ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ ΗΛΕΚΤΡΟΝΙΚΗΣ
ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Σχεδιασμός και Υλοποίηση μηχανισμού ενοποίησης της μη σχεσιακής βάσης δεδομένων MongoDB με υπηρεσίες διαχειριστή ταυτόχρονων συνδιαλλαγών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ανδρέας Μ. Σχοινάς

Επιβλέπουσα: Θεοδώρα Βαρβαρίγου

Καθηγήτρια Ε.Μ.Π.

Αθήνα, Ιούλιος 2017

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ ΗΛΕΚΤΡΟΝΙΚΗΣ
ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Σχεδιασμός και Υλοποίηση μηχανισμού ενοποίησης της μη σχεσιακής βάσης δεδομένων MongoDB με υπηρεσίες διαχειριστή ταυτόχρονων συνδιαλλαγών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ανδρέας Μ. Σχοινάς

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

.....

ΑΝΔΡΕΑΣ ΣΧΟΙΝΑΣ

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

Περίληψη

Η MongoDB, είναι μία από τις πιο ευρέως διαδεδομένες NoSQL βάσεις δεδομένων, η οποία χρησιμοποιεί ένα ευέλικτο μοντέλο εγγράφων για την αποθήκευση των δεδομένων, το οποίο προσεγγίζει το πρότυπο JSON. Το μοντέλο αυτό, υποστηρίζει δεδομένα σε μορφή κλειδιού – τιμής και εκτός από τα συνηθισμένα είδη δεδομένων (τα οποία είναι διαθέσιμα στην μεγάλη πλειοψηφία των δημοφιλών γλωσσών προγραμματισμού), σαν εγγραφές, μπορεί επιπλέον να δεχθεί πίνακες, δυαδικά δεδομένα, ακόμα και άλλα έγγραφα. Το γεγονός αυτό προσφέρει τρομερή ευελιξία στους χρήστες, καθώς καθιστά την αλλαγή του μοντέλου δεδομένων, το ίδιο εύκολη και γρήγορη με την μεταβολή των ίδιων των εφαρμογών.

Η προσαρμοστικότητα που προσφέρει λόγω του δυναμικού της σχήματος και της αντικειμενοστραφούς της φύσης καθιστούν την MongoDB ιδανική για εφαρμογές και αναλύσεις πραγματικού χρόνου. Έχει καθιερωθεί, συνεπώς, ως βάση δεδομένων γενικής χρήσης με πληθώρα εφαρμογών σε τομείς όπως τα ηλεκτρονικά καταστήματα, οι εφαρμογές για κινητά, εφαρμογές υπολογιστικού νέφους, αλλά και οι εφαρμογές γεωχωρικής φύσεως. Εφαρμογές της MongoDB, επίσης, περιλαμβάνουν αποθήκευση και διαχείριση μεγάλων δεδομένων (Big Data), καθώς και διαχείριση περιεχομένου και κατασκευή υποδομών για κοινωνικά δίκτυα και κινητά.

Παρά το ολοένα και αυξανόμενο πλήθος εφαρμογών της MongoDB, η εφαρμογή της σε τομείς συναλλαγών (Transactions) είναι ακόμα σε εμβρυακό στάδιο. Το γεγονός, αυτό, παρατηρείται στις περισσότερες NoSQL βάσεις δεδομένων και συμβαίνει κυρίως λόγω της έλλειψης απαραίτητου ενδιάμεσου λογισμικού, που να επιτυγχάνει τον έγκυρο και αποδοτικό συγχρονισμό σε περιπτώσεις ταυτόχρονων συναλλαγών. Η επίλυση τυχών διενέξεων μπορεί να επιτευχθεί με την υλοποίηση ενός απλού συστήματος κλειδώματος, κατά το οποίο μόνο μία συναλλαγή θα επιτρέπεται να αλλάζει τα δεδομένα της βάσης την εκάστοτε στιγμή, ενώ οποιαδήποτε άλλη επιθυμεί επίσης να αλλάξει κάποιο δεδομένο θα πρέπει να περιμένει. Ωστόσο, η υλοποίηση αυτή υστερεί υπερβολικά σε θέμα ταχύτητας, ειδικά όταν εφαρμοστεί σε μεγάλα συστήματα, όπου τα δεδομένα της βάσης διαμοιράζονται σε πολλούς κόμβους και προσπελάζονται από χιλιάδες χρήστες.

Η συγκεκριμένη διπλωματική εργασίας έχει ως σκοπό τον σχεδιασμό και την υλοποίηση μηχανισμού διαχείρισης ταυτόχρονων συναλλαγών προς μία βάση MongoDB, ο οποίος ελέγχει για τυχόν διενέξεις και στην συνέχεια αναθέτει χρονοσφραγίδες (timestamps) στις συναλλαγές καθιστώντας τες ικανές να δρομολογηθούν. Επιπλέον, σχεδιάστηκε και δημιουργήθηκε μία διεπαφή, με χρήση του συστήματος σειριοποίησης δεδομένων, Apache Avro, πάνω σε ήδη υλοποιημένο σύστημα συναλλαγών σε Mongo, η οποία μπορεί δυνητικά να χρησιμοποιηθεί, ώστε να ενσωματωθεί οποιοσδήποτε αντίστοιχος μηχανισμός για την διαχείριση των ταυτόχρονων συναλλαγών.

Λέξεις Κλειδιά:
Βάσεις Δεδομένων, Απομόνωση Στιγμιότυπου, Έλεγχος Συγχρονικότητας πολλαπλών εκδόσεων, MongoDB, Μη Σχεσιακές Βάσεις Δεδομένων, Διαχειριστής Συναλλαγών

Abstract

MongoDB is one of the most widespread NoSQL databases worldwide. It utilizes a flexible documents record model for data storing, which resembles the JSON object format. This model stores data as key – value pairs and besides the common data types (that are supported by the vast majority of the popular programming languages), MongoDB documents also support arrays, binary data, even other documents as input. This makes MongoDB extremely flexible to the users, provides a dynamic schema and renders the data model able to change as rapidly as the application requirements.

MonogoDB's adeptness, due to its dynamic schema and its object – oriented nature, make it a perfect fit for real–time analytics and applications. Therefore, it has been established as a general purpose database, with a wide variety of applications, including e-shops, mobile, computational cloud and geospacial applications. MongoDB is also used to store and manage Big Data content, infrastructure development for social networks and mobile systems, as well as lightweight business intelligence applications.

Despite the increasing number of applications that rely on it, MonogoDB's has yet to be used in transactional applications. Like most NoSQL database systems, MongoDB is not one of the choices for transactional applications, and this is mainly due to the lack of the appropriate middleware, which would provide valid and efficient synchronization in case of concurrent transactions. Such middleware could resolve conflicts simply by utilizing a simple locking mechanism, which will ensure that only one transaction will be able to modify database content, at each given moment, so if another transactions wants to alter the contents of the database, it must wait until the previous operation is completed. Nevertheless, this solution falls short in terms of speed, especially, when applied to big data systems, in which the database contents are shared between many nodes and are accessed by thousands of users.

This thesis' main purpose is to design and implement a transaction management mechanism for concurrent MongoDB transactions. The aforementioned transaction manager is invoked on every stage of the transaction to perform the appropriate conflict checks and assign timestamps to the transactions, making them available for routing. Additionally, a connection interface, based on the Apache Avro data serialization system, was designed and exposed on top of a Mongo transaction system, through which any transaction manager can be assigned to manage the transactions' concurrency.

Keywords:
Databases, Snapshot Isolation, Multiversion Concurrency Control, MongoDB, NoSQL, Transactional Manager, Apache Avro

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο εργαστήριο Distributed Knowledge and Media System Group του Τομέα Επικοινωνιών, Ηλεκτρονικής και Συστημάτων Πληροφορικής της σχολής των Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη της Καθηγήτριας Θεοδώρας Βαρβαρίγου.

Θα ήθελα να ευχαριστήσω θερμά την Κ. Θεοδώρα Βαρβαρίγου για την υποστήριξη και την καθοδήγηση που μου προσέφερε κατά τη διάρκεια της διπλωματικής αυτής εργασίας, καθώς και για την ευκαιρία που μου έδωσε να ασχοληθώ με το ερευνητικό κομμάτι των κατανεμημένων βάσεων δεδομένων.

Επίσης θα ήθελα να ευχαριστήσω ιδιαίτερα τον υποψήφιο διδάκτορα Παύλο Κρανά, για την πολύτιμη βοήθεια του, την προθυμία και την καθοδήγηση του, σε όλη τη διάρκεια της εργασίας, καθώς και τον συνάδελφο, Μιλτιάδη Γιαλούση για την άψογη συνεργασία και συνεισφορά του στην εκπόνηση αυτής της εργασίας.

Περιεχόμενα

Σχήματα Και Πίνακες

1 Εισαγωγή

1.1 Βάση Δεδομένων

Ως βάση δεδομένων, καλείται οποιοδήποτε σύνολο δεδομένων (πληροφορίας), το οποίο είναι οργανωμένο με τέτοιο τρόπο, ώστε να μπορεί εύκολα να προσπελαστεί, να διαχειριστεί και να τροποποιηθεί. Μια βάση δεδομένων μπορεί να περιγραφεί σαν μια συλλογή από εγγραφές, κάθε μία από τις οποίες περιέχει ένα ή περισσότερα πεδία με πληροφορίες για κάποιο οντότητα, όπως για παράδειγμα, κάποιο άτομο ή κάποια εταιρία.

Ανά τα χρόνια έχουν αναπτυχθεί πολλών ειδών βάσεις δεδομένων, η κάθε μια με τα δικά της χαρακτηριστικά, όπως η δομή, το είδος των πεδίων που υποστηρίζει, αλλά και τις λειτουργίες που προσφέρει πάνω στα δεδομένα που φιλοξενεί. Ωστόσο, σε γενικές γραμμές οι βάσεις δεδομένων προσφέρουν **σχήματα**, **πίνακες**, **ερωτήματα** και **όψεις**, αλλά και άλλου είδους αντικείμενα αναλόγως την βάση.

Τα σχήματα (schemas) αποτελούν μια περιγραφή του μοντέλου που φιλοξενείται στην βάση και περιέχουν τους τύπους των οντοτήτων που υπάρχουν μέσα σε αυτό, καθώς και συσχετισμούς μεταξύ αυτών.

Πίνακες ή Συλλογές (tables ή collections) ονομάζονται το σύνολο των δεδομένων σχετικά με μία οντότητα. Στις σχεσιακές βάσεις, αποτελούνται από στήλες και γραμμές, όπου κάθε γραμμή περιέχει πληροφορία για ένα αντικείμενο της οντότητας (για παράδειγμα αν έχουμε κάποιον πίνακα που περιέχει άτομα, κάθε γραμμή θα περιγράφει ένα άτομο), ενώ στα κελιά της γραμμής αυτής περιέχονται οι τιμές για τα διάφορα χαρακτηριστικά της εγγραφής.

Με τον όρο «ερωτήματα» καλείται το σύνολο των διεργασιών που προσφέρει η κάθε βάση πάνω στα δεδομένα που φιλοξενεί. Η μεγάλη πλειοψηφία των βάσεων προσφέρει στον χρήστη ερωτήματα για διεργασίες ανάγνωσης, εισαγωγής, τροποποίησης και διαγραφής δεδομένων (**C**reate **R**ead **U**pdate **D**elete), ενώ σε πολλές περιπτώσεις προσφέρονται η δυνατότητα για πολύπλοκα ερωτήματα με συνδυασμό πινάκων, σχημάτων, ακόμα και βάσεων.

Οι όψεις στις βάσεις δεδομένων αποτελούν μια προκαθορισμένη εικόνα της βάσης ορισμένη από τον χρήστη, η οποία περιλαμβάνει συνδυαστική ή ενισχυμένη (ή και αφαιρετική σε συγκεκριμένες περιπτώσεις) πληροφορία από το σύνολο της βάσης που εξυπηρετεί κάποια λειτουργία του χρήστη. Για παράδειγμα σε περιπτώσεις εξαγωγής αναφορών, ο χρήστης δεν χρειάζεται όλη την πληροφορία που μπορεί να υπάρχει για κάθε εγγραφή, αλλά συγκεκριμένα πεδία μόνο που αφορούν την αναφορά, ή σε κάποια άλλη περίσταση να απαιτεί συνδυασμό πληροφορίας από δύο ή παραπάνω πίνακες.

1.2 Σύστημα Διαχείρισης Βάσεων Δεδομένων

Ένα Σύστημα Διαχείρισης Βάσεων Δεδομένων (DBMS) αποτελείται από ένα σύνολο από σχετιζόμενα δεδομένα (Βάση Δεδομένων) και από ένα σύνολο από προγράμματα που χρησιμοποιούνται για πρόσβαση στα δεδομένα, αυτά. Ο βασικός στόχος ενός DBMS είναι η παροχή τρόπου αποθήκευσης και ανάκλησης των δεδομένων από τις βάσεις, ο οποίος να είναι βολικός και αποτελεσματικός.

Τα συστήματα βάσεων δεδομένων σχεδιάζονται με τρόπο τέτοιο, ώστε να χειρίζονται μεγάλη ποσότητα πληροφοριών. Η διαχείριση των πληροφοριών περιλαμβάνει τόσο τον ορισμό των δομών αποθήκευσης τους, όσο και μηχανισμούς για τον χειρισμό τους. Επιπλέον τα συστήματα διαχείρισης βάσεων δεδομένων θα πρέπει να εξασφαλίζουν την ασφάλεια των πληροφοριών που διατηρούν ανεξάρτητα από τυχόν προβλήματα του συστήματος ή τις προσπάθειες μη πιστοποιημένης πρόσβασης. Ακόμα, αν τα δεδομένα είναι κοινόχρηστα μεταξύ χρηστών, τότε το σύστημα διαχείρισης θα πρέπει να αποφεύγει πιθανά λανθασμένα αποτελέσματα που μπορεί να προκύψουν από διαδοχικές οι ταυτόχρονες προσπελάσεις των δεδομένων από παραπάνω από έναν χρήστη.

1.3 Η εμφάνιση των Βάσεων Δεδομένων και τα πρώτα προβλήματα

Τα συστήματα Βάσεων Δεδομένων προέκυψαν κατά τη δεκαετία του ’60 σαν απόκριση στις αρχικές μεθόδους αυτοματοποιημένης διαχείρισης εμπορικών δεδομένων. Στα πρώτα αυτά συστήματα η αποθήκευση των δεδομένων πραγματοποιούνταν σε αρχεία του λειτουργικού συστήματος, ενώ τα προγράμματα που επέτρεπαν στους χρήστες να χειρίζονται τις πληροφορίες δημιουργούνταν ανάλογα με την εφαρμογή, ώστε να καλύπτουν τις απαιτήσεις του εκάστοτε συστήματος. Ένα τέτοιο τυπικό σύστημα επεξεργασίας αρχείων αποθήκευε μόνιμες εγγραφές σε διάφορα αρχεία και απαιτούσε την ύπαρξη ξεχωριστών προγραμμάτων για την εξαγωγή και την προσθήκη εγγραφών στα κατάλληλα αρχεία. Όπως ήταν αναμενόμενο, αυτά τα συστήματα παρουσίασαν αρκετά μειονεκτήματα ανοίγοντας τον δρόμο για την ανάπτυξη των βάσεων δεδομένων:

- **Επαναληπτικότητα και ασυνέπεια των δεδομένων:** Καθώς τα συστήματα που αναφέραμε παραπάνω απαιτούν μεγάλη συντήρηση, ήταν αρκετά πιθανό διαφορετικοί προγραμματιστές να δημιουργήσουν κάποιο πρόγραμμα διαχείρισης για ένα τέτοιο σύστημα, με αποτέλεσμα την ύπαρξη πολλών διαφορετικών μορφών δεδομένων, αλλά και γλωσσών προγραμματισμού στο ίδιο σύστημα. Επιπλέον, υπήρχε το ενδεχόμενο η ίδια πληροφορία να αποθηκευτεί σε δύο διαφορετικά αρχεία αυξάνοντας έτσι το κόστος αποθήκευσης και πρόσβασης. Ακόμα, λόγω αυτής της επαναληπτικότητας ήταν πιθανή η τροποποίηση κάποιας πληροφορίας σε κάποιο σημείο, χωρίς την εγγύηση ότι τροποποιήθηκαν όλες οι αναφορές της, με αποτέλεσμα την ασυνέπεια των δεδομένων, δηλαδή την ασυμφωνία μεταξύ αναφορών στα ίδια δεδομένα.
- **Δυσκολία στην πρόσβαση δεδομένων:** Ένα σύστημα επεξεργασίας δεδομένων, είχε την δυνατότητα να αποθηκεύει δεδομένα και να τα ανακτά, ωστόσο δεν υπήρχε η δυνατότητα για ανάκτηση δεδομένων υπό συνθήκη. Οι μόνες διαθέσιμες επιλογές ήταν είτε η εξαγωγή όλων των δεδομένων μιας λίστας και στη συνέχεια

επιλογή με το μάτι όσων πληρούσαν τα επιθυμητά κριτήρια είτε η κατασκευή κατάλληλης εφαρμογής που θα επέστρεφε τα αντίστοιχα αποτελέσματα. Το πρόγραμμα αυτό όμως θα μπορούσε να επιστρέψει μόνο τις εγγραφές που πληρούσαν το συγκεκριμένο κριτήριο. Αν υπήρχε ανάγκη για ανάκτηση δεδομένων με κάποιο άλλο κριτήριο, θα έπρεπε να δημιουργηθεί ένα καινούριο πρόγραμμα, γεγονός που καθιστούσε τον σύστημα αυτό δύσχρηστο.

- **Απομόνωση των δεδομένων:** Επειδή τα δεδομένα βρίσκονταν κατανεμημένα σε διάφορα αρχεία με διαφορετικές μορφές, η συγγραφή προγραμμάτων για ανάκληση τέτοιων, διασκορπισμένων, δεδομένων αποτελούσε μία πολύ απαιτητική διαδικασία.
- **Προβλήματα ακεραιότητας:** Ένα ακόμα πρόβλημα που αντιμετώπισε αυτό το σύστημα αφορούσε τους περιορισμούς συνέπειας που ενδεχομένως χρειαζόταν να εφαρμοστούν στα δεδομένα. Συγκεκριμένα, το πρόβλημα κορυφωνόταν σε περιπτώσεις που απαιτούνταν αρκετοί περιορισμοί, καθώς χρειάζονταν μεγάλη πολυπλοκότητα και προγραμματιστικό κόστος για την υλοποίηση τους.
- **Προβλήματα ατομικότητας:** Ένας υπολογιστής, όπως όλες οι συσκευές κινδυνεύει να χαλάσει. Σε πολλές εφαρμογές, είναι απαραίτητο σε κάποιο τέτοιο σενάριο τα δεδομένα να επιστρέψουν στην συνεπή κατάσταση που βρίσκονταν πριν το πρόβλημα. Σε τέτοιες περιπτώσεις πρέπει η προσπέλαση των δεδομένων να είναι ατομική, δηλαδή είτε να ολοκληρώνεται εντελώς είτε να μην γίνεται καθόλου, να επιστρέφει σε μια προηγούμενη κατάσταση.
- **Προβλήματα ταυτόχρονης πρόσβασης:** Για την βελτίωση της απόδοσης του συστήματος και της ταχύτερης απόκρισης, πολλά συστήματα επιτρέπουν την ταυτόχρονη προσπέλαση και τροποποίηση των δεδομένων από πολλαπλούς χρήστες. Το αποτέλεσμα, όμως, ταυτόχρονων ενημερώσεων μπορεί να είναι η ύπαρξη ασυνεπών δεδομένων. Με τα δεδομένα της εποχής, κάποια υλοποίηση επίβλεψης στο σύστημα ήταν αρκετά περίπλοκη και δύσκολη στην υλοποίηση.
- **Προβλήματα ασφάλειας:** Με την ύπαρξη πολλών χρηστών οι οποίοι θα έχουν πρόσβαση στην βάση, εγείρεται το ζήτημα της πρόσβασης στα δεδομένα. Συνήθως δεν είναι επιθυμητό να έχουν όλοι οι χρήστες πρόσβαση σε όλα τα δεδομένα και απαιτείται κάποιου είδους έλεγχος της ταυτότητας του χρήστη και έγκριση κατάλληλης πρόσβασης. Με την συνεχή προσθήκη, όμως, προγραμμάτων συγκεκριμένου σκοπού στο σύστημα, ο έλεγχος αυτός γίνεται αρκετά δύσκολος.

1.4 Τα είδη των Βάσεων Δεδομένων και ιστορική αναδρομή

Η επεξεργασία των πληροφοριών ακολουθεί την εξέλιξη των εμπορικών υπολογιστών. Για την ακρίβεια, η αυτοματοποίηση της διαδικασίας επεξεργασίας δεδομένων ξεκίνησε πριν από τους υπολογιστές με τις διάτρητες κάρτες που ανέπτυξε ο Hollerith στην αρχή του 20^{ου} αιώνα. Αρχικά χρησιμοποιήθηκαν για καταγραφή δημογραφικών δεδομένων των Η.Π.Α. αλλά στη συνέχεια, η μέθοδος αυτή χρησιμοποιήθηκε ως μέσο εισαγωγής δεδομένων στους υπολογιστές. Έκτοτε έχουν αναπτυχθεί διάφορες τεχνικές για την αποθήκευση και επεξεργασία των δεδομένων με την πάροδο των χρόνων:

- **Δεκαετία του ’50 – αρχές της δεκαετίας του ‘60**

Κάνουν την πρώτη εμφάνιση τους οι μαγνητικές ταινίες ως μέσο αποθήκευσης δεδομένων. Εργασίες επεξεργασίας δεδομένων, όπως η μισθοδοσία πλέον αυτοματοποιούνται και βασίζονται στο διάβασμα και καταγραφή δεδομένων σε μαγνητικές ταινίες. Οι διάτρητες κάρτες συνεχίζουν να χρησιμοποιούνται, σε συνδυασμό με τις ταινίες για την εμφάνιση συνδυαστικών δεδομένων από τα δύο αυτά είδη αποθήκευσης δεδομένων. Τόσο οι ταινίες, όσο και οι διάτρητες κάρτες μπορούσαν να προσπελαστούν μονάχα σειριακά, επομένως τα προγράμματα επεξεργασίας δεδομένων έπρεπε να ακολουθούν συγκεκριμένη σειρά διαβάσματος και συγχώνευσης δεδομένων.

- **Τέλη δεκαετιών ’60 και ‘70**

Στα τέλη της δεκαετίας του ’60, άρχιζε να παρατηρείται ευρεία χρήση των σκληρών δίσκων, αλλάζοντας ριζικά τον τρόπο και τα προγράμματα επεξεργασίας δεδομένων, καθώς πλέον έγινε δυνατή η άμεση προσπέλαση των δεδομένων. Πλέον τα δεδομένα μπορούσαν να προσπελαστούν σε κλάσματα του δευτερολέπτου, άσχετα από την θέση που ήταν γραμμένα στον δίσκο, εξαλείφοντας έτσι το βάρος της σειριοποίησης. Με την εμφάνιση των δίσκων πλέον μπορούσαν να δημιουργηθούν δίκτυα και ιεραρχικές βάσεις δεδομένων, που επέτρεπαν την δημιουργία δομών δεδομένων που αποθηκεύονταν στους δίσκους, όπως οι λίστες και τα δέντρα. Έτσι οι χρήστες μπορούσαν να κατασκευάσουν και να διαχειριστούν τις δομές αυτές, με την δημιουργία των κατάλληλων προγραμμάτων.

Την δεκαετία αυτή ορίστηκε και το σχεσιακό μοντέλο, καθώς και ο μη-διαδικαστικός τρόπος δημιουργίας ερωτημάτων σε αυτό κι έτσι δημιουργήθηκαν οι μη-σχεσιακές βάσεις δεδομένων. Το μεγάλο θετικό που εισήγαγε αυτή η καινοτομία ήταν η απλότητα και η απόκρυψη των λεπτομερειών χειρισμού από τον προγραμματιστή.

- **Δεκαετία του ‘80**

Παρά τα εμφανή πλεονεκτήματα που προσέφερε το σχεσιακό μοντέλο, δεν χρησιμοποιήθηκε αρχικά, λόγω μειονεκτημάτων που εμφάνιζε στην απόδοση. Οι πρώτες επιτυχημένες απόπειρες για κατασκευή ενός αποτελεσματικού σχεσιακού συστήματος βάσεων δεδομένων πραγματοποιήθηκαν ξεχωριστά το 1981 από την IBM και το Πανεπιστήμιο της Καλιφόρνιας στο Μπέρκλεϋ. Η έρευνα της IBM κατέληξε στην πρώτη σχεσιακή βάση δεδομένων της, την SQL/DS, ενώ το πανεπιστήμιο της Καλιφόρνιας, συγχρόνως ανέπτυξε το σύστημα Ingress. Στην συνέχεια ακολούθησαν και άλλες εταιρίες όπως η Oracle και πλέον το σχεσιακό μοντέλο φαίνεται να καθιερώνεται, καθώς οι σχεσιακές βάσεις κατάφεραν να γίνουν ανταγωνιστικές ως προς την απόδοση τους, έναντι των συστημάτων βάσεων δεδομένων δικτύου και των ιεραρχικών συστημάτων βάσεων, αλλά και η απλότητα που προσέφεραν ήταν πρωτόγνωρη.

- **Δεκαετία του ‘90**

Στις αρχές της δεκαετίας του ’90 σχεδιάστηκε η γλώσσα SQL, για υποστήριξη των αποφάσεων των εφαρμογών, με την δημιουργία ερωτημάτων για βάσεις δεδομένων,

καθώς και πολλά εργαλεία για ανάλυση μεγάλης ποσότητας δεδομένων. Στη συνέχεια, παράλληλα με την εκρηκτικής ανάπτυξη του world wide web, αλλά και λόγω αυτής, οι βάσεις δεδομένων αναπτύχθηκαν πολύ περισσότερο. Τα συστήματα των βάσεων δεδομένων πλέον είχαν πολύ υψηλούς ρυθμούς επεξεργασίας όπως και πολύ μεγάλη αξιοπιστία και διαθεσιμότητα 24-7 (όλο το 24ωρο, 7 μέρες την εβδομάδα), ενώ αρχίζουν να υποστηρίζουν web περιβάλλοντα για πρόσβαση στα δεδομένα.

- **Δεκαετία του '00**

Κάνει την εμφάνιση της η XML και η σχετική γλώσσα διατύπωσης ερωτημάτων η XQuery ως μία νέα τεχνολογία βάσεων δεδομένων. Παρόλο που η XML χρησιμοποιείται ευρέως για ανταλλαγή δεδομένων, αλλά και για αποθήκευση σύνθετων τύπων δεδομένων, οι σχεσιακές βάσεις εξακολουθούν να διαμορφώνουν τον πυρήνα μιας μεγάλης πλειοψηφίας εφαρμογών βάσεων δεδομένων μεγάλης κλίμακας. Κατά την περίοδο αυτή, παρατηρείται ανάπτυξη τεχνικών για «αυτόνομης και αυτόματης διαχείρισης» για ελαχιστοποίηση της προσπάθειας διαχείρισης των συστημάτων. Παράλληλα, κάνουν την εμφάνιση τους συστήματα βάσεων δεδομένων ανοιχτού κώδικα όπως η PostgreSQL και η MySQL.

Στο τελευταίο μέρος της δεκαετίας έχουν εμφανιστεί εξειδικευμένες βάσεις δεδομένων για ανάλυση δεδομένων, ειδικότερα αποθηκεύσεις στηλών, οι οποίες ουσιαστικά αποθηκεύουν κάθε στήλη πίνακα ως ένα ξεχωριστό πίνακα και πολύ παράλληλα συστήματα βάσεων δεδομένων τα οποία έχουν σχεδιαστεί ια ανάλυση πολύ μεγάλων συνόλων δεδομένων. Έχουν δημιουργηθεί αρκετά κατανεμημένα συστήματα αποθήκευσης δεδομένων για να χειρίζονται τις απαιτήσεις διαχείρισης δεδομένων πολύ μεγάλων δικτυακών τόπων όπως η Amazon, το Facebook, η Yahoo, ή η Microsoft. Μερικά από αυτά τα συστήματα προσφέρονται και σαν υπηρεσίες που μπορούν να χρησιμοποιηθούν από προγραμματιστές εφαρμογών. Ακόμα, πολύ μεγάλη έμφαση έχει δοθεί στην διαχείριση και την ανάλυση δεδομένων συνεχούς ροής, όπως είναι τα δεδομένα μετοχών ή παρακολούθησης δικτύων. Τέλος, αναπτύχθηκαν ευρέως τεχνικές εξόρυξης δεδομένων, όπως για παράδειγμα, εφαρμογές με προτάσεις προϊόντων βασισμένων στην συμπεριφορά του χρήστη στο web, καθώς και αυτόματη τοποθέτηση διαφημίσεων σε ιστοσελίδες.

- **Δεκαετία του '10**

Από την αρχή της δεκαετίας που διανύουμε, είχε καθιερωθεί η παρουσία των διαφόρων κοινωνικών δικτύων στην καθημερινότητα των ανθρώπων. Ο μεγάλος όγκος των δεδομένων και η ανάγκη για περιγραφή, αποθήκευση και πολύ ταχεία ανάκτηση πολύπλοκων κοινωνικών δομών οδήγησαν στην στροφή προς τις NoSQL βάσεις δεδομένων, λόγω της υψηλής απόδοσης καθώς και της ευκολίας στην κλιμάκωση που προσφέρουν. Παρ' όλο που οι σχεσιακές βάσεις δεδομένων εξακολουθούν να βρίσκονται στο προσκήνιο και να χρησιμοποιούνται κατά κόρον, οι μη-σχεσιακές παρουσιάζουν τρομερή άνθηση. Το δυναμικό τους σχήμα και η ευελιξία τους, ωστόσο, δεν είναι αρκετά για να καταφέρουν να αντικαταστήσουν τις σχεσιακές βάσεις δεδομένων, καθώς παρουσιάζουν ένα αρκετά μεγάλο μειονέκτημα: μέχρι στιγμής δεν υποστηρίζουν συναλλαγές. Ωστόσο, αρκετή έμφαση δίνεται στην υλοποίηση

μηχανισμών και εξωτερικών διαχειριστών συναλλαγών ώστε να ενσωματωθεί η λειτουργικότητα των συναλλαγών στις μη-σχεσιακές βάσεις δεδομένων και είναι θέμα χρόνου η προσπάθεια αυτή να τελεσφορήσει.

1.5 Αντικείμενο Διπλωματικής

Στόχος της παρούσας διπλωματικής εργασίας είναι η επέκταση της πλατφόρμας διαχείρισης συναλλαγών για NoSQL βάσεις δεδομένων, CoherentPaas, υλοποιώντας το κατάλληλο αφαιρετικό επίπεδο το οποίο θα χρησιμοποιηθεί σαν διεπαφή για την ενσωμάτωση οποιουδήποτε διαχειριστή συναλλαγών στην πλατφόρμα αυτή. Με την προσθήκη αυτή στην πλατφόρμα CoherentPaas, θα δίνεται η δυνατότητα στον χρήστη να επιλέξει ποια εφαρμογή επιθυμεί να διαχειρίζεται τις συναλλαγές της εφαρμογής του, καθώς και στους προγραμματιστές να πραγματοποιήσουν μετρήσεις απόδοσης και σύγκρισης των διαφόρων διαχειριστών συναλλαγών.

Επιπλέον, στα πλαίσια της εργασίας αυτής, κατασκευάστηκε ένας πλήρως λειτουργικός διαχειριστής συναλλαγών, ο οποίος χρησιμοποιήθηκε για επαλήθευση της λειτουργικότητας της παραπάνω υλοποίησης.

1.6 Οργάνωση Κειμένου

Πέραν της εισαγωγής, τα υπόλοιπα κεφάλαια οργανώνονται ως εξής:

- Στο 2^{ο} κεφάλαιο αναλύεται η βάση δεδομένων που χρησιμοποιήσαμε, η MongoDB, ενώ γίνεται μία εισαγωγή και στις NoSQL βάσεις δεδομένων, γενικότερα.
- Στο 3^{ο} κεφάλαιο αναλύεται η θεωρία των συναλλαγών, οι απαραίτητες συνθήκες που απαιτείται να ισχύουν, τυχόν προβλήματα που μπορεί να προκύψουν, καθώς και ορισμένα δημοφιλή πρωτόκολλα εξασφάλισης της συγχρονικότητας.
- Στο 4^{ο} κεφάλαιο αναφέρεται η έλλειψη υποστήριξης συναλλαγών από την MongoDB, παρατίθενται συγκεκριμένες εφαρμογές που προσπαθούν να αντιμετωπίσουν την έλλειψη αυτή και αναλύεται ο τρόπος που η εργασία αυτή θα συνεισφέρει στην αντιμετώπιση αυτή.
- Στο 5^{ο} κεφάλαιο περιγράφεται η υλοποίηση της εργασίας, με περιγραφή των τεχνολογιών που χρησιμοποιήθηκαν.
- Στο 6^{ο} κεφάλαιο περιγράφονται οι έλεγχοι που πραγματοποιήθηκαν για επαλήθευση της λειτουργικότητας της υλοποίησης.
- Στο 7^{ο} κεφάλαιο συγκεντρώνονται τα συμπεράσματα που προέκυψαν από την διεξαγωγή της εργασίας.
- Στο 8^{ο} κεφάλαιο αναγράφεται η βιβλιογραφία που χρησιμοποιήθηκε για την εκπόνηση της εργασίας.
- Στο 9^{ο} κεφάλαιο (παράρτημα) περιέχεται ένα βοηθητικό λεξικό αγγλικών όρων που έχουν μεταφραστεί στα Ελληνικά για χάρη της εργασίας.

2 MongoDB

2.1 NoSQL Βάσεις Δεδομένων

[3] Με την ονομασία NoSQL (non SQL) εννοούμε ένα μεγάλο σύνολο μη σχεσιακών βάσεων δεδομένων, όπως είναι η MongoDB, η Hadoop, η Cassandra κτλ. Πολλές φορές υπονοείται ότι η ονομασία προέρχεται από τα αρχικά Not Only SQL, εννοώντας ότι η sql δεν είναι μονόδρομος για την ανάπτυξη κάποιου λογισμικού ή εφαρμογής. Ενώ οι απαρχές της NoSQL βρίσκονται, σύμφωνα με κάποιους, στα μέσα της δεκαετίας του 60, οι πρώτες εμφανίσεις της σε εφαρμογές μεγάλου βεληνεκούς, εντοπίζονται μετά το τέλος του 20ου αιώνα.

Δυναμικά σχήματα

Οι σχεσιακές βάσεις δεδομένων προϋποθέτουν ότι τα σχήματα είναι πλήρως ορισμένα πριν μπορέσουν να εισαχθούν δεδομένα σε αυτά. Αυτό αποτελεί τροχοπέδη για την μοντέρνα ευέλικτη προσέγγιση στην ανάπτυξη λογισμικού, κατά την οποία αλλαγές στο τελικό προϊόν είναι συχνές και συνηθισμένες. Έτσι σε κάθε προσθήκη ή τροποποίηση, απαιτείται αντίστοιχη τροποποίηση στο σχήμα που χρησιμοποιείται, ενώ σε ακραίες περιπτώσεις μπορεί να χρειαστεί ακόμα και αλλαγή της βάσης. Επομένως, ειδικά σε παραγωγικά περιβάλλοντα τα οποία κάνουν χρήση αρκετά μεγάλων βάσεων δεδομένων, αυτό αμέσως ερμηνεύεται σε σημαντικό χρόνο κατά τον οποίο η εφαρμογή δεν θα είναι διαθέσιμη λόγω των αλλαγών οι οποίες πρέπει να πραγματοποιηθούν. Επιπλέον, δεν υπάρχει κανένας αποδοτικός τρόπος, κάποια σχεσιακή βάση δεδομένων, να διαχειριστεί δεδομένα τελείως αδόμητα ή με δομή άγνωστη εκ των προτέρων.

Αντίθετα με τις σχεσιακές βάσεις, οι NoSQL είναι σχεδιασμένες να επιτρέπουν την εισαγωγή δεδομένων χωρίς προκαθορισμένο σχήμα. Αυτό διευκολύνει την διαδικασία πραγματοποίησης σημαντικών αλλαγών σε πραγματικό χρόνο χωρίς την ανησυχία για διακοπές της υπηρεσίας. Έτσι επιταχύνεται η διαδικασία υλοποίησης, γίνεται πιο αξιόπιστη η διαδικασία ενσωμάτωσης του κώδικα, ενώ ελαχιστοποιείται ο χρόνος για διαχειριστικές εργασίες στην βάση δεδομένων. Αυτό από την άλλη, προσθέτει επιπλέον κόπο στο έργο των προγραμματιστών, καθώς απαιτείται να υλοποιηθούν έλεγχοι εγκυρότητας στον κώδικα, όταν κάποια πεδία είναι υποχρεωτικά. Σε πιο εξελιγμένες NoSQL βάσεις δεδομένων, επιτρέπουν τέτοιου είδους ελέγχους να πραγματοποιούνται στο επίπεδο της βάσης, επιτρέποντας στους χρήστες να έχουν τον απαραίτητο έλεγχο πάνω στα δεδομένα, ενώ ταυτόχρονα απολαμβάνουν τα πλεονεκτήματα που προσφέρει ένα δυναμικό σχήμα.

2.2 Είδη NoSQL βάσεων δεδομένων

Με την άνοδο των NoSQL βάσεων, πλέον υπάρχουν πάνω από 255 (σύμφωνα με τις τελευταίες μετρήσεις) διαφορετικές NoSQL βάσεις δεδομένων, καθεμία με τα δικά της χαρακτηριστικά. Ο πιο απλός διαχωρισμός στις πιο δημοφιλείς βάσεις δεδομένων μπορεί να πραγματοποιηθεί με βάση το είδος των εγγραφών που υποστηρίζουν, με πιο βασικά τα παρακάτω:

Βάσεις δεδομένων σε μορφή εγγράφων: Η κεντρική ιδέα πίσω από τις βάσεις είναι η έννοια του «εγγράφου». Ενώ κάθε βάση δεδομένων που χρησιμοποιεί δομές εγγράφων διαφέρει από τις υπόλοιπες, όλες βασίζονται σε έγγραφα που εμπεριέχουν και κωδικοποιούν πληροφορία σε κάποια προκαθορισμένη μορφή ή κωδικοποίηση. Οι πιο συνηθισμένες από αυτές τις κωδικοποιήσεις είναι τα αρχεία YAML, XML ή JSON, ή δυαδικές δομές όπως το BSON (στην περίπτωση της Mongo). Τα έγγραφα μπορούν να περιέχουν ζεύγη κλειδιών-τιμών διαφορετικών τύπων, ζεύγη κλειδιών-πινάκων αλλά και εμφωλευμένα άλλα έγγραφα. Ένα από τα μοναδικά χαρακτηριστικά των βάσεων αυτών είναι ότι επιπλέον της αναζήτησης με βάση το κλειδί κάποιας πληροφορίας, η βάση δεδομένων προσφέρει την δυνατότητα ανάκτησης δεδομένων με ερωτήματα βασισμένα στο περιεχόμενο τους.

Δομές αποθήκευσης γραφημάτων: Οι δομές αποθήκευσης γραφημάτων χρησιμοποιούνται για την αποθήκευση δεδομένων, οι σχέσεις μεταξύ των οποίων, μπορούν να αναπαρασταθούν ως γράφημα στοιχείων συνδεδεμένων με πεπερασμένο πλήθος σχέσεων μεταξύ τους. Χρησιμοποιούνται σε εφαρμογές με διασυνδεδεμένα δεδομένα, όπως για παράδειγμα σχέσεις επαφών σε κοινωνικά μέσα δικτύωσης, δίκτυα συγκοινωνιών, οδικοί χάρτες και τοπολογίες δικτύων.

Δομές κλειδιών-τιμών: Τις πιο απλές από τις NoSQL βάσεις δεδομένων αποτελούν οι δομές κλειδιών-τιμών. Στα μοντέλα αυτά τα δεδομένα αναπαριστώνται ως συλλογές ζευγών κλειδιών – τιμών στις οποίες κάθε πιθανή τιμή κλειδιού έχει μία το πολύ εμφάνιση. Κάποιες από τις βάσεις αυτές επιτρέπουν τον ορισμό του είδους της τιμής (όπως για παράδειγμα «ακέραιοι αριθμοί»), γεγονός που προσδίδει επιπλέον λειτουργικότητα. Το μοντέλο αυτό είναι από τα πιο απλά μοντέλα δεδομένων, πάνω στο οποίο βασίζονται πιο πλούσια μοντέλα. Όπως για παράδειγμα κάποιο ενισχυμένο μοντέλο που να διατηρεί τα κλειδιά σε λεξικογραφική κατάταξη και να δίνει την δυνατότητα ταχείας ανάκτησης δεδομένων από κάποιο εύρος κλειδιών.

2.3 Η βάση MongoDB

Εικόνα 2.1 *MongoDB*

[2] [5] Η MongoDB (από την Αγγλική λέξη hu**mongo**us = τεράστιος) αποτελεί μία βάση δεδομένων ανοιχτού κώδικα, η οποία υπάγεται στην κατηγορία των NoSQL βάσεων. Αναπτύχθηκε από την εταιρία MongoDB Inc. (μέχρι πρότινος 10gen) και αποτελεί δημοφιλή επιλογή για χρήση σε εφαρμογές από κάποιες από τις μεγαλύτερες εταιρίες παγκοσμίως, όπως είναι οι: Adobe, ebay, Craigslist, FIFA(EA Sports), Foursquare, LinkedIn, McAfee, κ.α. Η

MongoDB χρησιμοποιεί έγγραφα (documents) σαν μονάδες αποθήκευση των δεδομένων της, σε μορφή αντικειμένων BSON, τα οποία είναι μία δομή δεδομένων, αποτελούμενη από ζεύγη κλειδιών και τιμών, παρόμοια με τα αντικείμενα JSON. Οι τιμές των πεδίων του αντικειμένου, αυτού, μπορούν να πάρουν σαν τιμές άλλα έγγραφα, πίνακες, ακόμα και πίνακες από έγγραφα. Στην MongoDB τα έγγραφα αποθηκεύονται σε συλλογές (collections). Οι συλλογές εκτελούν τον ρόλο των πινάκων στις σχεσιακές βάσεις δεδομένων.

Εικόνα 2.2 *Παράδειγμά εγγράφου της MongoDB σε μορφή BSON*

Τα έγγραφα που χρησιμοποιεί η MongoDB σαν δομική μονάδα έρχονται με αρκετά πλεονεκτήματα, όπως:

- Τα πεδία των εγγράφων αντιστοιχούν σε βασικούς τύπους δεδομένων σε πολλές γλώσσες προγραμματισμού
- Η επιλογή που παρέχουν για χρήση άλλων εγγράφων και πινάκων σαν τιμές για τα πεδία της μειώνει την ανάγκη για πολύπλοκα συνδυαστικά ερωτήματα με μεγάλο υπολογιστικό κόστος υλοποίησης
- Το δυναμικό τους σχήμα της προσδίδει πολυμορφισμό

2.3.1 Χαρακτηριστικά της MongoDB

Η MongoDB επιτυγχάνει υψηλής απόδοσης διατήρηση δεδομένων. Συγκεκριμένα, με την υποστήριξη ενσωματωμένων μοντέλων δεδομένων μειώνει την δραστηριότητα I/O των συστημάτων βάσεων δεδομένων. Επίσης, με την χρήση ευρετηρίων προσφέρει γρήγορα ερωτήματα στα οποία μπορούν να περιληφθούν ακόμα και κλειδιά από τα ενσωματωμένα έγγραφα ή πίνακες.

Στα χαρακτηριστικά της MongoDB, προστίθεται η πλούσια γλώσσα για πραγματοποίηση ερωτημάτων, καθώς η βιβλιοθήκη τους, υποστηρίζει μεταξύ άλλων:

- Βασικές λειτουργίες ανάγνωσης και εγγραφής
- Συνδυασμό δεδομένων (data aggregation), μοντελοποιημένο σύμφωνα με την έννοια των επεξεργασίας δεδομένων μέσω σωληνώσεων δεδομένων (pipelines). Τα έγγραφα διοχετεύονται σε μία σωλήνωση πολλών σταδίων, η οποία τα

μεταμορφώνει σε συνδυαστικά αποτελέσματα. Αυτό η συνδυαστική σωλήνωση (aggregation pipeline) προσφέρει μία εναλλακτική επιλογή αντί του map-reduce και προτιμάται σε περιπτώσεις όπου η πολυπλοκότητα του map-reduce δεν μπορεί να εγγυηθεί.

- Αναζήτηση κειμένου στο περιεχόμενο των πεδίων, για τα οποία χρησιμοποιείται ως ευρετήριο ένα έγγραφο ερωτήματος κειμένου, το οποίο περιλαμβάνει το κείμενο που επιθυμείται να αναζητηθεί, καθώς και επιπλέον παραμέτρους όπως η διάκριση πεζών, κεφαλαίων.
- Γεωχωρικά ερωτήματα. Στην MongoDB, υπάρχει η δυνατότητα αποθήκευσης, δημιουργίας ευρετηρίου και ερωτημάτων δεδομένων σχετικών με γεωγραφικές παραμέτρους, κάνοντας χρήση των εννοιών των γεωχωρικών σχημάτων, σημείων και ευρετηρίου.

Ένα σημαντικό χαρακτηριστικό της Mongo αποτελεί η υψηλή διαθεσιμότητα της, η οποία επιτυγχάνεται με την χρήση ενός χώρου αντιγραφής, ο οποίος καλείται replica set. Το replica set αποτελεί ένα σύνολο από εξυπηρετητές MongoDB, οι οποίοι διατηρούν το ίδιο σύνολο δεδομένων, προσφέροντας πλεονασμό δεδομένων, αυτόματα χαρακτηριστικά διαχείρισης αποτυχιών, καθώς και αυξημένη διαθεσιμότητα των δεδομένων.

Για την διαχείριση των θεμάτων που προκύπτουν με την ανάγκη για κλιμάκωση σε εφαρμογές με πολύ μεγάλα σύνολα δεδομένων και λειτουργιών υψηλών διεκπεραιωτικών (throughput) αναγκών, τα συστήματα που χρησιμοποιούν MongoDB έχουν την δυνατότητα να επιλέξουν ανάμεσα σε δύο βασικές προσεγγίσεις:

- **Κάθετη κλιμάκωση**, κατά την οποία για αύξηση της απόδοσης, προστίθενται επιπλέον πόροι (μονάδες αποθήκευσης και CPU) στα υπάρχοντα συστήματα
- **Οριζόντια κλιμάκωση, ή sharding**. Αντίθετα με την κάθετη κλιμάκωση, στην περίπτωση αυτή, τα δεδομένα προς αποθήκευση τμηματοποιούνται και διαμοιράζονται σε ένα σύμπλεγμα μηχανημάτων. Κάθε ξεχωριστό μηχάνημα, ή shard, είναι μία ανεξάρτητη βάση δεδομένων, ενώ συνολικά όλα τα shards συντάσσουν την επιθυμητή λογική βάση.

3 Συναλλαγές (Transactions)

[1] Ο όρος συναλλαγή (transaction) αναφέρεται σ' ένα σύνολο από λειτουργίες που αποτελούν μια λογική μονάδα. Είναι, συνεπώς, σημαντικό είτε να εκτελεσθούν όλες οι ενέργειες μιας συναλλαγής, είτε σε περίπτωση κάποιου προβλήματος να ακυρωθεί οποιοδήποτε μέρος της συναλλαγής έχει ολοκληρωθεί εν μέρει. Επιπλέον, πρέπει να διασφαλίζεται η ταυτόχρονη εκτέλεση πολλαπλών συναλλαγών με τρόπο που να αποφεύγει τη δημιουργία ασυνεπειών στην βάση δεδομένων.

3.1 Χαρακτηριστικά

Για να μπορέσει ένα σύστημα βάσης δεδομένων να διατηρήσει την αξιοπιστία του, πρέπει να χαρακτηρίζεται από τις παρακάτω ιδιότητες:

Ατομικότητα (Atomicity)

Η ατομικότητα είναι η απαίτηση η συναλλαγή να χαρακτηρίζεται ως «όλα ή τίποτα», αν, δηλαδή, κάποιο τμήμα της συναλλαγής αποτύχει για οποιονδήποτε λόγο, τότε πρέπει η συναλλαγή στο σύνολο της να αποτύχει. Όσες αλλαγές είχαν προλάβει να πραγματοποιηθούν θα πρέπει να αναιρεθούν και η βάση στο τέλος να έχει την εικόνα που είχε πριν την εκκίνηση της συναλλαγής αυτής. Για να χαρακτηριστεί ένα σύστημα ως ατομικό, θα πρέπει να εγγυάται το παραπάνω αποτέλεσμα υπό οποιεσδήποτε συνθήκες, συμπεριλαμβανομένων των διακοπών ρεύματος, σφαλμάτων και crashes. Ένας εξωτερικός παρατηρητής θα πρέπει να μπορεί να διακρίνει κάποια ολοκληρωμένη συναλλαγή, ως ένα αδιαίρετο σύνολο αλλαγών στη βάση, ενώ καμία αλλαγή σε αυτή δεν θα πρέπει να εμφανίζεται αν κάποια συναλλαγή αποτύχει.

Συνέπεια (Consistency)

Συνέπεια καλείται η ιδιότητα που εγγυάται με οποιαδήποτε συναλλαγή η βάση θα μεταβεί από μία έγκυρη κατάσταση σε μία άλλη. Όσα δεδομένα έχουν γραφτεί στην βάση με το πέρας της θα πρέπει να είναι έγκυρα με βάση τους θεσμοθετημένους κανόνες, συμπεριλαμβανομένων των περιορισμών (constraints , cascades, triggers και όλους τους συνδυασμούς αυτών). Αυτό δεν εξασφαλίζει απαραίτητα την εγκυρότητα της συναλλαγής, καθώς γι αυτήν υπεύθυνος είναι ο κώδικας της εφαρμογής που την πραγματοποιεί, παρά μόνο ότι η συναλλαγή δεν παραβιάζει τους κανόνες που έχουν οριστεί.

Απομόνωση (Isolation)

Η απομόνωση είναι η ιδιότητα που εξασφαλίζει ότι η ταυτόχρονη εκτέλεση των συναλλαγών θα αποφέρει την ίδια ακριβώς έγκυρη κατάσταση με την σειριακή εκτέλεση τους. Η σειριακή εκτέλεση των συναλλαγών φαντάζει ευκολότερη, καθώς η διασφάλιση της συνέπειας σε ταυτόχρονη εκτέλεση συναλλαγών απαιτεί παραπάνω δουλειά. Ωστόσο

υπάρχουν δύο πολύ καλοί λόγοι για να αναζητήσουμε την συγχρονικότητα. Η βελτιωμένη απόδοση και χρήση των πόρων, καθώς και η ραγδαία μείωση του χρόνου αναμονής.

Ανθεκτικότητα (Durability)

Η ιδιότητα της ανθεκτικότητας είναι η εγγύηση ότι άπαξ και μία συναλλαγή έχει ολοκληρωθεί, όλες οι αλλαγές που πραγματοποιήθηκαν στην βάση δεδομένων θα παραμείνουν ακόμα και αν παρουσιαστεί κάποιο πρόβλημα στην συνέχεια.

3.2 Επίπεδα απομόνωσης

Αν κάθε συναλλαγή έχει την ιδιότητα να διατηρεί την συνέπεια της βάσης δεδομένων κατά την ατομική εκτέλεση της, τότε η σειριοποιησιμότητα εξασφαλίζει ότι οι ταυτόχρονες εκτελέσεις θα διατηρούν επίσης την συνέπεια. Ωστόσο τα πρωτοκολλά που εξασφαλίζουν την σειριοποιησιμότητα, μπορεί να επιτρέπουν λίγη έως ελάχιστη συγχρονικότητα σε ορισμένες εφαρμογές. Σε αυτές τις περιπτώσεις γίνεται χρήση πιο αδύναμων επιπέδων συνέπειας, μεταφέροντας, έτσι, περισσότερο φόρτο στους προγραμματιστές. Τα επίπεδα απομόνωσης που ορίζονται στο SQL πρότυπο είναι τα εξής:

- **Σειριοποίηση (serializable):** Η ιδιότητα της σειριοποίησης εξασφαλίζει συνήθως τη σειριοποιήσιμη εκτέλεση Ωστόσο, ορισμένα συστήματα βάσεων δεδομένων εφαρμόζουν αυτό το πρότυπο με τρόπο που μπορεί σε συγκεκριμένες περιπτώσεις να επιτρέπει και μη σειριοποιήσιμες εκτελέσεις.
- **Επαναλαμβανόμενο Διάβασμα (Repeatable Read):** Το επαναλαμβανόμενο διάβασμα επιτρέπει να διαβάζονται μόνο τα εκτελεσμένα δεδομένα και επιπλέον απαιτεί μεταξύ δύο διαβασμάτων ενός στοιχείου δεδομένων από μία συναλλαγή, καμία άλλη συναλλαγή να μην επιτρέπεται να το ενημερώσει.
- **Διάβασμα Ολοκληρωμένων (Read Committed):** Το διάβασμα ολοκληρωμένων επιτρέπει το διάβασμα μόνο των εκτελεσθέντων δεδομένων, αλλά δεν απαιτεί επαναλαμβανόμενα διαβάσματα. Μεταξύ δύο διαδοχικών διαβασμάτων ενός στοιχείου δεδομένων από μία συναλλαγή, δηλαδή, επιτρέπεται κάποια άλλη να το ενημερώσει ενδιάμεσα και να το έχει εκτελέσει.
- **Διάβασμα Μη-ολοκληρωμένων (Read Un-committed):** Το διάβασμα μη-ολοκληρωμένων επιτρέπει το διάβασμα μη εκτελεσθέντων δεδομένων και είναι το χαμηλότερο επίπεδο απομόνωσης που επιτρέπεται από το πρότυπο SQL.

Κατά τον σχεδιασμό των εφαρμογών μπορεί να αποφασιστεί να γίνει δεκτό ένα πιο αδύναμο επίπεδο απομόνωσης, προκειμένου να βελτιωθεί η απόδοση του συστήματος, καθώς η εξασφάλιση της σειριοποίησης μπορεί να αναγκάσει κάποια συναλλαγή να περιμένει άλλες συναλλαγές, ή σε ορισμένες περιπτώσεις, ακόμα και να διακοπεί επειδή η συναλλαγή δεν μπορεί πλέον να εκτελεστεί σαν μέρος μιας σειριοποιήσιμης εκτέλεσης. Παρόλο που πιθανώς να φαντάζει τραβηγμένο να διακινδυνευτεί η συνέπεια της βάσης δεδομένων για λόγους απόδοσης, η ανταλλαγή αυτή αποκτάει νόημα όταν μπορούμε να είμαστε βέβαιοι ότι η ασυνέπεια, που μπορεί να υπάρξει, δεν έχει σχέση με την εφαρμογή.

3.3 Φαινόμενα ανάγνωσης

Σύμφωνα με το πρότυπο ANSI/ISO η SQL 92 αναφέρει τρία διαφορετικά και προβληματικά φαινόμενα ανάγνωσης που μπορεί να προκύψουν ανάλογα με το επίπεδο απομόνωσης. Τα φαινόμενα ανάγνωσης παρατηρούνται όταν μία συναλλαγή (Transaction 1) διαβάζει δεδομένα τα οποία μπορεί να τροποποιηθούν από μία δεύτερη συναλλαγή (Transaction 2). Για την ανάλυση των φαινομένων αυτών θα θεωρήσουμε την παρακάτω δομή:

Πίνακας: users		
Id	**name**	**age**
1	Joe	20
2	Jill	25

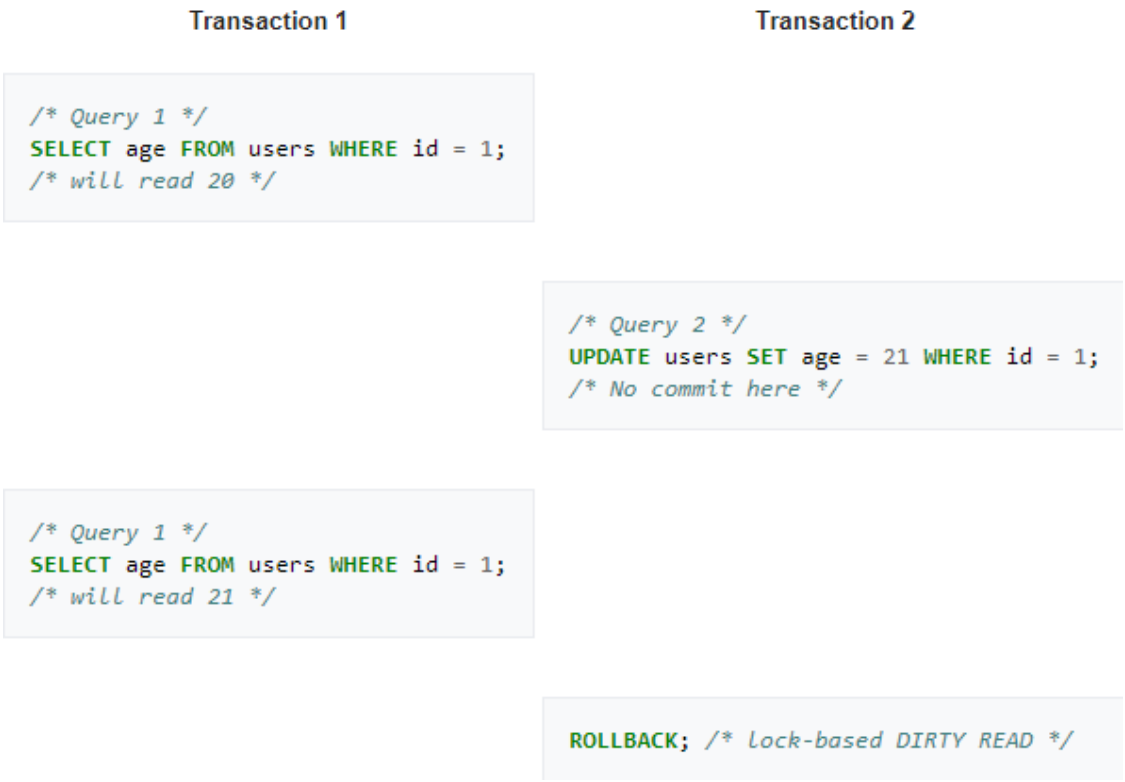
Πίνακας 3.1 *Δεδομένα Παραδείγματος*

Στις ακόλουθες περιπτώσεις θεωρούμε ότι η πρώτη συναλλαγή εκτελεί το ίδιο ερώτημα 2 φορές, ενώ ενδιάμεσα υπάρχει τροποποίηση των δεδομένων από την δεύτερη συναλλαγή.

Πρόχειρο γράψιμο (dirty reads)

Το φαινόμενο του πρόχειρου γραψίματος (επίσης γνωστό και ως μη ολοκληρωμένη εξάρτηση - uncommitted dependency) παρατηρείται σε περιπτώσεις που το επίπεδο απομόνωσης επιτρέπει σε κάποια συναλλαγή να διαβάσει δεδομένα από κάποια εγγραφή η οποία έχει επεξεργαστεί από κάποια άλλη συναλλαγή και δεν έχει ακόμα ολοκληρωθεί. Το πρόχειρο γράψιμο λειτουργεί με παρόμοιο τρόπο όπως και οι μη-επαναλαμβανόμενες αναγνώσεις, με τη διαφορά ότι η δεύτερη συναλλαγή δεν χρειάζεται να έχει ολοκληρωθεί, για να επιστρέψει το πρώτο ερώτημα διαφορετικό αποτέλεσμα. Η μόνη κατάσταση που μπορεί να αποφευχθεί κατά το «Διάβασμα μη-ολοκληρωμένων» επίπεδο απομόνωσης είναι η εμφάνιση των τροποποιήσεων (updates) με διαφορετική σειρά από αυτήν που πραγματοποιήθηκαν. Επομένως, πρότερες αλλαγές θα εμφανίζονται πάντα νωρίτερα από αυτές που τις ακολουθούν σε κάποιο σύνολο αποτελεσμάτων.

Στο παρακάτω παράδειγμα η δεύτερη συναλλαγή πραγματοποιεί μία αλλαγή σε κάποιο πεδίο της βάσης, αλλά δεν ολοκληρώνει την αλλαγή, ενώ στην συνέχεια η πρώτη συναλλαγή διαβάζει τα μη ολοκληρωμένα αυτά δεδομένα. Σε περίπτωση που η δεύτερη συναλλαγή αποφασίσει να μην ολοκληρώσει την αλλαγή αλλά να την ματαιώσει, η πρώτη συναλλαγή θα παραμείνει με μία λάθος και μη συνεπή εικόνα των δεδομένων της βάσης.



Εικόνα 3.1 *Παράδειγμα πρόχειρου γραψίματος*

Στο παραπάνω παράδειγμα η δεύτερη συναλλαγή τροποποιεί τα δεδομένα ανάμεσα στις δύο αναγνώσεις της πρώτης συναλλαγής, ωστόσο, στο τέλος, δεν ολοκληρώνει την αλλαγή, με αποτέλεσμα, στο τέλος η πρώτη συναλλαγή να έχει μη έγκυρες εγγραφές δεδομένων.

Μη-επαναλαμβανόμενες αναγνώσεις (Non-repeatable reads)

Το φαινόμενο μη-επαναλαμβανόμενων αναγνώσεων συμβαίνει όταν σε μία συναλλαγή κάποια εγγραφή ανακτάται πολλαπλές φορές, αλλά οι τιμές που ανακτώνται διαφέρουν από ανάγνωση σε ανάγνωση. Το φαινόμενο αυτό παρατηρείται σε μεθόδους ελέγχου συγχρονικότητας βασισμένους σε κλειδώματα, όταν δεν προσφέρονται κλειδώματα ανάγνωσης κατά την εκτέλεση ενός SELECT ερωτήματος, ή όταν αυτά υπάρχουν, αλλά ελευθερώνονται αμέσως μετά την εκτέλεση του ερωτήματος. Αντίθετα, σε περιπτώσεις ελέγχου συγχρονικότητας μέσω πολλαπλών εκδόσεων, το φαινόμενο αυτό μπορεί να παρατηρηθεί όταν η απαίτηση ότι μια συναλλαγή πρέπει να ματαιωθεί αν υποπέσει σε διένεξη ολοκλήρωσης (commit conflict) είναι χαλαρά ορισμένη.

Εικόνα 3.2 *Παράδειγμα μη-επαναλαμβανόμενης ανάγνωσης*

Στο παράδειγμα αυτό, η δεύτερη συναλλαγή, πραγματοποιεί μια αλλαγή και την ολοκληρώνει επιτυχώς, γεγονός που σημαίνει ότι η αλλαγή πλέον είναι εμφανής στην βάση. Όμως, πριν από την αλλαγή αυτή η πρώτη συναλλαγή έχει ήδη διαβάσει τα περιεχόμενα της εγγραφής που άλλαξε με αποτέλεσμα κατά την δεύτερη ανάγνωση μέσα στην ίδια συναλλαγή, να βλέπει διαφορετικές τιμές στα δεδομένα χωρίς να τα έχει επηρεάσει η ίδια. Σε περιπτώσεις επιπέδων απομόνωσης Σειριοποίησης και Επαναλαμβανόμενου Διαβάσματος το σύστημα διαχείρισης της βάσης δεδομένων θα πρέπει να επιστρέψει την αρχική τιμή που διάβασε η συναλλαγή ακόμα και στο στην δεύτερη ανάγνωση, ενώ στα επίπεδα απομόνωσης Διάβασμα Ολοκληρωμένων και Διάβασμα Μη-ολοκληρωμένων, το σύστημα διαχείρισης μπορεί να επιστρέψει την ανανεωμένη τιμή.

Υπάρχουν δύο τρόποι αποφυγής των μη-επαναλαμβανόμενων αναγνώσεων. Ο πρώτος είναι η προσθήκη κάποιας χρονικής καθυστέρησης στην εκτέλεση της δεύτερης συναλλαγής , μέχρις ότου να ολοκληρωθεί ή να ματαιωθεί η πρώτη. Η πρακτική αυτή εφαρμόζεται σε μεθόδους συγχρονικότητας βασισμένες σε κλειδώματα και οδηγεί σε μία σειριακή εκτέλεση των συναλλαγών.

Η δεύτερη μέθοδος αντιμετώπισης του φαινομένου αυτού, είναι η τακτική που ακολουθείται και κατά την χρήση πολλαπλών εκδόσεων για την εξασφάλισης της παραλληλίας των συναλλαγών. Σε αυτήν, η δεύτερη συναλλαγή επιτρέπεται να πραγματοποιήσει αλλαγές και να τις ολοκληρώσει πρώτη, ωστόσο, η πρώτη συναλλαγή θα πρέπει να συνεχίσει να εκτελείται στο στιγμιότυπο της βάσης δεδομένων που διατηρεί την εικόνα της βάσης όπως ήταν όταν ξεκίνησε η συναλλαγή. Όταν η πρώτη συναλλαγή, αισίως, αιτηθεί να ολοκληρωθεί, το σύστημα διαχείρισης της βάσης δεδομένων ελέγχει αν το

αποτέλεσμα της ολοκλήρωσης αυτής θα επιφέρει το ίδιο αποτέλεσμα με το αποτέλεσμα που θα είχε αν ολοκληρωνόταν πρώτα η πρώτη συναλλαγή και στη συνέχεια η δεύτερη. Αν υπάρξει διαφορά ανάμεσα στα δύο αυτά αποτελέσματα τότε η ολοκλήρωση της πρώτης συναλλαγής αποτυγχάνει επιστρέφοντας κατάλληλο μήνυμα αποτυχίας της σειριοποίησης.

Χρησιμοποιώντας έλεγχο συγχρονικότητας βασισμένο σε κλειδώματα, κατά το επίπεδο απομόνωσης επαναλαμβανόμενου διαβάσματος, η εγγραφή με ID=1 θα κλειδωνόταν, μην επιτρέποντας στο ερώτημα της δεύτερης συναλλαγής να εκτελεστεί μέχρι η πρώτη συναλλαγή ολοκληρωθεί επιτυχώς ή ματαιωθεί. Αντίθετα, χρησιμοποιώντας διάβασμα ολοκληρωμένων, κατά την δεύτερη εκτέλεση του πρώτου ερωτήματος, τα δεδομένα θα έχουν τροποποιηθεί από την δεύτερη συναλλαγή.

Σε περιπτώσεις εξασφάλισης συγχρονικότητας μέσω πολλαπλών εκδόσεων, σε επίπεδο απομόνωσης σειριοποίησης και τα δύο ερωτήματα της πρώτης συναλλαγής βλέπουν μία απεικόνιση – στιγμιότυπο της βάσης την στιγμή που ξεκίνησε η πρώτη συναλλαγή, επομένως και τα δύο θα επιστρέψουν το ίδιο αποτέλεσμα. Αν, όμως στην συνέχεια η πρώτη συναλλαγή προσπαθούσε να τροποποιήσει την τιμή της εγγραφής αυτής, η απόπειρα αυτή θα αποτύγχανε, καθώς η εγγραφή αυτή έχει ήδη τροποποιηθεί από κάποια άλλη συναλλαγή. Επομένως θα εμφανιζόταν ένα σφάλμα σειριοποίησης και η πρώτη συναλλαγή θα έπρεπε να ματαιωθεί και να αναιρέσει όλες τις αλλαγές που έχει τυχόν πραγματοποιήσει. Αντίθετα σε επίπεδο απομόνωσης διαβάσματος ολοκληρωμένων, κάθε ερώτημα βλέπει ένα στιγμιότυπο της βάσης δεδομένων το οποίο έχει ληφθεί κατά την εκκίνηση του εκάστοτε ερωτήματος. Συνεπώς κάθε ερώτημα έχει διαφορετική εικόνα για την τροποποιημένη εγγραφή. Σε αυτήν την περίπτωση δεν υπάρχει σφάλμα σειριοποίησης καθώς στην περίπτωση αυτή δεν υπάρχει εγγύηση αυτής και, συνεπώς, δεν υπάρχει η απαίτηση η πρώτη συναλλαγή να ανακληθεί.

Ανάγνωση «Φάντασμα» (Phantom reads)

Λέμε ότι έχουμε μία ανάγνωση φάντασμα όταν στην διάρκεια κάποιας συναλλαγής εκτελούνται δύο πανομοιότυπα ερωτήματα, αλλά το σύνολο των επιστρεφόμενων εγγραφών διαφέρει ανάμεσα στα δύο. Αυτό μπορεί να συμβεί όταν δεν ανατίθενται τα κατάλληλα κλειδώματα κατά την διάρκεια της εκτέλεσης κάποιου ερωτήματος SELECT WHERE με έλεγχο εύρους τιμών. Η ανάγνωση «φάντασμα» μπορεί να θεωρηθεί μια ειδική περίπτωση η-επαναλαμβανόμενης ανάγνωσης στην οποία η πρώτη συναλλαγή επαναλαμβάνει το ίδιο ερώτημα SELECT WHERE με απαίτηση εύρους τιμών και ανάμεσα στις δύο εκτελέσεις η δεύτερη συναλλαγή προσθέτει ή αφαιρεί δεδομένα που επηρεάζουν το αποτέλεσμα του ερωτήματος.

Εικόνα 3.3 *Παράδειγμα ανάγνωσης «φάντασμα»*

Αν είχε επιβληθεί το μέγιστο επίπεδο απομόνωσης, το ίδιο σύνολο αποτελεσμάτων θα είχε επιστραφεί και στα δύο παραπάνω ερωτήματα, όπως και είναι υποχρεωτικό να συμβαίνει στις βάσεις δεδομένων που λειτουργούν κάτω από το σειριοποιήσιμο επίπεδο απομόνωσης. Αντίθετα, στα υπόλοιπα επίπεδα απομόνωσης, υπάρχει το ενδεχόμενο τα δύο αυτά ερωτήματα να επιστρέψουν κάθε φορά διαφορετικό σύνολο αποτελεσμάτων.

Σε περιβάλλοντα με σειριοποίηση ως επίπεδο απομόνωσης κατά την εκτέλεση του πρώτου ερωτήματος θα τοποθετούνταν κλειδώματα σε όλες τις εγγραφές με το συγκεκριμένο ζητούμενο εύρος τιμών. Έτσι, το ερώτημα της δεύτερης συναλλαγής δεν θα μπορούσε να εκτελεστεί έως ότου η πρώτη συναλλαγή ολοκληρωθεί. Αντίθετα σε περιβάλλον με επίπεδο απομόνωσης επαναλαμβανόμενου διαβάσματος δεν τοποθετείται κανένα κλείδωμα στα αποτελέσματα του πρώτου ερωτήματος , επιτρέποντας έτσι νέα δεδομένα να εισαχθούν, με αποτέλεσμα η δεύτερη εκτέλεση του ίδιου ερωτήματος στην πρώτη συναλλαγή να περιλαμβάνει τα νέα αυτά δεδομένα.,,

Επίπεδο απομόνωσης	**Πρόχειρο γράψιμο**	**Μη-επαναλαμβανόμενε αναγνώσεις**	**Ανάγνωση «Φάντασμα»**
Διάβαμα Μη-ολοκληρωμένων	✓	✓	✓
Διάβασμα Ολκληρωμένων	✖	✓	✓
Επαναλαμβανόμενο Διάβασμα	✖	✖	✓
Σειριοποίηση	✖	✖	✖

Πίνακας 3.2 *Επίπεδα απομόνωσης και φαινόμενα ανάγνωσης*

3.4 Χειρισμός επιπέδων απομόνωσης

Υπάρχουν διάφορες πολιτικές ελέγχου συγχρονικότητας που μπορούν να χρησιμοποιηθούν ώστε να διασφαλιστεί ότι, όταν εκτελούνται πολλαπλές συναλλαγές ταυτόχρονα, δημιουργούνται μόνο αποδεκτά χρονοδιαγράμματα, ανεξάρτητα από τον τρόπο που το εκάστοτε λειτουργικό σύστημα μοιράζει τους πόρους (όπως τον χρόνο της CPU) μεταξύ τν συναλλαγών. Ο στόχος πολιτικών ελέγχου συγχρονικότητας είναι να παρέχουν υψηλό βαθμό συγχρονικότητας, αλλά να διασφαλίζουν ότι όλα τα χρονοδιαγράμματα που μπορεί να δημιουργηθούν να είναι σειριοποιήσιμα ως προς τις διενέξεις ή την προβολή, με δυνατότητα ανάκαμψης και χωρίς διαδοχικές αναιρέσεις.

Ένα απλοϊκό παράδειγμα μιας πολιτικής ελέγχου συγχρονικότητας, είναι η απαίτηση κλειδώματος ολόκληρης της βάσης δεδομένων πριν από την εκτέλεση κάποιας συναλλαγής και η ελευθέρωση αυτού μετά το πέρας της. Κατά το διάστημα που μια συναλλαγή διατηρεί το κλείδωμα, καμία άλλη δεν επιτρέπεται να ξεκινήσει, επομένως όλες οι υπόλοιπες περιμένουν να ελευθερωθεί. Ως αποτέλεσμα αυτής της πολιτικής, μόνο μία συναλλαγή μπορεί να εκτελεστεί κάθε φορά κι έτσι τα χρονοδιαγράμματα που δημιουργούνται είναι σειριακά, που διατηρούν (προφανώς) την σειριοποιησιμότητα και δεν απαιτούν διαδοχικές αναιρέσεις. Μία τέτοια πολιτική ελέγχου συγχρονικότητας, είναι εύκολα αντιληπτό ότι μπορεί να καταλήξει σε άσχημη απόδοση, καθώς εξαναγκάζει όλες τις συναλλαγές να περιμένουν την ολοκλήρωση των προηγουμένων, για να ξεκινήσουν.

Παρακάτω αναλύονται ορισμένες πολιτικές ελέγχου συγχρονικότητας καλύτερης απόδοσης από αυτόν που περιγράψαμε. Υπάρχουν αρκετοί μηχανισμοί ελέγχου της συγχρονικότητας, ωστόσο κανένας από αυτούς δεν είναι καθαρά καλύτερος από τους υπόλοιπους. Κάθε μηχανισμός έχει τα πλεονεκτήματα του και η επιλογή τους εξαρτάται από τις απαιτήσεις της εκάστοτε εφαρμογής. Οι δύο όμως που έχουν την μεγαλύτερη επιλεξιμότητα από τους υπόλοιπους είναι το *κλείδωμα δύο φάσεων* και η *απομόνωση στιγμιότυπων*.

3.4.1 Πρωτόκολλα Βασισμένα σε Κλειδώματα

Ένας τρόπος να διασφαλίσουμε την απομόνωση είναι η απαίτηση προσπέλασης των δεδομένων με έναν μεταξύ τους αποκλειόμενο τρόπο. Αυτό σημαίνει, ότι ενώ μια συναλλαγή χρησιμοποιεί κάποια δεδομένα, καμία άλλη δεν επιτρέπεται να χρησιμοποιήσει τα δεδομένα αυτά. Η πιο συνηθισμένη πρακτική που χρησιμοποιείται για την ικανοποίηση αυτής της απαίτησης, είναι η επίτρεψη σε κάποια συναλλαγή να έχει πρόσβαση σε κάποιο στοιχείο δεδομένων, μόνο εάν την συγκεκριμένη χρονική στιγμή, έχει ένα κλείδωμα (lock) σε αυτό.

Τα δεδομένα μπορούν να κλειδωθούν σε διάφορες καταστάσεις όπως είναι η *κοινόχρηστη* ή η *αποκλειστική*. Αν κάποια συναλλαγή T_i πετύχει κάποιο κοινόχρηστο κλείδωμα (S) σε κάποιο στοιχείο Q, τότε η T_i επιτρέπεται να διαβάσει, αλλά όχι να γράψει στο Q. Αντίθετα αν η συναλλαγή T_i πετύχει ένα αποκλειστικό κλείδωμα (X) στο στοιχείο Q, τότε μπορεί τόσο να διαβάσει όσο και να γράψει στο στοιχείο αυτό.

Υπάρχει απαίτηση από κάθε αίτημα συναλλαγής, να *ζητά (request)* ένα κλείδωμα σε κατάλληλη κατάσταση για το στοιχείο Q που επιθυμεί να χρησιμοποιήσει, ανάλογα με τον

τύπο των λειτουργιών που επιθυμεί να εκτελεστούν στο Q. Η συναλλαγή κάνει το αίτημα στον διαχειριστή ελέγχου συγχρονικότητας και μπορεί να προχωρήσει στην εκτέλεση της λειτουργίας της μόνο αφού αυτός *επιτρέψει (grants)* το κλείδωμα στην συναλλαγή. Με την χρήση των δύο προαναφερθέντων καταστάσεων κλειδωμάτων δίνεται η δυνατότητα σε πολλαπλές συναλλαγές να διαβάζουν το ίδιο στοιχείο δεδομένων, αλλά η δυνατότητα γραψίματος περιορίζεται σε μία συναλλαγή κάθε φορά.

Θεωρούμε αυθαίρετες καταστάσεις κλειδώματος A, B και την συναλλαγή T_i η οποία ζητάει ένα κλείδωμα κατάστασης A για το στοιχείο Q, στο οποίο η συναλλαγή T_j ($T_i \neq T_j$) διατηρεί ένα κλείδωμα κατάστασης B. Αν η T_i μπορεί να επιτύχει αμέσως ένα κλείδωμα για το Q, ανεξαρτήτως από την παρουσία του B, τότε λέμε ότι η κατάσταση A είναι *συμβατή* με την κατάσταση B. Οι σχέσεις συμβατότητας μεταξύ των δύο καταστάσεων κλειδώματος παρουσιάζονται στον παρακάτω πίνακα (comp) όπου ένα στοιχείο comp(A,B) παίρνει την τιμή *αληθές (true)* αν και μόνο αν, η κατάσταση A είναι συμβατή με την κατάσταση B.

	S	**X**
S	True	True
X	False	False

Πίνακας 3.3 *Πίνακας comp συμβατότητας κλειδωμάτων*

Για πρόσβαση σε ένα στοιχείο, η συναλλαγή T_i πρέπει πρώτα να κλειδώσει αυτό το στοιχείο. Αν τα δεδομένα είναι ήδη κλειδωμένα από άλλη συναλλαγή σε μία μη συμβατή κατάσταση, ο διαχειριστής ελέγχου συγχρονικότητας δεν θα παραχωρήσει το κλείδωμα μέχρι να ελευθερωθούν όλα τα μη συμβατά κλειδώματα των άλλων συναλλαγών. Έτσι, η T_i θα πρέπει να περιμένει μέχρι να ελευθερωθούν τα μη συμβατά κλειδώματα, που υπάρχουν από τις άλλες συναλλαγές.

Η πολιτική χρήσης κλειδωμάτων για την εξασφάλιση της συγχρονικότητας μπορεί να καταλήξει σε κάποια ανεπιθύμητη κατάσταση. Στο παρακάτω παράδειγμα η συναλλαγή T_1 έχει ένα αποκλειστικό κλείδωμα στο στοιχείο B και η συναλλαγή T_2 ζητάει ένα κοινόχρηστο κλείδωμα σε αυτό και περιμένει η T_1 να το ελευθερώσει. Αντίστοιχα, η T_2 διατηρεί κοινόχρηστο κλείδωμα στο στοιχείο A, η T_1 ζητάει ένα αποκλειστικό κλείδωμα στο A και περιμένει να το ξεκλειδώσει η T_2. Έτσι εμφανίζεται μία κατάσταση, η οποία ονομάζεται *αδιέξοδη (deadlock),* όπου καμία από τις δύο συναλλαγές δεν μπορεί να προχωρήσει στην κανονική της εκτέλεση.

T1	T2
Lock –X (B)	
Read (B)	
B := B – 50	
Write (B)	
	Lock –S (A)
	Read (A)
	Lock –S (B)
Lock –X (A)	

Πίνακας 3.4 *Εμφάνιση αδιέξοδης κατάστασης στις συναλλαγές T1 και T2*

Όταν εμφανίζεται αδιέξοδη κατάσταση, το σύστημα πρέπει να αναιρέσει κάποια από τις δύο συναλλαγές. Αφού αναιρεθεί μία συναλλαγή, ξεκλειδώνονται τα δεδομένα που είχαν κλειδωθεί από αυτή την συναλλαγή. Τα δεδομένα, αυτά, είναι πλέον διαθέσιμα στην άλλη συναλλαγή, η οποία μπορεί να συνεχίσει κανονικά την εκτέλεση της. Γενικότερα, οι αδιέξοδες καταστάσεις είναι αναγκαίο κακό των κλειδωμάτων, αν θέλουμε να αποφευχθούν ασυνεπείς καταστάσεις. Μια αδιέξοδη κατάσταση είναι προτιμότερη από ασυνεπείς καταστάσεις, αφού μπορεί να διαχειριστεί με αναιρέσεις των συναλλαγών, ενώ οι ασυνεπείς καταστάσεις μπορεί να καταλήξουν σε πραγματικά προβλήματα που δεν μπορούν να αντιμετωπιστούν από τις βάσεις δεδομένων.

3.4.2 Πρωτόκολλα Βασισμένα σε Χρονοσφραγίδες

Μία άλλη μέθοδος γα τον προσδιορισμό της σειριοποιησιμότητας είναι η εκ των προτέρων επιλογή σειράς εκτέλεσης των συναλλαγών. Η πιο κοινή μέθοδος πραγματοποίησης αυτού, είναι η χρήση σχήματος διάταξης με χρονοσφραγίδες (timestamp ordering).

Με κάθε συναλλαγή T_i του συστήματος, συσχετίζουμε μία μοναδική, σταθερή σφραγίδα χρόνου, η οποία δηλώνεται ως $TS(T_i)$. Αυτή η σφραγίδα ανατίθεται από την βάση δεδομένων (ή από το σύστημα διαχείρισης της βάσης δεδομένων) πριν την εκκίνηση της εκτέλεσης της T_i. Για την τιμή της χρονοσφραγίδας ανάθεσης, υπάρχουν δύο συνηθισμένες περιπτώσεις: η ανάθεση της τιμής του χρόνου του συστήματος εκείνης της στιγμής, ή η χρήση ενός λογικού μετρητή ο οποίος σε κάθε ανάθεση δίνει την επόμενη τιμή. Σε κάθε περίπτωση πρέπει να εξασφαλίζεται ότι αν μετά την T_i μπει στο σύστημα μία άλλη συναλλαγή T_j, τότε $TS(T_i) < TS(T_j)$.

Καθώς οι χρονοσφραγίδες προσδιορίζουν την σειριοποιησιμότητα, πρέπει το σύστημα να εξασφαλίσει ότι το παραγόμενο χρονοδιάγραμμα είναι ισοδύναμο με ένα σειριακό χρονοδιάγραμμα κατά το οποίο, στο προηγούμενο παράδειγμα, η T_i εμφανίζεται πριν την T_j. Για την επίτευξη αυτού, ορίζονται δύο τύποι χρονοσφραγιδών: η χρονοσφραγίδα εγγραφής (W-Timestamp), καθώς και η χρονοσφραγίδα ανάγνωσης (R-Timestamp). Επιπλέον, ορίζονται οι παρακάτω περιπτώσεις για διαχείριση τυχών διενέξεων:

1. Η T_i προσπαθεί να διαβάσει ένα δεδομένο Q.
    a. Αν $TS(T_i) < W\text{-}Timestamp(Q)$, τότε ή T_i πρέπει να διαβάσει την τιμή του Q, που έχει ήδη γραφτεί. Συνεπώς η λειτουργία ανάγνωσης απορρίπτεται και η T_i αναιρείται.
    b. Αν $TS(T_i) \geq W\text{-}Timestamp(Q)$, τότε εκτελείται η λειτουργία ανάγνωσης και το R-Timestamp(Q) παίρνει την τιμή MAX{ R-Timestamp(Q), $TS(T_i)$}
2. Η T_i προσπαθεί να γράψει ένα δεδομένο Q.
    a. Αν $TS(T_i) < R\text{-}Timestamp(Q)$, τότε ή τιμή του Q που παράγει η T_i χρειάστηκε προηγουμένως και το σύστημα υπέθεσε ότι η τιμή δεν θα παραχθεί ποτέ. Συνεπώς η λειτουργία εγγραφής απορρίπτεται και η T_i αναιρείται.
    b. Αν $TS(T_i) < W\text{-}Timestamp(Q)$, τότε η T_i προσπαθεί να γράψει μία παλιά τιμή του Q. Επομένως το σύστημα απορρίπτει την λειτουργία και αναιρεί την T_i.
    c. Διαφορετικά, το σύστημα επιτρέπει την διαδικασία εγγραφής και ορίζει την τιμή της W-Timestamp(Q) ως την τιμή του $TS(T_i)$.
3. Αν κάποια συναλλαγή T_i αναιρεθεί από τον έλεγχο συγχρονικότητας επειδή δεν έχει δώσει καμία εντολή ανάγνωσης ή εγγραφής, τότε το σύστημα της αναθέτει μία νέα χρονοσφραγίδα και την επανεκκινεί.

Το πρωτόκολλο διασφαλίζει απουσία αδιεξόδων καταστάσεων, αφού καμία συναλλαγή δεν περιμένει ποτέ. Ωστόσο, υπάρχει η πιθανότητα επ' αόριστον αναμονής των μεγάλων συναλλαγών, αν μια σειρά από σύντομες συναλλαγές σε διένεξη προκαλούν συνεχή επανεκκίνηση της μεγάλης συναλλαγής. Σε αυτήν την περίπτωση, που εντοπίζουμε συνεχείς επανεκκινήσεις κάποιας μεγάλης συναλλαγής, θα πρέπει οι μικρότερες που δημιουργούν διενέξεις να μπλοκαριστούν προσωρινά, ώστε να μπορέσει η συναλλαγή να τερματίσει.

3.4.3 Σχήματα Πολλαπλών Εκδόσεων και Απομόνωση Στιγμιότυπων

Διατηρώντας περισσότερες από μία εκδόσεις ενός στοιχείου δεδομένων, καθίσταται δυνατόν να επιτραπεί σε κάποια συναλλαγή να διαβάσει μια παλιά έκδοση ενός στοιχείου δεδομένων αντί μιας νεότερης έκδοσης που έχει γραφτεί από μία μη εκτελεσθείσα συναλλαγή ή από μία συναλλαγή που είναι αργότερα στη σειρά σειριοποίησης. Κατά την μέθοδο της απομόνωσης στιγμιότυπου (Snapshot isolation), κατά την εκκίνηση της, κάθε συναλλαγή αποκτά μία δική της έκδοση (ή στιγμιότυπο) της βάσης δεδομένων[1].

Έτσι κάθε συναλλαγή διαβάζει από την δική της έκδοση κι έτσι, απομονώνεται από τις ενημερώσεις που προκαλούνται από άλλες συναλλαγές. Αντίστοιχα οι αλλαγές που πραγματοποιεί και για τις οποίες ενημερώνει την βάση, εμφανίζονται μόνο στην δική της έκδοση και όχι στην ίδια τη βάση δεδομένων. Οι αλλαγές μεταφέρονται στην «πραγματική» βάση δεδομένων μόνο όταν εκτελεστεί η συναλλαγή.

Όταν μια συναλλαγή Τα μπαίνει σε κάποια μερικώς εκτελεσθείσα κατάσταση, προχωρά σε εκτελεσθείσα, μόνο αν καμία άλλη ταυτόχρονη συναλλαγή δεν έχει τροποποιήσει

[1] Προφανώς, δεν δημιουργείται από ένα αντίγραφο ολόκληρης της βάσης δεδομένων για κάθε συναλλαγή, αλλά διατηρούνται πολλαπλές εκδόσεις των αντικειμένων αυτών που επηρεάζονται.

δεδομένα που σκοπεύει να ενημερώσει η Τ. Οι συναλλαγές που δεν μπορούν να εκτελεστούν, διακόπτονται. Η απομόνωση στιγμιότυπου εξασφαλίζει ότι δεν χρειάζεται να περιμένουν ποτέ οι προσπάθειες ανάγνωσης δεδομένων, σε αντίθεση με το μοντέλο του κλειδώματος. Οι συναλλαγές που περιέχουν μόνο διαδικασίες ανάγνωσης δεν μπορούν να διακοπούν. Μόνο όσες τροποποιούν δεδομένα διατρέχουν έναν μικρό κίνδυνο διακοπής. Εφόσον κάθε συναλλαγή διαβάζει την δική της έκδοση της βάσης δεδομένων, η ανάγνωση των δεδομένων δεν προκαλεί καμία αναμονή στις επόμενες προσπάθειες ενημέρωσης από άλλες συναλλαγές. Και καθώς οι περισσότερες συναλλαγές περιέχουν μόνο διαδικασίες ανάγνωσης (ή οι διαδικασίες ανάγνωσης δεδομένων είναι σαφώς περισσότερες από τις αντίστοιχες τροποποίησης δεδομένων) η απομόνωση στιγμιότυπων προσφέρει μία σημαντική βελτίωση της απόδοσης σε σύγκριση με τα μοντέλα κλειδώματος.

Είναι πιθανόν δύο συναλλαγές που εκτελούνται ταυτόχρονα, να ενημερώνουν το ίδιο στοιχείο. Εφόσον κάθε μία δουλεύει στο δικό της στιγμιότυπο δεν γνωρίζει για τις αλλαγές της άλλης. Εάν επιτραπεί και στις δύο συναλλαγές αν γράψουν στην βάση, τότε η πρώτη ενημέρωση θα επικαλυφθεί από την δεύτερη, με αποτέλεσμα να έχουμε μία χαμένη ενημέρωση (lost update). Για την αντιμετώπιση του φαινομένου αυτού έχουν σχεδιαστεί δύο τεχνικές («Νίκη πρώτης ολοκλήρωση» και «Νίκη πρώτης ενημέρωσης») και βασίζονται στον έλεγχο της συναλλαγής ως προς ταυτόχρονες συναλλαγές.

4 Απαιτήσεις και Σχεδίαση

4.1 MongoDB και συναλλαγές

Παρά την ολοένα και αυξανομένη χρήση της MongoDB σε διάφορες εφαρμογές , ακόμα και μεγάλων πολυεθνικών εταιριών – (ebay, adobe, κ.α.) υπάρχει ακόμα έλλειψη στο κομμάτι των συναλλαγών. Στην Mongo κάθε λειτουργία εγγραφής σε μοναδικό έγγραφο, χαρακτηρίζεται από ατομικότητα –αν δεν ολοκληρωθεί πλήρως, τότε δεν θα υπάρχει καμία αλλαγή στην εικόνα της βάσης- , ωστόσο αν μια λειτουργία απαιτεί εγγραφές σε πολλαπλά έγγραφα, τότε η ατομικότητα της συναλλαγής αυτής δεν μπορεί να εγγυηθεί. Η κάθε εγγραφή είναι ατομική, αλλά όχι η συναλλαγή σαν σύνολο. Επομένως αν κάποια τροποποίηση σε κάποιο έγγραφο ολοκληρωθεί επιτυχώς, τότε δεν υπάρχει κάποια μέριμνα για αυτόματη αναίρεση της, σε περίπτωση που μια μελλοντική αλλαγή σε άλλο έγγραφο της ίδιας λειτουργίας εγγραφής αποτύχει. Καθώς η Mongo βασίζεται σε έγγραφα, τα οποία μπορούν να περιέχουν εμφωλευμένα υπο-έγγραφα, η ατομικότητα ανά έγγραφο προσφέρει μεγαλύτερη χρησιμότητα, απ’ ότι σε κάποια σχεσιακή βάση δεδομένων, όπου οι αλλαγές σε παραπάνω από έναν πίνακες αποτελούν συχνό φαινόμενο.

4.1.1 Ο τελεστής $isolated

Μπορεί, η Mongo να μην εξασφαλίζει από μόνη της, όλες τις απαιτήσεις για την υποστήριξη συναλλαγών, ωστόσο παρέχει τον τελεστή $isolated, με χρήση του οποίου μπορεί κανείς να απομονώσει μια λειτουργία εγγραφής που επηρεάζει πολλαπλά έγγραφα. Με αυτόν τον τρόπο, παρόλο που δεν θα αναιρεθεί οποιαδήποτε αλλαγή έχει ήδη πραγματοποιηθεί, αν κάτι πάει στραβά σε μετέπειτα στάδιο της διαδικασίας εγγραφής, οι αλλαγές αυτές δεν θα είναι εμφανείς στους υπόλοιπους χρήστες, ως ότου η διαδικασία ολοκληρωθεί η αποτύχει.

Αυτό επιτυγχάνεται με την απονομή ενός αποκλειστικού κλειδώματος πάνω στις συλλογές στις οποίες πραγματοποιούνται οι αλλαγές. Για το διάστημα κατά το οποίο η διεργασία διατηρεί το κλείδωμα αυτό, ο τελεστής μετατρέπει οποιοδήποτε μηχανισμό κλειδώματος των εγγράφων (όπως ο WiredTiger) σε μονονηματικό, αποκλείοντας την πρόσβαση σε άλλους χρήστες, μέχρι να ολοκληρωθεί η διαδικασία, είτε να αποτύχει. Ωστόσο ο τελεστής $isolated δεν μπορεί να εφαρμοστεί σε οριζόντια κατανεμημένες mongo βάσεις.

4.1.2 Ολοκλήρωση δύο φάσεων

Μία πρώτη λύση για την προσέγγιση της λειτουργικότητας των συναλλαγών στην Mongo, αποτελεί η ολοκλήρωση της συναλλαγής σε δύο φάσεις για την υποστήριξη αλλαγών σε πολλαπλά έγγραφα. Η ολοκλήρωση συναλλαγών σε δύο φάσεις, προσφέρει σημασιολογία που παρομοιάζει την συναλλακτική και όχι πλήρη υποστήριξη συναλλαγών. Αυτό σημαίνει ότι κάνοντας χρήση της ολοκλήρωσης δύο φάσεων, υπάρχει η εγγύηση για την συνοχή των δεδομένων, αλλά είναι πιθανόν σε κάποιο σημείο της εκτέλεσης της εφαρμογή να

επιστραφούν ενδιάμεσα δεδομένα κατά τη διάρκεια της ολοκλήρωσης ή της αναίρεσης κάποιας συναλλαγής.

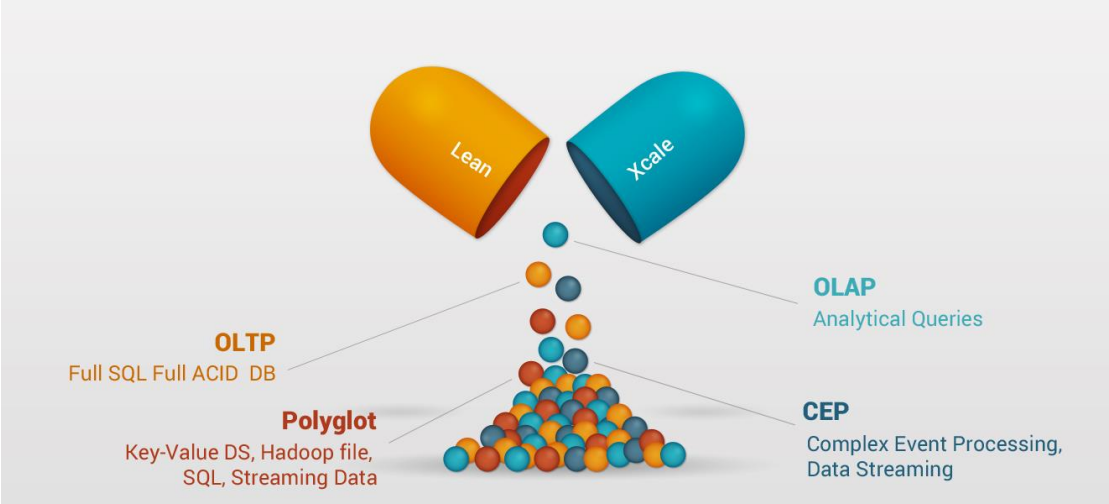
4.1.3 Έλεγχος Συγχρονικότητας

Ο έλεγχος συγχρονικότητας επιτρέπει σε πολλαπλές εφαρμογές να τρέχουν ταυτόχρονα χωρίς φόβο για ασυμβατότητα δεδομένων ή διενέξεις (conflicts). Μία προσέγγιση αποτελεί ο προσδιορισμός της αναμενόμενης τιμής κάποιου πεδίου στην εντολή για την διαδικασία εγγραφής, έτσι με βάση αυτό θα . Μια παραλλαγή της προσέγγισης αυτής υλοποιείται κατά την ολοκλήρωση σε δύο φάσεις, όπου το αναγνωριστικό της εφαρμογής, καθώς και η αναμενόμενη εικόνα των δεδομένων παρατίθενται κατά την διαδικασία εγγραφής.

Μία εναλλακτική προσέγγιση είναι η δημιουργία μοναδικού ευρετηρίου σε κάποιο πεδίο το οποίο μπορεί να έχει μοναδικές τιμές. Έτσι, αποφεύγεται η δημιουργία διπλότυπων από οι εισαγωγές ή τροποπαύσεις δεδομένων. Μοναδικό ευρετήριο, μπορεί επίσης να οριστεί σε συνδυασμό πεδίων, διασφαλίζοντας την μοναδικότητα του συνδυασμού των τιμών των πεδίων αυτών. Πάνω σε αυτήν την προσέγγιση έχει βασιστεί το CoherentPaas, καθώς και το Omid της Yahoo (για βάσεις Hbase).

4.1.4 LeanXcale

[6] Η LeanXcale είναι μία πλατφόρμα άμεσης διαχείρισης μεγάλων δεδομένων, η οποία μπορεί να κλιμακωθεί τόσο σε όγκο δεδομένων όσο σε ταχύτητα επεξεργασίας, αλλά και σε ποικιλομορφία δεδομένων. Μπορεί να υποστηρίξει εκατοντάδες terabytes σε όγκο δεδομένων, εκατομμύρια συναλλαγές ανά δευτερόλεπτο και επιπλέον υποστηρίζει μια μεγάλη ποικιλία δεδομένων (δομημένα, μη-δομημένα, ζεύγη κλειδιών-τιμών, ροές).

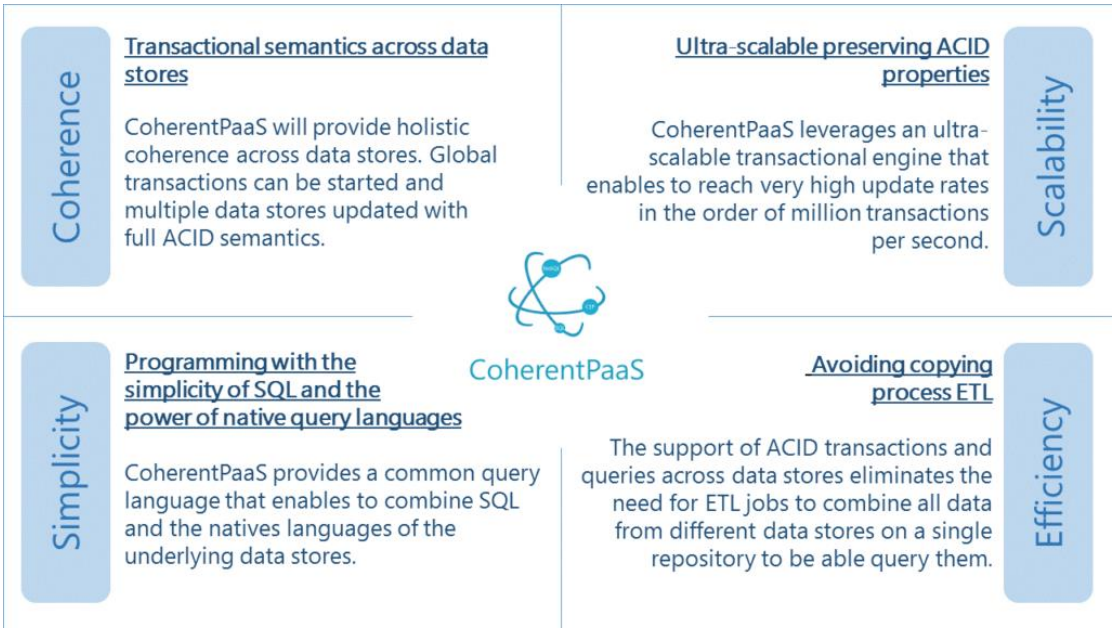


Πίνακας 4.1 *Ultra-Scalable Full ACID Full SQL Database*

Η LeanXcale αποτελεί μία υπέρ-κλιμακούμενη επιλογή SQL βάσης δεδομένων (OLTP), που υποστηρίζει πλήρως τα ACID χαρακτηριστικά μιας συναλλακτικής βάσης δεδομένων. Το κύριο χαρακτηριστικό της είναι ότι μπορεί να λειτουργήσει σε έναν μοναδικό κόμβο, αλλά ακόμα και σε εκατοντάδες και η δυνατότητα της να επεξεργαστεί πολύ μεγάλο αριθμό συναλλαγών. Η λειτουργία της βασίζεται σε μία νέα προσέγγιση, η οποία μπορεί να προσφέρει συνοχή στα συναλλακτικά δεδομένα, ενώ ταυτόχρονα μπορεί να κλιμακωθεί ώστε να εξυπηρετεί πολύ μεγάλο πλήθος συναλλαγών.

4.1.5 CoherentPaas

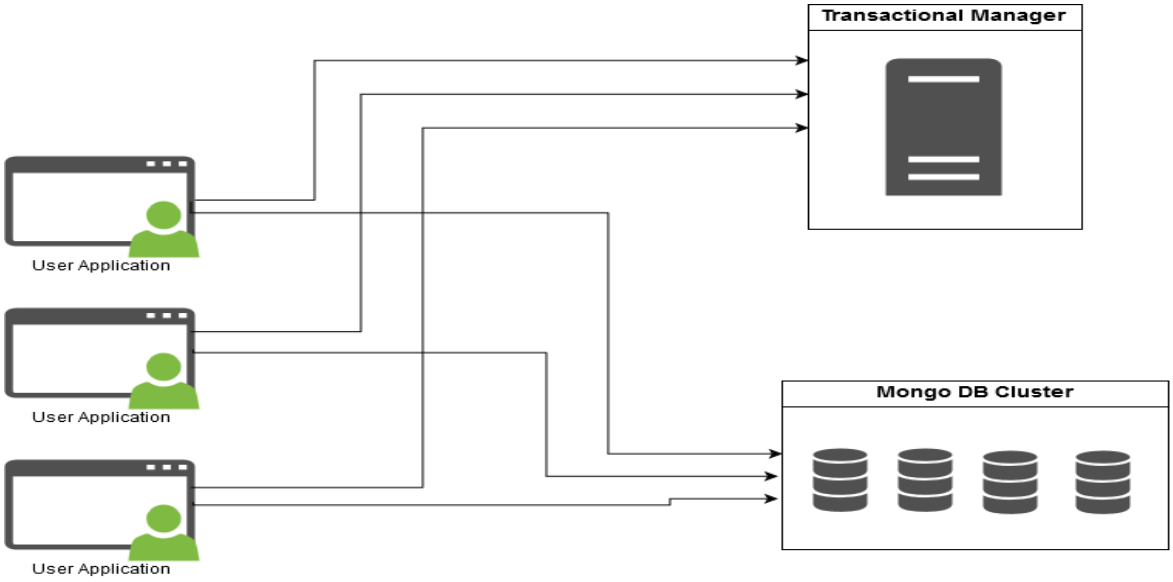
[11] Το CoherentPaas αποτελεί ένα έργο το αναπτύσσεται από το Ινστιτούτο Συστημάτων Επικοινωνιών και Πληροφορικής σε συνεργασία με 11 εταιρίες και ακαδημαϊκά ιδρύματα και βασίζεται στην πλατφόρμα του LeanXcale. Σαν στόχο έχει να ανταποκριθεί στις σύγχρονες απαιτήσεις για το χτίσιμο εφαρμογών νέφους μεγάλης ή μεταβλητής κλίμακας. Αποσκοπεί στο να λύσει ένα από τα μεγαλύτερα προβλήματα που αντιμετωπίζουν οι προγραμματιστές εφαρμογών νέφους, την ανάγκη να χρησιμοποιήσουν μια σειρά από τελείως διαφορετικές και ασύνδετες μεταξύ τους τεχνολογίες διαχείρισης δεδομένων νέφους. Τα κυριότερα προβλήματα που μπορεί να προκύψουν λόγω αυτού, είναι η έλλειψη συνοχής των δεδομένων (data coherence), η μεγάλη προσπάθεια που καταβάλλεται για την ανάπτυξη ερωτημάτων (queries) στις διάφορες δομές αποθήκευσης που βρίσκονται ανά το νέφους και η ανάγκη για χειροκίνητο προγραμματισμό τους και τέλος η δυσκολία πραγματοποίησης αποσφαλμάτωσης επίδοσης (performance debugging) σε εφαρμογές που εκτελούνται σε διάφορα μέρη του νέφους.



Εικόνα 4.1 *Οι πυλώνες του CoherentPaas*

Αρχικά, παρέχονται ολιστικές συναλλαγές για όλες τις ενέργειες αποθήκευσης δεδομένων. Αυτό επιτυγχάνεται με χρήση ενός διαχειριστή συναλλαγών ο οποίος επιτρέπει συναλλαγές με οποιοδήποτε συνδυασμό δεδομένων αποθήκευσης παρέχοντας έγκυρη εφαρμογή της ACID σημασιολογίας. Επιπλέον, στοχεύει στην 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 εξαφανίζοντας την ανάγκη χειροκίνητης ανάπτυξης ερωτημάτων. Καθώς ορισμένες δομές αποθήκευσης έχουν κάποια εξειδικευμένη διεπαφή επικοινωνίας ή γλώσσα ερωτημάτων, απαραίτητη για την επίτευξη υψηλών επιδόσεων, η κοινή γλώσσα ερωτημάτων που προσφέρει, μπορεί να ενσωματώσει υποερωτήματα τα οποία είναι εξ' ολοκλήρου γραμμένα στη γλώσσα της εκάστοτε δομής. Έτσι γίνεται πλήρης αξιοποίηση των γλωσσών ερωτημάτων της κάθε γλώσσας, αλλά και της εκφραστικότητας της SQL ώστε να επιτύχει το καλύτερο δυνατό αποτέλεσμα. Η μηχανή της γλώσσας ερωτημάτων είναι ενσωματωμένη με τις ολιστικές συναλλαγές για να εγγυηθούν πλήρη συνοχή των συναλλαγών.

Αναφορικά με την δυσκολία αποσφαλμάτωσης, το CoherentPaas, προσφέρει ένα πλαίσιο ενσωματωμένο στην πλατφόρμα του, όπου κάθε μέθοδος και κάθε κλήση σε κάποια δομή αποθήκευσης νέφους καταγράφεται και μετράται ο χρόνος εκτέλεσης του. Έτσι καθίσταται δυνατόν να εντοπιστούν τα φαινόμενα bottleneck, δηλαδή τα σημεία εκείνα της εφαρμογής που περιορίζουν την «ροή» της και μειώνουν την συνολική της απόδοση.



Εικόνα 4.2 *Η επικοινωνία των εφαρμογών χρηστών με τον διαχειριστή συναλλαγών και την κατανεμημένη βάση MongoDB*

4.1.5.1 Βασικά συστατικά του CoherentPaas

Τα βασικά μέρη από τα οποία αποτελείται η εφαρμογή CoherentPaas είναι:

- Εφαρμογή χρήστη
- MongoClient
- MongoDB
- LTMClient
- TransactionalManager

Εφαρμογή χρήστη καλείται η εφαρμογή μέσα από την οποία ο χρήστης μπορεί να δημιουργήσει, να ολοκληρώσει ή να ματαιώσει συναλλαγές. Χρησιμοποιεί τα υπόλοιπα μέρη της υλοποίησης CoherentPaas, είτε άμεσα είτε έμμεσα.

MongoClient είναι ένα μοναδικό αντικείμενο (singleton), το οποίο χρησιμοποιεί η εφαρμογή χρήστη για να δημιουργήσει κάποιο αντικείμενο τύπου MongoDB. Όσον αφορά την εφαρμογή χρήστη, αυτή είναι η μοναδική λειτουργικότητα του συστατικού αυτού, καθώς στα επόμενα στάδια η συμμετοχή του προκαλείται από άλλα εσωτερικά συστήματα και όχι από την εφαρμογή χρήστη.

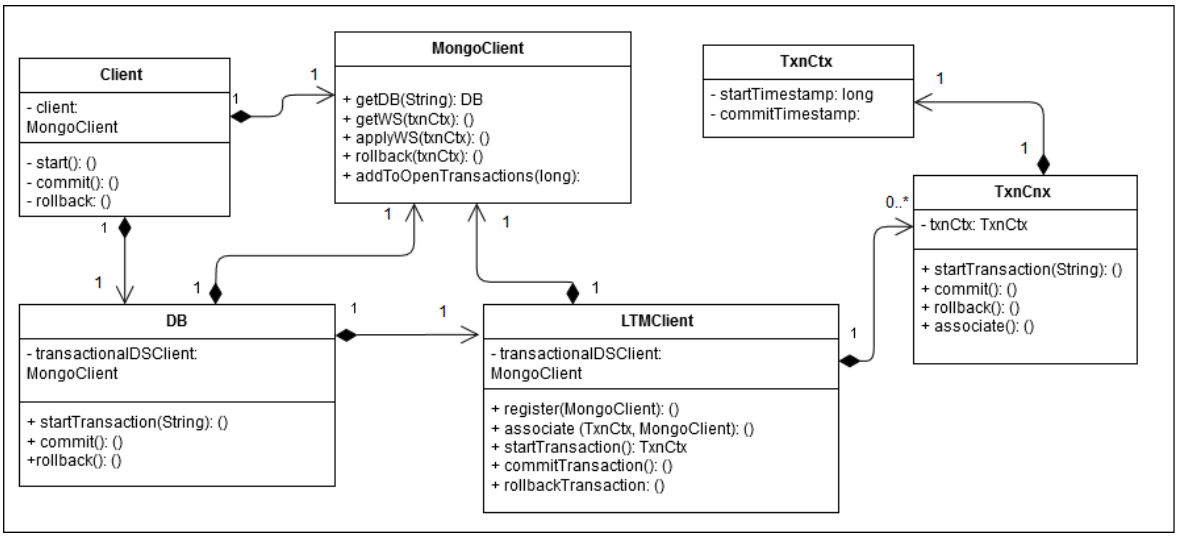
MongoDB είναι το αντικείμενο - διεπαφή πάνω στο οποίο ο χρήστης εκτελεί όλες τις λειτουργίες δημιουργίας, ολοκλήρωσης ή ματαίωσης κάποιας συναλλαγής. Αυτό, στη συνέχεια επικοινωνεί εσωτερικά με τα υπόλοιπα εμπλεκόμενα συστήματα χωρίς ο χρήστης να έχει εικόνα για το πώς διαχειρίζονται αυτές του τις ενέργειες.

Ο LTMClient είναι το συστατικό που είναι υπεύθυνο για την επικοινωνία με τον απομακρυσμένο διαχειριστή συναλλαγών, αλλά και για την υποστήριξη σε τοπικό επίπεδο ορισμένων λειτουργιών του διαχειριστή συναλλαγών, όπως για παράδειγμα ο έλεγχος για τυχών διενέξεις μεταξύ συναλλαγών. Επομένως όταν μια συναλλαγή ξεκινήσει από την εφαρμογή χρήστη αυτή καταγράφεται στον LTMClient. Επιπλέον, όταν κάποια συναλλαγή θελήσει να ολοκληρωθεί, τότε ο LTMClient θα προχωρήσει σε ελέγχους στην δομή με τις ενεργές συναλλαγές της συγκεκριμένης εφαρμογής χρήστη, για τυχών συγκρούσεις με άλλες συναλλαγές. Τέλος όταν μία συναλλαγή ολοκληρωθεί ή ματαιωθεί τότε αφαιρείται από την παραπάνω δομή με τις ενεργές συναλλαγές.

Τα παραπάνω συστατικά έχουν μία σχέση «ένα προς ένα», δηλαδή κάθε εφαρμογή χρήστη έχει μόνο έναν dsClient, ο οποίος δημιουργεί μοναδικό αντίγραφο τύπου MongoDB το οποίο επιστρέφει στην εφαρμογή χρήστη. Τέλος, το αντικείμενο αυτό τύπου MongoDB έχει μοναδική συσχέτιση με κάποιον LTMClient. Συνεπώς, για οποιαδήποτε αλληλεπίδραση μεταξύ των παραπάνω συστατικών, χρησιμοποιούνται τα μοναδικά αυτά αντίγραφα (instances) που έχουν δημιουργηθεί και αλληλοσυσχετιστεί.

Ο Transactional Manager είναι το τελευταίο σημαντικό στοιχείο της υλοποίησης και αποτελεί μία μονάδα, ξεχωριστή από την εφαρμογή χρήστη, η οποία υπάρχει βρίσκεται μοναδικά σε κάποια απομακρυσμένη υποδομή. Ο Transactional Manager είναι υπεύθυνος για την διαχείριση των συναλλαγών από όλες τις εφαρμογές χρήστη. Ο τρόπος που το επιτυγχάνει αυτό, είναι με την διατήρηση μίας δομής με τις ενεργές συναλλαγές, καθώς και με την απονομή χρονοσφραγίδων σε όλες τις συναλλαγές κατά την εκκίνηση τους και κατά

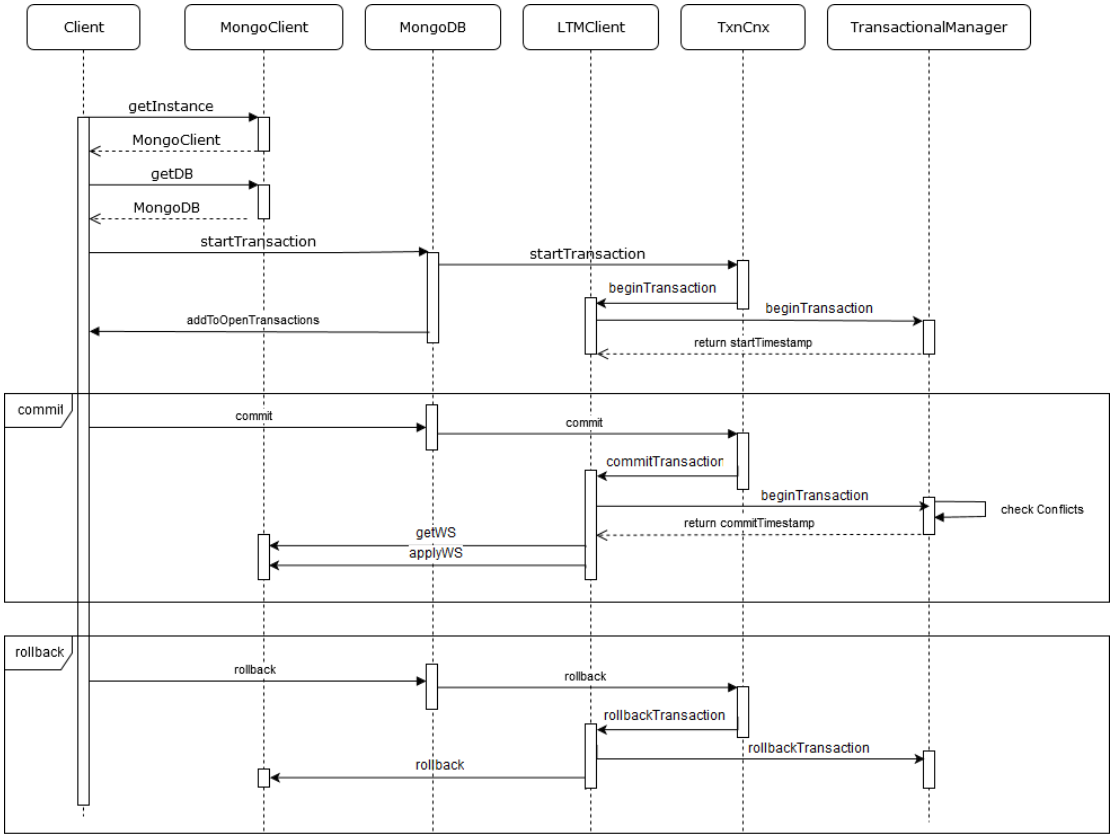
την επιτυχημένη ολοκλήρωση τους. Όταν ξεκινήσει κάποια συναλλαγή, πρώτα ελέγχεται αν τα δεδομένα που θέλει να προσπελάσει, βρίσκονται υπό επεξεργασία από κάποια άλλη εν ενεργεία συναλλαγή. Αν δεν ισχύει κάτι τέτοιο, τότε της απονέμεται μία χρονοσφραγίδα εκκίνησης, και το περιεχόμενο (συλλογή) που επιθυμεί να τροποποιήσει, «κλειδώνει» για μελλοντικές συναλλαγές, μέχρι αυτή να ολοκληρωθεί ή να ματαιωθεί.



Εικόνα 4.3 *Τα βασικά συστατικά της αρχιτεκτονικής του CoherentPaas*

4.1.5.2 Ο βασικός αλγόριθμος πίσω από την υλοποίηση του CoherentPaas

[4] Στη συνέχεια περιγράφεται περιληπτικά ο αλγόριθμος του CoherentPaas και τα βασικά στάδια επικοινωνίας μεταξύ των συστατικών της. Στο σχήμα που ακολουθεί φαίνονται τα κύρια συστατικά της εφαρμογής CoherentPaas, και τα στάδια επικοινωνίας μεταξύ τους ανά διεργασία.



Εικόνα 4.4 *Ο βασικός αλγόριθμος διαχείρισης συναλλαγών του CoherentPaas*

Αρχικά, κάποια εφαρμογή χρήστη (client) δημιουργεί ένα μοναδικό αντικείμενο (singleton) MongoClient, μέσω του οποίου δημιουργείται και ένα αντικείμενο τύπου DB. Πάνω στο DB, αυτό αντικείμενο ο χρήστης εκτελεί όλες τις λειτουργίες δημιουργίας, ολοκλήρωσης ή ματαίωσης κάποιας συναλλαγής, χωρίς να έχει εικόνα για τα πίσω συστήματα και το πώς διαχειρίζονται αυτές του τις ενέργειες.

Εσωτερικά το DB αντικείμενο δημιουργεί ένα αντικείμενο τύπου LTMClient και δημιουργείται μία συσχέτιση των δύο αυτών μερών μεταξύ τους αλλά και καθενός ξεχωριστά με τον MongoClient. Η σχέση τους είναι «ένα προς ένα», επομένως όταν κάποιο από αυτά θα χρειαστεί να χρησιμοποιήσει κάποια λειτουργικότητα κάποιου άλλου, τότε θα επικοινωνήσει με το μοναδικό αντικείμενο με το οποίο έχει συσχετιστεί.

Ο LTMClient είναι υπεύθυνος για την επικοινωνία με τον απομακρυσμένο διαχειριστή συναλλαγών, αλλά και η υποστήριξη σε τοπικό επίπεδο ορισμένων λειτουργιών του διαχειριστή συναλλαγών, όπως για παράδειγμα ο έλεγχος για τυχών διενέξεις μεταξύ συναλλαγών. Επομένως όταν μια συναλλαγή ξεκινήσει από την εφαρμογή χρήστη αυτή καταγράφεται στον LTMClient. Επιπλέον, όταν κάποια συναλλαγή θελήσει να ολοκληρωθεί, τότε ο LTMClient θα προχωρήσει σε ελέγχους στην δομή με τις ενεργές συναλλαγές της συγκεκριμένης εφαρμογής χρήστη, για τυχών συγκρούσεις με άλλες συναλλαγές. Τέλος όταν μία συναλλαγή ολοκληρωθεί ή ματαιωθεί τότε αφαιρείται από την παραπάνω δομή με τις ενεργές συναλλαγές.

Έπειτα, ξεκινάει μία συναλλαγή, με διαδοχικές κλήσεις προς την DB, τον LTMClient και ορισμένα εσωτερικά {components} (TnxCnx), η οποία καταλήγει σε κάποιον TransactionalManager ο οποίος της αναθέτει μια μοναδική χρονοσφραγίδα εκκίνησης. Στην συνέχεια η συναλλαγή με αναγνωριστικό την χρονοσφραγίδα που της ανατέθηκε, εισάγεται σε ένα σύνολο ανοιχτών συναλλαγών τόσο στον TransactionalManager, αλλά και στον MongoClient.

Στη συνέχεια με τη βοήθεια του DB δημιουργείται ένα σύνολο εγγραφών (writeset) το οποίο περιέχει ένα σύνολο από εντολές mongo, οι οποίες αποτελούν μία μονάδα (συναλλαγή). Όταν η εφαρμογή επιθυμεί να ολοκληρώσει την συναλλαγή, ο LTMClient αρχικά ελέγχει για τυχόν διενέξεις στο τοπικό σύνολο ανοιχτών συναλλαγών και στην συνέχεια επικοινωνεί με τον απομακρυσμένο TransactionalManager, ώστε να ολοκληρώσει διάφορους ελέγχους για διενέξεις από την δική του πλευρά και στην συνέχεια να του αναθέσει μια χρονοσφραγίδα ολοκλήρωσης. Όταν η συναλλαγή αποκτήσει την σφραγίδα αυτή, σημαίνει ότι μπορεί να ολοκληρωθεί χωρίς προβλήματα.

Αντίθετα, αν η εφαρμογή χρήστη αποφασίσει να ακυρώσει την συναλλαγή, τότε επικοινωνεί με τον TransactionalManager, ώστε να την αφαιρέσει από το σύνολο των ανοιχτών συναλλαγών και στην συνέχεια η συναλλαγή ματαιώνεται.

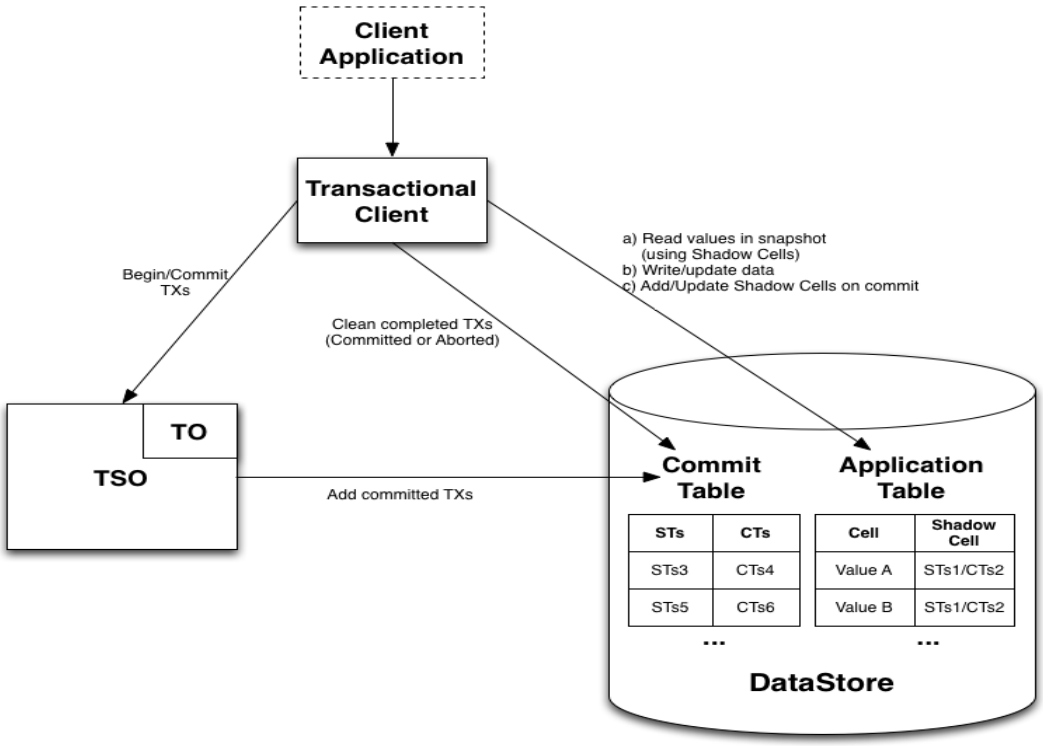
4.1.6 Omid Transactional Manager (Yahoo)

[7] Το Omid (“Optimistically transaction Management In Data-stores”) αποτελεί λογισμικό ανοιχτού κώδικα, ανεπτυγμένο από την Yahoo, που ξεκίνησε το 2011 και στόχος του είναι η υποστήριξη συναλλαγών σε μεγάλες «αποθήκες» δεδομένων στο πεδίο των NoSQL βάσεων. Η μέχρι στιγμής υλοποίηση του, παρέχει υποστήριξη μόνο για βάσεις δεδομένων Hbase.

Το omid καλείται να καλύψει το κενό της ανάγκης για υποστήριξη συναλλαγών από τις NoSQL βάσεις δεδομένων, προσφέροντας υποστήριξη συναλλαγών χωρίς την ανάγκη κλειδώματος (Lock-free) σε HBase βάση δεδομένων, διασφαλίζοντας την απομόνωση στιγμιότυπου (Snapshot Isolation).

4.1.6.1 Επισκόπηση της Αρχιτεκτονικής του Omid

Τα βασικά συστατικά της αρχιτεκτονικής του Omid, όπως διακρίνονται στο ακόλουθο σχήμα () περιγράφονται συνοπτικά στις παρακάτω παραγράφους:



Εικόνα 4.5 *Η αρχιτεκτονική του Omid*

Ο ρόλος και η λειτουργικότητα κάθε τμήματος της παραπάνω υλοποίησης:

- Timestamp Oracle (TO): Η μονάδα αυτή είναι υπεύθυνη για την ανάθεση ενός timestamp στις καινούριες συναλλαγές την στιγμή που ξεκινάνε ή ολοκληρώνονται
- The Status Oracle (TSO): Κύριος σκοπός αυτού του συστατικού είναι η επίλυση τυχών conflicts μεταξύ συναλλαγών
- Commit Table (CT): Ευθύνη του είναι η αποθήκευση μίας προσωρινής συσχέτισης των timestamp έναρξης με αυτών της λήξης για κάθε ολοκληρωμένη συναλλαγή
- Transactional Client: Επιτρέπει στις εφαρμογές να θεσπίζουν όρια για τις συναλλαγές και να γράφουν/διαβάζουν στην/από την πηγή δεδομένων (στην περίπτωση αυτή μία βάση HBase)
- Shadow Cells (SC): Η μονάδα αυτή επιτρέπει στους χρήστες (clients) να πραγματοποιούν διαδικασίες ανάγνωσης δεδομένων χωρίς την ανάγκη προσπέλασης του commit table

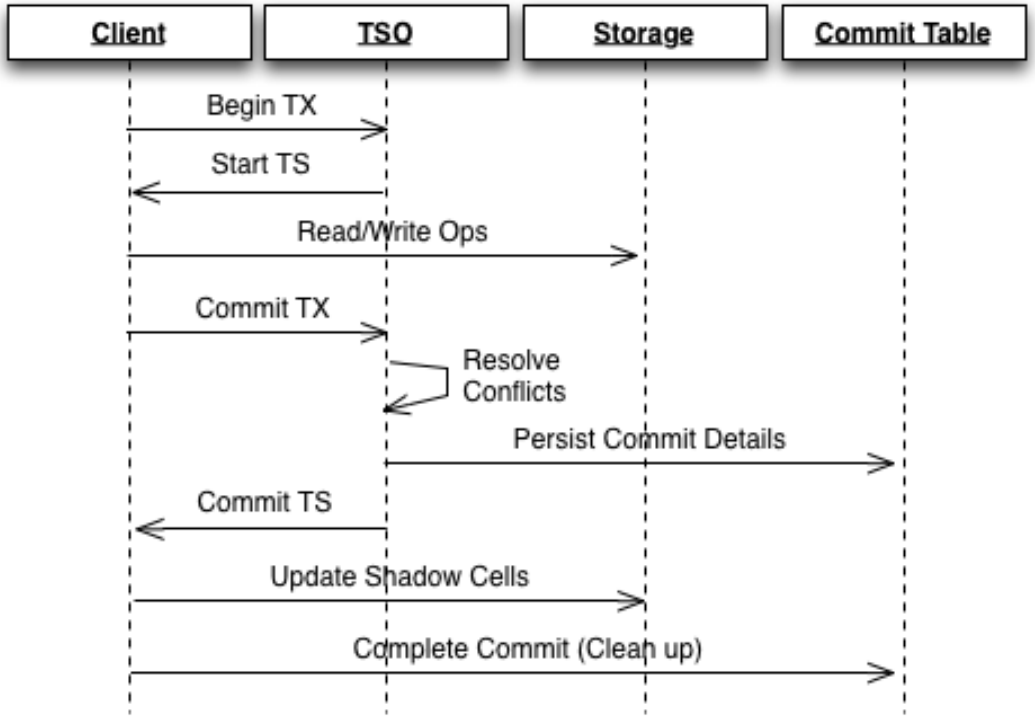
Η εφαρμογή αυτή, βασίζεται σε ένα κεντρικό σχήμα, στο οποίο ένας μόνο εξυπηρετητής (server) , ο TSO, παρακολουθεί τις γραμμές/κελιά που έχουν τροποποιηθεί από κάποια συναλλαγή και επιλύει conflicts μεταξύ των εγγραφών (writes).

Οι Transactional Clients (TC) δίνουν στις εφαρμογές των χρηστών την δυνατότητα να εκκινήσουν (begin), να κάνουν commit ή να ματαιώσουν (abort) κάποια συναλλαγή, προσφέροντας απομακρυσμένη επικοινωνία με τον TSO και επιτρέποντας την πραγματοποίηση διαδικασιών συναλλαγής στα δεδομένα που βρίσκονται αποθηκευμένα στο βάση δεδομένων.

Κατά τη δημιουργία μιας συναλλαγής, ο Timestamp Oracle (TO) της αναθέτει ένα timestamp έναρξης, το οποίο λειτουργεί ως αναγνωριστικό και χρησιμοποιείται από τον TSO για την εξασφάλιση της απομόνωσης στιγμιότυπου, αναγνωρίζοντας τυχόν διενέξεις στο σύνολο εγγραφών (writeset) μίας συναλλαγής με αυτά άλλων ταυτόχρονων συναλλαγών. Με το πέρας της ανίχνευσης των διενέξεων, ο TSO αναθέτει στην συναλλαγή μία νέα χρονοσφραγίδα ολοκλήρωσης και αποθηκεύει την συσχέτιση της με την προηγούμενη στο Commit Table (CT), πριν την επιστροφή της απόκρισης στον χρήστη. Ο χρήστης που λαμβάνει αυτή την απόκριση, προσθέτει ένα Shadow Cell σε κάθε κελί στο σύνολο εγγραφών της συναλλαγής, ώστε να μπορεί να έχει ένα έγκυρο στιγμιότυπο για τις περαιτέρω λειτουργίες ανάγνωσης, χωρίς να απασχολεί τον TSO.

4.1.6.2 Ο βασικός αλγόριθμος πίσω από την υλοποίηση του Omid

Στην παράγραφο αυτή αναλύεται ο αλγόριθμος που υλοποιεί το Omid για την διαχείριση συναλλαγών.



Εικόνα 4.6 *Ο βασικός αλγόριθμος διαχείρισης συναλλαγών*

Αρχικά, κάποια εφαρμογή χρήστη δημιουργεί μία νέα συναλλαγή, με την χρήση της διεπαφής της εφαρμογής (API) του Transactional Client. Η συναλλαγή παραλαμβάνει μια χρονοσφραγίδα εκκίνησης από τον TSO και στη συνέχεια, σύμφωνα με την λογική της εφαρμογής, πραγματοποιούνται οι αντίστοιχες ενέργειες ανάγνωσης και εγγραφής στο δοσμένο στιγμιότυπο.

Σε περίπτωση ανάγνωσης, αν τα κελιά περιέχουν shadow cells και η τιμή τους βρίσκεται στο στιγμιότυπο της συναλλαγής τότε αυτή διαβάζεται άμεσα. Αν λείπει η χρονοσφραγίδα ολοκλήρωσης από το shadow cell, ο Transactional Client θα προσπαθήσει να το βρει στον πίνακα ολοκλήρωσης. Αν το εντοπίσει εκεί, ενώ η τιμή του κελιού βρίσκεται στο στιγμιότυπο, τότε λαμβάνεται η τιμή, αυτή. Αν όμως, λείπει η χρονοσφραγίδα ολοκλήρωσης, τότε το shadow cell ξαναελέγχεται. Αν και πάλι δεν βρεθεί, τότε το κελί αγνοείται και λαμβάνεται από την πηγή των δεδομένων μία άλλη έκδοση του που συμβαδίζει με το στιγμιότυπο.

Όταν η εφαρμογή ολοκληρώσει την συναλλαγή. Ο Transaction Client επικοινωνεί με τον TSO για να ελέγξει την συναλλαγή για πιθανά conflicts. Αν το σύνολο εγγραφής δεν συγκρούεται με άλλες ταυτόχρονες συναλλαγές, η συναλλαγή ολοκληρώνεται, ο TSO

ανανεώνει τον πίνακα ολοκλήρωσης και ενημερώνει τον Transaction Client, ο οποίος με τη σειρά του, ενημερώνει τον χρήστη.

Τέλος, ο Transactional Client, μόλις ενημερωθεί για την ολοκλήρωση του commit, ενημερώνει τα shadow cells με τα απαραίτητα δεδομένα. Στη συνέχεια, μπορεί με ασφάλεια να αφαιρέσει την εγγραφή που προσέθεσε ο TSO στον πίνακα ολοκλήρωσης. Σε περίπτωση που ο TSO ανιχνεύσει κάποια διένεξη, η συναλλαγή ματαιώνεται, ο Transaction Client καθαρίζει τα δεδομένα που γράφτηκαν στην βάση και θα ενημερώσει την εφαρμογή του χρήστη.

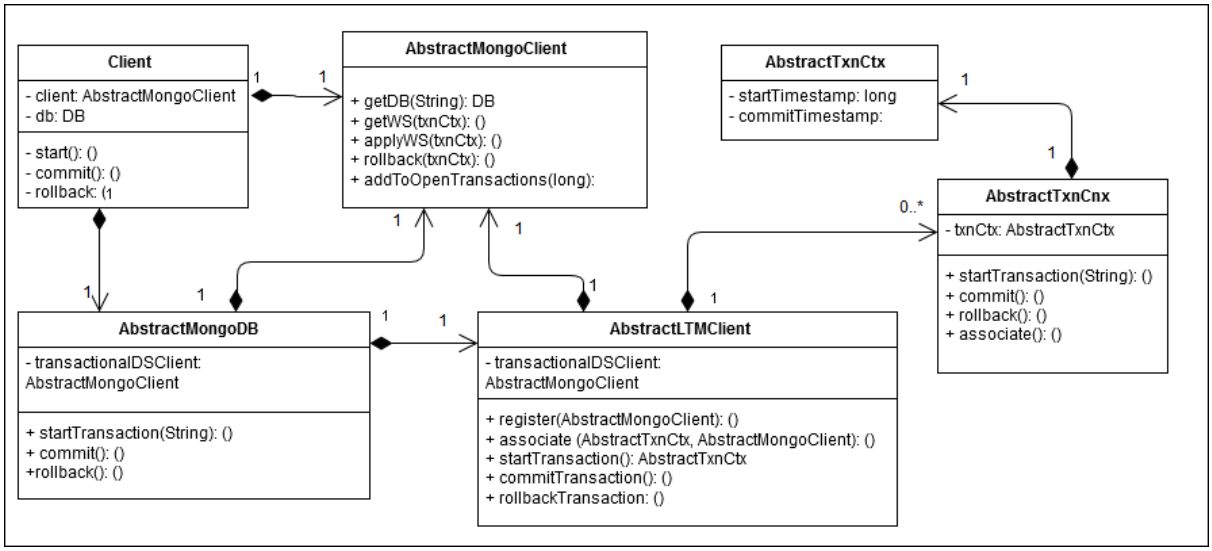
4.2 Ανάγκη για Abstraction

Και οι δύο παραπάνω προσεγγίσεις βασίζονται στην ίδια λογική, αυτή της ύπαρξης ενός απομακρυσμένου συστατικού (διαχειριστή συγχρονισμού συναλλαγών) υπεύθυνο για τον συγχρονισμού όλων των συναλλαγών ανεξαρτήτως από το πλήθος των εφαρμογών χρηστών. Και στις δύο περιπτώσεις ο εκάστοτε διαχειριστής διατηρεί την πληροφορία για τις ενεργές συναλλαγές, έχει τον συνολικό έλεγχο για διενέξεις και είναι υπεύθυνος για την ανάθεση χρονοσφραγιδών εκκίνησης και ολοκλήρωσης, ανεξάρτητα από τον επιμέρους έλεγχο συγχρονικότητας των εφαρμογών χρήστη.

Οι δύο αυτές υλοποιήσεις που αναφέραμε δεν είναι οι μοναδικές. Υπάρχουν πολλαπλές υλοποιήσεις διαχειριστών συναλλαγών με παρεμφερή λειτουργία, επομένως δημιουργήθηκε η ανάγκη για ανάπτυξη κάποιου είδους μηχανισμού, μέσω του οποίου θα μπορεί να επιλεχθεί οποιοσδήποτε από αυτούς για την διαχείριση των συναλλαγών της mongo με τον δικό του τρόπο.

4.3 Σχεδίαση

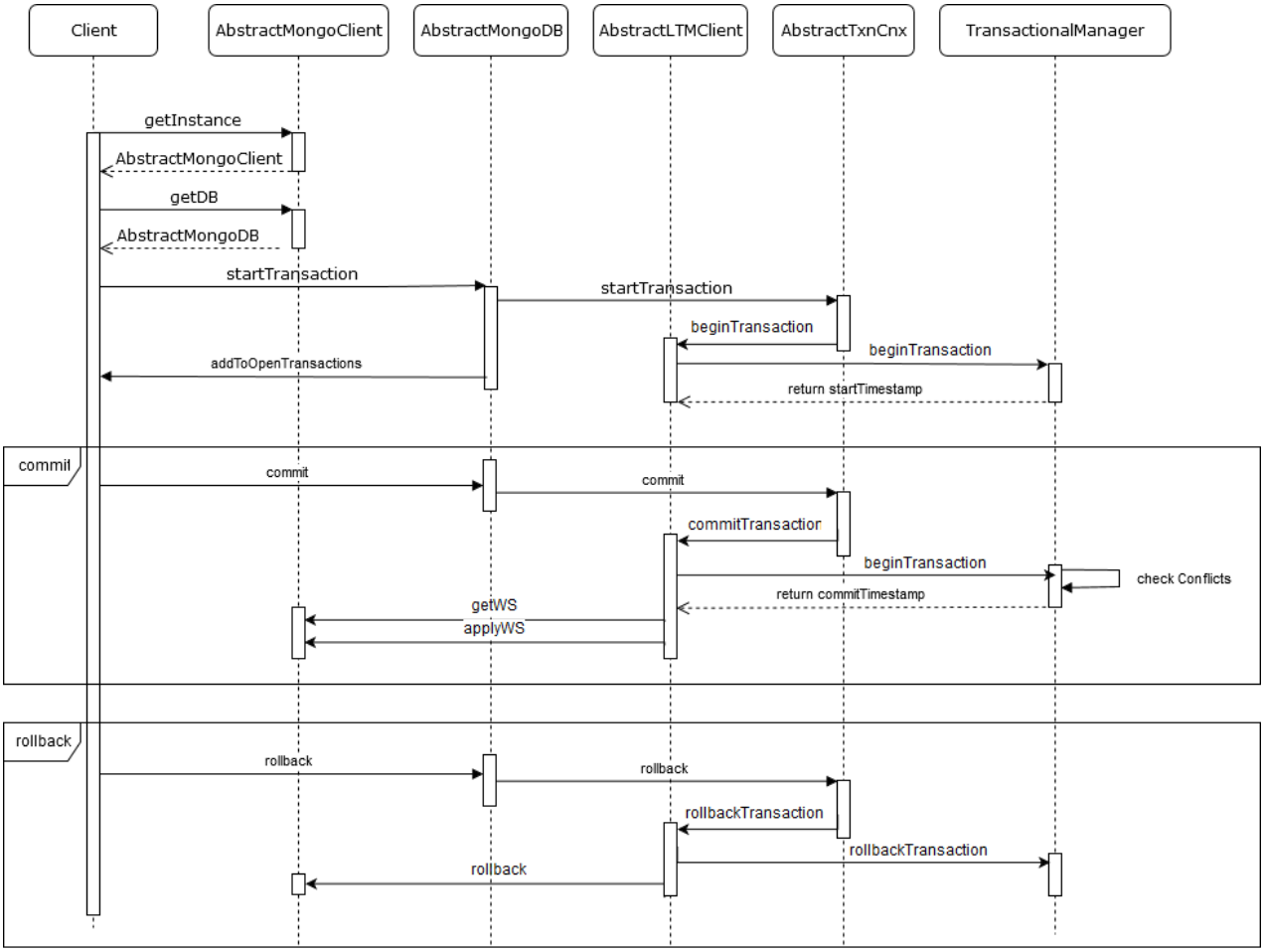
Βασισμένοι στην υλοποίηση του CoherentPaas, σχεδιάσαμε και υλοποιήσαμε ένα επιπλέον αφαιρετικό επίπεδο (level of abstraction), στα συστατικά τα οποία συμμετέχουν άμεσα στην διαδικασία των συναλλαγών και έχουν επικοινωνία με τον transactional manager. Έτσι θα δημιουργηθούν διεπαφές (interfaces) και αφηρημένες κλάσεις (abstract classes) οι οποίες θα κληρονομούνται και επεκτείνονται από τα ήδη υπάρχοντα συστατικά του CoherentPaas για χρήση με τον δικό της διαχειριστή συναλλαγών, ενώ αντίστοιχη δυνατότητα θα προσφέρεται και σε οποιονδήποτε απομακρυσμένο διαχειριστή, ο οποίος θα προσφέρει τις ζητούμενες λειτουργίες (ανάθεση χρονοσφραγιδών εκκίνησης και ολοκλήρωσης και έλεγχος για διενέξεις ανάμεσα στις ανοιχτές συναλλαγές).



Εικόνα 4.7 *Διάγραμμα κλάσεων*

Για κάθε συστατικό που λαμβάνει μέρος στην διαδικασία του ελέγχου και της υποβολής της συγχρονικότητας, τόσο τοπικά όσο και μέσω της επικοινωνίας με τον απομακρυσμένο διαχειριστή συναλλαγών, δημιουργήθηκε το κατάλληλο για την περίπτωση αφαιρετικό επίπεδο, ώστε πάνω σε αυτό να μπορέσει να προσκολληθεί τόσο η υπάρχουσα υλοποίηση όσο και οποιαδήποτε άλλη στο μέλλον. Όπως φαίνεται και στο παραπάνω σχήμα, δημιουργήθηκαν αφηρημένες κλάσεις (abstract classes), αλλά και διεπαφές (interfaces) , στις οποίες διατηρήθηκε η κοινή λογική που αφορά την λειτουργικότητα της εφαρμογής, αλλά παρέμειναν σε αφηρημένο επίπεδο όλα τα μέρη όπου υπάρχει επικοινωνία με κάποιον διαχειριστή συναλλαγών.

Οι κλάσεις από την προϋπάρχουσα υλοποίηση τροποποιήθηκαν ελάχιστα ώστε να κληρονομούν την βασική λειτουργικότητα τους από τα παραπάνω αφηρημένα συστατικά ενώ επεκτείνουν τις αφηρημένες μεθόδους επικοινωνίας με τον διαχειριστή συναλλαγών, διατηρώντας την πρότερη λειτουργία τους σε όλα τα επίπεδα. Επιπλέον, δημιουργήθηκαν νέες κλάσεις και αντικείμενα τα οποία υλοποιήθηκαν με παρόμοιο τρόπο, αλλά με τις κατάλληλες τροποποιήσεις, ώστε να κάνουν χρήση ενός καινούριου διαχειριστή συναλλαγών που δημιουργήθηκε στα πλαίσια της εργασίας αυτής.



Εικόνα 4.8 *Ακολουθιακό διάγραμμα διεργασιών συναλλαγής*

Τέλος, σχεδιάστηκε και υλοποιήθηκε με βάσει τα πρότυπα επικοινωνίας και διαχείρισης συναλλαγών της υλοποίησης του CoherentPaas, νέος διαχειριστής συναλλαγών (NtuaTransactionManager) για χρήση στην επαλήθευση της επιτυχίας της παραπάνω υλοποίησης. Ο νέος αυτός διαχειριστής δεδομένων θα έπρεπε να τρέχει απομακρυσμένα, να δέχεται συνδέσεις από τις εφαρμογές χρήστη, να έχει μόνιμα ολοκληρωμένη και έγκυρη εικόνα για όλες τις εφαρμογές χρήστη, όσον αφορά τις συναλλαγές που είτε θέλει να εκτελέσει η κάθε μία, είτε εκτελεί, είτε έχει εκτελέσει. Έχοντας αυτή την γνώση μπορεί να διακρίνει χωρίς απρόοπτα τυχόν συγκρούσεις μεταξύ συναλλαγών, αλλά και να τις δρομολογήσει κατάλληλα αναθέτοντας τους τις απαραίτητες χρονοσφραγίδες (εκκίνησης ή ολοκλήρωσης) δίνοντας τους έτσι το δικαίωμα είτε να εκκινήσουν είτε να ολοκληρωθούν.

5 Υλοποίηση

5.1 Τεχνολογίες και εργαλεία που χρησιμοποιήθηκαν

Η υλοποίηση πραγματοποιήθηκε με χρήση της γλώσσας Java και συγκεκριμένα της έκδοσης 7 (1.7.0_79). Ως λογισμικό συγγραφής του κώδικα επιλέχθηκε το IntelliJ της IDEA, ενώ για την διαχείριση των εξαρτήσεων αλλά και για την μετάφραση του κώδικα (compilation) χρησιμοποιήθηκε το Maven της Apache. Για την επικοινωνία με τον διαχειριστή συναλλαγών, υλοποιήθηκε σύστημα επικοινωνίας βασισμένο στο AVRO της Apache το οποίο αξιοποίησε την δυνατότητα που προσφέρει για σειριοποίηση των δεδομένων, αλλά και την RPC λειτουργικότητα που προσφέρει.

5.1.1 Η γλώσσα προγραμματισμού JAVA

Για την εργασία αυτή επιλέχθηκε η Java, ως γλώσσα προγραμματισμού, καθώς αποτελεί μία ιδανική και αρκετά διαδεδομένη γλώσσα προγραμματισμού για εφαρμογές μεγάλης κλίμακας. [9] Η Java αποτελεί μία αντικειμενοστραφή γλώσσα προγραμματισμού που αναπτύχθηκε το 1995 από την Sun Microsystems, βασισμένη στην σύνταξη των γλωσσών C/C++ και μιμούμενη αρκετά χαρακτηρίστηκα της δεύτερης.

“Write Once, Run Everywhere”

Το βασικό χαρακτηριστικό της που την κάνει να ξεχωρίζει από τις άλλες γλώσσες είναι ότι ο κώδικας της μεταφράζεται στο αποκαλούμενο “bytecode” το οποίο μπορεί να εκτελεστεί μονάχα πάνω στο εικονικό περιβάλλον της Java (JVM), το οποίο όμως υπάρχει σε όλες τις πλατφόρμες, συνεπώς (και σύμφωνα και με το μότο της Java) ο κώδικας της μπορεί να τρέξει παντού. Είναι όπως λέγεται ανεξάρτητη πλατφόρμας (platform independent). Έτσι ακόμα και αν ο χρήστης αποφασίσει να προχωρήσει σε ραγδαίες αλλαγές στην υποδομή της εφαρμογής, αλλάζοντας ακόμα και λειτουργικό σύστημα, δεν θα δημιουργήσει κανένα πρόβλημα στην εφαρμογή, η οποία θα μπορεί να συνεχίσει να λειτουργεί κανονικα.

Κλιμάκωση/Απόδοση/Αξιοπιστία

Σε επίπεδο εκτέλεσης, η Java είναι γνωστή για την ταχύτητα της, την αξιοπιστία της τόσο σε επίπεδο απόδοσης, αλλά και σε επίπεδο ασφάλειας. Ο μεταφρασμένος κώδικας της Java, το bytecode αποτελεί μορφή που δεν είναι ανθρωπίνως αναγνώσιμη και μεταφράζεται σε γλώσσα μηχανής κατά την διάρκεια της εκτέλεσης (runtime) ενώ πριν εκτελεστεί περνάει από έναν ελεγκτή κώδικα, ο οποίος αναζητά επιπλέον σύμβολα, ή κομμάτια κώδικα που δεν συμφωνούν με την υπόλοιπη εφαρμογή. Με αυτόν τον τρόπο το bytecode της Java προστατεύεται από σφάλματα, αλλά εξασφαλίζει, επιπλέον και ασφάλεια, καθώς εντοπίζεται οποιαδήποτε απόπειρα τροποποίησης ή προσθήκης κάποιου κακόβουλου ή εσφαλμένου (corrupted) κομματιού κώδικα. Επίσης, κάτι που την καθιστά ιδανική για εφαρμογές μεγάλης κλίμακας, είναι η κλιμάκωση που προσφέρει, γεγονός που την έχει καθιερώσει σαν επιλογή σε κατανεμημένες εφαρμογές μεγάλης κλίμακας. Αυτό γίνεται αντιληπτό από την επιλογή πολλών μεγάλων τεχνολογικών κολοσσών, όπως το Twitter, το Spotify, το Facebook και το eBay για χρησιμοποίηση της Java για να αναπτύξουν τις τεράστιες εφαρμογές τους, οι οποίες εξυπηρετούν εκατομμύρια χρήστες καθημερινά.

Προς τα πίσω συμβατότητα

Οι προσπάθειες της Java ώστε οι εφαρμογές της να μπορούν να τρέξουν παντού, δεν έμειναν μόνο στην υποστήριξη του Java περιβάλλοντος σε πλήθος λειτουργικών συστημάτων. Έχει δοθεί αρκετή προσοχή στο να μπορούν όλες οι μεταγενέστερες εκδόσεις της Java και οι αναβαθμίσεις της να υποστηρίζουν εφαρμογές που έχουν δημιουργηθεί με παλιότερες εκδόσεις. Έτσι ακόμα και αν αποφασιστεί κάποιο σύστημα να αναβαθμιστεί, ή να πραγματοποιηθεί κάποια ανακαίνιση σε επίπεδο υποδομών, η εφαρμογή θα εξακολουθήσει να λειτουργεί ακριβώς όπως και πριν.

Αυτόματη διαχείριση μνήμης

Η Java χρησιμοποιεί αυτόματη διαχείριση της μνήμης, μια διαδικασία ελέγχου στην μνήμη του σορού, αναγνώρισης των μη-χρησιμοποιούμενων αντικειμένων και διαγραφή τους. Σαν χρησιμοποιούμενο αντικείμενο θεωρείται οποιοδήποτε αντικείμενο, για το οποίο κάποιο κομμάτι της εφαρμογής διατηρεί κάποιον δείκτη πάνω του. Συνεπώς οποιοδήποτε αντικείμενο δεν χρησιμοποιείται πλέον από την εφαρμογή σε κανένα σημείο, μπορεί ασφαλώς να διαγραφεί και να ελευθερωθεί η μνήμη που καταλάμβανε. Σε άλλες γλώσσες προγραμματισμού ή δέσμευση και η αποδέσμευση της μνήμης αποτελούν χειροκίνητες διαδικασίες, ενώ στην Java, την δουλειά αυτή την αναλαμβάνει ο συλλέκτης απορριμμάτων.

5.1.2 Apache Maven

[10] Το Maven της Apache αποτελεί ένα εργαλείο διαχείρισης εφαρμογών λογισμικού και βασίζεται πάνω στην έννοια ενός μοντέλου αντικειμένου (POM). Το Maven είναι ικανό να διαχειριστεί την διαδικασία «χτισίματος» (build) ενός έργου, να αυτοματοποιήσει διαδικασίες αναφορών και δημιουργίας εγγράφων (documentation) βασισμένο πάνω στο προαναφερθέν αυτό μοντέλο. Ο κύριος στόχος του Maven είναι η προσφορά βοήθειας προς τον προγραμματιστή, ώστε να ολοκληρώσει την διαδικασία της δημιουργίας λογισμικού με τον μικρότερο δυνατό κόπο και στον λιγότερο δυνατόν χρόνο.

Το Maven καλύπτει δύο πολύ σημαντικές οπτικές της διαδικασίας ανάπτυξης λογισμικού. Αρχικά, περιγράφει πως θα χτιστεί το λογισμικό και στην συνέχεια περιγράφει και διαχειρίζεται τις εξαρτήσεις αυτού. Στο POM αρχείο περιγράφεται μέσα από μια δομή XML, η διαδικασία χτισίματος του λογισμικού, η εξάρτηση του από εξωτερικά συστήματα, η σειρά χτισίματος των επιμέρους συστατικών, αλλά και διάφορα πρόσθετα εργαλεία που μπορεί να λαμβάνουν μέρος στην διαδικασία. Αντίθετα με άλλα παλιότερα εργαλεία, όπως το Apache Ant, χρησιμοποιεί μία σειρά από συμβάσεις για την παραπάνω διαδικασία και μόνο τυχόν εξαιρέσεις χρειάζεται να οριστούν.

```
<project>

   <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
  <packaging>pom</packaging>

  <modules>
    <module>../my-module</module>
  </modules>

  <properties>
    <mavenVersion>2.1</mavenVersion>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-artifact</artifactId>
      <version>${mavenVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-project</artifactId>
      <version>${mavenVersion}</version>
    </dependency>
  </dependencies>
  ...
</project>
```

Εικόνα 5.1 *Παράδειγμα εγγράφου POM.xml*

Το Maven εντοπίζει και κατεβάζει δυναμικά τις ζητούμενες Java βιβλιοθήκες και τα διάφορα επιπλέον εργαλεία από προκαθορισμένα σημεία στο cloud, τα λεγόμενα maven repositories και τα διατηρεί σε ένα τοπικό αρχείο για μελλοντική χρήση. Το αρχείο αυτό εμπλουτίζεται με αντίγραφα από τα έργα που χτίζονται με χρήση του Maven στο τοπικό περιβάλλον. Παρόλο που η χρήση του έχει συνδυαστεί με την Java, το Maven μπορεί επίσης να χρησιμοποιηθεί σε έργα γραμμένα σε C#, Ruby, Scala, καθώς και σε άλλες γλώσσες προγραμματισμού.

Διαχείριση Εξαρτήσεων

Ένα από τα κεντρικά χαρακτηριστικά του Maven, που το έχουν κάνει τόσο διαδεδομένο, είναι η διαχείριση εξαρτήσεων. Ο μηχανισμός διαχείρισης εξαρτήσεων του Maven είναι οργανωμένος γύρω από ένα συντεταγμένο σύστημα αναγνώρισης επιμέρους στοιχείων, όπως βιβλιοθήκες και υπό-συστήματα, τα οποία ορίζονται στο αρχείο POM. Έπειτα, κατά την διαδικασία του χτισίματος του λογισμικού ο μηχανισμός αυτός αναλαμβάνει να αναζητήσει τα απαραίτητα αυτά στοιχεία, αρχικά στο τοπικό αρχείο και αν δεν βρεθούν εκεί, τότε τα αναζητάει σε κάποιο maven repository. Επιπλέον, μέσα από αυτόματες

διαδικασίες το Maven εντοπίζει τυχόν εξαρτήσεις από εσωτερικά υπό-συστήματα και οργανώνει την σειρά χτισίματος αυτών με τρόπο τέτοιο σε κάθε χτίσιμο κάποιου υπό-συστήματος, όλα τα υπόλοιπα από τα οποία εξαρτάται να έχουν ήδη ολοκληρωθεί εκ των προτέρων.

5.1.3 Apache Avro

[8] Η βιβλιοθήκη Avro της Apache αποτελεί μία δομή απομακρυσμένης κλήσης διαδικασιών, καθώς και σειριοποίησης δεδομένων, το οποίο αναπτύχθηκε στα πλαίσια του έργου κατανεμημένης επεξεργασίας μεγάλων δεδομένων της Apache: το Hadoop. Στο Avro οι τύποι των δεδομένων και τα πρωτόκολλα ορίζονται μέσα από δομές JSON, ενώ τα δεδομένα σειριοποιούνται σε μια συμπαγή δυαδική μορφή. Η κύρια χρήση του εντοπίζεται στο έργο Hadoop, όπου χρησιμοποιείται τόσο ως μέσο σειριοποίησης των αποθηκευμένων δεδομένων, όσο και ως ένα μέσο εγγραφής κατά την επικοινωνία μεταξύ των διαφόρων κόμβων Hadoop μέσα σε ένα σύστημα, καθώς και από τις εφαρμογές χρηστών προς τις υπηρεσίες Hadoop που τις εξυπηρετούν.

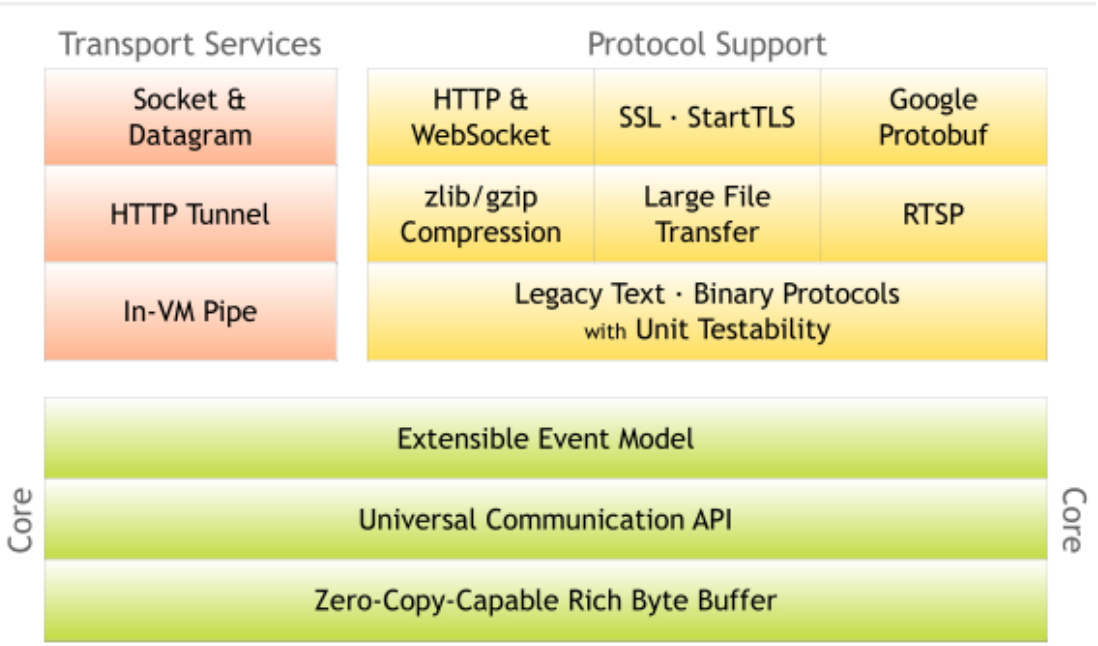
```
{"namespace": "example.avro",
 "type": "record",
 "name": "User",
 "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number",  "type": ["int", "null"]},
     {"name": "favorite_color", "type": ["string", "null"]}
 ]
}
```

Εικόνα 5.2 *Παράδειγμα ενός Avro σχήματος*

Επιπλέον στο Avro, υπάρχει υποστήριξη για δημιουργία RPC πρωτοκόλλων επικοινωνίας μεταξύ εφαρμογών σε συνδυασμό με κάποιον κατάλληλο εξυπηρετητή.

5.1.4 Netty Server

[12] Ο Netty είναι ένας εξυπηρετητής (server) βασισμένος πάνω στο πρωτόκολλο NIO (Network IO) της Java, ο οποίος επιτρέπει γρήγορη και εύκολη υλοποίηση εφαρμογών δικτύου (όπως εξυπηρετητές πρωτοκόλλου και εφαρμογών χρηστών). Η χρησιμότητα του σε μεγάλο βαθμό πηγάζει από την απλοποίηση του δικτυακού προγραμματισμού, όπως για παράδειγμα η χρήση TCP και UDP εξυπηρετητές υποδοχών (socket servers). Πέρα από την απλότητα, ο Netty έχει σχεδιαστεί με προσεκτική ανάλυση διαφόρων πρωτοκόλλων όπως το FTP, το SMTP, το HTTP και αρκετά ακόμα παλαιότερα. Ως αποτέλεσμα, ο Netty έχει επιτύχει την απλότητα στην υλοποίηση χωρίς να θυσιάζει την απόδοση, την σταθερότητα ή την ευελιξία της εφαρμογής.



Εικόνα 5.3 *Η δομή του έργου Netty*

5.2 Ανάπτυξη

Το κομμάτι της υλοποίησης πραγματοποιήθηκε σε τρία βασικά στάδια. Αρχικά υλοποιήθηκε το ζητούμενο αφαιρετικό επίπεδο σε όλα τα συστατικά που παίρνουν μέρος στην επικοινωνία με τον απομακρυσμένο διαχειριστή συναλλαγών στην υλοποίηση του CoherentPaas. Στη συνέχεια, δημιουργήθηκε ένας διαχειριστής συναλλαγών (NtuaTransactionalManager), ο οποίος περιείχε την βασική λειτουργικότητα ενός διαχειριστή συναλλαγών και κύριος στόχος του ήταν η χρήση για αξιολόγηση της υλοποίηση του βασικού στόχου της εργασίας. Έπειτα, έγινε χρήση του Avro για την επικοινωνία του απομακρυσμένου διαχειριστή συναλλαγών με τις εφαρμογές χρηστών. Ενώ τέλος, για λόγους ελέγχου της υλοποίησης, για όσα συστατικά είχε υλοποιηθεί αφαιρετικό επίπεδο στο πρώτο στάδιο, δημιουργήθηκαν νέα συστατικά τα οποία επέκτειναν την λειτουργία τους προσθέτοντας την λογική επικοινωνίας με τον δικό μας διαχειριστή συναλλαγών.

5.2.1 Αφαιρετικό Επίπεδο –Προσαρμογή υπάρχουσας υλοποίησης

Για το κομμάτι αυτό της εργασίας, αρχικά έγινε ανάλυση εις βάθος της λειτουργικότητας της εφαρμογής του CoherentPaas με μεγάλη προσοχή στην διαδρομή (flow) που ακολουθούσε η εφαρμογή, ώστε να συλλεχθούν και να απομονωθούν όλα τα συστατικά αυτά τα οποία είτε συμμετέχουν ενεργά στην επικοινωνία με τον απομακρυσμένο διαχειριστή συναλλαγών, είτε για οποιονδήποτε λόγο την επηρεάζουν. Έτσι εντοπίστηκαν οι κλάσεις που παρατίθενται παρακάτω.

5.2.1.1 MongoDB

Η κλάση MongoDB είναι το σημείο από το οποίο ξεκινάνε όλες οι διαδικασίες συναλλαγών. Αυτή είναι η κλάση που καλεί η εφαρμογή χρήστη ώστε να εκκινήσει, να ολοκληρώσει ή να ματαιώσει κάποια συναλλαγή. Στην συνέχεια μέσω αυτής καλούνται τα υπόλοιπα συστατικά, ενώ αν όλα πάνε καλά αυτή είναι που επικοινωνεί με την βάση δεδομένων για την πραγματοποίηση των εγγραφών και των αναγνώσεων από αυτήν.

Από τις μεθόδους που περιείχε οι περισσότερες κρίθηκε ότι δεν χρειάζονται κάποια τροποποίηση ώστε η λειτουργικότητα τους να είναι συμβατή με την σύνδεση κάποιου νέου διαχειριστή συναλλαγών και μεταφέρθηκαν αυτούσιες στην αφηρημένη κλάση (AbstractMongoDB) που δημιουργήσαμε στο αφαιρετικό επίπεδο. Αντίθετα οι μέθοδοι που παρατίθενται παρακάτω θα παρουσιάσουν διαφορές, είτε σε επίπεδο λογικής, είτε σε επίπεδο συστατικών επομένως τροποποιήθηκαν κατάλληλα και προστέθηκαν χωρίς λογική στην αφηρημένη κλάση, ενώ η λογική τους παρέμεινε στην αρχική κλάση MongoDB η οποία πλέον κληρονομεί την αφηρημένη AbstractMongoDB και κάνει Override τις συγκεκριμένες μεθόδους, προσθέτοντας την απαραίτητη λογική.

startTransaction(MongoTransactionalContext txnContext): **void**
Η μέθοδος αυτή πρακτικά καλεί την αντίστοιχη μέθοδο στο αντικείμενο TxnCnx (που αναλύεται παρακάτω) και ο λόγος που υπάρχει διαφοροποίηση ανάλογα με την επιλογή του διαχειριστή δεδομένων, έγκειται, πρώτον, στο γεγονός ότι και στο αντικείμενο αυτό προστέθηκε αφαιρετικό επίπεδο και αφετέρου, στο γεγονός ότι το αντικείμενο αυτό ανάλογα με την υλοποίηση επιστρέφεται από την κλάση LTMClient η οποία διαφοροποιείται από διαχειριστή σε διαχειριστή, καθώς μέσα σε αυτήν γίνεται η επικοινωνία της εφαρμογής χρήστη με τον απομακρυσμένο διαχειριστή.

commit (): **void**
Στην περίπτωση της commit ο λόγος που τροποποιήθηκε είναι ακριβώς ο ίδιος με παραπάνω. Και σε αυτήν την περίπτωση προκαλείται η αντίστοιχη μέθοδος στο αντικείμενο TxnCnx μέσω του LTMClient.

rollback (): **void**
Και στην περίπτωση της rollback η λογική είναι ίδια με την παραπάνω. Ακόμη μία φορά καλείται η ομώνυμη μέθοδος στο αντικείμενο TxnCnx μέσω του LTMClient.

5.2.1.2 TxnCtx (Transaction Context)

Η κλάση TxnCtx επεκτείνει μία κλάση ονομαζόμενη Transaction, η οποία περιέχει όλη την πληροφορία για την συναλλαγή, όπως το μοναδικό αναγνωριστικό (transaction Id), την χρονοσφραγίδα εκκίνησης και ολοκλήρωσης που της έχουν αποδοθεί από τον διαχειριστή συναλλαγών, τους εμπλεκόμενους κόμβους του συστήματος, το περιεχόμενο της συναλλαγής, καθώς και την κατάσταση της συναλλαγής. Μέσα στην TxnCtx προστίθεται επιπλέον η πληροφορία για τον transactionalDSClient με τον οποίο δένεται η συναλλαγή και ακόμα ορίζονται μέθοδοι βασισμένες στην πληροφορία του Transaction, όπως για παράδειγμα ο έλεγχος για διενέξεις εσωτερικά στην συναλλαγή καθώς και οι απαραίτητες

μέθοδοι για εκκίνηση, ολοκλήρωση ή ματαίωση της συναλλαγής οι οποίες καλούνται από ένα αντικείμενο του τύπου TxnCnx (βλ. παρακάτω).

Έπειτα από αρκετή διερεύνηση και μετά από την πραγματοποίηση πολλών δοκιμαστικών συναλλαγών, επιβεβαιώθηκε ότι τα πεδία και οι μέθοδοι του αντικείμενου Transaction δεν χρίζουν τροποποίησης, ωστόσο χρειάστηκαν συγκεκριμένες προσαρμογές στο TxnCtx για το οποίο δημιουργήθηκε η αφηρημένη κλάση AbstractTxnCtx, στην οποία μεταφέρθηκε η μεγαλύτερη λογική του, και την οποία ορίστηκε πλέον να κληρονομεί. Τα παρακάτω πεδία – μέθοδοι του TxnCtx, ωστόσο χρειάστηκαν κάποιου είδους τροποποίηση:

registeredDataStores: **Set<TransactionalDSClient>**
Το registeredDataStores αποτελεί μία δομή (set) αποθήκευσης των συσχετιζόμενων με την συγκεκριμένη συναλλαγή (και συνεπώς και με το συγκεκριμένο αντικείμενο TxnCtx) αντικειμένων TransactionalDSClient. Καθώς, όπως αναλύεται και στην συνέχεια, το αντικείμενο του τύπου TransactionalDSClient χρίζει τροποποίησης και υλοποίησης του αφηρημένου επιπέδου AbstractTransactionalDSClient, ήταν απαραίτητη η τροποποίηση του πεδίου αυτού και η μετατροπή του σε *Set <AbstractTransactionalDSClient >.*

associate (TransactionalDSClient client): **void**
Η μέθοδος associate είναι υπεύθυνη για τον συσχετισμό της συναλλαγής (TxnCtx) με τον TransactionalDSClient ο οποίος την έχει αναλάβει. Πρακτικά αυτό που κάνει είναι να προσθέσει τον TransactionalDSClient που έρχεται ως παράμετρος στην δομή που περιγράψαμε μόλις. Κατά αντίστοιχο τρόπο χρειάστηκε να τροποποιηθεί και η συγκεκριμένη μέθοδος ώστε να δέχεται παράμετρο τύπου *AbstractTransactionalDSClient* και να το προσθέτει στην παραπάνω δομή.

generateWriteSets (): **void**
Η μέθοδος generateWriteSets διέτρεχε όλα τα στοιχεία της δομής registeredDataStores, ανακτούσε το σύνολο εγγραφών τους, το μετέτρεπε σε πίνακα από bytes και στην συνέχεια τον προσέθετε σε μία δομή από την οποία η εφαρμογή αναγνώριζε τα δεδομένα που θα καταχωρηθούν στην βάση κατά την ολοκλήρωση της συναλλαγής (commit) από τον LTMClient.

abort (): **void**
Η μέθοδος abort επίσης διέτρεχε όλα τα στοιχεία της δομής registeredDataStores και σε κάθε TransactionalDSclient που υπάρχει αποθηκευμένος στην δομή αυτή, καλείται η rollback μέθοδος του με όρισμα το ίδιο το TxnCTx. Λόγω όμως της τροποποίησης της αναφοράς από TransactionalDSClient σε AbstractTransactional DSClient πραγματοποιήθηκε και αλλαγή στην μέθοδο αυτή, ενώ απαραίτητη είναι και η αλλαγή στην rollback μέθοδο του AbstractTransactionalDSClient.

5.2.1.3 TxnCnx (Transaction Connection)

Αποτελεί το αντικείμενο – διεπαφή ανάμεσα στον LTMClient και το TxnCtx. Πρακτικά περιέχει μονάχα ένα αντικείμενο τύπου TxnCtx και την συσχέτιση μεταξύ αυτού και τον LTMClient. Στην κλάση αυτή έχουν οριστεί δημόσιες μέθοδοι οι οποίες καλούν τις αντίστοιχες μεθόδους του LTMClient για πραγματοποίηση των διαφόρων λειτουργιών πάνω στο TxnCtx. Ωστόσο, λόγω της προσθήκης του αφαιρετικού επιπέδου AbstractTxnCtx, χρειάστηκαν οι κατάλληλες τροποποιήσεις και στο TxnCnx, ώστε να δέχεται το αφηρημένο αντικείμενο σαν παράμετρο όπου είναι απαραίτητο και ανάλογα την υλοποίηση να μπορεί να διαχειριστεί το κατάλληλο παιδί του. Για τον λόγο αυτό, αλλά και για τις διαφορές στην υλοποίηση ανάλογα με τον διαχειριστή συναλλαγών δημιουργήθηκε το αντικείμενο AbstractTxnCnx, το οποίο κληρονομείται από το προϋπάρχον TxnCnx.

associate(TransactionalDSClient client) : **void**

Η μέθοδος αυτή καλείται κατά την εκκίνηση κάποιας συναλλαγής και καλεί την αντίστοιχη και με ίδιο όνομα μέθοδο του LTMClient, ώστε να συσχετίσει την συναλλαγή με τον TransactionalDSClient που έρχεται σαν παράμετρος. Χρειάστηκε συνεπώς η αλλαγή της παραμέτρου σε TransactionalAbstracDSClient λόγω της προσθήκης του αφαιρετικού επιπέδου στο αντικείμενο αυτό. Επιπλέον όπως θα δούμε και παρακάτω το αντικείμενο LTMClient είναι συγκεκριμένο ανάλογα με τον διαχειριστή συναλλαγών, συνεπώς η λογική της μεθόδου μεταφέρθηκε στα παιδιά της νέας αφηρημένης κλάσης.

startTransaction() : **void**

Η μέθοδος startTransaction καλείται κατά την αρχικοποίηση της συναλλαγής και με την σειρά της καλεί την ομώνυμη μέθοδο στον LTMClient που χρησιμοποιεί ο συγκεκριμένος διαχειριστής συναλλαγών. Επομένως η υπογραφή της μεθόδου παρέμεινε στην AbstractTxnCnx αλλά η λογική της μεταφέρθηκε στο εκάστοτε παιδί της.

commit() : **void**

Αντίστοιχα με την startTransaction, η μέθοδος αυτή καλεί την ομώνυμη μέθοδο στον LTMClient της εκάστοτε υλοποίησης, για ολοκλήρωση της συναλλαγής. Συνεπώς η υπογραφή και αυτής της μεθόδου παρέμεινε στην AbstractTxnCnx αλλά η λογική της μεταφέρθηκε στο εκάστοτε παιδί της.

rollback() : **void**

Αντίστοιχα με την startTransaction, η μέθοδος αυτή καλεί την ομώνυμη μέθοδο στον LTMClient της εκάστοτε υλοποίησης, για αναίρεση της συναλλαγής. Συνεπώς η υπογραφή και αυτής της μεθόδου παρέμεινε στην AbstractTxnCnx αλλά η λογική της μεταφέρθηκε στο εκάστοτε παιδί της.

hasConflict(byte[] key) : **void throws TransactionException**

Αντίστοιχα με την startTransaction, η μέθοδος αυτή καλεί την ομώνυμη μέθοδο στον LTMClient της εκάστοτε υλοποίησης, για έλεγχο για τυχόν διενέξεις με άλλες συναλλαγές. Συνεπώς η υπογραφή και αυτής της μεθόδου, με το προκαθορισμένο Exception σε περίπτωση εύρεσης διένεξης, παρέμεινε στην AbstractTxnCnx αλλά η λογική της μεταφέρθηκε στο εκάστοτε παιδί της.

5.2.1.4 TransactionalDSClient / AbstractMongoClient

Η αφηρημένη κλάση AbstractMongoClient διατηρεί δομές στις οποίες αποθηκεύονται οι ανοιχτές συνδέσεις προς τις βάσεις δεδομένων της εφαρμογής χρήστη, καθώς και οι ανοιχτές συναλλαγές σε αυτές. Επιπλέον σε αυτήν διατηρούνται κλειδώματα και atomic μεταβλητές για εξασφάλιση της συγχρονικότητας μέσα στην εφαρμογή χρήστη. Τέλος η διεπαφή (interface) TransactionalDSClient ορίζει μεθόδους πάνω στα πεδία του AbstractMongoClient και οι δύο παραπάνω οντότητες για να έχουν νόημα, χρησιμοποιούνται συνδυαστικά (μια λειτουργική κλάση πρέπει να επεκτείνει τον AbstractMongoClient και να υλοποιεί τον TransactionalDSClient).

Καθώς στον συγκεκριμένο συνδυασμό αντικειμένων χρησιμοποιούνται αντικείμενα τα οποία έχουν ήδη τροποποιηθεί ή επρόκειτο να τροποποιηθούν, χρειάστηκαν οι κατάλληλες αλλαγές στις υπογραφές των μεθόδων ή και σε παιδία ώστε να χρησιμοποιούν τις νέες αφηρημένες κλάσεις και διεπαφές. Επομένως σε αυτές τις περιπτώσεις δεν δημιουργήθηκε επιπλέον αφαιρετικό επίπεδο καθώς υπήρχε ήδη, αλλά τροποποιήθηκε κατάλληλα το ήδη υπάρχον.

getDB (String dbname) : **MongoDB**
Η συγκεκριμένη μέθοδος καλείται από την εφαρμογή χρήστη, κατά την αρχικοποίηση της και επέστρεφε αντικείμενο τύπου *MongoDB* πάνω στο οποίο θα καλούνται όλες οι διεργασίες που περιλαμβάνονται στην διαδικασία κάποιας συναλλαγής. Συνεπώς η μέθοδος αυτή τροποποιήθηκε κατάλληλα ώστε να επιστρέφει αντικείμενο τύπου *AbstractMongoDB.*

applyWS(TxnCtx) : **long**
Η μέθοδος αυτή καλείται κατά την ολοκλήρωση κάποιας συναλλαγής και χρησιμοποιείται για την αφαίρεση της συναλλαγής που ολοκληρώθηκε από την δομή με τις ενεργές συναλλαγές και επιστρέφει το αναγνωριστικό της συναλλαγής που ολοκληρώθηκε. Πραγματοποιήθηκε η αλλαγή της παραμέτρου με την οποία καλείται σε *AbstractTxnCtx*.

rollback(TxnCtx) : **void**
Η μέθοδος αυτή καλείται κατά την αναίρεση κάποιας συναλλαγής και επίσης χρησιμοποιείται για την αφαίρεση της συναλλαγής που ολοκληρώθηκε από την δομή με τις ενεργές συναλλαγές. Πραγματοποιήθηκε αντίστοιχη τροποποίηση και αλλαγή της παραμέτρου με την οποία καλείται σε *AbstractTxnCtx*.

getWS (TxnCtx) : **byte[]**
Η μέθοδος αυτή συνήθως καλείται κατά την ολοκλήρωση κάποιας συναλλαγής. Σκοπός της είναι η αναζήτηση της συναλλαγής που παίρνει σαν παράμετρο στην λίστα με τις ανοιχτές συναλλαγές και έπειτα από σειριοποίηση, επιστροφή του συνόλου εγγραφής της ως πίνακα από bytes. Και στην περίπτωση της *getWS* πραγματοποιήθηκε η αντίστοιχη αλλαγή της παραμέτρου με την οποία καλείται σε *AbstractTxnCtx*.

garbaceCollect (long) : **void**
Η μέθοδος garbageCollect καλείται μετά την ολοκλήρωση κάποιας συναλλαγής από κάποια εφαρμογή χρήστη, ώστε να ενημερωθεί η βάση δεδομένων (Mongo) με την πιο πρόσφατη τιμή χρονοσφραγίδας επιτυχημένης ολοκλήρωσης. Έτσι με την σειρά της μπορεί να επιλέξει ποιες εγγραφές δεν είναι πλέον έγκυρες και να τις αφαιρέσει.

5.2.1.5 MongoClient

Η κλάση MongoClient είναι η κλάση η οποία χρησιμοποιούσε τον συνδυασμό των TransactionalDSClient και AbstractMongoClient, που αναφέρθηκε στην προηγούμενη παράγραφο. Επομένως χρειάστηκε την κατάλληλη τροποποίηση, ώστε να αξιοποιεί τις τροποποιημένες (αφηρημένες) εκδόσεις των αντικειμένων αυτών.

5.2.1.6 LTMClient

Ο LTMClient είναι η κλάση εκείνη, η οποία είναι υπεύθυνη για την άμεση επικοινωνία της εφαρμογής χρήστη με τον απομακρυσμένο διαχειριστή συναλλαγών. Για όλες τις φάσεις μιας συναλλαγής (εκκίνηση, ολοκλήρωση, ματαίωση) η διαδρομή της εφαρμογής κατέληγε στον LTMClient, ο οποίος επικοινωνούσε με τον διαχειριστή συναλλαγών και ανάλογα με την απόκριση του αποφάσιζε αν θα συνεχιστεί η λειτουργία ή αν θα ακυρωθεί.

Για την κλάση αυτή, αποφασίστηκε να δημιουργηθεί μία αφηρημένη κλάση (AbstractLTMClient) η οποία θα διατηρήσει τις απαραίτητες μεθόδους καθώς και τα απαραίτητα εσωτερικά αντικείμενα της κλάσης, αλλά στην αφηρημένη τους μορφή (για όσα από αυτά υλοποιήθηκε). Οι μέθοδοι που κρίθηκαν απαραίτητοι για την ορθή λειτουργία των LTMClients, εμφανίζονται παρακάτω:

register(TransactionalDSClient dsClient) : **void**
Η παραπάνω μέθοδος καλείται κατά την αρχικοποίηση του LTMClient και επικοινωνεί για πρώτη φορά με τον απομακρυσμένο διαχειριστή συναλλαγών, στον οποίο δημιουργείται μία εγγραφή στην δομή που διατηρεί με τις ενεργές εφαρμογές χρηστών.

hasConflict(byte[]key, DataStoreId dsId, TxnCtx txnCtx):**void**
Η μέθοδος αυτή ελέγχει κατά την αίτηση εκκίνησης ή ολοκλήρωσης κάποιας συναλλαγής για τυχόν συγκρούσεις με άλλες ενεργές συναλλαγές, τόσο σε επίπεδο εφαρμογής χρήστη, αλλά και έπειτα από επικοινωνία με τον διαχειριστή συναλλαγών, αποφαίνεται για τυχόν συγκρούσεις με άλλες συναλλαγές και με τις υπόλοιπες ενεργές εφαρμογές χρήστη.

startTransaction() : TxnCtx
Η μέθοδος η οποία καλείται για την εκκίνηση κάποιας συναλλαγής. Εσωτερικά καλεί την hasConflict για έλεγχο για συγκρούσεις με άλλες εφαρμογές και αν δεν βρεθεί κάποια, τότε προχωράει στην αίτηση χρονοσφραγίδας εκκίνησης από τον διαχειριστή συναλλαγών.

commit(TxnCtx txnCtx) : **void**
Η μέθοδος commit καλείται για την ολοκλήρωση κάποιας ενεργής συναλλαγής, παρόμοια με την startTransaction, γίνονται οι έλεγχοι για διενέξεις και στην συνέχεια υπάρχει επικοινωνία με τον διαχειριστή συναλλαγών για ανάθεση χρονοσφραγίδας ολοκλήρωσης.

rollback(TxnCtx txnCtx) : **void**
Τέλος ή rollback καλείται για την ματαίωση κάποιας ενεργής συναλλαγής, αφαιρώντας οποιαδήποτε εγγραφή έχει πραγματοποιηθεί γι' αυτήν σε δομή με τις ενεργές συναλλαγές τόσο εσωτερικά στον LTMClient, όσο και στον διαχειριστή συναλλαγών.
Από τα αντικείμενα που χρησιμοποιούσε εσωτερικά ο LTMClient

Οι υπόλοιπες μέθοδοι της κλάσης αυτής, δεν κρίθηκαν απαραίτητοι για την λειτουργία της εφαρμογής και δεν προστέθηκαν στην αφηρημένη κλάση που δημιουργήσαμε, ωστόσο διατηρήθηκαν στην μέθοδο η οποίο πλέον ορίστηκε να επεκτείνει (extend) την παραπάνω κλάση και να (Override) τις μεθόδους που βρίσκονται σε αυτήν.

5.2.2 Διαχειριστής Συναλλαγών (NtuaTransactionalManager)

Για να μπορέσουμε να πραγματοποιήσουμε τους απαραίτητους ελέγχους στην υλοποίηση μας, ήταν απαραίτητη η διασύνδεση της με κάποιον διαχειριστή συναλλαγών. Εν προκειμένω, δημιουργήθηκε ένας νέος διαχειριστής, ο NtuaTransactionalManager. Ο ρόλος του NtuaTransactionalManager είναι να τρέχει μοναδικά σε απομακρυσμένη υποδομή, να δέχεται αιτήματα από τις εφαρμογές χρήστη και να τα διαχειρίζεται κατάλληλα. Μέσα στις αρμοδιότητες του συμπεριλαμβάνονται, η απόφανση για το αν μια συναλλαγή μπορεί να εκκινήσει, ή εάν βρίσκεται σε διένεξη με κάποια άλλη ενεργή συναλλαγή και αντίστοιχα για το εάν μπορεί να ολοκληρωθεί. Επιπλέον διαχειρίζεται τις αναιρέσεις των συναλλαγών, ενώ ανά συγκεκριμένα χρονικά διαστήματα ενημερώνει τις εφαρμογές χρηστών για το μεγαλύτερο σε τιμή αναγνωριστικό συναλλαγής, για το οποίο δεν υπάρχουν ανοιχτές συναλλαγές με μικρότερο αναγνωριστικό.

Τα πεδία που περιλαμβάνει ο διαχειριστής συναλλαγών είναι τα παρακάτω:

trId: AtomicLong
Μία νηματικά ασφαλής 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 νηματικά ασφαλής 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 νηματικά ασφαλές 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 νηματικά ασφαλές 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>

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();
                        }
                }
        }
```

Εικόνα 5.4 *Ο χρονοδρομολογητής που τρέχει την διεργασία για την ενημέρωση των εφαρμογών χρήστη*

5.2.3 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"
   }
  }
}
```

Εικόνα 5.5 *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"
}
```

Εικόνα 5.6 *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"}
 ]
}
```

Εικόνα 5.7 *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>
```

Εικόνα 5.8 *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>
```

Εικόνα 5.9 *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 σειριοποιήσιμα 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.

6 Έλεγχος λειτουργίας / Αξιολόγηση Υλοποίησης

Για τον έλεγχο της λειτουργίας της υλοποίησης που περιγράφεται στα πλαίσια αυτής της εργασίας, πραγματοποιήθηκαν έλεγχοι δύο σημείων. Αρχικά ήταν απαραίτητο να επαληθευτεί ότι η πρόσθετη λειτουργικότητα που υλοποιήσαμε, δεν επηρέασε την πρότερη λειτουργία της διαχείρισης συναλλαγών και της επικοινωνίας με τον υφιστάμενο διαχειριστή συναλλαγών του CoherentPaas. Σε δεύτερο χρόνο, έτρεξαν οι κατάλληλες διεργασίες για επαλήθευση των απαραίτητων σεναρίων, ώστε να εξακριβωθεί ότι έχει επιτευχθεί η ζητούμενη λειτουργικότητα.

6.1 Έλεγχος οπισθοδρόμησης

Για τον έλεγχο οπισθοδρόμησης, ήταν αναγκαίο να επαληθεύσουμε ότι οι τροποποιήσεις και προσθήκες στις οποίες προχωρήσαμε, δεν επηρέασαν την λειτουργία της εφαρμογής CoherentPaas. Η επαλήθευση αυτή έγινε σε δύο στάδια. Αρχικά, στην υλοποίηση του CoherentPaas προϋπήρχαν έλεγχοι μονάδων (unit tests) τα οποία εκτελούσαν διάφορες συγκεκριμένες αλληλουχίες του προγράμματος και στο τέλος αντιπαρέβαλλαν το αναμενόμενο αποτέλεσμα της εκτέλεσης με το πραγματικό. Οι έλεγχοι αυτοί τρέχουν σε κάθε απόπειρα χτισίματος της εφαρμογής μέσω του Maven (εκτός αν υπάρχει ρητή εντολή για το αντίθετο –DskipTests) και αν δεν ολοκληρωθούν όλοι επιτυχώς η εφαρμογή δεν μπορεί να δημιουργηθεί. Επομένως, αυτό που κάναμε, ήταν να προσθέσουμε δικούς μας επιπλέον ελέγχους για τα συστατικά τα οποία είχαμε επηρεάσει με την υλοποίηση μας αυτή και στην συνέχεια να δοκιμάσουμε να χτίσουμε την εφαρμογή μέσω του Maven (mvn clean package), κάτι που ολοκληρώθηκε επιτυχώς.

Για το δεύτερο στάδιο των ελέγχων μας, χρειάστηκε να δημιουργήσουμε δύο σημεία εισόδου στην εφαρμογή διαχείρισης συναλλαγών, δημιουργώντας μία εφαρμογή χρήστη και τρέχοντας την παράλληλα σαν δύο διαφορετικές διεργασίες. Στη συνέχεια προχωρήσαμε σε δημιουργία συναλλαγών και επαλήθευση ότι σε αυτές γίνεται ανάθεση χρονοσφραγιδών εκκίνησης και ολοκλήρωσης όπως γινόταν και πριν τις αλλαγές μας, ενώ κατά την ολοκλήρωση ή την αναίρεση κάποιας συναλλαγής πραγματοποιούνταν αφαίρεση της αναφοράς αυτής από την τοπική δομή αποθήκευσης ενεργών συναλλαγών.

Έπειτα, προχωρήσαμε στον έλεγχο της ομαλής λειτουργικότητας σε περιπτώσεις ύπαρξης διενέξεων μεταξύ ενεργών συναλλαγών. Για να το επιτύχουμε αυτό αρχικά δοκιμάσαμε να δημιουργήσουμε μία συναλλαγή η οποία επηρεάζει μία συλλογή την οποία επίσης επηρεάζει μια προϋπάρχουσα και μη ολοκληρωμένη συναλλαγή στην ίδια εφαρμογή χρήστη. Διαπιστώσαμε ότι εύστοχα ο τοπικός διαχειριστής συναλλαγών (LTM) δεν επέτρεψε κάτι τέτοιο και επέστρεψε κατάλληλο μήνυμα στην συγκεκριμένη εφαρμογή χρήστη. Εν συνεχεία εκτελέσαμε αντίστοιχο σενάριο αλλά αυτή τη φορά οι δύο συναλλαγές που επιθυμούσαν να επηρεάσουν την ίδια συλλογή προέρχονταν από διαφορετικές εφαρμογές χρήστη. Στην περίπτωση αυτή, η δεύτερη κατά σειρά εκτέλεσης εφαρμογή, κατά την προσπάθεια εκκίνησης της συναλλαγής, ο LTM προχώρησε κανονικά χωρίς να εντοπίσει

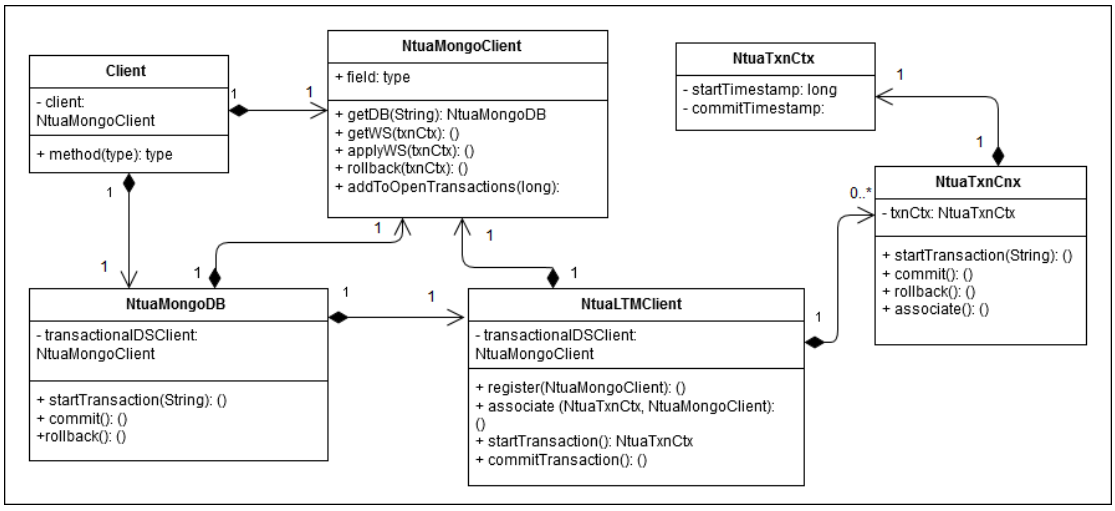
κάποια σύγκρουση, καθώς δεν είχε γνώση για ενεργές συναλλαγές σε διαφορετικές εφαρμογές χρήστη. Αντίθετα ο διαχειριστής συναλλαγών του CoherentPaas, ο οποίος είναι υπεύθυνος για την διαχείριση και των συγχρονισμό όλων των συναλλαγών από όλες τις εφαρμογές χρηστών, δεν επέτρεψε στην δεύτερη αυτή εφαρμογή να συνεχίσει, καθώς υπήρχε ήδη μία ενεργή συναλλαγή για την συγκεκριμένη συλλογή.

Ακόμα, πραγματοποιήθηκε μία σειρά από αλληλουχίες συναλλαγών οι οποίες δεν παρουσίαζαν συγκρούσεις ούτε σε επίπεδο εφαρμογής αλλά ούτε συνολικά σε όλες τις εφαρμογές χρηστών και όλες ολοκληρώθηκαν κανονικά, καθώς εφόσον δεν διαπιστώθηκαν συγκρούσεις ούτε στους τοπικούς LTMs, ούτε στον διαχειριστή συναλλαγών δεν υπήρχε κάποιος λόγος να ακυρωθεί κάποια από αυτές.

6.2 Έλεγχος υλοποίησης

Για τον έλεγχο της υλοποίησης μας, χρειάστηκε να ελέγξουμε τόσο το αφαιρετικό επίπεδο που σχεδιάσαμε και υλοποιήσαμε, όσο και τον διαχειριστή συναλλαγών που αναπτύξαμε. Για να το επιτύχουμε αυτό χρειάστηκε να επεκτείνουμε τις αφαιρετικές δομές που δημιουργήσαμε πάνω στο CoherentPaas ώστε να επιτύχουμε επικοινωνία με τον δικό μας διαχειριστή συναλλαγών.

Έτσι σχεδιάσαμε το παρακάτω μοντέλο με δομές με το πρόθεμα NTUA, η κάθε μία από τις οποίες επέκτεινε διαφορετικό συστατικό του αφαιρετικού επιπέδου που έχουμε δημιουργήσει. Οι τροποποιήσεις σε αυτές σε σχέση με την αρχική υλοποίηση του CoherentPaas έγκεινται στην χρησιμοποίηση των καταλλήλων NTUA-συστατικών εσωτερικά, στις ειδικές περιπτώσεις που περιγράφονται σε προηγούμενο κεφάλαιο, όπου η χρησιμοποίηση των γενικών αφαιρετικών συστατικών δεν ήταν εφικτή.



Εικόνα 6.1 *Η επέκταση του αφαιρετικού επιπέδου για έλεγχο της υλοποίησης*

Συγκεκριμένη υλοποίηση απαιτήθηκε μόνο στην κλάση NTUALTMClient, όπου χρειάστηκε να γίνουν οι απαραίτητες ρυθμίσεις και υλοποιήσεις, ώστε να εγκαθιδρυθεί η επικοινωνία

με τον διαχειριστή συναλλαγών (NTUATransactionalManager). Συγκεκριμένα, κατά την εκκίνηση του NTUALTMClient ξεκινάει ένα νέο νήμα (thread) διεργασιών το οποίο μέσω ενός NettyTransceiver αντικειμένου, επικοινωνεί με τον Netty εξυπηρετητή του NTUATransactionalManager και επιστρέφει ένα αντικείμενο τύπου MongoBroker, μέσω του οποίου θα γίνεται η επικοινωνία του NTUALTMClient με τον διαχειριστή συναλλαγών. Επιπλέον γίνεται αναγνώρισης των στοιχείων δικτύου (IP) της εφαρμογής χρήστη και καλείται η register μέθοδος του διαχειριστή, για δήλωση της εφαρμογής χρήστη.

Στη συνέχεια, επεκτάθηκαν όλες οι αφηρημένες μέθοδοι που αναφέραμε στον LTMClient σε προηγούμενο κεφάλαιο μέσα στις οποίες υλοποιήθηκε η επικοινωνία με τον διαχειριστή συναλλαγών μέσω του MongBroker.

```
NettyTransceiver  nettyClient = new NettyTransceiver(new InetSocketAddress(10000));
MongoBroker broker = (MongoBroker) SpecificRequestor.getClient(MongoBroker.class, nettyClient);

(…)

RegisterRequest  registerRequest = new RegisterRequest(clientIp, clientPort);
broker.register(registerRequest);
```

Εικόνα 6.2 *Αρχικοποίηση MongoBroker και κλήση της register*

Ως πρώτο στάδιο των ελέγχων μας, ορίσαμε ελέγχους μονάδων (unit testing) μέσω του Maven. Σχεδιάσαμε και υλοποιήσαμε τους κατάλληλους ελέγχους κατ' αντιστοιχία με τους ελέγχους που χρησιμοποιήθηκαν στον έλεγχο οπισθοδρόμησης, καθώς οι λειτουργικές μονάδες είχαν μικρές διαφορές στην λογική ανάμεσα στις δύο αυτές υλοποιήσεις. Έτσι μας δόθηκε η δυνατότητα να ελέγξουμε την λογική της εφαρμογής μας, πριν καν την εκτελέσουμε και να διορθώσουμε τυχόν σφάλματα.

Μετά την επιτυχή ολοκλήρωση των ελέγχων αυτών, χρειάστηκε να επιβεβαιώσουμε την επικοινωνία του διαχειριστή συναλλαγών με τις εφαρμογές χρηστών. Έτσι τρέξαμε τον διαχειριστή συναλλαγών και στην συνέχεια εκτελέσαμε πολλαπλές εφαρμογές χρηστών, κατά την αρχικοποίηση των οποίων ελέγξαμε ότι δημιουργούνται τα αντικείμενα επικοινωνίας με τον διαχειριστή συναλλαγών και ότι καλείται επιτυχώς η register μέθοδος αυτού. Στη συνέχεια πραγματοποιήθηκε μία σειρά από αλληλουχίες συναλλαγών οι οποίες δεν παρουσίαζαν συγκρούσεις και επιβεβαιώσαμε ότι όλες ολοκληρώθηκαν κανονικά, καθώς δεν υπήρχαν συγκρούσεις ούτε στους τοπικούς LTMs, ούτε στον διαχειριστή συναλλαγών και υπήρχε κάποιος λόγος να ακυρωθεί κάποια από αυτές.

```
{
  "_id":{
"$binary":"rO0ABXNyABdvcmcuYnNvbi50eXBlcy5PYmplY3RJZDLuvkyfZGqAAgAESQAHY291b
nRlckkAEW1hY2hpbmVJZGVudGlmaWVyUwARcHJvY2Vzc0lkZW50aWZpZXJJAAl0aW1lc3Rhb
XB4cACW/FoA6jAUEXBZa5xwAAAAAAAAAAE=", "$type":"00" },
  "user":1,
  "value":"someValue",
  "_dataID":{
    "$oid":"596b9c70ea3014117096fc5a"
  },
  "_tid":{
    "$numberLong":"1"
  },
  "_cmtTmstmp":{
    "$numberLong":"2"
  },
  "_nextCcmtTmstmp":"3"
}
```


Εικόνα 6.3 *Εγγραφή στην βάση μετά από επιτυχημένη συναλλαγή*

Έπειτα, προχωρήσαμε στον έλεγχο της ομαλής λειτουργικότητας σε περιπτώσεις ύπαρξης διενέξεων μεταξύ ενεργών συναλλαγών. Για να το επιτύχουμε αυτό αρχικά δοκιμάσαμε να δημιουργήσουμε μία συναλλαγή η οποία επηρεάζει μία συλλογή την οποία επίσης επηρεάζει μια προϋπάρχουσα και μη ολοκληρωμένη συναλλαγή στην ίδια εφαρμογή χρήστη. Διαπιστώσαμε ότι εύστοχα ο τοπικός διαχειριστής συναλλαγών (NTUALTM) δεν επέτρεψε κάτι τέτοιο και επέστρεψε κατάλληλο μήνυμα στην συγκεκριμένη εφαρμογή χρήστη. Εν συνεχεία εκτελέσαμε αντίστοιχο σενάριο αλλά αυτή τη φορά οι δύο συναλλαγές που επιθυμούσαν να επηρεάσουν την ίδια συλλογή προέρχονταν από διαφορετικές εφαρμογές χρήστη. Στην περίπτωση αυτή, η δεύτερη κατά σειρά εκτέλεσης εφαρμογή, κατά την προσπάθεια εκκίνησης της συναλλαγής, ο NTUALTM προχώρησε κανονικά χωρίς να εντοπίσει κάποια σύγκρουση, καθώς δεν είχε γνώση για ενεργές συναλλαγές σε διαφορετικές εφαρμογές χρήστη. Αντίθετα ο διαχειριστής συναλλαγών (NTUATransactional Manager), ο οποίος είναι υπεύθυνος για την διαχείριση και των συγχρονισμό όλων των συναλλαγών από όλες τις εφαρμογές χρηστών, δεν επέτρεψε στην δεύτερη αυτή εφαρμογή να συνεχίσει, καθώς υπήρχε ήδη μία ενεργή συναλλαγή για την συγκεκριμένη συλλογή.

6.3 Αξιολόγηση

Η λογική της εισαγωγής αφαιρετικού επιπέδου στην εφαρμογή του CoherentPaas στέφθηκε με επιτυχία και έλεγχοι που πραγματοποιήσαμε με πολλαπλές εφαρμογές χρήστη καθεμία από τις οποίες ορίστηκε να εκτελεί πολλαπλές συναλλαγές ήταν αρκετά ενθαρρυντικοί. Τόσο ο διαχειριστής δεδομένων που υλοποιήσαμε, όσο και η εσωτερική διαχείριση του συγχρονισμού των συναλλαγών από τις εφαρμογές χρήστη, καθ’ όλη την διάρκεια των ελέγχων εκτελέστηκαν χωρίς την παραμικρή καθυστέρηση, λόγω του φόρτου των συναλλαγών. Σε επέκταση της εργασίας αυτή μπορεί να γίνει αναλυτική μέτρηση του ρυθμού εκτέλεσης της εφαρμογής σε συνδυασμό με τον εκάστοτε διαχειριστή συναλλαγών.

7 Επίλογος

7.1 Σύνοψη και συμπεράσματα

Με την ολοένα αυξανόμενη ανάγκη για υποστήριξη εφαρμογών τεράστιου όγκου δεδομένων, που συνεχώς μεταβάλλονται, καθώς και η επιτακτική ανάγκη για ταχεία επίδοση των νέων τεχνολογιών, η στροφή προς τις NoSQL είναι μονόδρομος. Με πάνω από 250 επιλογές σε NoSQL βάσεις δεδομένων και αρκετούς μεγάλους κολοσσούς στον τομέα της πληροφορίας και της πληροφορικής, όπως το Facebook, το ebay κτλ, να έχουν ήδη εισάγει τις NoSQL στα παραγωγικά τους περιβάλλοντα, το μέλλον τους φαντάζει ευοίωνο. Μια τέτοια στροφή, ωστόσο, δεν μπορεί να είναι εφικτή για οποιοδήποτε σύστημα βάσης δεδομένων, το οποίο δεν προσφέρει υποστήριξη για συναλλαγές. Για να μπορέσει να ευδοκιμήσει μία τέτοια μεταφορά, από ήδη καθιερωμένα συστήματα και καταξιωμένες επιλογές σε σχεσιακές βάσεις δεδομένων, είναι απαραίτητη η εγγύηση της συγχρονικότητας των ταυτόχρονων συναλλαγών.

Στην παρούσα εργασία έγινε προσπάθεια επέκτασης της εφαρμογής ενσωμάτωσης των συναλλαγών σε NoSQL βάσεις δεδομένων, συγκεκριμένα σε MongoDB. Η πολλά υποσχόμενη υλοποίηση του CoherentPaas μπορεί να προσφέρει πολλά παραπάνω αν έχει την ευελιξία να επιλέξει ανάμεσα σε πολλαπλές υλοποιήσεις διαχειριστών συναλλαγών, ανάλογα με το ποιος ευνοεί περισσότερο την εκάστοτε υλοποίηση.

7.2 Μελλοντικές επεκτάσεις

Έχοντας υλοποιήσει το ζητούμενο αφαιρετικό επίπεδο για την ενσωμάτωση οποιουδήποτε διαχειριστή συναλλαγών στην υλοποίηση του CoherentPaas και έχοντας δημιουργήσει μια δική μας υλοποίηση και αντίστοιχο διαχειριστή συναλλαγών, μπορέσαμε να αναπαράγουμε και να επαληθεύσουμε την ζητούμενη λειτουργικότητα. Σε επόμενο βήμα μπορούν να ενσωματωθούν και άλλοι διαχειριστές όπως ο TO-TSO του Omid που αναφέραμε παραπάνω, ώστε να υπάρχει η δυνατότητα επιλογής ανάλογα τον χρήστη και τις ιδιαιτερότητες της υλοποίησης του, ο κατάλληλος διαχειριστής συναλλαγών.

Επιπλέον μπορεί να γίνουν αναλυτικές μετρήσεις των διαφόρων πιθανών διαχειριστών συναλλαγών και να αποτυπωθεί η συμπεριφορά τους και να μετρηθεί η απόδοση τους σε διαφορετικά περιβάλλοντα, με διαφορετικές NoSQL βάσεις, ώστε να διαπιστωθεί ποιος συνδυασμός ταιριάζει καλύτερα σε ποια περίπτωση. Έτσι θα έχει γίνει ένα σημαντικό ερευνητικό βήμα, σχετικά με την καταγραφή της απόδοση και την επίδοση των διαφόρων ειδών βάσεων δεδομένων, σε διαφορετικά είδη εφαρμογών και σε συνδυασμό με διαφορετικούς διαχειριστές συναλλαγών.

8 Βιβλιογραφία

[1] Abraham Silberschatz, H. F. (2011). *Συστήματα Βάσεων Δεδομένων.* New York: Μ. Γκιούρδας.

[2] *MongoDB official website*. (n.d.). Ανάκτηση από https://docs.mongodb.com

[3] *NoSQL website*. (n.d.). Ανάκτηση από http://nosql-database.org/

[4] Iván Brondino, R. J.-P.-M. (2015). *Local Transaction Manager for all Cloud Data Stores.* CoherentPaaS: Coherent and Rich PaaS with a Common Programming Model page 1/40.

[5] *MongoDB university*. (n.d.). Ανάκτηση από https://university.mongodb.com

[6] *LeanXcale official website*. (n.d.). Ανάκτηση από http://www.leanxcale.com

[7] *Yahoo Omid official repository*. (n.d.). Ανάκτηση από http://yahoo.github.io/omid/

[8] *Apache Avro official website*. (n.d.). Ανάκτηση από https://avro.apache.org/

[9] *Oracle Java official website*. (n.d.). Ανάκτηση από https://www.java.com/

[10] *Apache Maven project official website*. (n.d.). Ανάκτηση από https://maven.apache.org

[11] *CoherentPaas project official website*. (n.d.). Ανάκτηση από http://coherentpaas.eu/

[12] *Official Netty web site*. (n.d.). Ανάκτηση από https://netty.io/

9 Appendix

9.1 Λεξικό Ελληνικών Όρων

Καθώς μεγάλο μέρος από την ορολογία στο κομμάτι της τεχνολογίας και δει στο κομμάτι του προγραμματισμού, προέρχεται από την Αγγλική γλώσσα, έγινε μεγάλη προσπάθεια να αποτυπωθούν στα Ελληνικά, όσο πιο πιστά γινόταν οι διάφορες λέξεις που χρησιμοποιήθηκαν στην συγγραφή της παρούσας εργασίας. Ωστόσο, για αποφυγή παρανοήσεων και για διευκόλυνση των αναγνωστών, στο παράρτημα αυτό αντιστοιχίζεται η αρχική αγγλική ορολογία των διαφόρων όρων.

Ανάγνωση: read

Ανάγνωση Φάντασμα: Phantom read

Αναίρεση: Rollback

Ανθεκτικότητα: Durability

Αντικείμενο Java: Java Object

Απομόνωση: Isolation

Απομόνωση Στιγμιότυπου: Snapshot isolation

Αποσφαλμάτωση: Debugging

Ατομικότητα: Atomicity

Αφαιρετικό επίπεδο/Επίπεδο αφαιρετικότητας: Abstraction level/layer

Δημόσια μέθοδος: Public method

Διαδικασία ανάγνωσης: Read operation

Διαδικασία εγγραφής: Write operation

Διαδικασία ενημέρωσης: Update operation

Διένεξη: Conflict

Διεπαφή: Interface

Δομή αποθήκευσης κλειδιού-τιμής: Key-value data store

Δομή αποθήκευσης νέφους: Cloud data store

Εγγραφή: Write

Εκτέλεση: Execution

Έναρξη επικοινωνίας: Connection establishment

Ενδιάμεσο Λογισμικό: Middleware

Εξαίρεση: Exception

Εξόρυξη δεδομένων: Data mining

Επεκτείνει: Extends

Έργο: project

Ερώτημα: Query

Ευέλικτη Μεθοδολογία: Agile methodology

Εφαρμογή χρήστη: client application

Java Κλάση: Java Class

Κληρονομικότητα: inheritance

Κλιμάκωση: Scaling

Κόμβος: Node

Ματαίωση: Abort

Μεγάλα Δεδομένα: Big Data

Μη-επαναλαμβανόμενη ανάγνωση: Non-repeatable read

Νέφος: cloud

Νήμα: Thread

Νηματικά ασφαλές: Thread safe

Ολοκλήρωση: Commit

Ολοκλήρωση δύο φάσεων: Two-phase commit

Οπισθοδρόμηση: Regression

Πρόχειρο γράψιμο: Dirty read

Ροή Εφαρμογής: Application flow

Σειριοποίηση: Serialization

Σύγκρουση: βλ. Διένεξη

Συγχρονικότητα: Concurrency

Συναλλαγή: Transaction

Συγχρονικότητα: Concurrency

Συλλέκτης απορριμμάτων: Garbage collector

Συστατικό: Component / Module

Συγχρονισμός πολλαπλών εκδόσεων: Multiversion concurrency

Χρησιμοποιεί: Implements

Χρονοδρομολογητής: Scheduler

Χρονοσφραγίδα (Σφραγίδα χρόνου): Timestamp