριμμάτων, αλλά κυρίως διότι θέλαμε να πάρουμε τις απαραίτητες προφυλάξεις για την ασφάλεια στην ταυτόχρονη εκτέλεση πολλαπλών νημάτων (thread safety).

Επιπροσθέτως, αξίζει να τονίσουμε τη χρήση του static initializer για να «πιάσουμε» το UnknownHostException το όποιο «ρίχνει» ο constructor και στη συνέχεια να «ρίξουμε» με τη σειρά μας το ExceptionInInitializerError το οποίο υποδηλώνει πρόβλημα στην διαδικασία αρχικοποίησης. Σε περίπτωση που ο constructor της singleton κλάσης μας δεν «έριχνε» κάποια *εξαίρεση* (exception) θα μπορούσαμε απλά να χρησιμοποιούσαμε μία static μεταβλητή για αρχικοποίηση του αντικειμένου.

**Observer Pattern**

Το Observer Pattern ορίζει μία σχέση εξάρτησης «μία προς πολλά» μεταξύ αντικειμένων έτσι ώστε όταν ένα αντικείμενο αλλάζει κατάσταση, όλα τα αντικείμενα που εξαρτώνται από αυτό να ειδοποιηθούν και να ενημερωθούν αυτόματα. Το αντικείμενο το οποίο παρακολουθεί τη κατάσταση ενός άλλου αντικειμένου ονομάζεται «παρατηρητής» (Observer) και το αντικείμενο που παρακολουθείται ονομάζεται «υποκείμενο» (Subject) . Το διάγραμμα κλάσης του Pattern αυτού καθιστά την κατανόηση του αρκετά ευκολότερη

.




Εικόνα 14: Observer Pattern

Το *Subject* περιέχει μία λίστα με τους *Observers* που ειδοποιεί όταν συμβεί κάποια αλλαγή στην κατάστασή (state) του. Επομένως παρέχει μεθόδους με τις οποίες οι Observers μπορούν να εγγραφούν και να διαγραφούν κατά βούληση από τη λίστα αυτή. Το *Subject* περιέχει ακόμα μία μέθοδο για να ειδοποιήσει όλους τους *Observers* όταν συμβεί κάποια αλλαγή και μπορεί είτε να στείλει την ενημέρωση ενώ ειδοποιεί τους *Observers* είτε να προσφέρει μία άλλη μέθοδο για να πάρει την ενημέρωση.

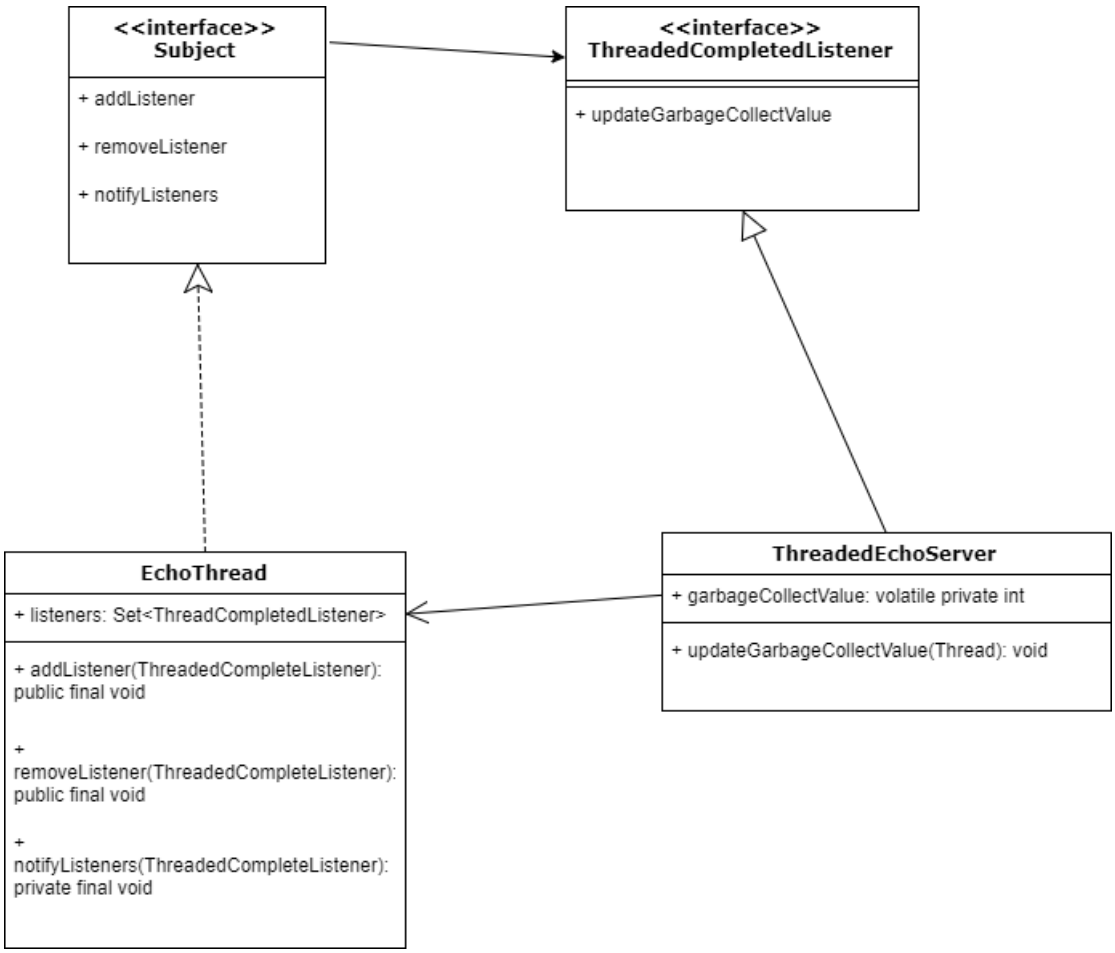
Οι *Observer* οφείλουν να διαθέτουν μία μέθοδο για να ορίσουν το αντικείμενο που θα παρακολουθούν και άλλη μία η οποία θα χρησιμοποιείται από το *Subject* για να τους ειδοποιήσει για οποιοδήποτε ενημερώσεις υπάρχουν.

Εμείς κάναμε χρήση μιας ειδικής, εκφυλισμένης μορφής, του Observer Pattern όπου έχουμε πολλά Subjects αλλά ένα μόνο Observer. Πιο συγκεκριμένα τα threads (**EchoThread.java**) τα οποία δημιουργούνται για να εξυπηρετήσουν την εκάστοτε κλήση για συλλογή απορριμμάτων (garbage collect) αποτελούν τα Subjects. Τα subject αυτά κατά την εκκίνηση της τους προσθέτουν ως Observer το κύριο αντικείμενο μέσα στο οποίο και αρχικοποιούνται τα νήματα αυτά (**ThreadedEchoServer**) έτσι ώστε να ενημερωθεί κατάλληλα η τιμή της μεταβλητής **garbageCollectValue** η οποία υποδηλώνει και τη μέγιστη τιμή με την οποία έχει πραγματοποιηθεί συλλογή απορριμμάτων. Η ενημέρωση αυτή γίνεται με την μέθοδο notifyOfThreadCompleted.

```
    public void notifyOfThreadCompleted(Thread thread) {
    int temp = ((EchoThread) thread).getValue();
    synchronized (this) {
       if (temp > garbageCollectValue) {
          garbageCollectValue = temp;
       }
    }
}
```

Η μέθοδος αυτή, ακολουθώντας τις προδιαγραφές του design pattern, ορίζεται στο interface **ThreadCompletedListener** το οποία και κάνει *implement* η κλάση μας **ThreadedEchoServer.** Άξιο αναφοράς είναι η χρήση του keyword *synchronized* της Java στη διάρκεια προσπέλασης της static μεταβλητής **garbageCollectValue.** Αυτό γίνεται φυσικά διότι περισσότερα του ενός thread μπορούν να εκτελούν παράλληλα τη συνάρτηση αυτή αλλάζοντας ταυτόχρονα την τιμή της μεταβλητής επομένως με το τρόπο αυτό επιτυγχάνουμε ουσιαστικά την ατομικότητα (atomicity) της εκτέλεσης της if εντολής μαζί με την ανάθεση στην μεταβλητή. Να υπογραμμίσουμε πως διαλέγουμε να συγχρονίσουμε στο lock του αντικειμένου **ThreadedEchoServer** (this) αφού ουσιαστικά πρόκειται για ένα μόνο αντικείμενο το οποίο είναι αυτό στο οποίο έχουμε ορίσει την μεταβλητή **garbageCollectValue**.

Για άλλη μια φορά το διάγραμμα κλάσεων θα μας επιτρέψει να κάνουμε πιο κατανοητή την αρχιτεκτονική του συστήματος μας.




Εικόνα 15: Υλοποίηση του Observer Pattern στον συλλέκτη απορριμμάτων

## 5.4 Εκτέλεση

Στο σημείο αυτό κρίναμε δόκιμο να παρουσιάσουμε τα αποτελέσματα εκτέλεσης του συλλέκτη απορριμμάτων σε ένα παράδειγμα , όσο πιο απλό γίνεται, για τη καλύτερη κατανόηση του τρόπου και του σκοπού λειτουργείας του μέσα στο ευρύτερο σύστημα. Το υπό εξέταση σύστημα απαρτίζεται από δύο απλοϊκές εφαρμογές χρήστη οι οποίες πραγματοποιούν μία σειρά από ποικίλα ερωτήματα στη βάση ώστε να παράγουμε διάφορα test cases με όσους περισσότερους τύπους συναλλαγών μπορούμε. Όπως ήδη έχουμε περιγράψει στα προηγούμενα κεφάλαια οι συναλλαγές αυτές διαχειρίζονται όλες από τον transactional manager, ο οποίος είναι υπεύθυνος για την εισαγωγή και εξαγωγή αντίστοιχα των δεδομένων από την βάση αλλά και για την ενημέρωση των εφαρμογών χρήστη με τη τελευταία έγκυρη χρονοσφραγίδα. Την χρονοσφραγίδα αυτή αναλαμβάνει ο LTM Client από την πλευρά του να μεταφέρει στη standalone εφαρμογή συλλογής απορριμμάτων η οποία οφείλει να έχει εκκινήσει παράλληλα , ή και νωρίτερα από τις εφαρμογές χρήστη.

Αμέσως μετά την εκτέλεση των απλοϊκών συναλλαγών, εξετάζοντας τα περιεχόμενα της βάσης Mongo βλέπουμε τα παρακάτω δεδομένα:


```
[…]
    {
      "_id":{ […] },
      "user":18,
      "value":"value18",
      "_dataID":{
         "$oid":"596b9c70ea3014117096fc5a"},
      "_tid":{
        "$numberLong":"18"
      },
      "_cmtTmstmp":{
        "$numberLong":"86"
      },
      "_nextCcmtTmstmp":"87"
    },
    {
      "_id":{ […] },
      "user":19,
      "value":"value19",
      "_dataID":{
         "$oid":"596b9f0ad4dad22a9c8e846b"},
      "_tid":{
        "$numberLong":"19"
      },
      "_cmtTmstmp":{
        "$numberLong":"87"
      },
      "_nextCcmtTmstmp":"88"
    },
    {
      "_id":{ […] },
      "user":20,
      "value":"value20",
      "_dataID":{
          "$oid":"596b9f0ad4dad22a9c8e8477"},
      "_tid":{
        "$numberLong":"20"
      },
      "_cmtTmstmp":{
        "$numberLong":"88"
      },
      "_nextCcmtTmstmp":"89"
    }
[…]
```

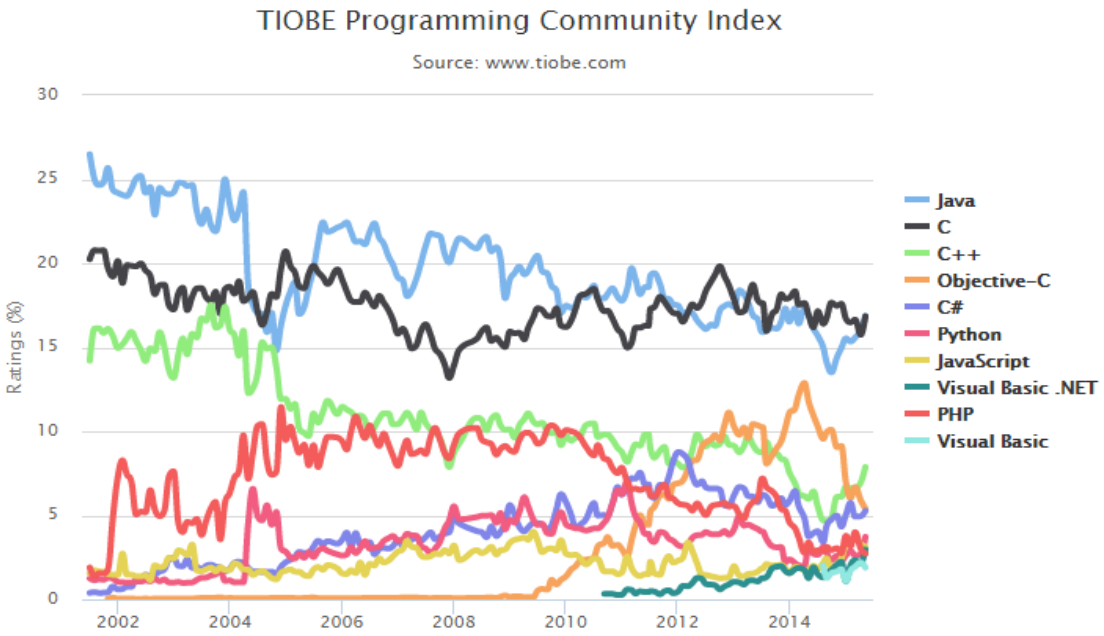
Στη συνέχεια, αφού περάσει το χρονικό διάστημα που έχουμε ορίσει, πραγματοποιείται η διαδικασία συλλογής απορριμμάτων. Επομένως έχουμε την διαγραφή των μή έγκυρων δεδομένων από τη βάση δεδομένων. Οι εγγραφές οι οποίες θα απομακρυνθούν από την βάση, καθορίζονται όπως έχουμε αναφέρει και προηγουμένως, από την τιμή που παρέχει ο Transactional Manager στην εφαρμογή χρήστη η οποία με τη σειρά της την μεταφέρει στην standalone εφαρμογή μας. Με την τιμή αυτή εκτελούμε στη βάση δεδομένων το κατάλληλο ερώτημα για την διαγραφή δεδομένων με tid χαμηλότερο από αυτήν. Στην προκειμένη περίπτωση η τιμή με την οποία εκτελέστηκε η συλλογή απορριμμάτων ήταν 87 και η εικόνα της βάσης έπειτα από την εκτέλεση του ερωτήματος αυτού είναι η παρακάτω:


```
{
    "_id":{ […] },
    "user":19,
    "value":"value19",
    "_dataID":{
       "$oid":"596b9f0ad4dad22a9c8e846b"},
    "_tid":{
      "$numberLong":"19"
    },
    "_cmtTmstmp":{
      "$numberLong":"87"
    },
    "_nextCcmtTmstmp":"88"
  },
  {
    "_id":{ […] },
    "user":20,
    "value":"value20",
    "_dataID":{
       "$oid":"596b9f0ad4dad22a9c8e8477"},
    "_tid":{
      "$numberLong":"20"
    },
    "_cmtTmstmp":{
      "$numberLong":"88"
    },
    "_nextCcmtTmstmp":"89"
  }
[…]
```

# ΚΕΦΑΛΑΙΟ 6
# Προγραμματιστικά Εργαλεία

## 6.1 Η γλώσσα προγραμματισμού Java

Η Java είναι μια γενικής χρήσης γλώσσα προγραμματισμού , αντικειμενοστραφής, βασισμένη σε κλάσεις, κατάλληλη για παράλληλο προγραμματισμό και ειδικά σχεδιασμένα ώστε να έχει τις λιγότερο δυνατό εξαρτήσεις εφαρμογής [12]. Η κύρια αρχής της, η οποία αποτελεί και το σήμα κατατεθέν της, είναι να επιτρέπει στους προγραμματιστές να «γράφουν μία φορά, και να εκτελείται παντού» (write once, run everywhere ). Πιο συγκεκριμένα ο κώδικας σε Java εφόσον έχει μεταγλωττιστεί μπορεί να εκτελεστεί σε οποιαδήποτε πλατφόρμα που υποστηρίζει Java χωρίς την ανάγκη επαναμεταγγλώτισης. Οι εφαρμογές σε Java μεταγλωττίζονται συνήθως σε bytecode το οποίο μπορεί να εκτελεστεί σε οποιοδήποτε JVM(εικονική μηχανή της Java) ανεξάρτητα από την αρχιτεκτονική του εκάστοτε επεξεργαστή. Η γλώσσα αυτή αναπτύχθηκε από τον James Gosling στην Sun Microsystems το 1995 και έχει δανειστεί αρκετά στοιχεία όσον αφορά τη σύνταξη της από την C και C++, εξαιρουμένων των λειτουργιών «χαμηλότερου επιπέδου» ( low level facilities).




Εικόνα 16: Δημοτικότητα των γλωσσών προγραμματισμού

Η Java αποτελεί μία από τις πιο δημοφιλείς γλώσσες προγραμματισμού παγκοσμίως καθώς συναγωνίζεται καθ' όλη τη διάρκεια της τελευταίας δεκαετίας για τη πρώτη θέση μαζί με την C. Κύριος παράγοντας που την έχει θέσει πρώτη στις προτιμήσεις των προγραμματιστών είναι το γεγονός ότι οι σχεδιαστικοί στόχοι της γλώσσας βρίσκονται σε ευθυγράμμιση με την φύση και τις ανάγκες των υπολογιστικών συστημάτων στα οποία το λογισμικό εγκαθίσταται . Η μαζική ανάπτυξη του διαδικτύου οδήγησε σε έναν καινούργιο τρόπο ανάπτυξης και διάδοσης του λογισμικού. Επιπροσθέτως η λειτουργία πολλαπλών πλατφόρμων σε ετερογενή δίκτυα ανέτρεψε τις παραδοσιακές μεθόδους διανομής , έκδοσης , αναβάθμισης και ενημέρωσης του εκτελέσιμου αρχείου (binary file). Τέλος ο κατακλυσμός της αγοράς με πολυπύρηνους επεξεργαστές κατέστησε ως απαραίτητη προϋπόθεση των εφαρμογών την αποδοτική αξιοποίηση τους. Όλα τα παραπάνω συντέλεσαν στην διαμόρφωση των στόχων και συνάμα χαρακτηριστικών της Java, τα κυριότερα από τα οποία είναι:

- Απλή, Αντικειμενοστραφής και οικεία

  Η Java είναι μία απλή γλώσσα η οποία μπορεί να χρησιμοποιηθεί χωρίς εκτενή προγραμματιστική εκπαίδευση ενώ τηρεί τις υπάρχουσες πρακτικές λογισμικού. Οι θεμελιώδης έννοιες της γλώσσας γίνονται γρήγορα κατανοητές και οι προγραμματιστές είναι παραγωγικοί από την πολύ αρχή. Ακόμα υπάρχει πρόσβαση σε υπάρχουσες βιβλιοθήκες δοκιμασμένων αντικειμένων που παρέχουν λειτουργικότητα από απλά δομές δεδομένων I/O και διεπαφές δικτύου μέχρι εργαλειοθήκες για γραφικές διεπαφές.

- Αξιόπιστη και ασφαλής

  Παρέχει εκτενή έλεγχο κατά τη διάρκεια μεταγλώττισης. Ακολουθούμενο από ένα δεύτερο επίπεδο ελέγχου κατά τη διάρκεια εκτέλεσης. Το μοντέλο διαχείριση μνήμης είναι εξαιρετικά απλό, δεν υπάρχουν περίπλοκες αριθμητικές πράξεις με δείκτες, τα αντικείμενα δημιουργούνται με τον τελεστή new και η συλλογή απορριμμάτων γίνεται αυτόματα. Επιπλέον είναι σχεδιασμένη ώστε να λειτουργεί σε κατανεμημένα περιβάλλοντα οπότε η ασφάλεια είναι υψίστης σημασίας.

- Αρχιτεκτονικά Ουδέτερη και Φορητή

  Για να εξυπηρετήσει την ποικιλία των λειτουργικών συστημάτων, ο μεταγλωττιστής της JAVA παράγει bytecodes, μια αρχιτεκτονικά ουδέτερη μορφή για την αποτελεσματική μεταφορά κώδικα σε διάφορες πλατφόρμες. Η αρχιτεκτονικά αυτή ουδέτερη πλατφόρμα της Java που προσφέρει και τη φορητότητα της γλώσσας είναι γνωστή ως εικονική μηχανή της Java (Java virtual machine).

- Υψηλής Απόδοσης

  Η απόδοση είναι πάντα ένας σημαντικός παράγοντας. Η πλατφόρμα της Java επιτυγχάνει ανώτερη επίδοση υιοθετώντας μια δομή στην οποία ο διερμηνέας (interpreter) μπορεί να τρέχει σε πλήρη ταχύτητα χωρίς την ανάγκη ελέγχου του περιβάλλοντος εκτέλεσης. Ο αυτόματος συλλέκτης απορριμμάτων τρέχει σε χαμηλής προτεραιότητας νήμα, βεβαιώνοντας με μεγάλη πιθανότητα ότι η απαραίτητη μνήμη είναι διαθέσιμη. Οι εφαρμογές που απαιτούν μεγάλη ποσότητα υπολογιστικής ισχύς μπορούν να σχεδιαστούν με τέτοιο τρόπο ώστε τα κρίσιμα κομμάτια να ξαναγραφτούν σε native κώδικα μηχανής σύμφωνα με τα πρότυπα της πλατφόρμας.

- Παραλληλοποιήσιμη, πολυνηματική και δυναμική

  Η πλατφόρμα της Java υποστηρίζει πολυνηματισμό στο επίπεδο της γλώσσας με την προσθήκη ενός εκλεπτυσμένου συνόλου μεταβλητών συγχρονισμού, βιβλιοθηκών (όπως η κλάση Thread) και ενός συστήματος εκτέλεσης που παρέχει monitor και condition locks. Ενώ ο μεταγλωττιστής της Java είναι αυστηρός κατά τη διάρκεια του στατικού ελέγχου, η γλώσσα και το περιβάλλον εκτέλεσης είναι δυναμικά στο στάδιο σύνδεσης. Οι κλάσεις συνδέονται μόνο όταν χρειάζεται και οι επιμέρους ομάδες κώδικα μπορούν να συνδεθούν κατά βούληση από μία ποικιλία πηγών, ακόμα και από πηγές που βρίσκονται κάπου απομακρυσμένες στο ίδιο δίκτυο.

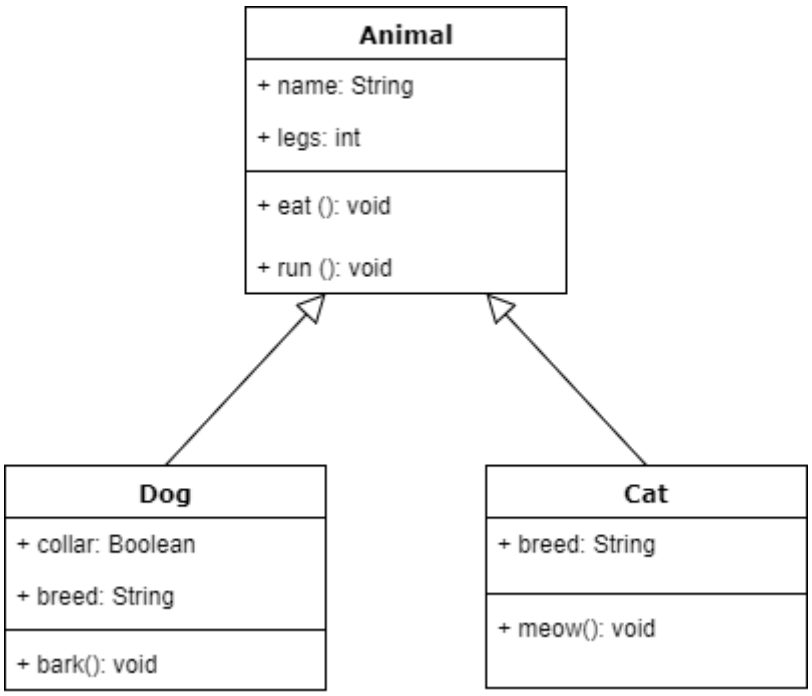
### 6.1.1 Χαρακτηριστικά της γλώσσας Java

Στο σημείο αυτό αξίζει να αναφέρουμε μερικά από τα σημαντικά χαρακτηριστικά υλοποίησης της γλώσσας Java [13][14].

**Κληρονομικότητα**

Η κληρονομικότητα είναι ένα από τα χαρακτηριστικά του αντικειμενοστραφή προγραμματισμού. Η κληρονομικότητα επιτρέπει μία κλάση να χρησιμοποιήσει τις ιδιότητες και τις μεθόδους μίας άλλης κλάσης. Με άλλα λόγια, η παραγόμενη κλάση κληρονομεί τις μεταβλητές (state) και τη συμπεριφορά της από την γονική κλάση (base class). Η παραγόμενη κλάση είναι γνωστή και ως «υποκλάση» (subclass) ενώ η γονική κλάση ονομάζεται και «υπερκλάση» (super-class). Η παραγόμενη κλάση μπορεί να προσθέσει επιπλέον μεταβλητές και μεθόδους ή ακόμα να τροποποιήσει την συμπεριφορά και λειτουργεία των ήδη υπαρχόντων. Αυτές οι προσθέσεις και τροποποιήσεις είναι αυτές που ουσιαστικά διαφοροποιούν την παραγόμενη κλάση από την γονική.

Η κληρονομικότητα είναι ένας μηχανισμός που υλοποιείται κατά την μεταγλώττιση. Μία κλάση μπορεί να έχει οποιοδήποτε αριθμό υποκλάσεων αλλά μία υποκλάση μπορεί να έχει μόνο μία γονική κλάση. Αυτό συμβαίνει διότι η Java, σε αντίθεση με τη C++, δεν υποστηρίζει πολλαπλή κληρονομικότητα. Η υπερκλάση με την υποκλάση έχουν μία “is-a” σχέση μεταξύ τους ενώ παρακάτω βλέπουμε και την απεικόνιση της σχέση αυτής σε ένα πολύ απλό παράδειγμα.




Εικόνα 17: Παράδειγμα κληρονομικότητας

**Animal.java**

```
public abstract Animal
{
	int name;
	int legs;
	Boolean tail;
	public void run()
	{
		System.out.println(name + “ is running”);
	}
	public abstract void eat ();
}
```

**Dog.java**

```
public Dog extends Animal
{
	public void eat()
	{
		System.out.println(“Eating bones”);
	}
	public void bark()
	{
		System.out.println(“woof woof”);
	}
}
```

**Cat.java**

```
public  Cat extends Animal
{
	public void eat()
	{
		System.out.println(“Eating fish”);
	}
	public void meow()
	{
		System.out.println(“Meow”);
	}
}
```

Στο παραπάνω παράδειγμα βλέπουμε πως η κληρονομικότητα επιτυγχάνεται με τη χρήση της λέξης κλειδί **extends** και πιο συγκεκριμένα έχουμε τις δύο υποκλάσεις Cat και Dog να επεκτείνουν την κλάση Animal. Βλέπουμε ακόμα την χρήση της λέξης κλειδί **abstract** με την οποία έχουμε τη δυνατότητα να δηλώνουμε μεθόδους οι οποίες δεν περιέχουν κάποια υλοποίηση. Εφόσον μία μέθοδος είναι δηλωμένη ως abstract συνεπάγεται αυτομάτως πως και η κλάση θα πρέπει να είναι abstract. Όπως είναι λογικό μία abstract κλάση δεν μπορεί να αρχικοποιηθεί (instantiate) ενώ υποχρεώνει ουσιαστικά τις υποκλάσεις της να παρέχουν την κατάλληλη υλοποίηση των abstract μεθόδων της.

**Αυτόματη διαχείριση μνήμης**

Όπως έχουμε ήδη αναφέρει η Java είναι εξοπλισμένη με έναν αυτόματο συλλέκτη απορριμμάτων, ο οποίος διαχειρίζεται την μνήμη κατά τη διάρκεια ζωής ενός αντικειμένου. Ενώ ο χρήστης είναι υπεύθυνος για την δημιουργία και τη συμπεριφορά των αντικειμένων, η εικονική μηχανή της Java (JVM) είναι εκείνη που

καθορίζει πότε τα αντικείμενα αυτά θα αποδεσμεύσουν τη μνήμη του συστήματος, όταν φυσικά αυτά δεν είναι πλέον σε χρήση. Η ανάκτηση της μνήμης πραγματοποιείται , όπως είδη έχουμε αναφέρει, από το συλλέκτη απορριμμάτων όταν δεν υπάρχουν πλέον αναφορές προς ένα αντικείμενο. Ωστόσο η πιθανότητα διαρροής μνήμης παραμένει εφόσον διατηρηθεί μία αναφορά σε ένα αντικείμενο το οποίο δεν χρησιμοποιείται στη συνέχεια του προγράμματος. Κυρίως στόχος της αυτόματης διαχείρισης μνήμης είναι η ελάφρυνση του προγραμματιστή από το βάρος της χειροκίνητης διαχείρισης μνήμης, όπως συμβαίνει σε γλώσσες σαν τη C. Παράλληλα όμως διασφαλίζει ότι αξιοπιστία και η ασφάλεια της εφαρμογής συντηρείται εφόσον ο προγραμματισμός δεν εμπλέκεται καθόλου με την διαχείριση μνήμης γεγονός που θα μπορούσε να προκαλέσει κάποιο σφάλμα στο JVM εξαιτίας κάποιου λάθους χειρισμού της μνήμης.

Κάνοντας μία γρήγορη παρουσίαση του συλλέκτη απορριμμάτων που χρησιμοποιεί η Java να υπογραμμίσουμε πως υπάρχουν τέσσερεις διαφορετική τύποι διαθέσιμοι

- Σειριακός Συλλέκτης
  Η συλλογή χρησιμοποιώντας αυτόν τον τρόπο συλλογής απορριμμάτων πραγματοποιείται σταματώντας προσωρινά την εκτέλεση της εφαρμογής. Κυρίως είναι επιλογή εφαρμογών σε μηχανήματα τύπου «πελάτη» (client) τα οποία δεν έχουν απαίτηση για μικρούς χρόνους παύσης εκτέλεσης.

- Παράλληλος Συλλέκτης
  Ο Παράλληλος συλλέκτης αναπτύχθηκε για να εκμεταλλευτεί την ύπαρξη πολλαπλών πυρήνων, οι οποίοι έμεναν ανενεργοί (idle) κατά τη διάρκεια συλλογής απορριμμάτων. Πρακτικά προτιμάτε έναντι του σειριακού από όσα μηχανήματα έχουν παραπάνω από ένα πυρήνα διαθέσιμο.

- Παράλληλος Συλλέκτης Συμπίεσης
  Πρόκειται για μία βελτιωμένη έκδοση του παράλληλου συλλέκτη. Ο νέος αλγόριθμος που χρησιμοποιεί βελτιώνει την διαδικασία καθαρισμού (sweep). Πήρε τη θέση του παράλληλου συλλέκτη σε όσες εφαρμογές των χρησιμοποιούσαν εφόσον πρόκειται για μία βελτιωμένη έκδοση του.

- Παράλληλος Mark-Sweep (Μαρκάρισμα και Καθαρισμός) συλλέκτης
  Ονομάζεται αλλιώς και συλλέκτης χαμηλής καθυστέρησης και κύριο χαρακτηριστικό του είναι η αντιμετώπιση των μεγάλων παύσεων που συμβαίνουν όταν εμπλέκονται μεγάλοι σε μέγεθος σωροί (heaps).

**Εξαιρέσεις (Exceptions)**

Στη Java ο χειρισμός εξαιρέσεων είναι ένας από τους πιο ισχυρούς μηχανισμούς για την διαχείριση των σφαλμάτων εκτέλεσης (runtime errors) έτσι ώστε να συντηρείται η ομαλή ροής της εφαρμογής. Η εξαίρεση (exception) αποτελεί έναν event (γεγονός)

το οποίο διακόπτει την ομαλή ροή του προγράμματος, είναι ένα αντικείμενο το οποίο «ρίπτεται» κατά τη διάρκεια εκτέλεσης. Τρείς είναι οι κύριες κατηγορίες εξαιρέσεων που συναντάμε στη Java :

- **Ελεγμένες Εξαιρέσεις (Checked exceptions):**
  Οι ελεγμένες εξαιρέσεις είναι εξαιρέσεις τις οποίες θα πρέπει να μπορεί να διαχειρίζεται η Java εφαρμογή. Για παράδειγμα, εάν μία εφαρμογή διαβάζει δεδομένα από ένα αρχείο θα πρέπει να μπορεί να χειριστεί την FileNotFoundException. Αφού, δεν υπάρχει κάποια εγγύηση πως το αρχείο θα βρίσκεται εκεί που υποτίθεται πως θα είναι. Οτιδήποτε θα μπορούσε να συμβεί στο σύστημα αρχείων το οποίο δεν θα είχε τη δυνατότητα να αντιληφθεί.

- **Εξαιρέσεις κατά τη διάρκεια εκτέλεσης (Runtime exceptions):**
  Μία εξαίρεση κατά τη διάρκεια εκτέλεσης συμβαίνει απλά επειδή ο προγραμματιστής έκανε κάποιο λάθος. Κάποια από τα πιο συχνά είναι η προσπάθεια προσπέλασης ενός στοιχείου πίνακα το οποίο δεν υπάρχει ή η κλήση μιας μεθόδου με null τιμή εξαιτίας κάποιου λογικού σφάλματος στον κώδικα. Οι εξαιρέσεις κατά τη διάρκεια εκτέλεσης, σε αντίθεση με τις ελεγμένες εξαιρέσεις, αγνοούνται κατά τη διάρκεια της μεταγλώττισης.

- **Λάθη (Errors):**
  Όταν μία εξαίρεση συμβαίνει, ο JVM θα δημιουργήσει ένα αντικείμενο εξαίρεσης (exception object). Όλα τα αντικείμενα αυτά προκύπτουν από την υπερκλάση Throwable. Η Throwable κλάση έχει δύο κύριες υποκλάσεις την Error και την Exception. Η Error κλάση υποδηλώνει μία εξαίρεση την οποία η εφαρμογή δεν θα είναι εφικτό να διαχειριστεί. Είναι δυνατό για την εφαρμογή να «πιάσει» το λάθος (error) αυτό και να ειδοποιήσει τον χρήστη αλλά συνήθως η εφαρμογή θα πρέπει να τερματίσει μέχρι το υποκείμενο πρόβλημα να αντιμετωπιστεί.

**Generics**

Τα Generics είναι μία λειτουργία του γενικού προγραμματισμού η οποία προστέθηκε στη Java το 2004 στην έκδοση 5.0. Σχεδιάστηκαν για να επεκτείνουν το σύστημα τύπων της Java ώστε να επιτρέπει έναν τύπο ή μία μέθοδο να λειτουργεί πάνω σε αντικείμενα ποικίλων τύπων ενώ παρέχει ασφάλεια τύπων (type safety) κατά τη μεταγλώττιση.

## 6.2 XML

Η XML (Extensible Markup Language) πρόκειται για μία γλώσσα σήμανσης η οποία ορίζει ένα σύνολο κανόνων για τη κωδικοποίηση εγγράφων σε μία

συγκεκριμένη μορφή η οποία είναι αναγνώσιμη τόσο από ανθρώπους όσο και από τους υπολογιστές [15]. Ο σχεδιαστικός στόχος του XML είναι να προάγει την απλότητα, τη γενικότητα και την χρηστικότητα στο διαδίκτυο. Πρόκειται για δεδομένα με διαμόρφωση κειμένου και αξιοποιεί χαρακτήρες Unicode για την υποστήριξη όλων διαφορετικών ανθρώπινων γλωσσών. Αν και ο σχεδιασμός της XML επικεντρώνεται στα έγγραφα, η γλώσσα χρησιμοποιείται ευρέως για την αναπαράσταση αυθαίρετων δομών δεδομένων όπως αυτά που συναντάμε στις υπηρεσίες δικτύου. Πλήθος διεπαφών εφαρμογών έχουν αναπτυχθεί από προγραμματιστές για την διευκόλυνση στην επεξεργασία των XML δεδομένων.

Ο χαρακτηρισμός «επεκτάσιμη γλώσσα» οφείλεται στο ότι δίνει τη δυνατότητα στου χρήστες να καθορίσουν τα δικά τους στοιχεία. Κυρίως σκοπός της είναι η παροχή εύκολης ανταλλαγής δομημένων δεδομένων ανάμεσα στα διάφορα συστήματα πληροφοριών, συνήθως στο χώρο του Διαδικτύου. Αξιοποιείται για την κωδικοποίηση εγγράφων αλλά και για τη σειριοποίηση δεδομένων. Αποτελεί ένα πλαίσιο (framework) για τον ορισμό γλωσσών σήμανσης, όπου κάθε γλώσσα αποσκοπεί στο δικό της τομέα εφαρμογής αλλά όλες έχουν αρκετά κοινά χαρακτηριστικά.

- XLM processor-parser: Πρόκειται για τον λεκτικό επεξεργαστή, ο οποίος είναι υπεύθυνος για την ανάλυση της σήμανσης καθώς και για τη μεταφορά δομημένων πληροφορίων σε μια εφαρμογή. Μία ακόμα από τις αρμοδιότητες του είναι και η αντικατάσταση όλων των *αναφορών* σε οντότητες από τους ορισμούς τους καθώς και ο έλεγχος για την εγκυρότητα του αρχείου. Ο επεξεργαστής αναφέρεται συχνά και ως XML parser.
- Markup and Content (Σήμανση και περιεχόμενο): Οι χαρακτήρες που αποτελούν ένα XML έγγραφο χωρίζονται σε χαρακτήρες σήμανσης και χαρακτήρες περιεχομένου, κάτι το οποίο είναι απόρροια απλών συντακτικών κανόνων. Οι συμβολοσειρές σήμανσης ξεκινούν είτε με τον χαρακτήρα « < » είτε με το χαρακτήρα « & » και καταλήγουν στο χαρακτήρα « > » ή στο χαρακτήρα « ; ». Οι συμβολοσειρές που δεν είναι σήμανση αποτελούν το περιεχόμενο.
- Tags (Ετικέτες): Πρόκειται για ένα στοιχείο σήμανσης το οποίο αρχίζει με το χαρακτήρα « < » και τελειώνει με το χαρακτήρα « > ». Υπάρχουν ετικέτες έναρξης ( <data>), ετικέτες τέλους ( </data) και ετικέτες χωρίς στοιχείο <BR/>.
- Elements (Στοιχεία): Πρόκειται για ένα συστατικό του εγγράφου, και αποτελείται από τους χαρακτήρες μεταξύ μίας ετικέτας έναρξης και μία ετικέτας λήξης.
- Attributes (Γνωρίσματα): Πρόκειται για ένα στοιχείο σήμανσης το οποίο απαρτίζεται από ένα ζευγάρι ονόματος/τιμής που υπάρχει μέσα σε μία ετικέτα έναρξης ή μία ετικέτα χωρίς στοιχείο. Επιτρέπεται να έχει μόνο μία τιμή ενώ εμφανίζεται το πολύ μία φορά σε κάθε στοιχείο.

Στο επόμενο κεφάλαιο παρουσιάζεται και παράδειγμα ενός xml αρχείου, πιο συγκεκριμένα του αρχείου pom.xml, το οποίο είναι απαραίτητο για τη σωστή λειτουργία του Maven.

## 6.3 Maven

Το Maven είναι ένα εργαλείο διαχείρισης και κατανόησης project λογισμικού [16]. Είναι βασισμένο στην ιδέα ενός μοντέλου αντικειμένου (POM) και μπορεί να διαχειριστεί την διαδικασία δημιουργίας, αναφοράς και τεκμηρίωσης ενός project από ένα κεντρικό κομμάτι πληροφορίας. Χρησιμοποιείται κυρίως με την γλώσσα Java και η κυριολεκτική μετάφραση του είναι «συλλέκτης γνώσης».

Το Maven απευθύνεται σε δύο τομείς ανάπτυξης του λογισμικού, αρχικά περιγράφει πως το λογισμικό θα «χτιστεί» και στη συνέχεια περιγράφει τις εξαρτήσεις. Αντίθετα με τα προγενέστερα εργαλεία όπως το Apache Ant, χρησιμοποιεί δεδομένες παραδοχές για την διαδικασία μεταγλώττισης και μόνο οι εξαιρέσεις χρειάζεται να προσδιοριστούν. Ένα xml αρχείο είναι υπεύθυνο για την περιγραφή του λογισμικού, των εξαρτήσεων του από εξωτερικά πακέτα και συστατικά, της σειράς μεταγλώττισης, των καταλόγων (directories) και τον απαιτούμενων επεκτάσεων ( plug-ins). Έχει τη δυνατότητα να κατεβάζει δυναμικά βιβλιοθήκες Java και Maven plug-ins από ένα ή περισσότερα αποθετήρια (repositories) όπως το Maven 2 Central Repository και τα αποθηκεύει σε μία τοπική μνήμη. Η τοπική αυτή μνήμη που αποτελείται από κατεβασμένα artifacts μπορεί να ενημερωθεί από artifacts κατασκευασμένα από τοπικά project. Τα δημόσια αποθετήρια μπορούν και αυτά με τη σειρά τους να ενημερωθούν.

Το xml αυτό αρχείο πρόκειται για το αντικείμενο POM (Project Object Model) το οποίο αναφέραμε προηγουμένως και είναι τα θεμελιώδης σημασία για το έργο που επιτελεί το Maven. Κάποια κύρια πεδία του, τα ελάχιστα που πρέπει να οριστούν για τη σωστή λειτουργεία του, είναι τα εξής:

- Η «ρίζα» του προγράμματος
- modelVersion – Πρέπει να τεθεί 4.0.0
- groupId – Το group του ανήκει το πρόγραμμα
- artifactId – Η «ταυτότητα» του προγράμματος (artifact)
- version – Η έκδοση του προγράμματος (artifact) κάτω από το συγκεκριμένο group

Το POM απαιτεί το groupId, artifactId και version να είναι καθορισμένα και οι τρεις αυτές τιμές αποτελούν το πλήρως εξειδικευμένο artifact όνομα του προγράμματος στη μορφή <groupid>:<artifact>:<version>. Ακολουθεί ένα κομμάτι του pom.xml του δικού μας Java Project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>eu.coherentpaas</groupId>
  <artifactId>coherentpaas-mongomvcc</artifactId>
  <version>3.3-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>coherentpaas-mongomvcc</name>

    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.3.2</version>
          <configuration>
            <source>1.7</source>
            <target>1.7</target>
          </configuration>
        </plugin>
      </plugins>
    </build>

  <modules>
[…]
    <module>diplo-mongovcc-driver</module>
    <module>diplo-transactional-manager</module>
  </modules>

  <dependencies>
          […]

     <!--dependencies for log4j-->
      <dependency>
          <groupId>log4j</groupId>
          <artifactId>log4j</artifactId>
          <version>1.2.17</version>
     </dependency>
     <dependency>
          <groupId>org.slf4j</groupId>
          <artifactId>slf4j-api</artifactId>
          <version>1.7.5</version>
     </dependency>
     <dependency>
          <groupId>org.slf4j</groupId>
          <artifactId>slf4j-log4j12</artifactId>
          <version>1.7.5</version>
     </dependency>

   </dependencies>
</project>
```

## 6.4 Apache Avro

Το Avro πρόκειται για ένα πλαίσιο σειριοποίησης δεδομένων και απομακρυσμένων κλήσεων συναρτήσεων που έχει αναπτυχθεί μαζί με το project Hadoop της Apache. Χρησιμοποιεί JSON για να ορίσει τους τύπους δεδομένων και τα πρωτόκολλα και σειριοποιεί τα δεδομένα σε συμπαγή δυαδική (binary) μορφή [17].

## 6.5 Eclipse

Για την εκπόνηση της εργασίας έγινε χρήση του λογισμικού Eclipse SDK [18]. Πρόκειται για ένα ολοκληρωμένο περιβάλλον ανάπτυξης (Integrated Development Environment – IDE) το οποίο και είναι το νούμερο ένα λογισμικό για ανάπτυξη σε JAVA. Περιέχει ένα βασικό χώρο εργασίας και ένα επεκτάσιμο σύστημα με plug-ins το οποίο προσφέρει πλούσιες επιλογές παραμετροποίησης. Το Eclipse είναι γραμμένο στο μεγαλύτερο μέρος του σε Java και ενώ η κύρια χρήση του είναι η ανάπτυξη Java εφαρμογών, χρησιμοποιείται και για την ανάπτυξη εφαρμογών σε άλλες γλώσσες όπως η Ada, ABAP, C, C++, COBOL, D, Fortran, Haskell, JavaScript, Julia, Lasso, Lua, NATURAL, Perl, PHP, Prolog, Python, R, Ruby, Rust, Scala, Clojure, Groovy, Scheme και Erlang. Εμείς κάναμε χρήση του Eclipse software development kit (SDK), το οποίο περιέχει τα εργαλεία ανάπτυξης για Java, και απευθύνεται για Java προγραμματιστές. Το πακέτο αυτό είναι παρέχεται δωρεάν και είναι λογισμικό ανοιχτού κώδικα (open source), συνοδεύεται με τις προϋποθέσεις του Eclipse Public License , αν και δεν είναι συμβατό με το GNU General Public License.

# ΚΕΦΑΛΑΙΟ 7
# Επίλογος

## 7.1 Σύνοψη και συμπεράσματα

Στόχος της εργασίας ήταν η δημιουργία του συστήματος παροχής υπηρεσιών , με την υποστήριξη ταυτόχρονων συνδιαλλαγών, για τη μη-σχεσιακή βάση MongoDB. Αυτό επιτεύχθηκε μέσω της χρήσης του συστήματος που έχει αναπτυχθεί στα πλαίσια του προγράμματος CoherentPaas το οποίο προσφέρει την υποστήριξη συνδιαλλαγών και των ιδιοτήτων ACID. Αναλάβαμε και φέραμε εις πέρας την υλοποίηση ενός αφαιρετικού επιπέδου πάνω από το υπάρχων σύστημα έτσι ώστε να έχουμε τη δυνατότητα να το χρησιμοποιούμε με τις εφαρμογές χρήστη της αρεσκείας μας. Εν συνεχεία επιχειρήσαμε με επιτυχία να επεκτείνουμε το σύστημα αυτό με τον σχεδιασμό και την υλοποίηση κατανεμημένου συλλέκτη μη έγκυρων δεδομένων για τη βάση δεδομένων.

Σημαντικό κομμάτι της εκπόνησης της εργασίας, το οποίο και δεν είναι εύκολα να μετρηθεί ποσοτικά, ήταν η μελέτη, κατανόηση και ανάλυση των απαιτήσεων, των χαρακτηριστικών αλλά και γενικότερα στο σύνολο του, του συστήματος Coherent Paas και των επιμέρους συστατικών που το συνθέτουν. Ύστερα από την ολοκλήρωση του αφαιρετικού επιπέδου ακολούθησε η επαλήθευση της απόδοσης του μέσω πλήθος δοκιμών που πραγματοποιήθηκαν.

## 7.2 Μελλοντικές επεκτάσεις

Παρότι ο στόχος της εργασίας επετεύχθη, εφόσον το σύστημα που δημιουργήθηκε υποστηρίζει τις συνδιαλλαγές, τη χρήση του Transactional Manager και συνεπώς της MVCC, δεν έχουν πραγματοποιηθεί δοκιμές αξιολόγησης της επίδοσης του συγκεκριμένου συστήματος καθώς δεν αποτελούσε πρωταρχική σημασία της παρούσας εργασίας η επίτευξη γρήγορων χρόνων απόκρισης. Σαν μελλοντικό στόχο θα μπορούσαμε να θέσουμε την χρήση κάποιου εργαλείου αξιολόγησης συστημάτων νέφους εξυπηρέτησης, όπως για παράδειγμα το YCSB, για το fine-tuning και την περαιτέρω βελτίωση του συστήματος όσον αφορά τους χρόνους απόκρισής του.

# ΚΕΦΑΛΑΙΟ 8
# Βιβλιογραφία


[1] **Συστήματα Βάσεων Δεδομένων** –Abraham Silberschatz, Henry F. Korth, S. Sudarshan

[2] https://docs.oracle.com/database/121/CNCPT/transact.htm#CNCPT117

[3] https://technet.microsoft.com/en-us/library/ms190612(v=sql.105).aspx

[4] Berenson, Hal; Bernstein, Phil; Gray, Jim; Melton, Jim; O'Neil, Elizabeth; O'Neil, Patrick (1995), "A Critique of ANSI SQL Isolation Levels"

[5] **Θεμελιώδεις Αρχές Συστημάτων Βάσεων Δεδομένων –**R.Elmarsi, S.B.Navanthe

[6] https://www.mongodb.com/scale/what-is-a-non-relational-database

[7] http://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL

[8] http://coherentpaas.eu/

[9] **Java Concurrency in Practice** - Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea

[10] http://mongodb.github.io/mongo-java-driver/3.4/driver/getting-started/quick-start/

[11] **Head First Design Patterns** - Elisabeth Freeman , Kathy Sierra

[12] https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf

[13] http://www.oracle.com/technetwork/java/intro-141325.html

[14] **Thinking in Java (4th Edition)** – Bruce Eckel

[15] https://www.w3schools.com/xml/xml_whatis.asp

[16] https://maven.apache.org/

[17] https://www.tutorialspoint.com/avro/avro_overview.htm

[18] https://eclipse.org/