Πανεπιστήμιο Μακεδονίας

Σχολή Επιστημών Πληροφορίας

Τμήμα Εφαρμοσμένης Πληροφορικής

Μη σχεσιακές Βάσεις δεδομένων : Η περίπτωση της MongoDB

Ανάπτυξη εφαρμογής (blog)

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Σαπουντζή Ιμπράμ Ιμπραήμ

Επιβλέπων : Γεώργιος Ευαγγελίδης, Καθηγητής

Εξεταστική επιτροπή :

Κολωνιάρη Γεωργία, Λέκτορας

Χατζηγεωργίου Αλέξανδρος, Αναπληρωτής Καθηγητής

Θεσσαλονίκη, 2015

Ευχαριστίες

Η εργασία εκπονήθηκε στο Information Management Laboratory του τμήματος Εφαρμοσμένης Πληροφορικής στο Πανεπιστήμιο Μακεδονίας, υπό την επίβλεψη του καθηγητή Γεώργιου Ευαγγελίδη.

Θα ήθελα να ευχαριστήσω θερμά τον κ. Γεώργιο Ευαγγελίδη για την καθοδήγηση και την υποστήριξη που μου πρόσφερε αλλά και για την ευκαιρία να ασχοληθώ με τις μη σχεσιακές βάσεις δεδομένων.

Τέλος θα ήθελα να ευχαριστήσω πολύ την οικογένεια και τους φίλους μου για την συνεχή υποστήριξη σε όλα τα φοιτητικά μου χρόνια.

Περίληψη

Αυτή η εποχή χαρακτηρίζεται από πολλούς ως ζωτικής σημασίας στον κόσμο των βάσεων δεδομένων. Εδώ και πολλά χρόνια, ακόμα και πριν το 1980, το σχεσιακό μοντέλο για την αναπαράσταση των δεδομένων ήταν η *defacto* επιλογή για κάθε είδους προβλήματα, είτε μεγάλα είτε μικρά. Κανείς δεν περιμένει πώς το σχεσιακό μοντέλο αναπαράστασης δεδομένων (RDBMS) θα 'μαραζώσει ' στο κοντινό μέλλον, απλώς οι προγραμματιστές εξελίσσονται από τις σχεσιακές βάσεις δεδομένων και έχουν την τάση να ανακαλύπτουν εναλλακτικές επιλογές. Επιλογές που θα τους δίνουν την δυνατότητα να έχουν στην διάθεση τους τεχνικές όπως – δυναμικά σχήματα και εναλλακτικές δομές δεδομένων, απλή αναπαραγωγή των δεδομένων, υψηλή διαθεσιμότητα, οριζόντια κλιμάκωση της βάσης και καινούριες μεθόδους αναπαράστασης ερωτημάτων προς την βάση (queries). Αυτές οι δυνατότητες είναι μερικές από τις οποίες διαθέτουν οι μη σχεσιακές βάσεις δεδομένων, γνωστές και ως *NoSQL* βάσεις δεδομένων. Ο όρος NoSQL ή ' Not only SQL ' ξεκίνησε να χρησιμοποιείται από τις αρχές του 2009 για να περιγράψει αυτόν το νέο, μη σχεσιακό τρόπο αποθήκευσης δεδομένων. Τέτοια συστήματα βάσεων δεδομένων χρησιμοποιούνται όλο και περισσότερο σε εφαρμογές μεγάλου όγκου δεδομένων (big data) και σε σύγχρονες διαδικτυακές εφαρμογές πραγματικού χρόνου (real-time web applications). Στην παρούσα πτυχιακή εργασία θα κάνουμε μια περιήγηση σε αυτόν τον καινούριο κόσμο βάσεων δεδομένων, και συγκεκριμένα θα μελετήσουμε την περίπτωση της MongoDB, μια πλατφόρμα βασισμένη στην αποθήκευση των δεδομένων ως έγγραφα (document-oriented database). Αφού αναλύσουμε τις διάφορες δυνατότητες που μας προσφέρει ένα τέτοιο σύστημα, ξεκινώντας από τις βασικές λειτουργίες αποθήκευσης, επεξεργασίας και ανάκτησης των δεδομένων και συνεχίζοντας σε πιο πολύπλοκες διαδικασίες, όπως τεχνικές κλιμάκωσης και μηχανισμούς για την υψηλή διαθεσιμότητα των δεδομένων, στο τέλος θα αναπτυχθεί μια διαδικτυακή εφαρμογή (blog) με την χρήσης της νέας τεχνολογίας *nodeJS* και της βάσης δεδομένων *MongoDB*. Το κυριότερο σημείο της παρούσας εργασίας είναι να καταλάβουν οι χρήστες, τις δυνατότητες που μπορεί να έχουν με ένα τέτοιο εργαλείο στην διάθεση τους. Τελικά, οι προγραμματιστές καλούνται να υιοθετήσουν τις σύγχρονες τεχνολογίες ή τον καλύτερο συνδυασμό βάσεων δεδομένων που ταιριάζει καλύτερα στις ανάγκες τους και την δομή του προβλήματος που καλούνται να αντιμετωπίσουν.

Λέξεις Κλειδιά

Μη σχεσιακές βάσεις δεδομένων, NoSQL, μεγάλος όγκος δεδομένων, big data, νέες τεχνολογίες, MongoDB, διαδικτυακές εφαρμογές, nodeJS, real-time web applications.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Κεφάλαιο 1 Big Data

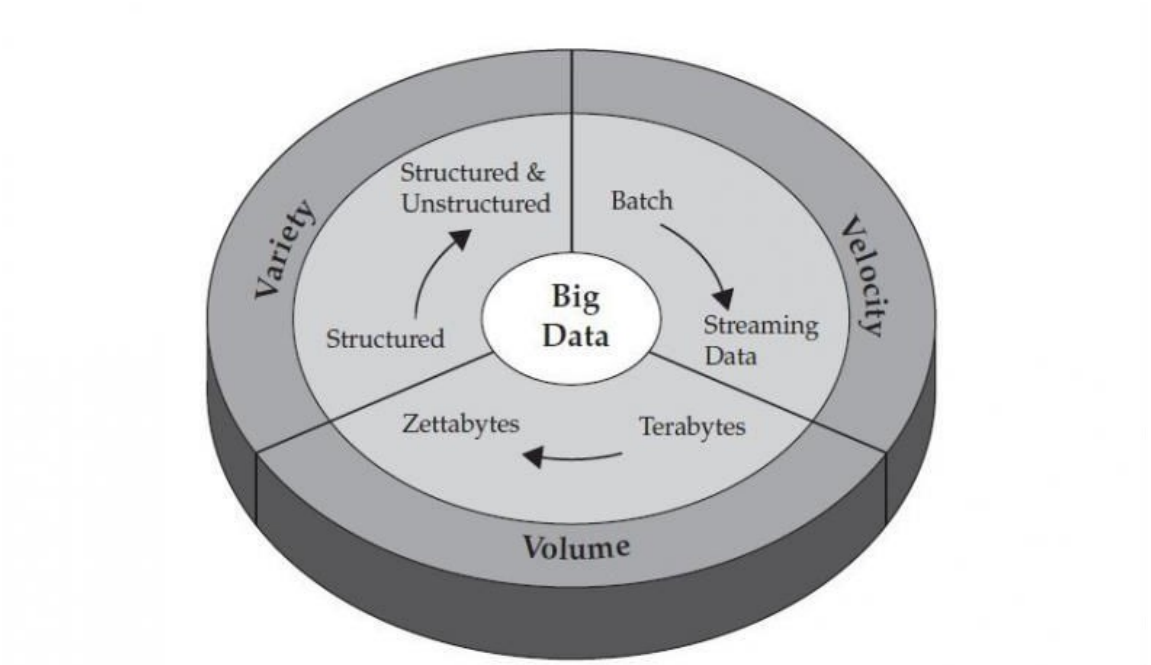
1.1 Γενικά

Οι προμηθευτές των γνωστών συστημάτων βάσης και αποθήκευσης δεδομένων έλεγαν πως ο όρος big data απλά αναφέρεται σε βάσεις που περιέχουν πάρα πολύ μεγάλο αριθμό γραμμών στους πίνακές τους. Το μέγεθός τους συνήθως είναι της τάξης ενός ή μερικών terabytes και οι γραμμές δεδομένων που είναι αποθηκευμένες μπορεί να φτάνουν έως και μερικά δισεκατομμύρια. Τα τελευταία χρόνια όμως ο ορισμός των big data έχει αλλάξει. Η εξέλιξη του διαδικτύου έχει επαναπροσδιορίσει την ταχύτητα με την οποία κινείται η πληροφορία μέσα στα online συστήματα. Οι πελάτες των εταιρειών δεν καθορίζονται πλέον από γεωγραφικά κριτήρια, μιας και το διαδίκτυο επιτρέπει άμεσες συναλλαγές διεθνούς εμβέλειας, με αποτέλεσμα ο αριθμός τους να αυξάνεται κατακόρυφα. Μεγάλη εξέλιξη έχει πραγματοποιηθεί και στα είδη των δεδομένων που επεξεργάζονται και παρακολουθούνται μέσω του διαδικτύου. Προσωπικές πληροφορίες χρηστών, γεωγραφικά δεδομένα (geolocational), κοινωνικά γραφήματα, δεδομένα από αρχεία καταγραφής, δεδομένα που παράγονται από αισθητήρες, είναι κάποια παραδείγματα του διαρκώς διευρυμένου πίνακα δεδομένων που συλλαμβάνονται. Επίσης το χρονικό διάστημα μεταξύ της στιγμής που εισέρχονται τα δεδομένα σε κάποιο σύστημα και της στιγμής που μετατρέπονται σε πληροφορία έτοιμη να αναλυθεί και να χρησιμοποιηθεί θα πρέπει να ανταποκρίνεται στις νέες απαιτήσεις των χρηστών και να συμβαδίζει με τη γενικότερη μείωση της ταχύτητας που έχει επέλθει σε όλες τις λειτουργίες.

Ο όρος περιγράφει δεδομένα των οποίων η κλίμακα, η κατανομή, η ποικιλομορφία και η επικαιρότητα απαιτούν τη χρήση νέων αρχιτεκτονικών και εργαλείων προκειμένου να καταστεί δυνατή η εξόρυξη γνώσης [1]. Τρία βασικά στοιχεία χαρακτηρίζουν τον όρο big data:

- **Όγκος** (**V**olume): terabytes και petabytes δεδομένων
- **Ταχύτητα** (**V**elocity): μεγάλος όγκος δεδομένων παράγεται, συλλέγεται και αλλάζει πολύ γρήγορα σε πραγματικό χρόνο από διαφορετικές τοποθεσίες όπως online συστήματα, αισθητήρες, κοινωνικά δίκτυα, αρχεία καταγραφής.

- **Ποικιλία** (Variety): ανάγκη αποθήκευσης δεδομένων τα οποία είναι δομημένα, ήμι-δομημένα ή αδόμητα, όπως αλληλογραφία ηλεκτρονικού ταχυδρομείου (emails), μηνύματα κειμένων (text messages), έγγραφα (documents), γεωγραφικά δεδομένα, 3D δεδομένα, ήχος και εικόνα.



Εικόνα 1. Χαρακτηριστικά των Big Data

Ό όγκος των δεδομένων είναι το πρωτεύον χαρακτηριστικό των big data. Τα μεγέθη τέτοιων συνόλων δεδομένων αγγίζουν την τάξη των TB/PB ή μπορούν να ποσοτικοποιηθούν με κριτήριο τον αριθμό των εγγραφών, συναλλαγών, πινάκων ή αρχείων. Επιπρόσθετα, ο λόγος που μιλάμε για μεγάλο όγκο δεδομένων σχετίζεται άμεσα με τις πηγές προέλευσης αυτών των δεδομένων, καθώς δεδομένα παράγονται από πολλές διαφορετικές πηγές, συμπεριλαμβάνοντας αρχεία καταγραφής, clickstreams, κοινωνικά δίκτυα κ.ά. Χρησιμοποιώντας τέτοιες πηγές δεδομένων για αναλυτικούς σκοπούς συναντάμε ποικιλομορφία δεδομένων, όπου τα δομημένα δεδομένα πλέον ενώνονται με αδόμητα δεδομένα όπως μηνύματα κειμένων, και ήμι-δομημένα δεδομένα, όπως αρχεία XML ή RSS. Υπάρχουν και δεδομένα τα οποία είναι δύσκολο να κατηγοριοποιηθούν. Προέρχονται από ήχο, βίντεο ή άλλες συσκευές. Επιπλέον, πολυδιάστατα δεδομένα μπορούν να συλλεχθούν

από τις αποθήκες δεδομένων για να προσαρτηθεί ιστορική αξία στο δεδομένα των big data. Έτσι, καταλήγουμε πως όταν μιλάμε για big data, η ποικιλομορφία των δεδομένων είναι εξίσου μεγάλη όσο και ο όγκος.

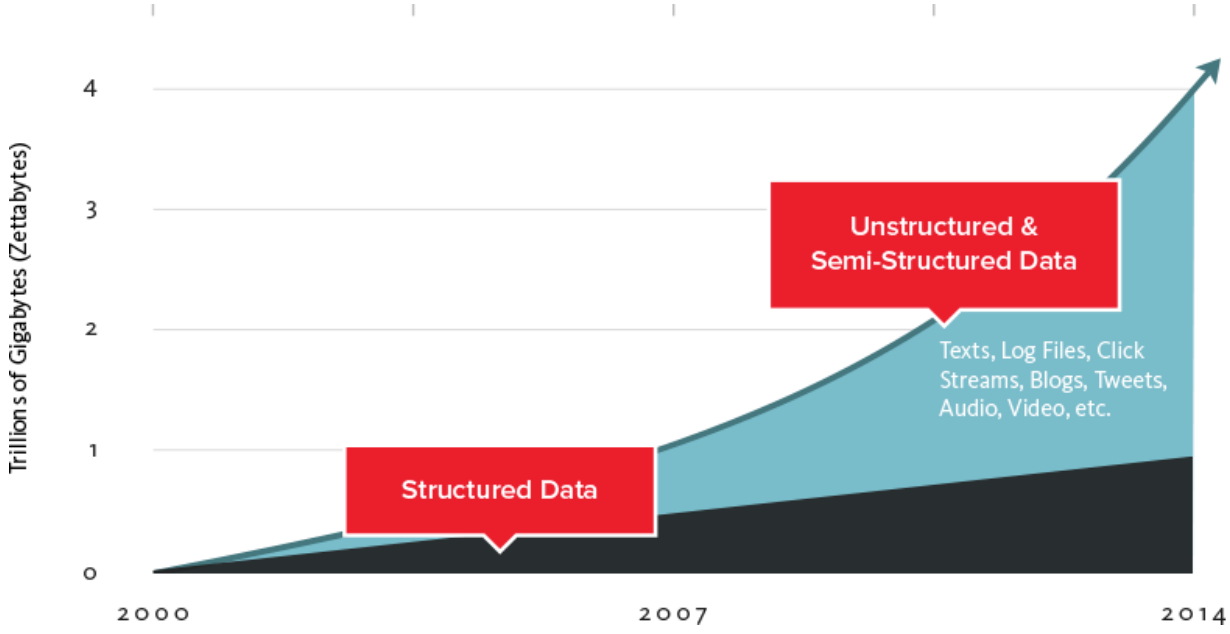
Επιπλέον, τα δεδομένα αυτά μπορούν να χαρακτηριστούν από την ταχύτητα τους. Η ταχύτητα περιγράφει την συχνότητα παραγωγής ή παροχής δεδομένων. Υπάρχει συνεχής ροή δεδομένων που συλλέγεται σε πραγματικό χρόνο από διάφορους διαδικτυακούς τόπους. Πολλοί ερευνητές μιλάνε και για την προσθήκη και τέταρτου στοιχείου, - το τέταρτο V - , (Veracity). Το τέταρτο χαρακτηριστικό επικεντρώνεται στην ποιότητα των δεδομένων. Μπορούμε να χαρακτηρίσουμε τα δεδομένα ως καλής, κακής ή απροσδιόριστης ποιότητας λόγο της ασυνέπειας τους [2].

Σήμερα, λόγω της ραγδαίας ανάπτυξης του διαδικτύου (Web 3.0) και των τεχνολογιών που το διέπουν, είναι εύκολα διαθέσιμα μεγάλα σύνολα δεδομένων, τα οποία χαρακτηρίζονται από τις λεπτομερείς πληροφορίες που μας παρέχουν. Κάθε δευτερόλεπτο, όλο και περισσότερα δεδομένα παράγονται και έχουν την ανάγκη να αποθηκευτούν και να αναλυθούν με σκοπό την εξόρυξη γνώσης. Η διαχείριση των δεδομένων έχει μεταλλαχτεί από μια σημαντική ικανότητα σε έναν κρίσιμο διαφοροποιητή, ο οποίος μπορεί να καθορίσει και να αναδείξει ποιος θα είναι ο νικητής σε μια αγορά. Εταιρείες αλλά και κρατικοί φορείς έχουν αρχίσει να επωφελούνται από τις καινοτομίες που προσφέρουν οι διαδικτυακές εφαρμογές. Οι οργανισμοί αυτοί καθορίζουν νέες πρωτοβουλίες και αξιολογούν εκ νέου τις στρατηγικές τους για να εξετάσουν πώς μπορούν να βελτιώσουν την επιχειρηματικότητα τους χρησιμοποιώντας big data. Κατά την διαδικασία αυτή, ανακαλύπτουν ότι ο όρος big data δεν αναφέρεται σε μία τεχνολογία, τεχνική ή πρωτοβουλία. Μάλλον, είναι μια τάση που περιλαμβάνει πολλούς τομείς. Μπορούμε να πούμε ότι είναι ένα σύνολο διαδικασιών που απαιτούν γνώσεις πληροφορικής, στατιστικής, μηχανικής εκμάθησης και επιχειρησιακής νοημοσύνης, που απλά εμπλέκουν και χρησιμοποιούν δεδομένα. Δεδομένα διαφορετικά μεταξύ τους που αλλάζουν πολύ γρήγορα και με όγκο μεγάλο για να αντιμετωπιστεί και να διαχειριστεί με τις παραδοσιακές συμβατικές τεχνολογίες. Όπως αναφέραμε και πριν τα δεδομένα αυτά χαρακτηρίζονται στις σχετικές βιβλιογραφίες ως τα 3Vs – Volume - Variety – Velocity [3].

Πλέον, καινούριες τεχνολογίες καθιστούν δυνατή την ανάκτηση πληροφορίας από τα big data. Για παράδειγμα, λιανοπωλητές παρακολουθούν τα 'clicks΄ των διαδικτυακών χρηστών, με σκοπό τον προσδιορισμό της συμπεριφοράς τους και έχουν ως στόχο την βελτίωση των εκστρατειών τους, την τιμολόγηση καθώς και την αποθεματοποίηση.

Κυβερνήσεις, ακόμα και η Google μπορούν να ανιχνεύσουν και να παρακολουθήσουν την εμφάνιση κρουσμάτων μίας νόσου, μέσω των κοινωνικών δικτύων. Εταιρείες πετρελαίου και φυσικού αερίου μπορούν να επεξεργαστούν τα δεδομένα των αισθητήρων από τον εξοπλισμό γεωτρήσεων τους, με σκοπό να πάρουν πιο αποτελεσματικές και ασφαλείς αποφάσεις για γεωτρήσεις.

Τέλος, ο όρος big data περιγράφει σύνολα δεδομένων τα οποία είναι τόσο μεγάλα (ο όρος ' big ' είναι κινούμενος καθώς ένα σύνολο δεδομένων μπορεί σήμερα να είναι μεγάλο αλλά αύριο να θεωρείται μικρό) και πολύπλοκα, τα οποία είναι αδύνατο να επεξεργαστούμε και να αναλύσουμε με τις παραδοσιακές τεχνικές και εφαρμογές. Η ανάλυση τέτοιου είδους και όγκου δεδομένων όπως αναφέραμε στα παραπάνω παραδείγματα, επιτρέπει στους αναλυτές και στους ειδικούς την λήψη αποφάσεων. Να προβλέψουν δηλαδή τάσεις των επιχειρήσεων, την πρόληψη ασθενειών, την καταπολέμηση του εγκλήματος (π.χ το noSQL σύστημα της MongoDB χρησιμοποιείται για την καταπολέμηση του εγκλήματος στο Σικάγο) και ούτω καθεξής [4].



Εικόνα 2. Ραγδαία αύξηση των δεδομένων,ως προς τον όγκο και την δομή τους

1.2 Τεχνολογίες Big Data

Το τοπίο του big data αποτελείται από τεχνολογίες οι οποίες μας επιτρέπουν την συλλογή, αποθήκευση, επεξεργασία και ανάλυση των δεδομένων. Μπορούμε να διαχωρίσουμε τις τεχνολογίες αυτές σε δύο κατηγορίες. Συστήματα τα οποία παρέχουν δυνατότητες αποθήκευσης, επεξεργασίας και διαχείρισης δεδομένων σε πραγματικό χρόνο (real-time), όπου τα δεδομένα κατά κύριο λόγο συλλαμβάνονται και αποθηκεύονται, και συστήματα που παρέχουν αναλυτικές ικανότητες για αναδρομικές και σύνθετες αναλύσεις. Στα συστήματα ανάλυσης μπορεί να χρησιμοποιηθεί ένα μέρος ή ακόμα και όλο το σύνολο των δεδομένων. Αυτές οι δύο τεχνολογίες είναι συμπληρωματικές και συχνά αναπτύσσονται μαζί.

Η κάθε μία από αυτές τις τεχνολογίες έχει οδηγήσει στην δημιουργία καινούριων αρχιτεκτονικών. Τα συστήματα αποθήκευσης, επεξεργασίας και διαχείρισης, όπως και οι μη σχεσιακές βάσεις τύπου NoSQL, έχουν επικεντρωθεί στην εξυπηρέτηση ταυτόχρονων αιτημάτων προς την βάση σε πραγματικό χρόνο και εμφανίζουν χαμηλό latency στην ανάκτηση των πληροφοριών ακόμη και σε πολύπλοκα αιτήματα. Από την άλλη πλευρά, τα συστήματα ανάλυσης επικεντρώνονται στην υψηλή διεκπεραιωτική ικανότητα, όπου τα αιτήματα προς την βάση μπορεί να είναι πολύ πολύπλοκα και να αφορούν τα περισσότερα αν όχι όλα τα δεδομένα ενός συστήματος. Και στις δύο περιπτώσεις, τα συστήματα έχουν την τάση να λειτουργούν σε πολλούς διακομιστές (servers) πάνω σε ένα cluster ως κατανεμημένα συστήματα και να διαχειρίζονται δεκάδες ή εκατοντάδες terabytes δεδομένων γύρω από δισεκατομμύρια εγγραφές [3].

1.2.1 Συστήματα αποθήκευσης και διαχείρισης big data

Οι παραδοσιακές τεχνικές για την αποθήκευση και ανάκτηση δομημένων δεδομένων περιλαμβάνουν τις σχεσιακές βάσεις δεδομένων (RDBMS), τις αποθήκες δεδομένων (data werehouses) και τα data marts. Τα δεδομένα αφού συλλεχθούν, έχουν την ανάγκη για

αποθήκευση μέσω διαδικασιών *ETL* (Extract, Transform, Load) ή *ELT* (Extract, Load, Transform). Διαδικασίες οι οποίες αρχικά εισάγουν τα δεδομένα από εξωτερικές πηγές, μεταμορφώνουν τα δεδομένα για να ικανοποιούν τις λειτουργικές ανάγκες ενός συστήματος και τέλος φορτώνουν τα δεδομένα σε μία βάση δεδομένων ή σε μία αποθήκη δεδομένων.

Τα συστήματα αποθήκευσης και διαχείρισης δεδομένων, έχουν καταληφθεί από την αγορά πριν από το 1990 από σχεσιακά συστήματα τύπου SQL. Όμως λόγο του Big Data και των καινούριων αρχιτεκτονικών που αναπτύσσονται συνέχεια στο σύγχρονο υπολογιστικό περιβάλλον, αναπτύχθηκαν οι NoSQL τεχνολογίες ή NoSQL Big Data συστήματα[μη σχεσιακά μεγάλου όγκου δεδομένων συστήματα] για να αντιμετωπίσουν τις αδυναμίες των σχεσιακών βάσεων δεδομένων. Τα μη σχεσιακά συστήματα αναπτύχθηκαν κατά κύριο λόγο για την αποθήκευση και την διαχείριση αδόμητων, ή μη σχεσιακών δεδομένων. Σκοπός αυτών των βάσεων δεδομένων είναι η ογκώδης επεκτασιμότητα, το ευέλικτο μοντέλο δεδομένων και η απλότητα στην ανάπτυξη και διαχείριση εφαρμογών. Τέτοιου είδους συστήματα αναπτύχθηκαν για να επωφεληθούν από τις νέες αρχιτεκτονικές του cloud computing που προέκυψαν την τελευταία δεκαετία, επιτρέποντας φθηνότερους και αποτελεσματικότερους μαζικούς υπολογισμούς. Επίσης, τέτοια συστήματα βάσεων δεδομένων υποστηρίζονται όλο και περισσότερο από τις αρχιτεκτονικές των κατανεμημένων συστημάτων παρέχοντας υψηλή διαθεσιμότητα των δεδομένων και ανοχή σε σφάλματα (fault-tolerance) μέσω της αντιγραφής (replication) των δεδομένων και τις ικανότητας για οριζόντια επεκτασιμότητα της βάσης.

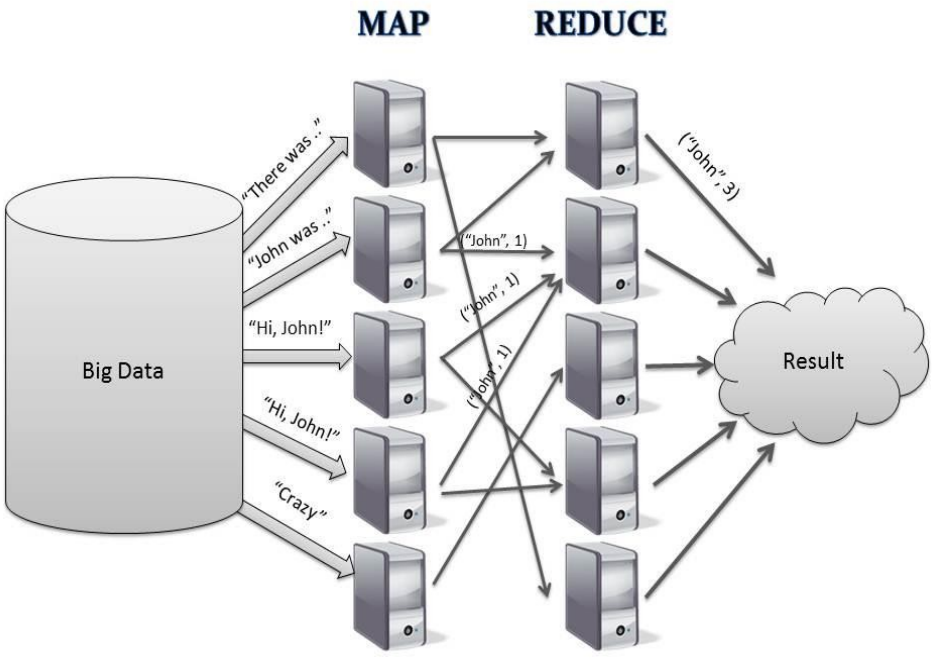
1.2.2 Συστήματα ανάλυσης big data

Από την άλλη πλευρά, ο φόρτος εργασίας που σχετίζεται με την ανάλυση των δεδομένων, τείνει να διαχειρίζεται από συστήματα βάσεων τύπου MPP (*Massive-Parallel-Processing)* και από το μοντέλο MapReduce. Αυτές οι τεχνολογίες είναι ακόμη μια αντίδραση στους περιορισμούς των παραδοσιακών σχεσιακών βάσεων δεδομένων και την ικανότητα τους για κλιμάκωση πέρα από τους πόρους ενός μόνο διακομιστή. Επιπλέον, το μοντέλο MapReduce παρέχει μια νέα μέθοδο για την ανάλυση των δεδομένων που είναι συμπληρωματικό προς τις δυνατότητες που παρέχονται από την SQL.

Το μοντέλο *MapReduce* είναι ένα παράλληλο προγραμματιστικό μοντέλο, εμπνευσμένο από τις συναρτήσεις 'Map ()' και 'Reduce ()' των συναρτησιακών γλωσσών προγραμματισμού,

το οποίο είναι κατάλληλο για την επεξεργασία των big data. Είναι ο πυρήνας του συστήματος *Apache Hadoop*, και εκτελεί την επεξεργασία των δεδομένων και τις λειτουργίες ανάλυσης τους. Το Apache Hadoop είναι ένα σύστημα που χρησιμοποιείται για την διαχείριση κατανεμημένων δεδομένων που αποθηκεύονται στην υποδομή που προσφέρεται μέσω cloud. Σύμφωνα με το EMC, το μοντέλο MapReduce βασίζεται στην προσθήκη περισσότερων μηχανημάτων ή πόρων, παρά την αύξηση της ισχύς ή της χωρητικότητας ενός μηχανήματος. Η βασική ιδέα του MapReduce είναι η κατανομή ενός task σε στάδια (stages) και η εκτέλεση των σταδίων αυτών παράλληλα, προκειμένου να μειωθεί ο χρόνος που απαιτείται για να ολοκληρωθεί το task [5].

Η πρώτη φάση του μοντέλου, είναι να αντιστοιχήσει τις εισαγόμενες τιμές δεδομένων (input) σε ένα σετ από ζευγάρια κλειδιών-τιμών (key/value) και να τα δώσει ως έξοδο (output). Η συνάρτηση Map () διαμερίζει μεγάλα υπολογιστικά έργα (tasks) σε μικρότερα και τα αναθέτει στα κατάλληλα ζεύγη κλειδιών-τιμών. Έτσι, αδόμητα δεδομένα όπως κείμενα, μπορούν να αντιστοιχιστούν (map) σε ένα δομημένο ζευγάρι κλειδιών-τιμών. Για παράδειγμα, το key θα μπορούσε να είναι μία λέξη από το κείμενο και το value ο αριθμός των εμφανίσεων τις συγκεκριμένης λέξης στο κείμενο. Αυτό το αποτέλεσμα εξόδου (output) είναι το όρισμα που εισάγεται (input) στην συνάρτηση Reduce (). Στην συνέχεια η συνάρτηση reduce () εκτελεί συνδυασμούς της τιμής του output, συνδυάζοντας όλες τις τιμές που μοιράζονται το ίδιο key, παρέχοντας το τελικό αποτέλεσμα από την όλη διαδικασία. Στην Εικόνα 3 παρουσιάζεται η διαδικασία.



Εικόνα 3. Παράδειγμα MapReduce

Καθώς οι εφαρμογές κερδίζουν όλο και περισσότερο έδαφος και οι χρήστες τους παράγουν αυξανόμενους όγκους δεδομένων, υπάρχει ένας αριθμός αναδρομικών αναλυτικών φόρτων εργασίας που παρέχουν πραγματική αξία για τις επιχειρήσεις. Το μοντέλο MapReduce είναι πλέον η πρώτη επιλογή για την ανάλυση big data. Ωστόσο, μερικά μη σχεσιακά συστήματα παρέχουν εγγενή λειτουργικότητα του μοντέλου MapReduce, που επιτρέπει την ανάλυση των δεδομένων. Το NoSQL σύστημα της MongoDB προσφέρει μια ελαφρώς μεταλλαγμένη μορφή του MapReduce. Εναλλακτικά, τα δεδομένα μπορούν φορτωθούν από ένα μη σχεσιακό σύστημα σε ένα αναλυτικό σύστημα όπως το Hadoop. Υπάρχει η δυνατότητα χρήσης του συστήματος της MongoDB και του Hadoop, όπου το πρώτο σύστημα χρησιμοποιείται για την αποθήκευση δεδομένων και το δεύτερο για offline επεξεργασία και ανάλυση των δεδομένων [6].

	OPERATIONAL	ANALYTICAL
Latency	1 ms – 100 ms	1 min – 100 min
Concurrency	1000 – 100,000	1 – 10
Access Pattern	Writes and Reads	Reads
Queries	Selective	Unselective
Data Scope	Operational	Retrospective
End User	Customer	Data Scientist
Technology	NoSQL	Hadoop,MapReduce MPP Database

Πίνακας 1. Σύγκριση τεχνολογιών, Operational vs Analytical

Κεφάλαιο 2. Μη σχεσιακές βάσεις δεδομένων

2.1 Εισαγωγή

Για πολλά χρόνια, το σχεσιακό μοντέλο δεδομένων ήταν αυτό που επικρατούσε σε όλους τους τομείς της πληροφορικής. Στο διαδίκτυο, στα πληροφοριακά συστήματα, στους προσωπικούς υπολογιστές. Λόγω της απήχησης αυτού του μοντέλου αναπαράστασης των δεδομένων, πληροφοριών και σχέσεων μεταξύ τους, οι σχεσιακές βάσεις δεδομένων ήταν στην ουσία η μόνη επιλογή. Τα τελευταία χρόνια όμως, λόγω της ταχύτατης αύξησης των χρηστών του διαδικτύου, του ρυθμού κυκλοφορίας των πληροφοριών, των big data και τελικά του cloud computing, οι επιστήμονες του χώρου και οι προγραμματιστές συνειδητοποίησαν πως οι σχεσιακές βάσεις δεδομένων δεν ήταν κατάλληλες πια για ορισμένα προβλήματα.

Ξεκίνησαν να αντιμετωπίζουν προβλήματα κλιμάκωσης όταν οι σχεσιακές εφαρμογές είχαν επιτυχία και αυξανόταν η χρήση τους. Οι ενώσεις (*joins*), που είναι αναπόφευκτα σε οποιαδήποτε, ακόμα και μικρού μεγέθους, κανονικοποιημένη σχεσιακή βάση, προκαλούν μεγάλες καθυστερήσεις. Ακόμη, η εξασφάλιση της συνέπειας (*consistency*), που είναι βασική αρχή για τις σχεσιακές βάσεις, δεν επιτρέπει ταυτόχρονα αναγνώσεις (*reads*) και εγγραφές (*writes*) στη βάση, πράγμα που απαιτεί κλείδωμα (*lock*) μέρους της βάσης σε κάθε περίπτωση, με αποτέλεσμα τα αντίστοιχα δεδομένα να μην είναι διαθέσιμα στους χρήστες. Αυτό μπορεί να γίνει αφόρητο υπό την πίεση μεγάλου φόρτου δεδομένων, αφού τα locks θα

έχουν σαν αποτέλεσμα οι χρήστες να «ανταγωνίζονται» για τα δεδομένα και να κάνουν ουρά περιμένοντας τη σειρά τους να διαβάσουν ή να γράψουν κάτι. Έτσι άρχισαν να ερευνούν εναλλακτικούς τρόπους αποθήκευσης δεδομένων. Αυτή τους η αναζήτηση είχε ως αποτέλεσμα τη δημιουργία μη σχεσιακών συστημάτων βάσεων δεδομένων τα οποία θα μπορούσαν να ανταποκριθούν καλύτερα σε κατανεμημένο περιβάλλον. Έτσι, το 2009 ξεκίνησε η χρήση του όρου NoSQL ή ' Not only SQL ' για την αναφορά σε αυτόν το νέο, μη σχεσιακό τρόπο αποθήκευσης δεδομένων. Η φιλοσοφία των NoSQL βάσεων άρχισε σταδιακά να κερδίζει όλο και περισσότερους οπαδούς στο διαδίκτυο και μάλιστα εταιρείες-κολοσσοί όπως οι: Google, Amazon, Facebook, Twitter, Digg, Reddit, Linkedin, Sourceforge, Bing χρησιμοποιούν μη σχεσιακές βάσεις τώρα πια. Ο όγκος των δεδομένων που διαχειρίζονται τέτοιες εταιρείες είναι της τάξης πολλών petabyte και επομένως θα ήταν αδύνατη η κλιμάκωσή τους με κάποιο σχεσιακό σύστημα βάσης δεδομένων. Κι αυτό γιατί οι σχεσιακές βάσεις δεδομένων συναντούν περιορισμούς όσον αφορά την αντιμετώπιση προβλημάτων όπως είναι η εξόρυξη δεδομένων, το Web 2.0 (και σύντομα 3.0), το cloud computing, τα μη γραμμικά σε εκτέλεση ερωτήματα κα. Αντιθέτως, ο κατανεμημένος μηχανισμός που προσφέρουν οι μη σχεσιακές βάσεις καθιστά τις εφαρμογές οριζόντια κλιμακώσιμες και δίνει τη δυνατότητα εκμετάλλευσης πολλαπλών υπολογιστικών συστημάτων με πολυπύρηνους επεξεργαστές οπότε καθιστά εφικτή την αντιμετώπιση των προαναφερθέντων προβλημάτων. Επίσης, ορισμένα hot use-cases όπως αραιά δεδομένα, συνεργατική επεξεργασία, κοινωνικός γράφος, αρχεία καταγραφής, τα μη σχεσιακά συστήματα τα καλύπτουν καλύτερα από τα σχεσιακά.

Ο όρος *NoSQL* περιλαμβάνει μια μεγάλη ποικιλία διαφορετικών τεχνολογιών βάσεων δεδομένων που αναπτύχθηκαν σε απόκριση σε μια ραγδαία αύξηση του όγκου των δεδομένων που αποθηκεύονται για τους χρήστες, τα αντικείμενα και τα προϊόντα, την συχνότητα που είναι προσβάσιμα αυτά τα δεδομένα καθώς επίσης και τις ανάγκες για υψηλότερη απόδοση όπως και ανάγκες επεξεργασίας. Από την άλλη πλευρά, οι σχεσιακές βάσεις δεδομένων δεν έχουν σχεδιαστεί για να αντιμετωπίζουν προκλήσεις κλιμάκωσης και ευκινησίας, προκλήσεις που παρατηρούνται στις σύγχρονες εφαρμογές, ούτε έχουν κατασκευαστεί για να επωφεληθούν από τον φθηνό τρόπο αποθήκευσης και την επεξεργαστική ισχύ που είναι διαθέσιμη στην σημερινή εποχή [7].

Συμπερασματικά, έχει ήδη ξεκινήσει η εξάπλωση των μη σχεσιακών συστημάτων σε διαδικτυακές εφαρμογές και αναμένεται να κατακλύσουν το διαδίκτυο. Ήδη μεγάλες εταιρίες έχουν μεταφέρει όλες τους τις εφαρμογές που χρησιμοποιούσαν σχεσιακές βάσεις

δεδομένων σε μη σχεσιακές. Αναμένεται να ακολουθήσουν κι οι υπόλοιπες εταιρίες, ούτως ώστε να εκμεταλλευθούν και αυτές την κλιμάκωση που προσφέρουν οι μη σχεσιακές βάσεις δεδομένων και την εκμετάλλευση περισσότερων υπολογιστικών πόρων. Παρομοίως, έχει εξαπλωθεί και η ανάπτυξη μη σχεσιακών συστημάτων, όπου έως σήμερα αριθμούνται πάνω από 150 διαφορετικές πλατφόρμες NoSQL, όλες ελεύθερου λογισμικού [8].

2.2 Βάσεις Δεδομένων

Όπως και με την μουσική, οι βάσεις δεδομένων μπορούν γενικά να κατηγοριοποιηθούν σε παραπάνω από ένα είδη. Για παράδειγμα, ένα τραγούδι μπορεί να έχει παρόμοιους, ακόμα και όμοιους στίχους με ένα άλλο τραγούδι, παρόλα αυτά μερικά τραγούδια είναι πιο κατάλληλα σε συγκεκριμένες καταστάσεις. Παρομοίως, μερικές βάσεις δεδομένων είναι καλύτερες για κάποια προβλήματα από άλλες. Η ερώτηση που πρέπει να γίνεται πάντα πριν επιλέξουμε βάση δεδομένων, δεν είναι το αν " Μπορώ να χρησιμοποιήσω αυτήν την βάση δεδομένων για να αποθηκεύσω και να ανακτήσω τα δεδομένα μου; " αλλά το αν " Πρέπει να χρησιμοποιήσω αυτήν την βάση; " καθώς αναλόγως με το πρόβλημα που καλούμαστε να αντιμετωπίσουμε σαν προγραμματιστές, θα επιλέξουμε και την ανάλογη με τη φύση του προβλήματος βάση δεδομένων.

Σε αυτήν την ενότητα, θα αναλύσουμε πέντε βασικά είδη βάσεων δεδομένων. Ξεκινώντας από τις σχεσιακές βάσεις δεδομένων, συνεχίζοντας με τα διαφορετικά είδη των μη σχεσιακών βάσεων, τα οποία διακρίνονται για τον τρόπο που αναπαριστούν τα δεδομένα, στο τέλος θα προτείνεται μια πλατφόρμα, η οποία διακρίνεται ως σήμερα σε κάθε είδος βάσεων δεδομένων.

2.2.1 Σχεσιακές Βάσεις δεδομένων

Το σχεσιακό μοντέλο αναπαράστασης δεδομένων είναι το πρώτο που έρχεται στο μυαλό των περισσότερων που έχουν εμπειρία με τις βάσεις δεδομένων. Τα σχεσιακά συστήματα τα οποία συνδυάζονται και με ένα περιβάλλον διαχείρισης τους (*RDBMSs*), είναι βασισμένα

στην θεωρία των συνόλων και εφαρμόζονται ως δισδιάστατοι πίνακες με γραμμές και στήλες. Μια σχεσιακή βάση δεδομένων αποτελείται από ένα σύνολο από πίνακες, καθένας εκ των οποίων έχει ένα μοναδικό όνομα. Πίνακας είναι μία δισδιάστατη δομή δεδομένων. Κάθε στήλη του πίνακα αφορά ένα συγκεκριμένο γνώρισμα (*attribute*). Μία γραμμή ενός πίνακα αντιπροσωπεύει μία σχέση ενός συνόλου από τιμές για τα αντίστοιχα γνωρίσματα. Αφού ένας πίνακας είναι ένα σύνολο από τέτοιες σχέσεις υπάρχει μια στενή σχέση μεταξύ ενός πίνακα και της μαθηματικής ιδέας της σχέσης (*relation*), από την οποία παίρνει το όνομά του το σχεσιακό μοντέλο δεδομένων. Δεν είναι απαραίτητο για μια σχέση να έχει γραμμές για να θεωρείται σχέση. Ακόμα και αν η σχέση δεν περιέχει δεδομένα, αυτή παραμένει ορισμένη με το σετ των ιδιοτήτων της. Ο τρόπος αλληλεπίδρασης με ένα *RDBMS* είναι υποβάλλοντας ερωτήματα σε SQL (*Structured Query Language*). Οι τιμές των δεδομένων μπορούν να είναι αριθμοί, αλφαριθμητικά, ημερομηνίες ή άλλοι τύποι δεδομένων οι οποίοι ορίζονται αυστηρά από το σύστημα. Τέλος, το πιο σημαντικό των σχεσιακών συστημάτων είναι ότι ένας πίνακας μπορεί να ενωθεί (*JOIN*) με κάποιον άλλον πίνακα με αποτέλεσμα την δημιουργία ενός καινούριου πιο πολύπλοκου πίνακα [9].

Παραδείγματα: Υπάρχουν πολλές σχεσιακές βάσεις δεδομένων ελεύθερου ανοιχτού λογισμικού όπου μπορεί να διαλέξει κανείς, συμπεριλαμβάνοντας την MySQL, PostgreSQL, SQLite, H2, HSQLDB, κ.ά .

2.2.2 Ιδιότητες ACID

Συναλλαγή (*transaction*), ονομάζεται κάθε σειρά ενεργειών, όπου κάθε ενέργεια διαβάζει (*read*) ή γράφει (*write*) αντικείμενα σε μία βάση δεδομένων. Η εφαρμογή των ιδιοτήτων ACID σε κάθε συναλλαγή εγγυάται την αξιοπιστία των σχεσιακών βάσεων δεδομένων. Οι ιδιότητες αυτές εξασφαλίζουν ότι:

Ατομικότητα, Atomicity - Εξασφαλίζεται ότι είτε θα πραγματοποιηθούν όλες οι πράξεις μιας συναλλαγής είτε ότι θα αποτύχουν όλες, αφήνοντας την βάση ανεπηρέαστη. Σε περίπτωση κατάρρευσης του συστήματος κατά την διάρκεια μιας συναλλαγής θα

αναιρεθούν ό,τι αλλαγές έχουν γίνει στην βάση και αυτή θα έρθει στην μορφή που ήταν πριν ξεκινήσει η εκτέλεση της συναλλαγής.

Συνέπεια, Consistency – Εξασφαλίζεται ότι πριν και μετά την εκτέλεση μιας συναλλαγής, οι κανόνες και οι περιορισμοί που διέπουν τη βάση δεδομένων πληρούνται. Μία συναλλαγή δεν μπορεί να αφήσει την βάση σε ασυνεπή μορφή.

Απομόνωση, Isolation - Αναφέρεται στην απαίτηση ότι όλες οι ενέργειες δεν μπορούν να έχουν πρόσβαση ή να δουν δεδομένα τα οποία τροποποιούνται εκείνη την στιγμή από μια συναλλαγή η οποία δεν έχει ακόμα ολοκληρωθεί. Κάθε συναλλαγή δεν πρέπει να ξέρει αν υπάρχουν άλλες συναλλαγές που εκτελούνται ταυτόχρονα, αλλά να περιμένουν την ολοκλήρωση μιας συναλλαγής ώστε να δουν/τροποποιήσουν τα δεδομένα τα οποία χρειάζεται και η άλλη συναλλαγή.

Μονιμότητα, Durability – Οι αλλαγές που προκαλεί στη βάση μια συναλλαγή που επιτυγχάνει θα παραμείνουν ακόμα και μετά από κατάρρευση του συστήματος [10].

2.2.3 Μη σχεσιακές βάσεις δεδομένων

2.2.4 BASE

Το *ACID* είναι το μοντέλο που παρέχει συνέπεια και αξιοπιστία στις βάσεις δεδομένων (*strong consistency*), γι' αυτό και αποτελεί βασική αρχή των σχεσιακών βάσεων. Το μοντέλο αυτό εξασφαλίζει ότι κάθε χρονική στιγμή όλες οι διεργασίες ενός κατανεμημένου συστήματος βλέπουν την τελευταία έκδοση των δεδομένων. Για παράδειγμα, έστω δεδομένο x το οποίο διατηρείται σε τρεις κόμβους. Κάθε φορά που γίνεται update η νέα έκδοση του x αντιγράφεται σε δύο κόμβους. Κάθε read διαβάζει από δύο κόμβους με τουλάχιστον τον έναν από αυτούς να έχει την τελευταία έκδοση του x. Σε περίπτωση σφάλματος της σύνδεσης τα writes δεν επιτυγχάνονται και η βάση γίνεται προσωρινά μη διαθέσιμη.

Από την άλλη πλευρά, τα μη σχεσιακά συστήματα ακολουθούν άλλες ιδιότητες γνωστές ως *BASE (Eventual consistency)*. Σε μία κατανεμημένη βάση δεδομένων, ο όρος eventual consistency αντιπροσωπεύει μια μορφή ασθενέστερης συνέπειας των δεδομένων, με τον οποίο βελτιώνεται η απόδοση και η διαθεσιμότητα της βάσης. Σε αυτό το μοντέλο εξασφαλίζεται ότι μετά από ένα χρονικό διάστημα στο οποίο δεν γίνονται updates, θα έχουν γίνει με τη σωστή σειρά όλα τα updates και όλα τα αντίγραφα στην βάση θα είναι συνεπή. Ωστόσο, μπορεί να υπάρξει μια μορφή ασυνέπειας, ένα χρονικό διάστημα όπου μπορεί να διαβαστούν παλαιότερες εκδόσεις των δεδομένων (αν το σύστημα διαβάζει από εκείνους τους κόμβους). Αυτό συμβαίνει συχνά λόγο της ασύγχρονης διατήρησης αντιγράφων (replication) στους δευτερεύοντες κόμβους, όπου ο κόμβος δεν έχει ενημερωθεί ακόμα για τα update. Σε περίπτωση αστοχίας του συστήματος ή λόγου σφαλμάτων στην σύνδεση, η βάση συνεχίζει να είναι διαθέσιμη, με τον κίνδυνο όμως να διαβαστούν παλαιότερες εκδόσεις των δεδομένων (stale data). Το ακρώνυμο ορίστηκε από τον Brewer, ο οποίος στην συνέχεια πρότεινε το θεώρημα CAP.

Ένα BASE σύστημα θυσιάζει κάπως την συνέπεια για την διαθεσιμότητα και την απόδοση του συστήματος.

- **Basically Available:** Η βάση είναι συνεχώς διαθέσιμη για πρόσβαση ακόμα και αν μερικοί κόμβοι είναι μη διαθέσιμοι. Εγγύηση διαθεσιμότητας συστήματος.
- **Soft-State**: Η βάση δεν χρειάζεται να είναι συνεπή πάντα και μπορεί να ανέχεται ασυνέπεια για συγκεκριμένη χρονική στιγμή διαθέτοντας δεδομένα ενδεχόμενης συνέπειας (eventually consistent). Είναι θέμα της εφαρμογής να εγγυηθεί συνέπεια και όχι της βάσης.
- **Eventually Consistent**: Ύστερα από κάποια χρονική περίοδο, η βάση επανέρχεται σε γνωστή συνεπή κατάσταση.

Σε σύγκριση με το ιδιότητες που ακολουθούν τα σχεσιακά συστήματα (ACID) παρατηρούμε ότι τα μη σχεσιακά τύπου NoSQL παραβλέπουν κάπως την συνέπεια των δεδομένων με σκοπό την υψηλή διαθεσιμότητα, την ανοχή σε σφάλματα και την απόδοση του συστήματος Για παράδειγμα, σε ένα κοινωνικό σύστημα όπως το Facebook/Twitter, η βάση έχει την ανάγκη να είναι διαθέσιμη πάντα στα requests των χρηστών έχοντας ίσως δεδομένα ενδεχόμενης συνέπειας. Δηλαδή, αν κάποιος χρήστης κάνει ένα status update, το γεγονός ότι θα εμφανιστεί η ενημέρωση στον τοίχο ενός άλλου χρήστη σε κάποια χρονική περίοδο, ίσως μετά από 10sec, είναι αποτέλεσμα ενδεχόμενης συνέπειας.

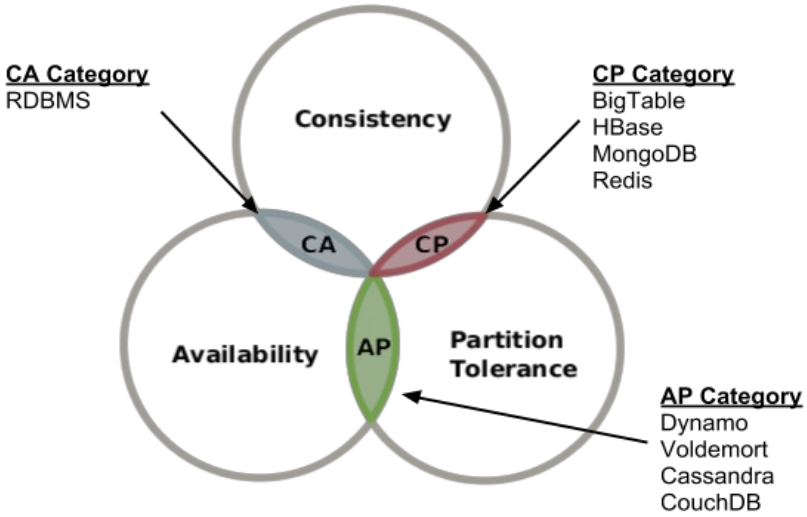
Ωστόσο, υπάρχουν συστήματα τα οποία ακολουθούν τις ACID ιδιότητες όπως το FoundationDB, κ.ά [11].

2.2.5 Θεώρημα CAP

Σύμφωνα με το θεώρημα CAP, το οποίο προτάθηκε από τον Eric Brewer το 2002, ένα κατανεμημένο υπολογιστικό σύστημα δε μπορεί να παρέχει ταυτόχρονα συνέπεια (*consistency*), διαθεσιμότητα (*availability*) και ανοχή τμημάτων (*partition tolerance*). Μόνο **δύο** από αυτές τις ιδιότητες μπορούν να ικανοποιούνται ταυτόχρονα.

Με τον όρο *Consistency* εννοούμε την υλοποίηση του μοντέλου strong consistency, όπου όλοι οι κόμβοι του κατανεμημένου συστήματος βλέπουν κάθε χρονική στιγμή τα ίδια δεδομένα. Με τον όρο *Availability* εννοούμε πως οι χρήστες μπορούν κάθε χρονική στιγμή να διαβάσουν και να γράψουν δεδομένα. Με τον όρο *Partition Tolerance* εννοούμε πως το σύστημα έχει ανοχή σε προσωρινά προβλήματα σύνδεσης και κατάρρευσης υλικού και επιτρέπει τον χωρισμό του συστήματος σε τμήματα τα οποία συνεχίζουν να δουλεύουν κανονικά.

Στις μη σχεσιακές βάσεις δεδομένων ή κατανεμημένες βάσεις, το Partition Tolerance θεωρητικά πρέπει να ικανοποιείται. Άρα για τέτοιου είδους συστήματα, υπάρχει η επιλογή σε CP ή AP, συστήματα δηλαδή που είναι αυστηρά συνεπή και αντέχουν στην κλιμάκωση, ή συστήματα πάντα διαθέσιμα και κατανεμημένα, προσφέροντας για τα δεδομένα τους ενδεχόμενη συνέπεια. Ωστόσο, τα περισσότερα συστήματα δίνουν την δυνατότητα στους χρήστες τους να ρυθμίσουν τις επιλογές για συνέπεια ή διαθεσιμότητα, ανάλογα με τις ανάγκες του κάθε προβλήματος. [12]



Εικόνα 4. Θεώρημα CAP

2.3 Χαρακτηριστικά μη σχεσιακών συστημάτων

Παρόλο που υπάρχουν πολλές διαφορετικές μη σχεσιακές βάσεις δεδομένων και οι διαφορές από σύστημα σε σύστημα μπορεί να είναι αρκετές, οι βάσεις τύπου NoSQL μοιράζονται κάποια κοινά χαρακτηριστικά. Είναι μη σχεσιακά, χειρίζονται ήμιδομημένα και αδόμητα δεδομένα, εφαρμόζονται σε κατανεμημένο περιβάλλον, είναι ανοιχτού κώδικα και οριζόντια κλιμακώσιμα.

2.3.1 Απλά και Δυναμικά μη σχεσιακά μοντέλα δεδομένων

Οι σχεσιακές βάσεις δεδομένων απαιτούν τον ορισμό σχήματος πριν αποθηκευτούν τα δεδομένα. Ο χρήστης πρέπει να καθορίσει το μοντέλο δεδομένων και να ορίσει τους τύπους που θα χρησιμοποιήσει. Οι μη σχεσιακές βάσεις από την άλλη πλευρά, προσφέρουν δυναμικά σχήματα ή ελεύθερα σχήματα (schema-free) και είναι σχεδιασμένα να χειρίζονται ποικιλία δομών δεδομένων. Επιτρέπουν την εισαγωγή δεδομένων χωρίς προκαθορισμένο σχήμα και ο χρήστης μπορεί δυναμικά να ορίσει καινούρια γνωρίσματα και τύπους.

2.3.2 Οριζόντια και αυτόματη κλιμάκωση

Προσφέρουν την ικανότητα να κλιμακώνονται οριζόντια (*scale-out*) αντίθετα με της

σχεσιακές βάσεις όπου συνήθως το σύστημα κλιμακώνεται κάθετα (*scale-up*). Στις περιπτώσεις κάθετης κλιμάκωσης, γίνεται αναβάθμιση επεξεργαστών, προσθήκη RAM και χωρητικότητας με αντίκτυπο το υψηλό κόστος. Από την άλλη πλευρά, λόγο του κατανεμημένου μηχανισμού των μη σχεσιακών βάσεων, αρκεί η προσθήκη επιπλέον μηχανημάτων στα ήδη υπάρχοντα. Δεν υπάρχει περιορισμός στους κόμβους που θα προστεθούν στο cluster και μερικά συστήματα υποστηρίζουν αυτόματη κλιμάκωση, δηλαδή διαμερίζουν τα δεδομένα και τα requets αυτόματα σε κάθε κόμβο που προστίθεται. Με αυτόν τον τρόπο, υπάρχει η δυνατότητΗα εκμετάλλευσης των υπηρεσιών που προσφέρονται από το *cloud computing*, όπου με ελάχιστο κόστος προσφέρονται δυνατότητες επεξεργασίας και αποθήκευσης. Συχνά οι μη σχεσιακές βάσεις αναφέρονται και ως cloud databases, διότι εφαρμόζονται και συνεργάζονται με υπηρεσίες cloud.

2.3.3 Υψηλή Διαθεσιμότητα και αυτόματο failover

Οι βάσεις δεδομένων υλοποιούν έναν μηχανισμό διατήρησης αντιγράφων των δεδομένων σε διαφορετικούς κόμβους. Αυτή η διαδικασία ονομάζεται replication. Κρατώντας πολλαπλά αντίγραφα των δεδομένων μας σε διαφορετικούς κόμβους πετυχαίνεται η διαθεσιμότητα των δεδομένων καθώς και ανοχή σε σφάλματα σε περιπτώσεις κατάρρευσης του συστήματος. Πολλές μη σχεσιακές βάσεις υποστηρίζουν το αυτόματο replication προσφέροντας υψηλή διαθεσιμότητα των δεδομένων και αυτόματο failover σε περίπτωση που ένας κόμβος σταματήσει να λειτουργεί.

2.4 Είδη μη σχεσιακών βάσεων δεδομένων

2.4.1 Key-Value Stores

Οι βάσεις δεδομένων τύπου *Key-Value* (KV) είναι οι πιο απλές μη σχεσιακές βάσεις.

Μια KV βάση δεδομένων αποθηκεύει τα δεδομένα της ως ζευγάρια κλειδιών - τιμών. Ο τρόπος αποθήκευσης της είναι παρόμοιος με ένα *map* ή *hashtable* όπως σε κάθε γνωστή γλώσσα προγραμματισμού. Κάποιες KV υλοποιήσεις επιτρέπουν σύνθετους τύπους τιμών όπως λίστες, που προσθέτουν λειτουργικότητα. Η ανάκτηση των δεδομένων γίνεται από τα αντίστοιχα κλειδιά. Τέτοια συστήματα μπορούν να αποθηκεύουν δομημένη και μη δομημένη πληροφορία. Ο KV τρόπος αποθήκευσης των δεδομένων επιτρέπει στις εφαρμογές να αποθηκεύουν τα δεδομένα τους με ενα schema-less τρόπο.

Car	
Key	Attributes
1	Make: Nissan Model: Pathfinder Color: Green Year: 2003
2	Make: Nissan Model: Pathfinder Color: Blue Color: Green Year:2005 Transmission: Auto

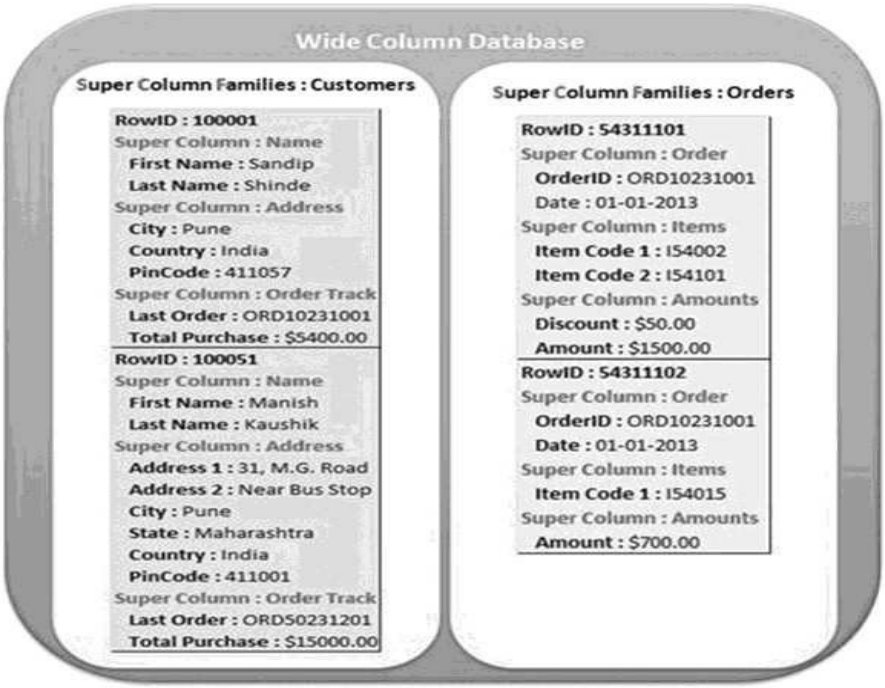
Εικόνα 5. Παράδειγμα αποθήκευσης Key-Value

Παραδείγματα: Όπως και με τις σχεσιακές βάσεις δεδομένων, υπάρχουν πολλές επιλογές ανοιχτού λογισμικού. Μερικές από τις πιο δημοφιλείς περιλαμβάνουν την memcached, Voldemort, Redis, Riak, Dynamo.

2.4.2 Columnar Stores

Στις column-oriented βάσεις δεδομένων τα δεδομένα στους πίνακες αποθηκεύονται κατά στήλες, σε αντίθεση με τις σχεσιακές βάσεις δεδομένων όπου τα δεδομένα αποθηκεύονται κατά γραμμές. Αυτή η αλλαγή, παρ' ότι φαίνεται μικρή δημιουργεί πολύ διαφορετικές

συνθήκες και ιδιότητες στο σύστημα. Η δομή των δεδομένων αποτελείται απο πολλαπλά γνωρίσματα (attributes) για κάθε κλειδί (key). Οι column-oriented βάσεις δεδομένων είναι αποδοτικές σε συστήματα OLAP (online-analytival-processing) όπου απετούνται συνενώσεις μεγάλου όγκου παρόμοιων δεδομένων, όπως σε αποθήκες δεδομένων, συστήματα διαχείρισης πελατιακών σχέσεων (CRM), και παρόμοια συστήματα όπου απετούνται λιγότερα πολύπλοκα ερωτήματα.



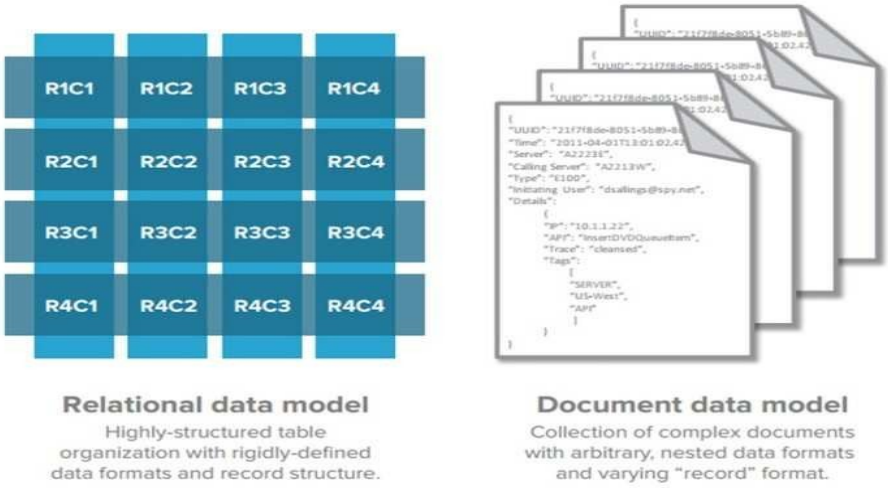
Εικόνα 6. Παράδειγμα αποθήκευσης Column

Παραδείγματα: Google Big Table, Cassandra, Hbase, Amazon SimpleDB κ.ά.

2.4.3 Document Databases

Στις document-oriented βάσεις δεδομένων τα δεδομένα αποθηκεύονται σε αρχεία και όχι σε συσχετισμένους πίνακες όπως στις σχεσιακές βάσεις. Παρέχουν έναν άλλον τρόπο

παράγωγο του μοντέλου Key-Value, όπου τα κλειδιά χρησιμοποιούνται για να εντοπιστούν τα έγγραφα (documents) μέσα στην βάση. Οι περισσότερες document-oriented βάσεις χρησιμοποιούν JSON ή BSON για την αναπαράσταση των documents. Αυτός ο τρόπος αποθήκευσης είναι κατάλληλος για εφαρμογές στις οποίες τα δεδομένα εισόδου μπορούν να αναπαρασταθούν υπό μορφή εγγράφων (documents) και επιτρέπει την εύκολη αποθήκευση ημι δομημένων δεδομένων. Τα documents μπορούν να περιέχουν σύνθετες δομές δεδομένων, όπως εμφωλευμένα αντικείμενα και πίνακες (nested-objects) και δεν χρειάζονται προκαθορισμένο σχήμα (schema-less). Τα documents τα οποία μοιράζονται παρόμοια δομή συνήθως οργανώνονται σε συλλογές (collections), σε ιεραρχίες καταλόγου κ.ά. Τα documents αντιπροσωπεύονται μέσω ενός μοναδικού κλειδιού (*Universally Unique Identifier*) που είναι παρόμοιο με το primary key των σχεσιακών βάσεων δεδομένων. Οι βάσεις δεδομένων τύπου document-oriented είναι καλές για την αποθήκευση και διαχείριση μεγάλων συλλογών από έγγραφα όπως emails, έγγραφα τύπου XML κ.ά. Επίσης χρησιμοποιούνται και στην ανάπτυξη διαδικτυακών εφαρμογών καθώς παρέχουν μια εύκολη και απλή αποθήκευση των δεδομένων που ταιριάζει κατάλληλα στις αντικειμενοστραφής γλώσσες προγραμματισμού.

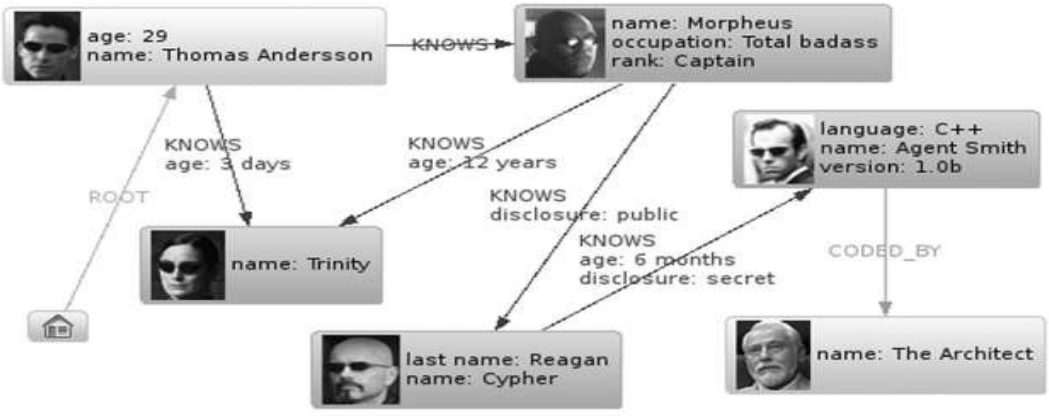


Εικόνα 7. Παράδειγμα αποθήκευσης Document

Παραδείγματα : Apache CouchDB, MongoDB, CouchBase, RavenDB κ.ά.

2.4.4 Graph Databases

Οι βάσεις δεδομένων τύπου graph χρησιμοποιούν την θεωρία των γράφων και αναπαριστούν τα δεδομένα τους υπό μορφή γράφων. Είναι κατάλληλη επιλογή σε περιπτώσεις οπού τα δεδομένα παρουσιάζουν μεγάλη συσχέτιση μεταξύ τους και η συσχέτιση αυτή πρέπει να αποτυπωθεί. Μοντελοποιούνται με κόμβους και με σχέσεις που τους συνδέουν. Χαρακτηριστική λειτουργία σε τέτοιου είδους βάσεις είναι να διασχίζονται οι κόμβοι χρησιμοποιώντας τις σχέσεις που τους συνδέουν. Τέλος, οι κόμβοι και οι σχέσεις μπορούν να έχουν ιδιότητες στις οποίες αποθηκεύονται δεδομένα. Είναι η μόνη κατηγορία από αυτές που μελετήθηκαν των μή σχεσιακών η οποία χρησιμοποιεί σχέσεις. Η κλασσική χρήση τέτοιων συστημάτων γίνεται για την αναπαράσταση κοινωνικών σχέσεων, όπως κοινωνικά δίκτυα, μεταφορές, χάρτες, όπου οι σχέσεις μεταξύ των χρηστών είναι πολλές και πολύπλοκες.



Εικόνα 8. Παράδειγμα αποθήκευσης Graph

Παραδείγματα : Neo4j, Titan, OrientDB, Sparksee κ.ά.

Κεφάλαιο 3. MongoDB

3.1 Γενικά

Η MongoDB ('hu**mongo**us') ανήκει στην οικογένεια των document-oriented noSQL βάσεων δεδομένων και αποθηκεύει τα δεδομένα της σε μορφή BSON (*Binary JSON*). Αναπτύχθηκε αρχικά από την εταιρεία *10gen* (πλέον *MongoDB Inc*.) το 2007 σαν συστατικό ενός προϊόντος *PaaS (*Platform as a Service) και στην συνέχεια ακολούθησε τις αρχές του ελεύθερου λογισμικού. Πρόκειται για μια βάση ανοιχτού κώδικα πλέον, υλοποιημένη στην γλώσσα προγραμματισμού *C++*, διαθέσιμη με τους όρους χρήστης του *GNU Affero General Public Licence* και του *Apache Licence*. Αποτελεί το πιο δημοφιλές σύστημα διαχείρισης noSQL βάσεων δεδομένων και έχουν αναπτυχθεί drivers για την υποστήριξη πολλών γλωσσών προγραμματισμού (*C, C#, C++, Haskell, Java, nodeJS, Perl, PHP, Python, Ruby* κ.α.) . Για την αλληλεπίδραση και την διαχείριση της βάσης η MongoDB χρησιμοποιεί έναν διαδραστικό *JavaScript shell,* την διεργασία *mongo*. Είναι σχεδιασμένη να προσφέρει υψηλή απόδοση στις εφαρμογές, αυτόματη επεκτασιμότητα, υψηλή διαθεσιμότητα και δυνατότητα υποβολής σύνθετων ερωτημάτων. Για τα δεδομένα της εγγυάται αυστηρή συνέπεια (strong consistency).

3.1.1 Document Database - Data As Document

Μια εγγραφή στην MongoDB είναι ένα document (έγγραφο), μια δομή δεδομένων που αποτελείται από πεδία και ζεύγη τιμών. Τα documents στην MongoDB είναι ίδια με τα JSON αντικείμενα. Οι τιμές των πεδίων μπορούν να περιέχουν άλλα documents, πίνακες ή πίνακες από documents. Μία αναπαράσταση ενός MongoDB document φαίνεται παρακάτω. Τα documents με την σειρά τους αποθηκεύονται σε collections (συλλογές). Ένα collection αντιπροσωπεύει μια ομάδα από documents τα οποία είναι συναφή μεταξύ τους και μοιράζονται ένα σύνολο από κοινά *indexes.*

Εικόνα 9. Αναπαράσταση ενός MongoDB document

Κάποια από τα πλεονεκτήματα της χρήσης documents είναι:

- Τα documents ή αντικείμενα ανταποκρίνονται σε τύπους δεδομένων πολλών γλωσσών προγραμματισμού.
- Τα embedded documents καθώς και οι χρήση πινάκων μειώνουν τις ανάγκες για 'ακριβά' JOINS.
- Η χρήση δυναμικών σχημάτων υποστηρίζει εύκολα πολυμορφισμό.

3.1.2 JSON

Το πρότυπο δεδομένων JSON (JavaScript Object Notatation) είναι ένα ελαφρύ πρότυπο ανταλλαγής δεδομένων. Είναι εύκολο και κατανοητό στον άνθρωπο αλλά και στις μηχανές να το αναλύσουν και να το παράγουν. Μαζί με τα XML αρχεία αποτελούν πλέον τα κύρια πρότυπα ανταλλαγής δεδομένων που συναντάμε στο σύγχρονο διαδίκτυο. Υποστηρίζει όλους τους βασικούς τύπους δεδομένων όπως αριθμούς, αλφαριθμητικά, πίνακες κ.α. Η βάσεις δεδομένων τύπου document-oriented όπως και η Mongo χρησιμοποιούν *JSON* documents για την αποθήκευση τον εγγραφών, όπως και οι αντίστοιχοι πίνακες και γραμμές σε ένα σχεσιακό σύστημα. Υπάρχουν δύο βασικές δομές δεδομένων σε ένα JSON document, *Array* τύπου[*value1,value2,..valueN*] και τα *Objects (hashtable ή Map)* τύπου {*String:Value, String:Value1*} τα οποία λειτουργούν σαν συλλογές από ζευγάρια κλειδιών και τιμών. Υπάρχει μια ιεραρχία σε έναν JSON έγγραφο που επιτρέπεται την εμφώλευση (nesting) σε όποιο βαθμό εμείς το επιθυμούμε καθώς και τη εμφώλευση υπό-εγγράφων (sub-

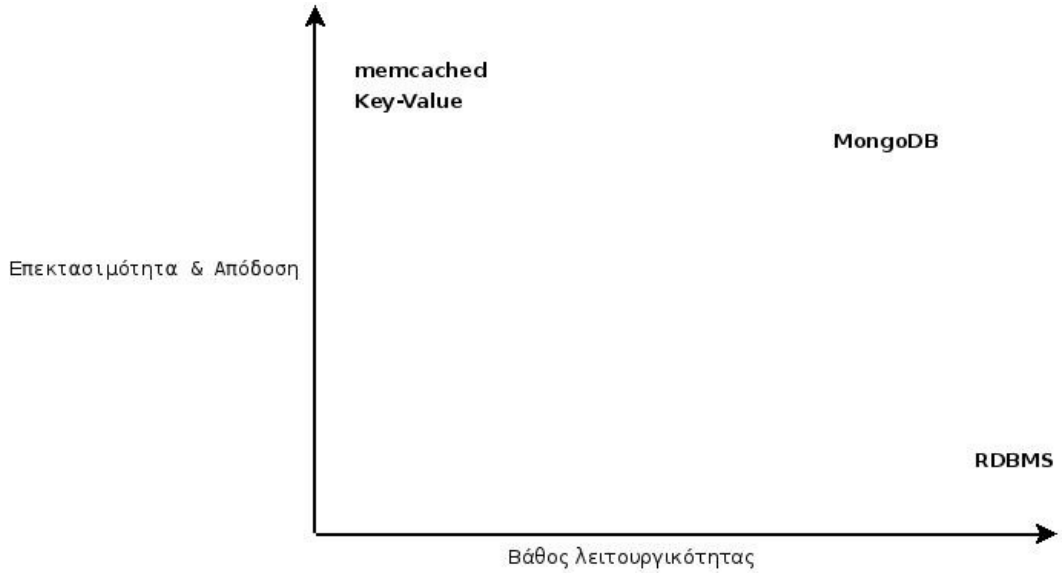
documents). Υποστηρίζει την ενσωμάτωση αντικειμένων (*Objects*) και πινάκων μέσα σε άλλα αντικείμενα ή πίνακες [15].

3.1.3 Binary JSON (BSON)

Η MongoDB από την μεριά της αναπαριστά τα JSON documents σε δυαδική μορφή η οποία ονομάζεται BSON. Η δομή BSON επεκτείνει το μοντέλο JSON και παρέχει επιπλέον τύπους δεδομένων για την αποτελεσματική κωδικοποίηση και αποκωδικοποίηση μεταξύ διαφορετικών γλωσσών [16].

3.1.4 Από το σχεσιακό στο σχετικό της MongoDB

Στο παρακάτω σχήμα παρουσιάζεται η εικόνα μιας βάσης δεδομένων. Στον κάθετο άξονα (x) τοποθετούμε την επεκτασιμότητα και την απόδοση ως παράγοντα που προσφέρεται από μια βάση δεδομένων ενώ στον οριζόντιο άξονα (y) θα μπορούσαμε να τοποθετήσουμε τον αντίστοιχο βαθμό λειτουργικότητας που προσφέρει η βάση. Συστήματα όπως τα memcached ή τα Key-Value προσφέρουν μεγάλη επεκτασιμότητα και την μεγαλύτερη απόδοση αλλά δεν μπορούν να είναι ταυτόχρονα τόσο λειτουργικά. Από την άλλη πλευρά, την μεγαλύτερη λειτουργικότητα που μπορεί να μας προσφέρει μια βάση την δίνουν τα σχεσιακά συστήματα (Oracle, DB2, MySQL) αλλά κλιμακώνονται κάθετα, πράγμα που οδηγεί στην προσθήκη όλο και περισσότερου hardware για να πάρουμε την μέγιστη απόδοση από έναν κόμβο με αποτέλεσμα να αυξάνεται όλο και περισσότερο το κόστος. Η MongoDB συνδυάζει εύκολη επεκτασιμότητα και υψηλή απόδοση καθώς και την μέγιστη λειτουργικότητα που μπορεί να προσφέρει ένα σχεσιακό σύστημα.



Εικόνα 10. Από το σχεσιακό RDBMS στο σχετικό της Mongo

Η διαφορά λειτουργικότητας ανάμεσα σε ένα RDBMS και στην MongoDB οφείλεται στο ότι η MongoDB δεν υποστηρίζει λειτουργίες των σχεσιακών συστημάτων όπως:

Ενώσεις Πινάκων (Joins)

Η Mongo αποθηκεύει τα δεδομένα της σε documents και κάθε document αποθηκεύεται σε ένα collection (συλλογή). Δεν υπάρχει η δυνατότητα της ένωσης δύο διαφορετικών συλλογών όπως πραγματοποιείται στα σχεσιακά συστήματα με την αντίστοιχη ένωση πινάκων. Ο λόγος που δεν επιτρέπει η Mongo αυτήν την δυνατότητα είναι επειδή οι ενώσεις μεταξύ πινάκων είναι ο βασικότερος παράγοντας που μειώνουν την επεκτασιμότητα και την απόδοση ενός συστήματος όταν προσπαθούμε να κλιμακώσουμε την εφαρμογή μας σε πολλαπλούς κόμβους.

Συναλλαγές (Transactions)

Η Mongo δεν υποστηρίζει συναλλαγές στο επίπεδο των RDBMS. Υποστηρίζει συναλλαγές σε **ατομικό** επίπεδο και όχι μεταξύ πολλαπλών documents. Η δομή των documents ως JSON είναι ιεραρχική και επιτρέπεται η πρόσβαση σε αυτά ατομικά. Άρα, κάτι που θα χρειαζότανε πολλαπλές ενημερώσεις (updates) σε ένα σχεσιακό σύστημα μπορούμε να το χειριστούμε με μία ατομική συναλλαγή σε ένα μόνο document.

RDMBS	**MongoDB**
Database	Database
Table	Collection
Row	Document
Column	Field
Index	Index
JOIN	Embedeed document
Foreign key	Reference
Primary key	PrimaryKey, automatically set to the _id field
Aggregation (Group By)	Aggregation pipeline
Server - mysqld	Server – mongod
Client – mysql	Client - mongo

Πίνακας 2. Ορολογία σε RDBMS και MongoDB σύστημα

3.2 Μοντέλο Δεδομένων – Data Model

Στην Mongo τα δεδομένα αποθηκεύονται με *schema-free* τρόπο σε αρχεία τύπου *BSON* (*Binary JSON*), τα οποία ονομάζονται documents. Κάθε document αντιπροσωπεύει μια εγγραφή στην ορολογία των σχεσιακών βάσεων δεδομένων και περιέχει πεδία σε μορφή ζευγαριών *key/value*.. Στο key αποθηκεύεται σε ενα String το όνομα του πεδίου, και στο value μπορούν να αποθηκευτούν τύποι δεδομένων που υποστηρίζονται από τα *JSON* αρχεία (*Number, String, Boolean, Array, Object, Whitespace, Null, ή JSON value*) καθώς και άλλοι τύπου *Date*. Ένα σύνολο από documents ορίζει μια collection, που είναι το ανάλογο των πινάκων στις σχεσιακές βάσεις δεδομένων. Ένα σύνολο από collections συνιστά μια βάση δεδομένων. Η Mongo ως σύστημα διαχείρισης διατηρεί ένα σύνολο από βάσεις δεδομένων. Στον παραπάνω πίνακα **<2>** παρουσιάζονται οι αντίστοιχες ορολογίες μεταξύ ενός *RDBMS* και ενός συστήματος *Mongo.*

Κάθε document υποχρεωτικά πρέπει να διαθέτει ένα πεδίο όπου αποθηκεύεται το μοναδικό *id* του document και χρησιμοποιείται σαν *primary key* για τον εντοπισμό του. Το key αυτού του πεδίου ονομάζεται *_id* και το value είναι τύπου *ObjectID* - ή οποιοσδήποτε άλλος τύπος εκτός από Array- και έχει μοναδική τιμή για κάθε document. Σε όλες τις βάσεις δημιουργείται αυτόματα index με βάσει το *_id*.. Το _id αν δεν το δηλώσουμε εμείς να είναι συγκεκριμένο πεδίο, παράγεται αυτόματα από το σύστημα. Μια αναπαράσταση ενός Mongo document με embedded document με όνομα *comments* και πεδία *author, body, email* και Array *tags* δίνεται στη παρακάτω εικόνα.

```
    "_id" : ObjectId("54bfc1a70f666d913e88f5f2"),
    "title" : "MongoDB Document",
    "author" : "ibraim",
    "body" : "This is how a MongoDB document looks like in a JSON format.",
    "permalink" : "MongoDB_Document",
    "tags" : [
        "mongo",
        "dbmbs",
        "thesis",
        "uom"
    ],
    "comments" : [
        {
            "author" : "ibraim",
            "body" : "Lorem ipsum dolor sit amet"
        },
        {
            "author" : "gevan",
            "body" : "Lorem ipsum dolor sit amet!",
            "email" : "gevan@uom.gr"
        }
    ],
    "date" : ISODate("2015-01-21T15:11:35.433Z")
}
```

Εικόνα 11. Αναπαράσταση MongoDB post Document εφαρμογής

3.2.1 Μέγεθος ενός document

Κάθε document έχει μέγιστο μέγεθος 16MB. Εάν το μέγεθος του document υπερβεί το χώρο που έχει εκχωρηθεί για το συγκεκριμένο document, η Mongo επανατοποθετεί αυτόματα το document στον δίσκο. Η αύξηση όμως του μεγέθους των documents άνω του επιτρεπτού ορίου (παρατηρείται κατά την χρήση της ενσωματωμένης αποθήκευσης *Object* και *Array* εντός του *document*) επηρεάζει την απόδοση των *writes* και μπορεί να οδηγήσει σε κατακερματισμό των δεδομένων. Για την αποθήκευση μεγαλύτερων σε μέγεθος δεδομένων (π.χ εικόνες) υπάρχει η δυνατότητα χρήσης του εργαλείου GridFS [17]. Αντί να

αποθηκεύετε ένα αρχείο σε ένα ενιαίο document, η τεχνική GridFS χωρίζει ένα αρχείο σε τμήματα (*parts*), ή κομμάτια (*chunks*), και αποθηκεύει το καθένα από αυτά τα κομμάτια ως ξεχωριστό document. Το GridFS χρησιμοποιεί δύο collections για την αποθήκευση των αρχείων, μία για την αποθήκευση των *chunks* και μία για την αποθήκευση των *metadata* του αρχείου.

3.2.2 Ευέλικτο Δυναμικό Σχήμα – Flexible Schema – Schema-less/free

Η Mongo προσφέρει μεγάλες ελευθερίες στον τρόπο αποθήκευσης των δεδομένων και δίνει στους προγραμματιστές τη δυνατότητα να εκμεταλλευτούν τις ιδιαιτερότητες της εφαρμογής τους· να δημιουργήσουν ανάλογα μοντέλα αναπαράστασης των δεδομένων και να παρέχουν αποδοτικότερες υπηρεσίες. Σε αντίθεση με τις σχεσιακές βάσεις δεδομένων όπου πρέπει να καθοριστεί και να δηλωθεί το σχήμα των πινάκων (φυσικό μοντέλο) προτού εισαχθούν τα δεδομένα, οι συλλογές στην Mongo εκτός του ότι δεν δηλώνονται, δεν επιβάλλουν στα documents να έχουν την ίδια δομή. Τα documents μιας collection δεν χρειάζεται να έχουν τα ίδια πεδία και δεν απαιτείται να διαθέτουν τους ίδιους τύπους δεδομένων. Μπορεί δηλαδή το κάθε ένα document να διαθέτει το δικό του σχήμα. Αυτή η ευελιξία στο σχήμα που προσφέρεται από την Mongo επιτρέπει την μοντελοποίηση των documents ώστε να αναπαριστούν καλύτερα τα αντικείμενα (objects) ή τις οντότητες (entities) που εμπλέκονται και τις σχέσεις μεταξύ τους. Στην πράξη, όμως, τα εμπλεκόμενα documents μιας collection τείνουν να ακολουθούν ομοιογενή δομή.

Η κύρια πτυχή κατά την μοντελοποίηση των δεδομένων μας με την Mongo, είναι να μοντελοποιήσουμε τα δεδομένα μας με βάση τον τρόπο χρήσης τους και πρόσβασης και όχι με βάση τον τρόπο αποθήκευσης τους όπως γίνεται στις σχεσιακές βάσεις δεδομένων όπου η εκτέλεση των ερωτημάτων προς την βάση είναι κυρίως βασισμένη στα *joins*. Επομένως, κατά την επιλογή του κατάλληλου σχήματος από τους προγραμματιστές, πρέπει να λαμβάνονται υπόψιν τα *queries* που θα υποβάλλονται στην βάση, η συχνότητα των *updates*, των *reads* και *writes*, οι *Map/Reduce* λειτουργίες καθώς και ο ρυθμός αύξησης των documents.

3.2.3 Μοντελοποίηση δεδομένων και αναπαράσταση σχέσεων

Στην MongoDB το κύριο συστατικό κατά την διάρκεια σχεδίασης των μοντέλων μας για την αναπαράσταση των σχέσεων μεταξύ των δεδομένων και την δομή που θα ακολουθήσουν τα documents καθορίζεται από το αν θα αποφασίσουμε να κάνουμε χρήση της *Embedded Object* και Arrays αποθήκευσης ή την χρήση *References.* Η απόφαση αυτή σχετίζεται με το πώς έχουμε πρόσβαση στα δεδομένα μας, την συχνότητα πρόσβασης στα δεδομένα και την ατομικότητα των δεδομένων μιας και δεν επιτρέπονται *transactions*. Η κάθε μία από τις προαναφερθέντες μεθόδους έχει τα πλεονεκτήματα και τα μειονεκτήματα της.

Embedded documents

Με την χρήση της *embedded* αποθήκευσης, δηλαδή της ενσωμάτωσης Objects ή και Arrays μέσα σε ένα document, πετυχαίνουμε την συλλογή δεδομένων σχετικών μεταξύ τους μέσα σε ένα document, δηλαδή στην ίδια εγγραφή. Η MongoDB επιτρέπει την ενσωμάτωση υπό-εγγράφων στο εσωτερικό ενός document ή και την ενσωμάτωση πινάκων. Αυτά τα μοντέλα δεδομένων, γνωστά και ως '***μη κανονικοποιημένα***' κάνουν την ανάκτηση και την επεξεργασία των δεδομένων που είναι σχετικά μεταξύ τους πιο εύκολη , καθώς λειτουργούν σαν ***pre-joined*** δεδομένα. Άρα, σε αναλογία με τις σχεσιακές βάσεις δεδομένων όπου για την ανάκτηση ή επεξεργασία των δεδομένων μας θα χρειαζόμασταν *JOINS* μεταξύ πολλαπλών πινάκων, η Mongo προσφέρει την δυνατότητα της ενσωμάτωσης σχετικής πληροφορίας μέσα σε ένα document με αποτέλεσμα οι εφαρμογές να χρειάζονται λιγότερα queries για την ανάκτηση ή ενημέρωση της πληροφορίας. Η τεχνική της embedded αποθήκευσης χρησιμοποιείται συχνότερα όταν θέλουμε να περιγράψουμε συσχετίσεις ***1:1***, ***1:N*** (όταν όμως το 'Πολλά' αναφέρεται σε λίγα και όχι πραγματικά πολλά, καθορίζεται από την εκάστοτε εφαρμογή). Γενικά, η τεχνική αυτή προσφέρει καλύτερη απόδοση σε λειτουργίες ανάγνωσης και στη ανάκτηση σχετικών μεταξύ τους δεδομένων σε ατομικό επίπεδο καθώς με ένα querie μπορούμε να ανακτήσουμε δεδομένα τα οποία συγκροτούνται τοπικά σε ένα αρχείο. Επίσης με αυτήν την μέθοδο κρατάμε τα δεδομένα μας άθικτα και συνεπή από την στιγμή που η Mongo δεν επιτρέπει περιορισμούς ξένων κλειδιών. Μειονέκτημα της μεθόδους αυτής αποτελεί το γεγονός ότι εύκολα αυξάνεται το μέγεθος των documents, μέγιστο μέγεθος

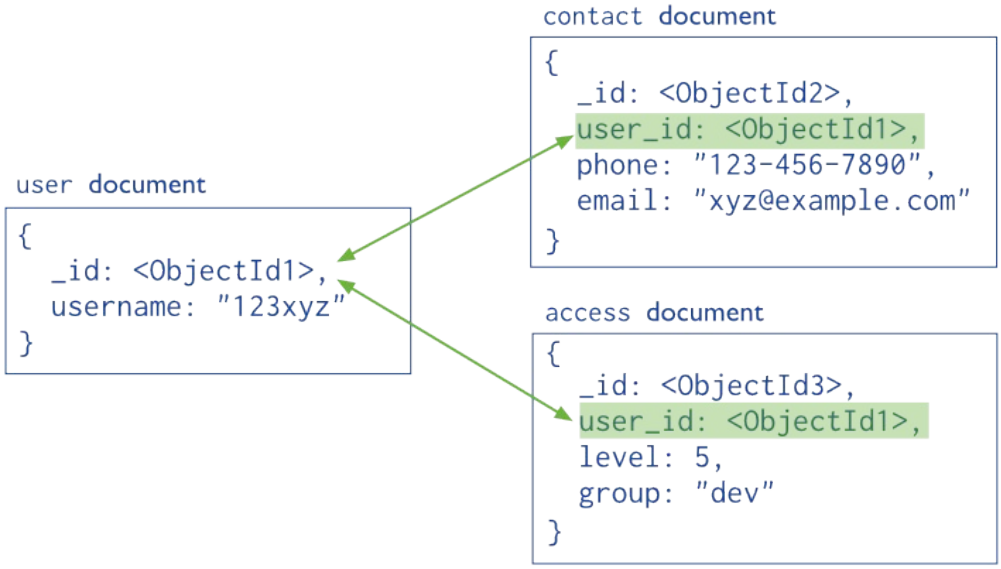
ως *16MB*, που περιέχουν ενσωματωμένα άλλα *Objects*.

Εικόνα 12. Παράδειγμα Embedded Αποθήκευσης

References

Με την χρήση *references* ή *linking*, δηλαδή την διατήρηση των *_id* κάποιων documents σε άλλα documents, πετυχαίνουμε την αναπαράσταση και σύνδεση των σχέσεων μεταξύ των documents και των εμπλεκόμενων δεδομένων. Τα documents που θα συσχετιστούν μπορεί να ανήκουν σε διαφορετικές *collections* ή και σε διαφορετικές βάσεις. Η χρήση των references είναι ανάλογη με τα *foreign keys* των σχεσιακών βάσεων. Στην πραγματικότητα όμως, πρόκειται για ένα *Array* από *_ids* στο document καθώς η Mongo δεν υποστηρίζει *foreign keys* ή κάποια παρόμοια λειτουργία για λόγους απόδοσης. Άρα δεν υπάρχει καμία εγγύηση ότι τα documents υπάρχουν στην 'linked' *collection*. Επομένως, η διαχείριση των συγκεκριμένων references για τη πρόσβαση στα σχετικά δεδομένα και την ανάκτηση τους γίνεται σε επίπεδο εφαρμογής (προγραμματιστικά). Αυτά τα μοντέλα δεδομένων, γνωστά και ως '***κανονικοποιημένα***' , είναι πιο ευέλικτα για τον ορισμό των σχέσεων των documents και μπορεί να ελεγχθεί καλύτερα το μέγεθος τους σε σχέση με την *embedded* αποθήκευση. Ωστόσο, ο επεξεργαστικός φόρτος στους κόμβους γίνεται μεγαλύτερος, αφού απαιτείται μεγαλύτερη μεταφορά δεδομένων στο δίκτυο καθώς χρειάζονται περισσότερα queries,

δηλαδή περισσότερα *I/O* για την ανάκτηση της πληροφορίας. Τα references χρησιμοποιούνται περισσότερο όταν θέλουμε να περιγράψουμε σχέσεις **1:N**, ***n:m***. Υπάρχουν δύο επιλογές για να προσδιορίσουμε references, *manual* ή με τον τελεστή *DBRef*. Τέλος, η μέθοδος αυτή παρουσιάζει καλύτερα αποτελέσματα όταν η embedded αποθήκευση θα είχε σαν αποτέλεσμα την αποθήκευση των ίδιων δεδομένων πολλές φορές, όταν είναι απαραίτητη η δημιουργία σύνθετων σχέσεων *n:m*, για την μοντελοποίηση μεγάλων ιεραρχικά συνόλων δεδομένων και όταν είναι δυνατή η αναπαράσταση των δεδομένων υπό την μορφή δένδρου [18].



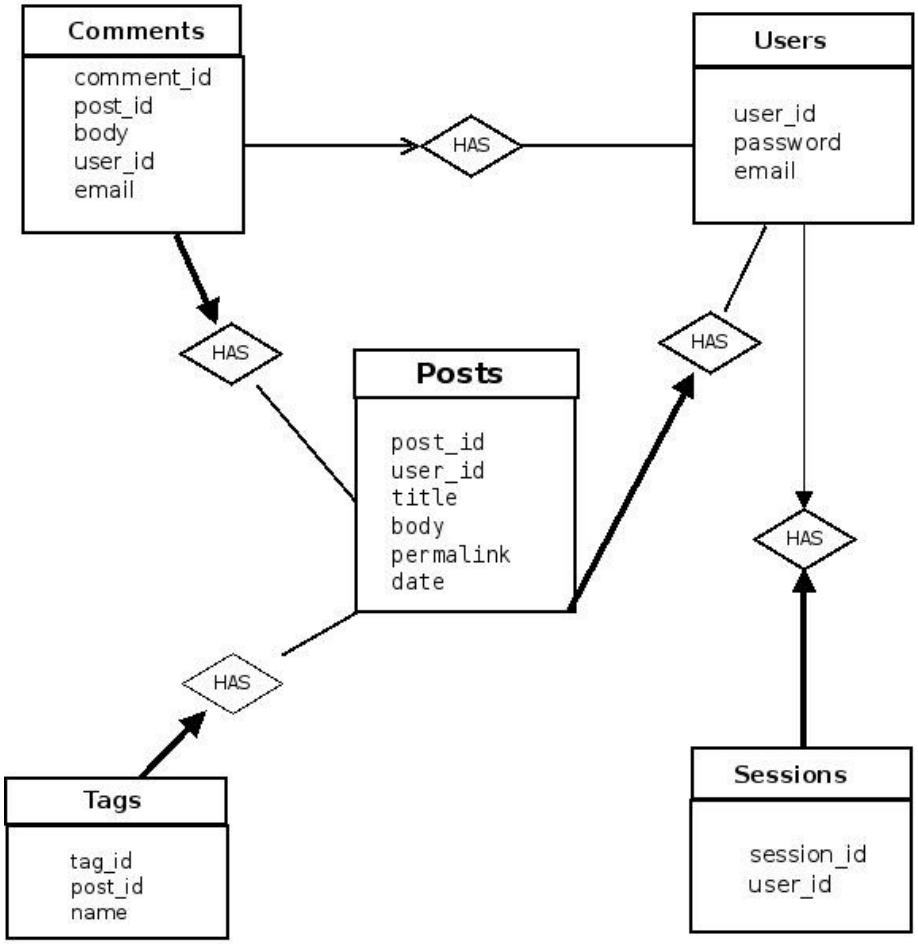
Εικόνα 13. Παράδειγμα αποθήκευσης με χρήση References

3.2.4 Ατομικότητα

Στην MongoDB, οι λειτουργίες των write (*update* ή *remove*) εκτελούνται πάντα **ατομικά** στο επίπεδο ενός document. Καμία λειτουργία εγγραφής δεν μπορεί να τροποποιήσει παραπάνω από ένα document. Ακόμα και σε λειτουργίες όπου τροποποιούμε παραπάνω από ένα document σε μία collection (*multi-update*), τα *writes* ενεργούν στο κάθε document ξεχωριστά. Δεν λειτουργούν δηλαδή ως απομονωμένα transactions που συναντάμε στα σχεσιακά συστήματα. Η Mongo, εγγυάται ότι ένα document δεν θα τροποποιηθεί μερικώς ενώ μια άλλη λειτουργία προσπαθεί να έχει πρόσβαση σε αυτό. Επομένως. κατά την μοντελοποίηση των δεδομένων μας θα πρέπει να κρατάμε τα δεδομένα

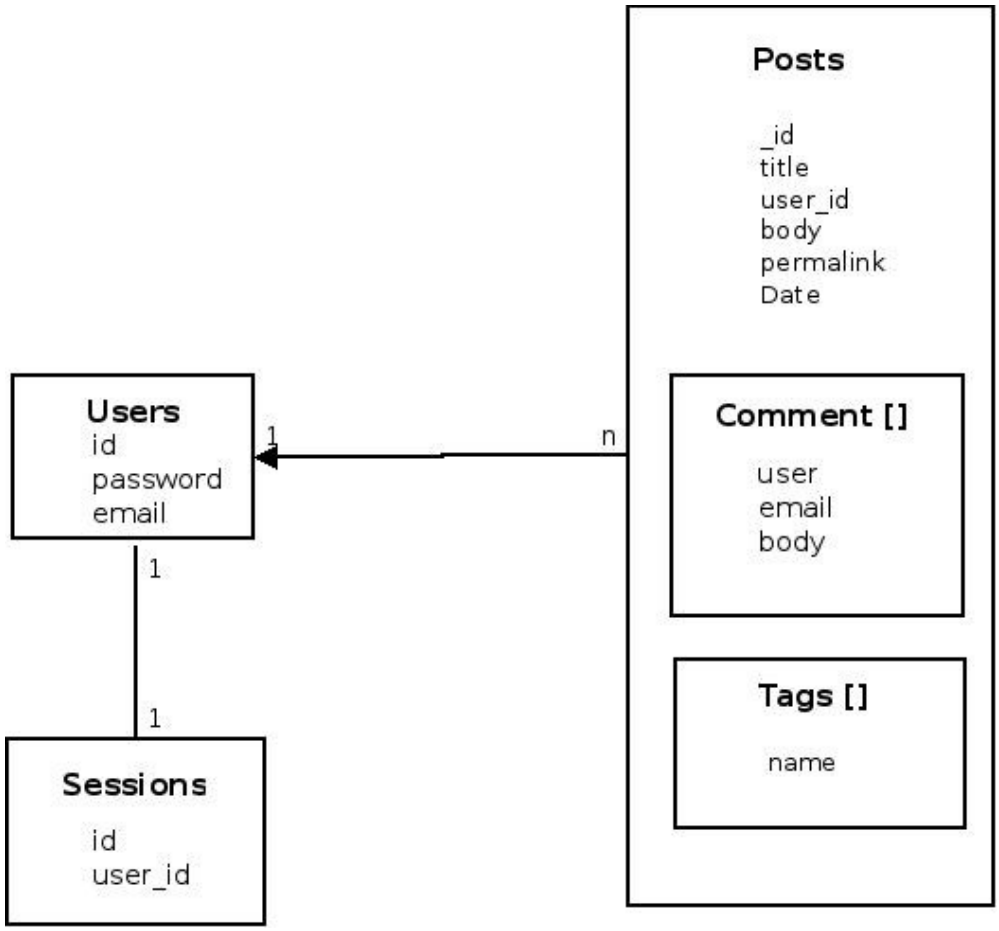
μας ατομικά εξαρτημένα σε επίπεδο ενός document (embedded documents) εάν αυτό είναι δυνατό. Σε ένα μοντέλο δεδομένων το οποίο αποθηκεύει references για να περιγράψει τις σχέσεις των εμπλεκόμενων δεδομένων, θα χρειαζόντουσαν περισσότερα *read* και *write* για να ανακτήσουμε και να τροποποιήσουμε την σχετική πληροφορία [19].

3.2.5 Αναπαράσταση Μοντέλου ΟΣ εφαρμογής σε σχεσιακό σύστημα



Εικόνα 14. Μοντέλο ER εφαρμογής σε RDBMS

3.2.6 Αναπαράσταση σχήματος εφαρμογής στην MongoDB



Εικόνα 15. Μοντέλο εφαρμογής στην MongoDB

3.3 Query Model – CRUD Operations

Η MongoDB, όπως και οι περισσότερες μη σχεσιακές βάσεις δεδομένων, χρησιμοποιεί το ακρώνυμο ***CRUD*** (Create,Read,Update,Delete) για να περιγράψει τους χειρισμούς τους οποίους μπορούμε να εκτελέσουμε στην βάση δεδομένων. Οι λειτουργίες αυτές υφίστανται ως μέθοδοι ή συναρτήσεις στις αντίστοιχες βιβλιοθήκες (APIs) κάθε γλώσσας προγραμματισμού και όχι σαν ξεχωριστή γλώσσα όπως συμβαίνει στις σχεσιακές βάσεις δεδομένων. Κατά την υποβολή των queries χρησιμοποιούνται *JSON* αρχεία, τα οποία δημιουργούνται είτε μέσω του διαδραστικού JavaScript φλοιού (*mongo*), είτε μέσω μιας γλώσσας προγραμματισμού. Με την σειρά τους, τα JSON αρχεία στέλνονται στην βάση σαν Objects τύπου *BSON* μέσω του *driver* που χρησιμοποιεί η εφαρμογή.

Στην MongoDB, ένα query εφαρμόζεται σε μια συγκεκριμένη collection από documents και συμπεριλαμβάνονται όλα τα embedded Objects και Arrays που περιέχει το document. Κατά την υποβολή ενός query ορίζονται τα κριτήρια ή οι συνθήκες που πρέπει να ικανοποιούν τα documents για να συμπεριληφθούν στο αποτέλεσμα που θα επιστραφεί. Ένα query μπορεί να επιστρέψει ένα document ή ένα υποσύνολο συγκεκριμένων πεδίων μέσα στο document. Το μοντέλο ερωτημάτων της MongoDB παρέχει πλούσια λειτουργικότητα και ευελιξία αφού υποστηρίζει, μεταξύ άλλων, τα ακόλουθα χαρακτηριστικά:

- υποβολή ερωτημάτων σε document και τα embedded sub-documents (*dot notation*)
- υποβολή ερωτημάτων σε Array (*all, elemMatch, size*)
- υποβολή geospatial ερωτημάτων (*geoIntersects, nearSphere, near*)
- χρήση τελεστών σύγκρισης (*gt, gte, in, lt, lte, ne, nin*)
- χρήση λογικών τελεστών (*and, nor, not, or*)
- χρήση τελεστών ικανοποίησης συνθηκών (*exists, type, mode, regex, text, where*)
- χρήση συναρτήσεων ομαδοποίησης (*group, count, sum, min, max, average, ...*)
- υποβολή συνάρτησης MapReduce

Η χρήση των παραπάνω τελεστών πραγματοποιείται βάζοντας πρώτα τον χαρακτήρα $

[20] .

3.3.1 Read Operations – Λειτουργίες Ανάγνωσης

Στην MongoDB, οι λειτουργίες ανάγνωσης, ή *queries*, χρησιμοποιούνται για να ανακτήσουμε δεδομένα από την βάση μας. Τα read operations εφαρμόζονται στα documents μίας μόνο collection. Κατά τον ορισμό ενός ερωτήματος μπορούμε να συμπεριλάβουμε ένα *projection,* η αντίστοιχη προβολή (*SELECT*) σε σχεσιακό, για να ορίσουμε τα πεδία που θα επιστραφούν από τα αντίστοιχα documents. Από προεπιλογή, το πεδίο *_id* επιστρέφεται στα αποτελέσματα, ενώ αν δεν θέλουμε να το συμπεριλάβουμε αρκεί να προσδιορίσουμε *_id*: 0. Η χρήση *projection* περιορίζει τον όγκο τον δεδομένων που πρέπει να επιστραφούν από το δίκτυο. Τέλος, υπάρχει η δυνατότητα τροποποίησης ενός query έτσι ώστε να κάνει χρήση των μεθόδων *limit*, *skip*, *sort*.

Η Mongo για την ανάκτηση των δεδομένων της προσφέρει την μέθοδο `db.collection.find ()`. Η μέθοδος δέχεται τις συνθήκες που πρέπει να ικανοποιεί το ερώτημα μας και μαζί με την χρήση του *projection,* ως δεύτερο όρισμα, επιστρέφει έναν *cursor* των αντίστοιχων documents. Επίσης η Mongo παρέχει ακόμη την μέθοδο `db.collection.findOne ()` η οποία επιστρέφει ένα μόνο document. Στην παρακάτω εικόνα φαίνεται ένα query στην Mongo και η αντιστοιχία του σε ένα σχεσιακό σύστημα.

Εικόνα 16. MongoDB Query Read

Αντίστοιχα σε *RDBMS* και *SQL*:

Εικόνα 17. RDBMS Query Read

Για την πιο γρήγορη εκτέλεση των queries είναι δυνατή η δημιουργία indexes. Ωστόσο καθώς αυτά καταναλώνουν χώρο και επιβραδύνουν την εκτέλεση των updates πρέπει να χρησιμοποιούνται με προσοχή. Θα αναλυθεί περισσότερο η δημιουργία και χρήση indexes στην επόμενη ενότητα.

3.3.2 Reads στην εφαρμογή

Παρακάτω δίνεται ένα read operation της εφαρμογής (blog). Το συγκεκριμένο query βρίσκει τα posts τα οποία έχουν και μοιράζονται το ίδιο tag. Για παράδειγμα, αν tag : 'uom', επιστρέφονται τα posts με tag 'uom', ταξινομημένα με βάση το 'date', και επιστρέφει τα 10 τελευταία posts. Τα queries είναι υλοποιημένα σε γλώσσα *JavaScript*, στην πλατφόρμα *nodeJS*.

```
this.getPostsByTag = function(tag, num, foundTags) {
    "use strict";


    posts.find({ tags : tag }).sort('date', -1).limit(num).toArray(function (err, items) {
        "use strict";

        if (err) return callback(err, null);

        console.log("Found " + items.length + " posts");

        foundTags(err, items);
    });
}
```

Εικόνα 18. Read στο blog, βρίσκει τα posts με συγκεκριμένο tag ταξινομημένα με βάση το 'date'

Αντίστοιχα το read στο *mongo* shell:

```
>db.posts.find ({ "tags":"uom"}).sort ({"date":-1}).limit (10)
```

Πίνακας 3. Read mongo shell

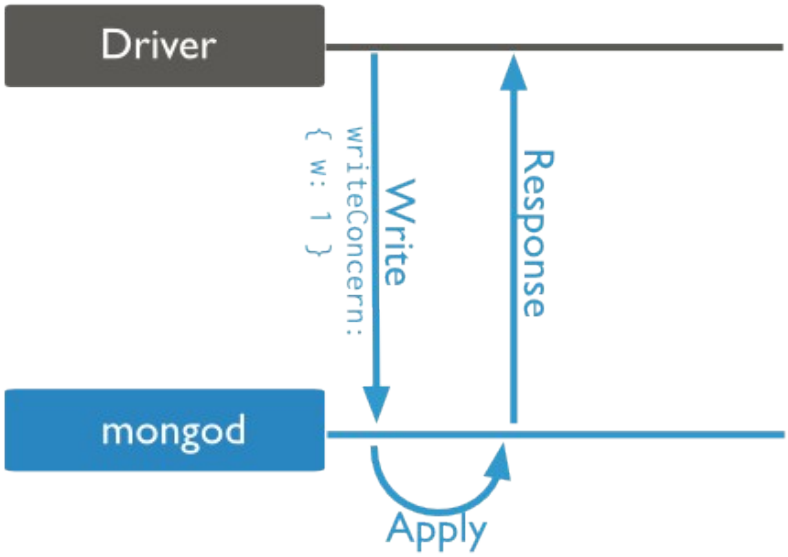
Για περισσότερα παραδείγματα μπορούν οι ενδιαφερόμενοι να ανατρέξουν στο manual της MongoDB [21].

3.3.3 Write Operations – Λειτουργίες Εγγραφής

Στην MongoDB, οι λειτουργίες εγγραφής (insert,update,remove), *write operations,* περιγράφουν λειτουργίες όπου δημιουργούνται ή τροποποιούνται δεδομένα. Τα writes εφαρμόζονται σε μία μόνο collection και εκτελούνται πάντα ατομικά στο επίπεδο ενός document. Οι λειτουργίες *insert* προσθέτουν δεδομένα σε ένα collection, τα *update* τροποποιούν υπάρχουσα δεδομένα ή προσθέτουν δεδομένα σε documents, και τα *remove* διαγράφουν δεδομένα από ένα collection. Καμία από τις προαναφερθέντες μεθόδους δεν μπορεί να επηρεάσει πάνω από ένα document ατομικώς. Για τις λειτουργίες update και remove, μπορούμε να καθορίσουμε κριτήρια, όπως στα read operations, όπου ικανοποιούν τα documents που θέλουμε να τροποποιήσουμε.

Στο σύστημα διαχείρισης της MongoDB, υπάρχουν διαθέσιμα πολλαπλά επίπεδα που εγγυώνται την μονιμότητα των επιτυχών εγγραφών, το *write concern.* Η μονιμότητα (durability) σχετίζεται με την εγγύηση ότι τα δεδομένα που γράφτηκαν,αποθηκεύτηκαν και θα επιβιώσουν μόνιμα. Τα επίπεδα κυμαίνονται από ασθενές έως ισχυρά. Αναλόγως με τις ανάγκες κάθε εφαρμογής, τη φύση των δεδομένων, ο χρήσης μπορεί να επιλέξει το επίπεδο που του ταιριάζει καλύτερα. Από προεπιλογή, η Mongo χρησιμοποιεί *acknowledgement* για τα writes. Σύμφωνα με αυτό το επίπεδο μονιμότητας, ο driver της MongoDB επιβεβαιώνει την παραλαβή ενός write operation στον server, και εφαρμόζει τις τροποποιήσεις στην αντίστοιχη όψη των δεδομένων στην μνήμη. Ωστόσο, αυτό το επίπεδο δεν εγγυάται ότι τα

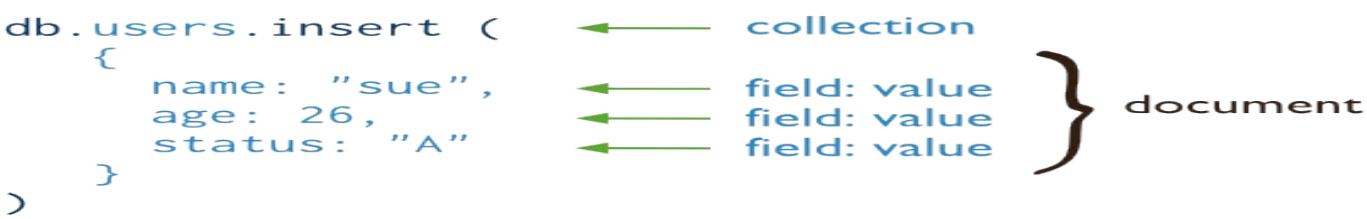
δεδομένα γράφτηκαν και στο δίσκο. Τέλος, υπάρχει η δυνατότητα να ορίσει ο χρήστης για κάθε write τι επίπεδο εγγύησης επιθυμεί, εισάγοντας για κάθε write operation την επιλογή writeConcern και το επίπεδο που επιθυμεί για κάθε write.



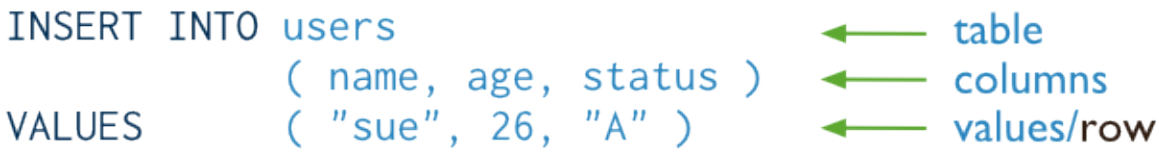
Εικόνα 19. Acknowledgement write concern

Insert

Για την εισαγωγή νέων document σε ένα collection, η Mongo παρέχει την μέθοδο `db.collection.insert ()`. Αν προστεθεί ένα νέο document χωρίς να ορίσουμε εμείς το πεδίο *_id*, η Mongo προσθέτει αυτόματα ένα πεδίο *_id* και το συμπληρώνει με μια μοναδική τιμή τύπου *ObjectID*. Στις περιπτώσεις όπου ορίζουμε εμείς το _id της εγγραφής, αν υπάρχει άλλο document με το συγκεκριμένο _id, η Mongo επιστρέφει σφάλμα διπλότυπης εγγραφής (duplicate key exception).



Εικόνα 20. MongoDB Insert

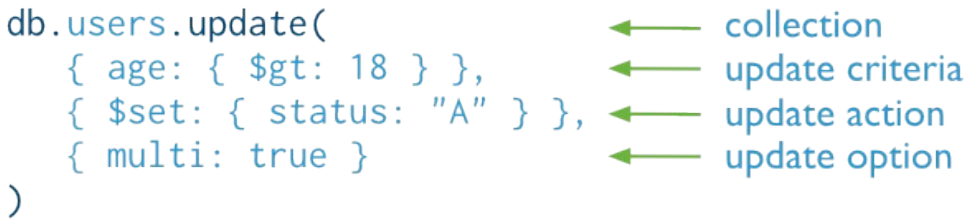


Εικόνα 21. RDBMS Insert

Επίσης παρέχεται η δυνατότητα προσθήκης ενός πίνακας από documents. Τέλος, στις τελευταίες εκδόσεις της MongoDB προσφέρεται και η δυνατότητα εισαγωγής πολλών documents σε μια collection. Η δυνατότητα αυτή προσφέρεται από το *bulk ()* API της MongoDB.

Update

Για την τροποποίηση των documents σε μια collection, προσφέρεται η μέθοδος `db.collection.update ()`. Η μέθοδος δέχεται κριτήρια που προσδιορίζουν το document που θα τροποποιηθεί και επιλογές που επηρεάζουν τη συμπεριφορά της λειτουργίας, όπως η επιλογή *multi,* η οποία χρησιμοποιείται για την τροποποίηση όλων των document που ικανοποιούν τη συνθήκη. Όπως και με τις άλλες λειτουργίες, τα update λειτουργούν ατομικά στο κάθε document. Προσφέρεται και ακόμη μια επιλογή αυτή της *upsert,* όπου αν δεν ικανοποιηθούν οι συνθήκες του query για το update, η μέθοδος δημιουργεί ένα νέο document.



Εικόνα 22. MongoDB Update

Εικόνα 23. RDBMS Update

Επίσης για τις λειτουργίες τροποποίησης προσφέρονται και η μέθοδος *db.collection.findAndModify ()*, η οποία βρίσκει και τροποποιεί ένα document. Τέλος, προσφέρεται και η μέθοδος *db.collection.save ()*, η οποία είτε εισάγει ένα document αν το document δεν έχει _id πεδίο, είτε το τροποποιεί αν βρει document με πεδίο _id. Και οι δύο μέθοδοι απευθύνονται στην τροποποίηση ενός μόνο document.

Remove

Για τις λειτουργίες διαγραφής των documents από μία collection, η MongoDB προσφέρει την μέθοδο *db.collection.remove ()*. Η μέθοδος δέχεται τα κριτήρια τα οποία προσδιορίζουν ποια documents θα διαγραφούν. Υπάρχει η δυνατότητα διαγραφής όλων των documents από μία collection, η διαγραφή των documents που προσδιορίζονται από τα κριτήρια και η δυνατότητα διαγραφής ενός μόνο document που ικανοποιεί τις συνθήκες.

Εικόνα 24. MongoDB Remove

Εικόνα 25. RDBMS Delete

Απομόνωση (Isolation) των write operations

Η τροποποίηση ενός document είναι πάντα ατομική ακόμη και αν η λειτουργία εγγραφής τροποποιήσει πολλαπλά embedded documents που εμπεριέχονται στο ίδιο το document. Αν μία λειτουργία εγγραφής τροποποιήσει πολλαπλά documents (*multi-update*), η λειτουργία δεν θεωρείται ατομική καθώς εξασφαλίζεται ατομικότητα μόνο για κάθε document. Σε αυτήν την περίπτωση, μπορούν και άλλες λειτουργίες να εμπλακούν. Ωστόσο, δίνεται η δυνατότητα χρήσης του τελεστή `$isolated`, ο οποίος απομονώνει μια λειτουργία εγγραφής η οποία επηρεάζει πολλαπλά documents. Χρησιμοποιώντας τον τελεστή, εξασφαλίζεται ότι κανένας χρήστης δεν θα δει τις αλλαγές των δεδομένων προτού η λειτουργία ολοκληρωθεί ή τερματιστεί λόγω λάθους. Η απομόνωση αυτή δεν εξασφαλίζει το "*all-or-nothing*" των σχεσιακών βάσεων δεδομένων.

3.3.4 Writes στην εφαρμογή

.

Insert

Στην παρακάτω εικόνα παρουσιάζεται μια εισαγωγή (insert) ενός post στο blog. Το document που περιγράφει την οντότητα post αποτελείται από fields, *title*, *body*, *author*, *date*, embedded Array *comments* και Array *tags*.

```
//Function that inserts a post
   this.insertEntry = function (title, body, tags, author, insertedPost) {
       "use strict";
       console.log("inserting blog entry" + title + body);


       // fix up the permalink to not include whitespace
       var permalink = title.replace( /\s/g, '_' );
       permalink = permalink.replace( /\W/g, '' );
       // Build a new post, schema-like for a post
       var post = {"title": title,
                "author": author,
                "body":  body,
                "permalink":  permalink,
                "tags":  tags,
                "comments":  [],
                "date": new Date()}

       // Now insert post in database
   posts.insert(post, function (err, result) {
           "use strict";

           // If err throw err, callback
           if (err) return (err, null);


           console.log("Inserted new post");
           //Insert post if no err, and return permalink
           insertedPost(err, permalink);
       });

   }
```

Εικόνα 26. Insert στο blog, εισαγωγή ενός νέου post

Update

Στην παρακάτω εικόνα παρουσιάζεται ένα update σε ένα post και συγκεκριμένα στον

embedded Array με sub-documents *comments*. Κάθε φορά που ένας χρήστης δημιουργεί ένα νέο comment, αυτό προσθέτεται στον Array με τα comments του κάθε post.

```
// Function that adds comment, updating the array Comments
this.addComment = function(permalink, name, email, body, addComment) {
    "use strict";

    var comment = {'author': name, 'body': body}

    if (email != "") {
        comment['email'] = email
    }
    // Mongo
    // Now add the comment using update
    posts.update(
            { 'permalink' : permalink},
            { '$push': { 'comments' : comment}},
            function (err, numModified) {

        "use strict";

        if (err) return (err, null);


        addComment(err, numModified);
    });

}
```

Εικόνα 27. Update στο blog, τροποποίηση του συγκεκριμένου post βάση του permalink, εισάγοντας νέο comment στο Array comments

Remove

Στην παρακάτω εικόνα υλοποιείται ένα remove, μια διαγραφή του session_id. Η συνάρτηση αυτή χρησιμοποιείται αργότερα όταν κάποιος χρήστης κάνει logout από το σύστημα.

```
this.endSession = function(session_id, removeSession) {
        "use strict";
        // Mongo
        // Remove session document
        sessions.remove({ '_id' : session_id }, function (err) {
            "use strict";
            removeSession(err);
        });
    }
```

Εικόνα 28. Remove στο blog, διαγραφή του session_id από το collection sessions.

3.4 Indexing

Η απόδοση μιας MongoDB βάσης δεδομένων σχετίζεται όπως και με όλες τις βάσεις δεδομένων με την χρήση ευρετηρίων (*indexes*). Η χρήση indexes υποστηρίζει την γρηγορότερη και αποδοτικότερη εκτέλεση των queries. Τα indexes είναι ειδικές δομές δεδομένων που εντοπίζουν γρήγορα documents με βάση ένα ή περισσότερα συγκεκριμένα πεδία. Χωρίς την χρήση indexes, η MongoDB πρέπει να σαρώσει κάθε document σε μία collection για να επιλέξει αυτά τα οποία ικανοποιούν την συνθήκη του query. Σε μια τέτοια περίπτωση, ο φόρτος εργασίας γίνεται μεγάλος καθώς η MongoDB θα πρέπει να σαρώσει όλη την collection για κάθε λειτουργία ανάγνωσης ή εγγραφής [22].

Στην MongoDB τα indexes αποθηκεύονται σε μορφή *B-trees* όπως και στις σχεσιακές βάσεις δεδομένων. Τα indexes ορίζονται σε επίπεδο μιας collection και υποστηρίζονται indexes σε κάθε field, σύνολο από fields, embedded Object (sub-documents) και Array ενός document. Όπως αναφέρθηκε σε παραπάνω κεφάλαιο, η MongoDB δημιουργεί αυτόματα index με βάση το *_id* του κάθε document. Συγκεκριμένα οι τύποι των indexes που υποστηρίζονται από την MongoDB ως τώρα είναι:

- βάση ενός πεδίου (ορίζει ο χρήστης ένα index σε ένα μόνο πεδίο)
- βάση παραπάνω του ενός πεδίου (σύνθετα indexes που αφορούν παραπάνω από ένα πεδία)
- Multikey index (χρήση indexes σε *Arrays*)

- Geospatial indexes (χρήση indexes που αφορούν γεωγραφικά δεδομένα,*2D,2Sphere*)
- Text indexes (χρήση indexes για την αναζήτηση ενός String με βάση το περιεχόμενο)
- Hashed Indexes (εφαρμόζονται σε hash-based sharding)

Μπορούμε να ορίσουμε indexes στην MongoDB καλώντας την μέθοδο `ensureIndex()` μέσω του φλοιού *mongo*. Επίσης, κατά την δημιουργία των indexes μπορούμε να εισάγουμε ως δεύτερο όρισμα τις ιδιότητες *unique*, *Sparse* και *TTL*. Η ιδιότητα unique διασφαλίζει την μη ύπαρξη διπλότυπης τιμής όσο αφορά το πεδίο που ορίζουμε για index.

Υπάρχουν πολλοί παράγοντες που καθορίζουν την σωστή δημιουργία και χρήση indexes. Ένας σημαντικός παράγοντας αφορά την εκλεκτικότητα (*selectivity*), δηλαδή την ικανότητα ενός query να περιορίσει τον πιθανό αριθμό των documents που θα επιστραφούν χρησιμοποιώντας το index. Επίσης θα πρέπει τα indexes να είναι επαναχρησιμοποιήσιμα, δηλαδή να μπορούμε να τα χησιμοποιήσουμε σε όσα περισσότερα queries. Τέλος, καθώς τα indexes διατηρούνται στην *RAM* και καταναλώνουν χώρο, θα πρέπει να μπορούν να χωράνε στην RAM για να μεγιστοποιείται η απόδοση της βάσης. Αν ένα index είναι πολύ μεγάλο και δεν μπορεί να διατηρηθεί στην RAM, η Mongo θα πρέπει να διαβάσει το index από τον δίσκο, λειτουργία αρκετά χρονοβόρα. Σε εφαρμογές με πολλά write requets, η δημιουργία πολλών indexes μπορεί να επηρεάσει την απόδοση αρνητικά γιατί για κάθε write request που επηρεάζει ένα index, η βάση πρέπει να κάνει update τα συγκεκριμένα indexes εκτός από το ίδιο το document.

Ανάλογα με τα queries που σχεδιάζονται, την συχνότητα που εκτελούνται και την αναλογία των read & write requests επιλέγουμε την δημιουργία των κατάλληλων indexes. Υπάρχουν περιπτώσεις όπου τα indexes δεν μας προσφέρουν απόδοση όπως στα updates. Γιαυτό πρέπει να δημιουργούνται προσεκτικά και για τα πιο συχνά queries. Τα indexes γενικά διατηρούνται στην RAM και γιαυτό εκτελούνται γρηγορότερα τα queries. Αν το index είναι πολύ μεγάλο για να χωρέσει στην RAM, η Mongo πρέπει να διαβάσει το index από τον δίσκο, και έτσι επιβραδύνεται κατά πολύ η εκτέλεση του query. Σε κάθε query χρησιμοποιείται ένα μόνο index. Η μέθοδο explain () παρέχει στατιστικά όσο αφορά την

απόδοση του query που εφαρμόζουμε. Η μέθοδος χρησιμοποιείται σε παρακάτω παραδείγματα.

3.4.1 Indexing στην εφαρμογή

Για τους σκοπούς του indexing στην εφαρμογή, φορτώθηκε ένα dataset με 1000 documents που αφορούν την συλλογή posts. Το συγκεκριμένο dataset περιέχει 1000 documents τύπου post, τα οποία εμπεριέχουν πολλά comments και μεγάλο αριθμό από tags για κάθε document. Δημιουργήθηκαν indexes για την αρχική σελίδα της εφαρμογής (http://locahost:3000), για την σελίδα η οποία επιστρέφει τα posts με συγκεκριμένο 'tag' (http://localhost:3000/tag/whatever) , για την σελίδα που επιστρέφει ένα post με βάση το 'permalink' του (http://localhost:3000/post/permalink) και τέλος για την σελίδα της αναζήτησης βάση του 'title' των posts (http://localhost:3000/search/title) . Χρησιμοποιήθηκε η μέθοδος explain () για να καταλάβουμε πώς συμπεριφέρεται η βάση σε αυτά τα read requets. Δίνεται ένα παράδειγμα της χρήσης explain () πριν και μετά την δημιουργία index για την αρχική σελίδα της εφαρμογής, οπού επιστρέφονται 10 posts ταξινομημένα βάση του date.

```
> db.posts.find ().sort ({'date':-1}).limit (10).explain ()
{
      "clauses" : [
            {
                  "cursor" : "BasicCursor",
                  "isMultiKey" : false,
                  "n" : 10,
                  "nscannedObjects" : 1000,
                  "nscanned" : 1000,
                  "scanAndOrder" : true,
                  "indexOnly" : false,
                  "nChunkSkips" : 0
            },
            {
                  "cursor" : "BasicCursor",
                  "isMultiKey" : false,
                  "n" : 0,
                  "nscannedObjects" : 0,
                  "nscanned" : 0,
                  "scanAndOrder" : true,
                  "indexOnly" : false,
                  "nChunkSkips" : 0
            }
      ],
      "cursor" : "QueryOptimizerCursor",
      "n" : 10,
      "nscannedObjects" : 1000,
      "nscanned" : 1000,
      "nscannedObjectsAllPlans" : 1000,
      "nscannedAllPlans" : 1000,
      "scanAndOrder" : false,
      "nYields" : 49,
      "nChunkSkips" : 0,
      "millis" : 1211,
      "server" : "stoug-debian-uom-pc:27017",
      "filterSet" : false
}
>
```

Πίνακας 4. Ανάκτηση των δέκα τελευταίων post της εφαρμογής, ταξινομημένα με βάση το date, χωρίς την χρήση index

```
> db.posts.ensureIndex ({'date':-1})
{
      "createdCollectionAutomatically" : false,
      "numIndexesBefore" : 1,
      "numIndexesAfter" : 2,
      "ok" : 1
}
```

Πίνακας 5. Δημιουργία index στο πεδίο date

```
> db.posts.find ().sort ({'date':-1}).limit (10).explain ()
{
      "cursor" : "BtreeCursor date_-1",
      "isMultiKey" : false,
      "n" : 10,
      "nscannedObjects" : 10,
      "nscanned" : 10,
      "nscannedObjectsAllPlans" : 10,
      "nscannedAllPlans" : 10,
      "scanAndOrder" : false,
      "indexOnly" : false,
      "nYields" : 0,
      "nChunkSkips" : 0,
      "millis" : 66,
      "indexBounds" : {
            "date" : [
                  [
                        {
                              "$maxElement" : 1
                        },
                        {
                              "$minElement" : 1
                        }
                  ]
            ]
      },
      "server" : "stoug-debian-uom-pc:27017",
      "filterSet" : false
}
```

Πίνακας 6. Ανάκτηση των δέκα τελευταίων post της εφαρμογής, ταξινομημένα με βάση το date, μετά την χρήση index στο 'date'

Στους παραπάνω πίνακες, παρατηρείται ότι για το συγκεκριμένο query, χωρίς την χρήση index σε κάποιο πεδίο, η βάση πρέπει να σαρώσει όλα τα documents προκειμένου να επιστρέψει τα δέκα posts ταξινομημένα με βάση το date (`nscannedObjects" : 1000`). Επίσης ο χρόνος που κάνει να επιστρέψει το read αντιστοιχεί σε 1211 millisecond. Από την άλλη πλευρά, με την χρήση index στο πεδίο 'date' για το συγκεκριμένο query, παρατηρούμαι ότι το query επέστρεψε πολύ πιο γρήγορα (“ millis" : 66) σαρώνοντας μόνο τα document που ζητήθηκαν (“nscannedObjects" : 10). Παρομοίως, εφαρμόστηκαν indexes για τις περιπτώσεις που αναλύθηκαν προηγουμένως. Στον παρακάτω πίνακα δίνονται τα indexes που δημιουργήθηκαν στην βάση γι αυτές τις περιπτώσεις καθώς και τα αντίστοιχα

queries.

Query	Index
db.posts.find ({tags:'tag'}).sort ({'date':-1}).limit (10)	db.posts.ensureIndex ({'tags':1,'date':-1}
db.posts.findOne ({permalink:'permalink'})	db.posts.ensureIndex ({permalink : 1 })
db.posts.find ({"$text":"$search":title }}).sort ('date', -1).limit (10)	db.posts.ensureIndex ({"title":"text"})

Πίνακας 7. Indexes στην εφαρμογη

3.5 Αρχιτεκτονική

Η MongoDB έχει ως κύριο στόχο την υποστήριξη κατανεμημένων συστημάτων που τρέχουν σε κατανεμημένα περιβάλλοντα -*clusters*- υπολογιστών. Έτσι, με την προσθήκη νέων κόμβων πραγματοποιείται αυτόνομος καταμερισμός του φόρτου εργασίας. (*auto-sharding*) μεταξύ των κόμβων του cluster παρέχοντας επεκτασιμότητα στην βάση. Η τεχνική του *sharding*, η οποία θα αναλυθεί σε επόμενη ενότητα, διαμερίζει τα δεδομένα και τα κατανέμει σε πολλαπλούς κόμβους shard. Κάθε shard είναι μια ανεξάρτητη βάση δεδομένων και συνολικά όλοι οι κόμβοι shard ενός cluster αποτελούν μια λογική βάση δεδομένων. Σε ένα MongoDB cluster διακρίνονται τρεις τύπου κόμβων, οι *shards*, οι *configuration servers*

και οι *routers*. Στην παρακάτω εικόνα φαίνεται πώς είναι δομημένο ένα MongoDB cluster.

3.5.1 Shard Nodes

Οι κόμβοι shard είναι υπεύθυνοι για την αποθήκευση των δεδομένων. Ένα shard αποτελείται είτε από μία *mongod* διεργασία είτε συνήθως από ένα σύνολο διακοσμητών mongod, τα *replica set*. Κάθε βάση δεδομένων αποτελείται από έναν primary shard το οποίο είναι υπεύθυνο να κρατάει τις un-sharded collections στην βάση σε sharded περιβάλλον. Ένα replica set αποτελείται από έναν ή περισσότερους κόμβους που διατηρούν αντίγραφα των ίδιων δεδομένων με σκοπό την υψηλή διαθεσιμότητα των δεδομένων μας, την συνέπεια τους και την ανοχή σε σφάλματα. Από τους κόμβους που δομούν ένα shard (replica set), ένας θεωρείται primary και οι υπόλοιποι secondary. Από προεπιλογή στην MongoDB, τα writes και τα consistent reads δρομολογούνται στον primary κόμβο, ενώ τα eventual consistency reads διαμερίζονται στους secondary. Σε περίπτωση αποτυχίας του πρωτεύον κόμβου, ένας από τους επόμενους δευτερεύοντες κόμβους αναλαμβάνει ως πρωτεύον μέσω μιας διαδικασίας εκλογής (*election*) [23].

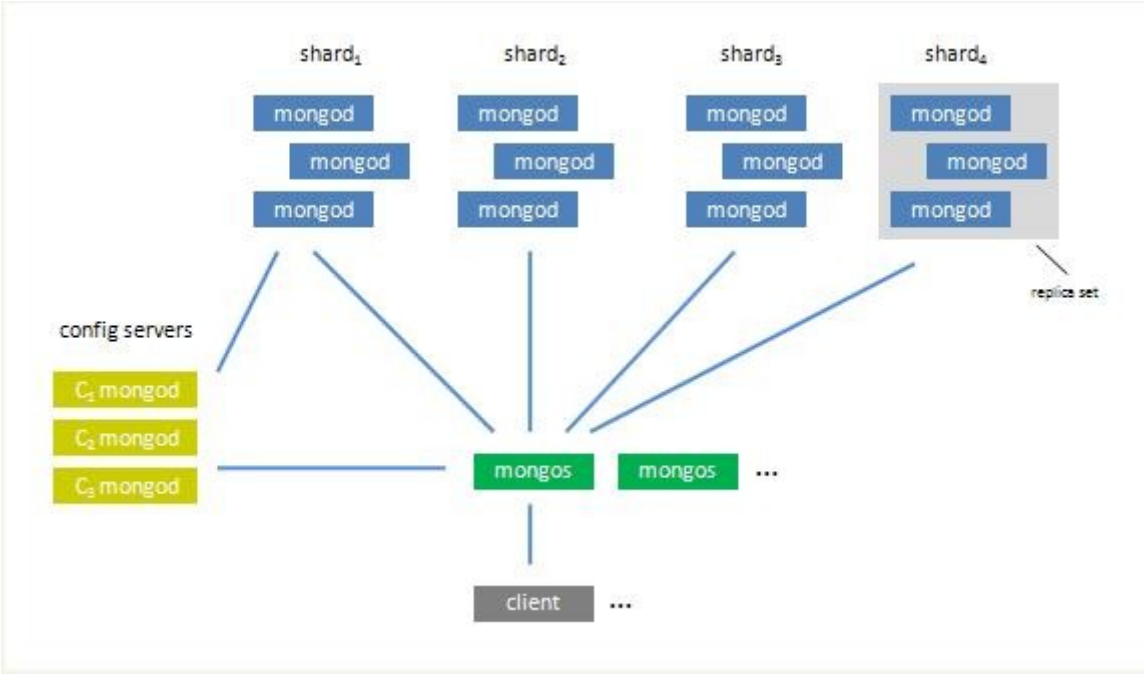
3.5.2 Configuration Servers

Οι *configuration servers* είναι ειδικές *mongod* διεργασίες οι οποίες είναι υπεύθυνες για την αποθήκευση των *metadata* σε ένα cluster. Τα metadata περιέχουν δεδομένα και πληροφορίες δρομολόγησης που δείχνουν που είναι αποθηκευμένα τα δεδομένα μας στο κάθε διαφορετικό κόμβο shard. Η πρόσβαση σε αυτούς γίνεται μέσω των shards, που ενημερώνουν ποία δεδομένα διαθέτουν. Έτσι, οι query routers χρησιμοποιούν τα metadata για να δρομολογήσουν τα reads και τα writes στο κατάλληλο κόμβο shard. Οι config servers χρησιμοποιούν την τεχνική *two-phase-commit* για να εξασφαλίσουν την άμεση συνέπεια και αξιοπιστία των δεδομένων. Κάθε sharded cluster έχει ακριβώς τρεις config servers [24].

3.5.3 Query Routers

Οι *query routers*, είναι ειδικές διεργασίες *mongos* οι οποίες χρησιμοποιούνται για την επικοινωνία ενός ή περισσότερων *clients* με την βάση δεδομένων. Είναι ο μόνος τρόπος επικοινωνίας της εφαρμογής με το κατανεμημένο sharded περιβάλλον. Δέχονται τα requests

για reads και writes από τους χρήστες, επικοινωνούν με τους config servers για να μάθουν σε ποιο shard είναι τα ζητούμενα δεδομένα και στέλνουν το query στα αντίστοιχα shards. Αφού τελειώσουν τα requests, οι routers στέλνουν τα αποτελέσματα πίσω στον χρήστη. Ένα sharded cluster μπορεί να περιέχει παραπάνω από ένα router για να διαμερίσει τον φόρτο εργασίας του request. Ένας χρήστης στέλνει requests μόνο σε ένα query router. Οι mongos διεργασίες καταναλώνουν ελάχιστους πόρους από το σύστημα και έτσι στα περισσότερα sharded clusters υπάρχουν πολλοί query routers [25].



Εικόνα 29. Αρχιτεκτονική κατανεμημένου συστήματος MongoDB

3.6 Storage – Εσωτερικοί μηχανισμοί συστήματος

Η MongoDB αποθηκεύει τα δεδομένα της με την μορφή *BSON* document. Ένα BSON

αρχείο δεν μπορεί να ξεπερνάει τα *16MB*. Κάθε document αποθηκεύετε ως ένα έγγραφο (*record*) το οποίο περιέχει το document και κάποιον επιπλέον χώρο, ή *padding*, ο οποίος προστίθεται στο τέλος του document και του επιτρέπει να μεγαλώσει λίγο σε μέγεθος.

3.6.1 Preallocated data files and Padding

Η MongoDB χρησιμοποιεί έναν μηχανισμό pre-allocation για την αποθήκευση και διαχείριση των αρχείων της. Κατά το pre-allocation, όταν ένα αρχείο φτάσει ένα ορισμένο μέγεθος, δεσμεύεται αυτόματα χώρος για το επόμενο, πριν η εφαρμογή ζητήσει την δημιουργία του. Τα αρχεία που δημιουργούνται με pre-allocation έχουν γνωστό και σταθερό μέγεθος. Έτσι αποφεύγεται η ανεξέλεγκτη αύξηση του μεγέθους των αρχείων και το filesystem fragmentation που θα προκαλούταν χωρίς την εφαρμογή της τεχνικής αυτής. Τα αρχεία στον δίσκο μοιράζονται σε μεγέθη (*extents)* τα οποία περιέχουν τα documents. Η MongoDB ονομάζει το πρώτο αρχείο που δημιουργείται <*databasename*.0>, <databasename.1>, κοκ. Για κάθε βάση δεδομένων η MongoDB ξεκινά το pre-allocation με πρώτο αρχείο μεγέθους 64MB. Το επόμενο αρχείο που δημιουργείται ως αποτέλεσμα του pre-allocation έχει διπλάσιο μέγεθος (128MB), μέχρι το αρχείο να φτάσει τα 2GB. Το αρχείο που δημιουργείται από την διαδικασία αυτή αρχικά δεν περιέχει δεδομένα. Είναι δυνατόν να δεσμευτεί αρχείο 1GB όπου το 90% να είναι ελεύθερο. Είναι φανερό πως κάθε βάση στην MongoDB μπορεί να έχει αχρησιμοποίητο χώρο μέχρι 2GB.

Σε φόρτους εργασίας όπου παρατηρούνται πολλά *updates,* υπάρχει ο κίνδυνος να αυξηθεί το μέγεθος ενός document ανεξέλεγκτα ξεπερνώντας τον δεσμευμένο του χώρο. Γιαυτό τον λόγο η MongoDB, πρέπει να δεσμεύσει νέο χώρο και να μεταφέρει το document στην νέα του θέση. Κατά την διαδικασία αυτή το σύστημα πρέπει να μεταφέρει το document και να κάνει ενημέρωση σε όλα τα indexes και τα shards που αναφέρονται στο συγκεκριμένο document. Αυτή η διαδικασία είναι αρκετά χρονοβόρα και μπορεί να προκαλέσει filesystem fragmentation. Γιαυτό τον λόγο, η MongoDB προκειμένου να αυξήσει την απόδοση τους συστήματος, κατά την διαδικασία δέσμευσης χώρου (allocation) για το νέο αρχείο, προστίθεται κάποιος επιπλέον χώρος στο τέλος του document, η *padding*, που του επιτρέπει να μεγαλώσει λίγο σε μέγεθος χωρίς να χρειαστεί η μεταφορά του σε άλλο σημείο.

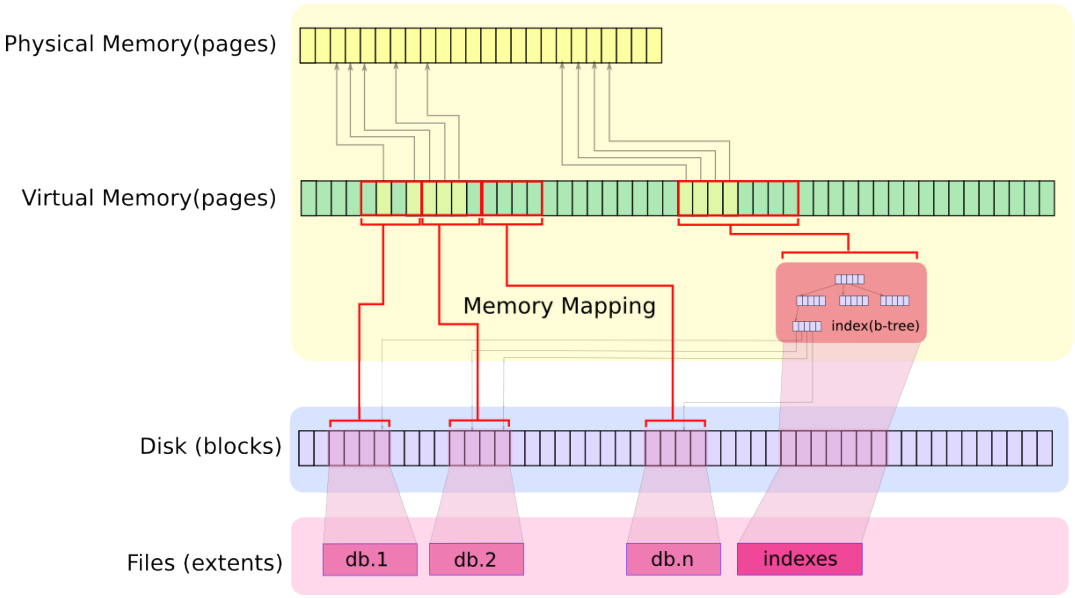
Η MongoDB υποστηρίζει πολλές στρατηγικές για το *allocation* των αρχείων, οι οποίες

προσδιορίζουν τον τρόπο που το σύστημα προσθέτει επιπλέον χώρο (padding) στο document κατά την δημιουργία του έγγραφου (record). Καθώς τα documents μπορεί να αυξηθούν σε μέγεθος μετά από write requests και από την στιγμή που τα records αποθηκεύονται σε διαδοχικές θέσεις μνήμης, ο μηχανισμός του padding μπορεί να μειώσει την ανάγκη μεταφοράς των documents στον δίσκο κατά την διάρκεια που πραγματοποιούνται updates. Αναλόγως με τις ανάγκες της εκάστοτε εφαρμογής, έχουν αναπτυχθεί διαφορετικοί μηχανισμοί allocation. Για εφαρμογές με πολλά *write requets* (insert/update/delete) χρησιμοποιείται η *Power of 2size allocation*, η οποία είναι από προεπιλογή στρατηγική. Τέλος, για collections όπου δεν παρατηρούνται *updates* και *deletes*, η προτεινόμενη είναι ή *exact fit allocation* [26].

3.6.2 Memory-mapped files

Η MongoDB χρησιμοποιεί memory-mapped files για τη διαχείριση και την αλληλεπίδραση των δεδομένων. Κατά τη διαδικασία του memory mapping, η MongoDB ζητά από το λειτουργικό του συστήματος, να κάνει map όλα τα αρχεία από τον δίσκο στη RAM καθώς έχει πρόσβαση στα documents. Η διαδικασία επικοινωνίας του λειτουργικού συστήματος με τη RAM, υλοποιείται μέσω της *virtual memory*. Τα αρχεία αντιστοιχίζονται σε blocks εικονικής μνήμης με μία byte-to-byte συσχέτιση. Αφού ολοκληρωθεί η διαδικασία του map, η σχέση που προκύπτει ανάμεσα στο αρχείο και στην μνήμη, επιτρέπει στην MongoDB να αλληλεπιδράει με τα δεδομένα του αρχείου σαν να ήταν αποθηκευμένα στη φυσική μνήμη. Η διαδικασία της χρήσης memory mapped files προσφέρει στην MongoDB υψηλή απόδοση για την πρόσβαση και επεξεργασία των δεδομένων της. Τα δεδομένα τα όποια δεν είναι προσβάσιμα δεν αντιστοιχίζονται στην μνήμη.

Η χρήση memory-mapped files αποτελεί κρίσιμο κομμάτι στον μηχανισμό διαχείρισης των δεδομένων της Mongo. Ωστόσο παρατηρούνται και μειονεκτήματα χρήσης του όπως η εμφάνιση page faults. Κατά την διαδικασία πρόσβασης στα δεδομένα μέσω reads και writes, για τα αρχεία των δεδομένων που δεν είναι διαθέσιμα εκείνη την στιγμή στην μνήμη μπορούν να εμφανιστούν page faults. Τέλος, υπάρχει και κίνδυνος για RAM fragmentation που οφείλεται στο ακριβές mapping του δίσκου στην RAM. Γιαυτό το λόγο θα πρέπει να ελέγχεται συχνά το ποσοστό της ελεύθερης μνήμης σε σχέση με την μνήμη που καταναλώνει το σύστημα της MongoDB [27].



Εικόνα 30. MongoDB memory-mapped files

3.6.3 Journal files

Το σύστημα διαχείρισης την Mongo προκειμένου να εξασφαλίσει την συνέπεια της βάσης και την μονιμότητα (durabily) των *writes* χρησιμοποιεί *write ahead logging* σε ένα αρχείο journal. Προκειμένου να διασφαλιστεί ότι όλες οι τροποποιήσεις γράφονται στο δίσκο μόνιμα, η MongoDB εγγράφει τις τροποποιήσεις αυτές στο journal συχνότερα από τι γράφει στα αρχεία των δεδομένων. Αν κάθε φορά που γινόταν ένα write γραφόταν αμέσως από το journal στο δίσκο η επίδοση της βάσης θα μειωνόταν. Γιαυτό το λόγο το journal ομαδοποιεί τα writes σε δέσμες και τα προωθεί μαζικά και περιοδικά στα αρχεία του δίσκου. Από προεπιλογή, η Mongo γράφει τα δεδομένα στα αρχεία των δεδομένων κάθε 60sec και πραγματοποιεί commit στο journal κάθε 100millsec. Ωστόσο, οι χρήστες έχουν τη δυνατότητα να ρυθμίσουν κάθε πότε τα writes γράφονται στο δίσκο ή στο journal.

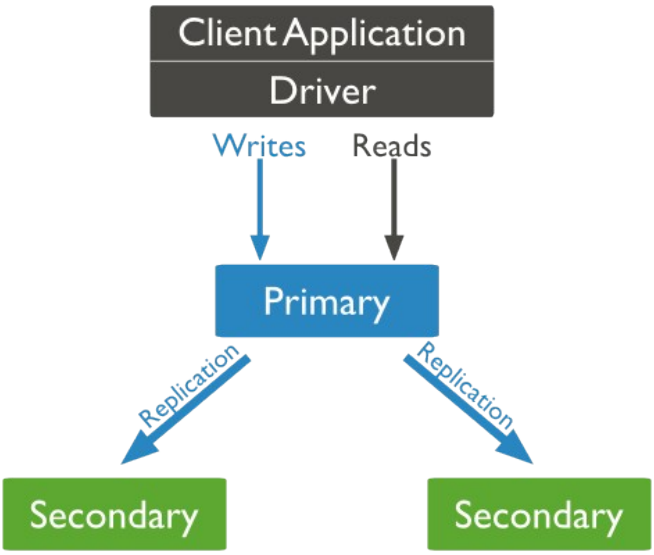
Σε περίπτωση αστοχίας του συστήματος , υπάρχει η δυνατότητα της επανεφαρμογής των τελευταίων αλλαγών από το journal προκειμένου να έρθει η βάση σε συνεπή μορφή και

πάλι. Όταν πραγματοποιείται επιτυχής shutdown της MongoDB, διαγράφονται όλα τα αρχεία τύπου journal. Στις *64byte* διανομές της MongoDB, το journal είναι αυτόματα ενεργοποιημένο, ενώ στις *32byte* διανομές ορίζεται από τον χρήστη εισάγοντας το όρισμα – *journal*. Γενικά σε περιβάλλοντα παραγωγής, ο μηχανισμός πρέπει να είναι ενεργοποιημένος, καθώς σε περίπτωση αστοχίας υλικού εξασφαλίζεται η επαναφορά της βάσης σε συνεπή μορφή [28].

3.7 Replication

Replication είναι η διαδικασία αντιγραφής των δεδομένων της βάσης σε άλλους κόμβους και έχει ως στόχο την εξασφάλιση της υψηλής διαθεσιμότητας των δεδομένων, τη συνέπεια και αξιοπιστία των δεδομένων και την ανοχή σε σφάλματα. Έχοντας πολλαπλά αντίγραφα των δεδομένων μας σε διαφορετικούς κόμβους, προστατεύεται η βάσης μας από την απώλεια ενός μεμονωμένου διακομιστή. Η MongoDB χρησιμοποιεί *master-slave* replication (πριν την έκδοση 1.6) και μια παραλλαγή της τεχνικής master-slave, την single-*master* που ονομάζεται *replica sets*. Η εφαρμογή των replica sets για τον μηχανισμό του replication έχει επικρατήσει και συνιστάται από την MongoDB.

Κάθε replica set, που συνήθως είναι ένας shard κόμβος, αποτελείται από έναν primary/master κόμβο και έναν ή περισσότερους secondary/slave κόμβους όπου όλοι είναι διεργασίες *mongod* και φιλοξενούν το ίδιο σετ δεδομένων. Ένα replica set μπορεί να διαθέτει έως 12 κόμβους, έναν primary και 11 secondary. Ο primary κόμβος είναι ο μόνος που δέχεται writes και reads από τον *client* και ένα replica set μπορεί να έχει μόνο έναν primary. Καθώς ένας μόνο κόμβος, ο primary, δέχεται λειτουργίες write, αυτόματα πετυχαίνεται και strong consistency στα reads. Ο primary κόμβος καταγράφει όλες τις εργασίες που ακολουθεί και τις τροποποιήσεις των δεδομένων του σε ένα log, στο *oplog*.



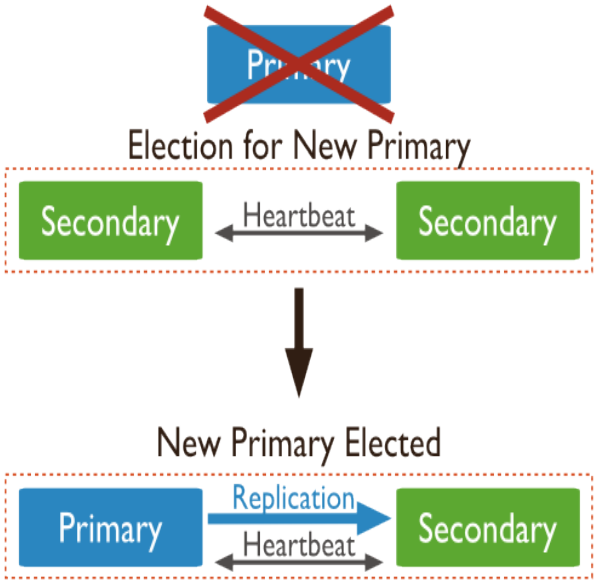
Εικόνα 31. Replication MongoDB Replica-set

Οι secondary κόμβοι είναι *read-only* κόμβοι. Οι secondary κόμβοι ενημερώνονται ασύγχρονα από το log του primary κόμβου, εφαρμόζουν στα δεδομένα τους τις αλλαγές που τους αφορούν και διατηρούν δεδομένα ενδεχόμενης συνέπειας. Ο λόγος που οι secondary κόμβοι κρατούν δεδομένα με ενδεχόμενη συνέπεια είναι γιατί μπορεί να μην προλάβουν να αντιγράψουν τα τελευταία writes από τον primary καθώς ενημερώνονται ασύγχρονα. Στην παραπάνω εικόνα, φαίνεται ένα replica set με τρεις κόμβους όπου ο primary δέχεται όλα τα writes και τα reads. Στην συνέχεια, οι secondary κόμβοι αντιγράφουν το oplog για να εφαρμόσουν τις αλλαγές και στα δικά τους δεδομένα.

Από προεπιλογή, η εφαρμογή (client) στέλνει τα writes και reads request στον primary κόμβο. Ωστόσο, υπάρχει η δυνατότητα παραμετροποίησης του client έτσι ώστε να στέλνονται τα reads στους secondary κόμβους. Με αυτόν τον τρόπο, κατανέμοντας δηλαδή μερικά από τα reads ή και όλα τα read requets στους secondary, βελτιώνουμε την απόδοση των read requests. Η τεχνική αυτή συνιστάται σε περιπτώσεις όπου η εφαρμογή μας δεν απαιτεί να έχει πλήρη ενημερωμένα δεδομένα.

Με την χρήση replica set διασφαλίζεται από την Mongo το αυτόματο failover σε περίπτωση σφάλματος. Οι κόμβοι ενός replica set χρησιμοποιούν *heartbeats (pings)* για να εντοπίσουν τους κόμβους που έχουν αστοχήσει. Σε περίπτωση αστοχίας του primary κόμβου αυτόματα εκλέγεται ένας από τους secondary ως primary. Όταν ο primary κόμβος επανέλθει τότε συνεχίζει την λειτουργία του ως secondary. Τέλος, υπάρχει η δυνατότητα εισαγωγής στο replica set ενός άλλου κόμβου, του *artiber,* ο οποίος δεν περιέχει δεδομένα αλλά υπάρχει μόνο για να ψηφίσει στην διαδικασία εκλογής ενός primary. Ένας artiber κόμβος

έχει πάντα την ιδιότητα του ψηφοφόρου. Ένας primary μπορεί να αποτύχει και να γίνει secondary και ένας secondary μπορεί να γίνει primary κατά την διαδικασία της εκλογής. Η χρήση των artiber γίνεται σε περιπτώσεις όπου στο replica set υπάρχει άρτιος αριθμός μελών κόμβων. Τέλος, σε περιπτώσεις που απαιτούνται περισσότεροι secondary κόμβοι η MongoDB δίνει την επιλογή χρήσης master-slave replication. Με την χρήση master-slave αρχιτεκτονικής δεν υποστηρίζεται ο μηχανισμός του αυτόματου failover. Στην παρακάτω εικόνα φαίνεται η διαδικασία εκλογής ενός primary κόμβου σε περίπτωση σφάλματος [29].



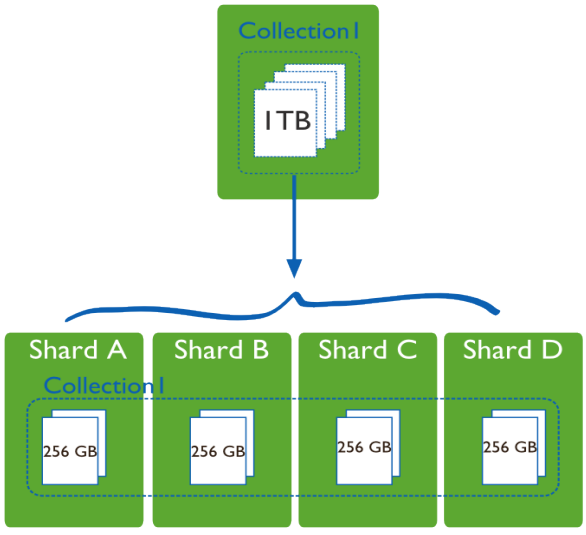
Εικόνα 32. Εκλογή Primary κόμβου σε περίπτωση σφάλματος

3.8 Sharding - Κατακερματισμός

Τα συστήματα βάσεων δεδομένων που διαθέτουν και διαχειρίζονται μεγάλο όγκο

δεδομένων και οι εφαρμογές υψηλής απόδοσης συχνά βρίσκονται αντιμέτωπα με τις ικανότητες ενός μόνο server. Τα υψηλής απόδοσης queries μπορεί να εξαντλήσουν την CPU και τα μεγάλα σύνολα δεδομένων μπορούν να υπερβούν την χωρητικότητα ενός μόνο μηχανήματος. Για να αντιμετωπίσουν τα συστήματα βάσεων δεδομένων αυτές τις ανάγκες χρησιμοποιούν την τεχνική της κλιμάκωσης (scaling). Η κλιμάκωση αναφέρεται στην επεκτασιμότητα της βάσης, δηλαδή την διαχείριση όλο και μεγαλύτερου όγκου δεδομένων ή και την υποστήριξη περισσότερων χρηστών. Παρατηρούνται δύο μορφές κλιμάκωσης, η κάθετη κλιμάκωση (vertical-scaling) και η οριζόντια κλιμάκωση (horizontal-scaling). Για να κλιμακώσουμε το σύστημα μας κάθετα αρκεί να προσθέσουμε περισσότερο hardware. Με την προσθήκη περισσότερων *CPU*, *RAM* και αποθηκευτικών μέσων πετυχαίνεται η κάθετη κλιμάκωση του συστήματος με αντίκτυπο όμως το υψηλό κόστος. Η εφαρμογή της κάθετης κλιμάκωσης παρατηρείται στα σχεσιακά συστήματα. Αντιθέτως, με την τεχνική της οριζόντιας κλιμάκωσης ή *sharding,* διαιρείται το σύνολο των δεδομένων και κατανέμονται τα δεδομένα μας σε πολλαπλούς κόμβους (shards). Κάθε shard αποτελεί μια ανεξάρτητη πλέον βάση και συλλογικά όλα τα shards σε ένα cluster συνθέτουν μια λογική βάση δεδομένων. Στην παρακάτω εικόνα φαίνεται πώς συμπεριφέρεται μια βάση δεδομένων, σε επίπεδο collection, μετά την εφαρμογή του sharding και την προσθήκη τεσσάρων κόμβων στο cluster [30].

Ο μηχανισμός του sharding μειώνει τον φόρτο εργασίας που κάθε κόμβος χειρίζεται. Κάθε κόμβος επεξεργάζεται λιγότερα read και write requests όσο μεγαλώνει το cluster. Επιπλέον, μειώνει τον όγκο των δεδομένων που κάθε κόμβος χρειάζεται να αποθηκεύσει όσο προσθέτονται περισσότεροι κόμβοι στο cluster. Για παράδειγμα, στην παρακάτω εικόνα ο όγκος δεδομένων μας είναι της τάξης 1TB. Με την διατήρηση τεσσάρων κόμβων στο cluster μας, κάθε κόμβος μπορεί να αποθηκεύσει ως 256GB των δεδομένων. Αν υπήρχαν για παράδειγμα σαράντα shard κόμβοι, τότε στον κάθε κόμβο θα μπορούσε να διατηρείται μόνο 25GB δεδομένων.



Εικόνα 33. Εφαρμογή Sharding σε 4 κόμβους

3.8.1 Αυτόματο Sharding

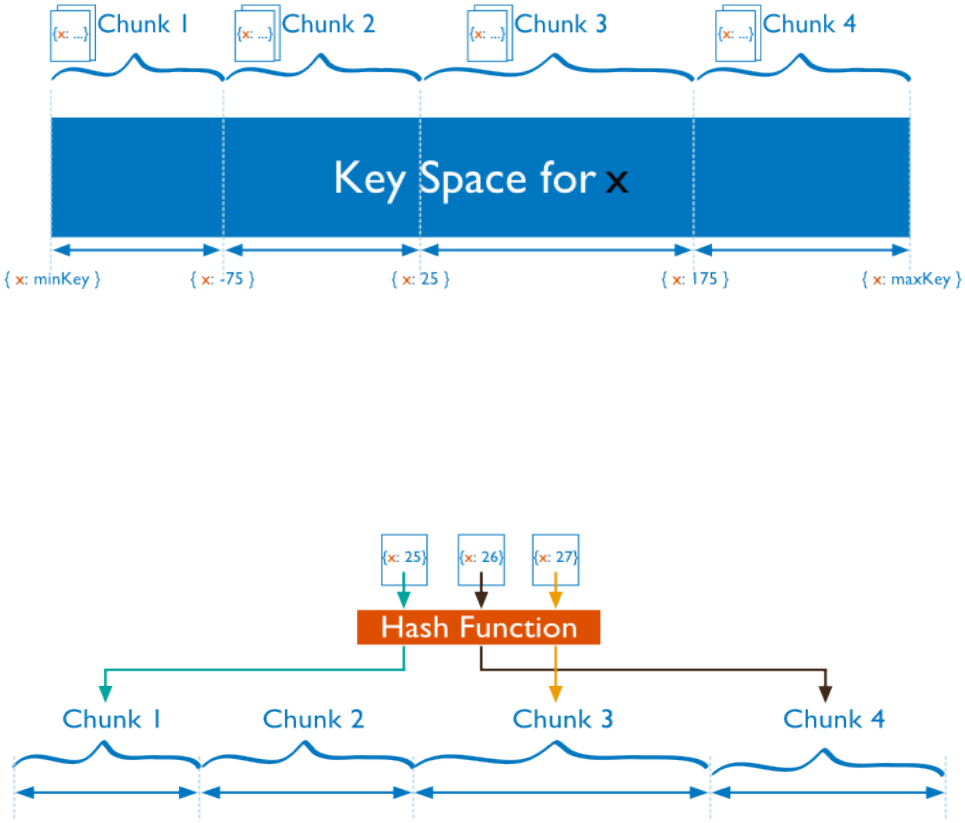
Στην MongoDB ένα cluster αποτελείται από : δύο και παραπάνω shard nodes, στους οποίους αποθηκεύονται τα δεδομένα, τρεις configuration servers όπου αποθηκεύονται τα metadata του cluster (που δείχνουν ποιά δεδομένα έχει ο κάθε κόμβος) και έναν ή περισσότερους query routers οι οποίοι επικοινωνούν με την εφαρμογή. Στην ενότητα 3.4 φαίνεται η εικόνα ενός MongoDB cluster καθώς και οι αρμοδιότητες που έχουν οι εμπλεκόμενοι κόμβοι. Η MongoDB υποστηρίζει αυτόματο sharding, ισοκατανέμοντας τον όγκο των δεδομένων και το φόρτο εργασίας σε όλους τους κόμβους του cluster. Με την προσθήκη νέων κόμβων τα δεδομένα καταμερίζονται αυτόματα, παρέχοντας έτσι επεκτασιμότητα στην βάση.

3.8.2 Διαμερισμός των δεδομένων σύμφωνα με το shard key

Οι προγραμματιστές επιλέγουν σε ποίες βάσεις δεδομένων και σε ποιες collections θα ενεργοποιήσουν το sharding. Η εφαρμογή του sharding γίνεται με βάση τα collections. Στις collections μίας βάσης όπου δεν εφαρμόζεται ο μηχανισμός του sharding, ο primary shard κόμβος αναλαμβάνει να φιλοξενήσει τις συγκεκριμένες unsharded collections. Αντιθέτως, σε collection που επιθυμούμε να ενεργοποιηθεί το sharding, κάθε τέτοια collection διαμερίζει τα documents της σύμφωνα ένα ορισμένο από τον χρήστη *shard key.*

Ένα shard key είναι είτε ένα index είτε ένα σύνθετο index (compound index), ένα πεδίο το οποίο υποχρεωτικά υπάρχει σε κάθε document της collection. Η MongoDB διαχωρίζει τις τιμές που έχουν τα documents σύμφωνα με το συγκεκριμένο shard key σε *chunks,* και στην συνέχεια κατανέμει τα chunks ομοιόμορφα σε κάθε κόμβο. Αν ένα chunk μεγαλώσει πολύ σε μέγεθος χωρίζεται. Τα chunks χρησιμοποιούνται για την ισοκατανομή των δεδομένων στο

cluster με την αυτόματη μετακίνηση τους, σε περίπτωση προσθήκης ή αφαίρεσής κόμβων από το cluster. Για την διαδικασία διαχωρισμού των τιμών του shard key, η Mongo χρησιμοποιεί *range-based* ή *hash-based* partitioning. Η κάθε τεχνική έχει τα πλεονεκτήματα και τα μειονεκτήματα της. Στην παρακάτω εικόνα φαίνεται πώς συμπεριφέρονται οι προαναφερθέντες μέθοδοι.



Εικόνα 34. Range and Hash based partitioning

Κατά την εφαρμογή της range-based partitioning, το σύστημα διαχωρίζει τις τιμές των documents σε εύρος τιμών καθορισμένο από το shard key. Κάθε chunk διατηρεί ταξινομημένα documents και έχει το δικό του εύρος τιμών ορισμένες με min και max. Σε ένα range-based σύστημα, τα documents που υπάρχουν και έχουν τιμές στα πεδία τους που είναι ορισμένες στο εύρος τιμών του αντίστοιχου chunk, λογικά υπάρχουν στο ίδιο κόμβο shard. .Κάθε shard μπορεί να πάρει documents από ένα συγκεκριμένο εύρος τιμών του shard

key. Όταν ένα shard ξεπεράσει ένα ορισμένο όγκο δεδομένων μετακινεί chunks του σε μικρότερα chunks. Για παράδειγμα, ας υποθέσουμε ότι έχουμε ένα collection orders, που κρατάει τις παραγγελίες από ένα σύστημα, και επιλέξουμε να δηλώσουμε για shard key το πεδίο order_id. Αν ο χρήστης ζητήσει να ανακτήσει την παραγγελία με order_id =20 και order_id=125, το πρώτο request θα πάει στο chunk 2, ενώ το δεύτερο request θα δρομολογηθεί στο chunk 3.

Η MongoDB υποστηρίζει και hash-based partitioning (από την έκδοση 2.6). Κατά την εφαρμογή αυτής της τεχνικής, το σύστημα υπολογίζει ένα *hash* από την τιμή ενός πεδίου, και χρησιμοποιεί αυτά τα hashes για να δημιουργήσει chunks. Σε ένα τέτοιο σύστημα, δύο documents που πιθανώς να έχουν κοντινές τιμές δεν συνυπάρχουν στο ίδιο chunk. Έτσι, εξασφαλίζεται μια τυχαία κατανομή ενός collection στο cluster. Χρησιμοποιείται περισσότερο σε πεδία όπου οι τιμές αυξάνονται μονότονα, όπως αν διαλέξουμε για key πεδία τύπου *ObjectID* ή *timestamps*. Τέλος, κατά την εφαρμογή του μηχανισμού sharding, οι σχεδιαστές θα πρέπει να είναι πολύ προσεκτικοί με την επιλογή του κατάλληλου shard key, καθώς μετά την εφαρμογή του sharding το shard key δεν μπορεί να μεταβληθεί.

Κεφάλαιο 4 Μελέτη περίπτωσης blog

4.1 Ανάπτυξη εφαρμογής με MongoDB και nodeJS

NodeJS

Για την αναπαράσταση του τρόπου αποθήκευσης και διαχείρισης των δεδομένων από την MongoDB αναπτύχθηκε ένα προσωπικό ιστολόγιο (blog). Η ανάπτυξη της εφαρμογής έγινε με την χρήση της νέας διαδικτυακής τεχνολογίας nodeJS για την διαχείριση του server-side της εφαρμογής. Η nodeJS είναι μια πλατφόρμα ανάπτυξης λογισμικού διακομιστών και διαδικτυακών εφαρμογών. Η εφαρμογές χτίζονται σε περιβάλλον *JavaScript* και τρέχουν στο περιβάλλον της nodeJS. Προσφέρει μια event-driven αρχιτεκτονική και ένα ασύγχρονο τρόπο επικοινωνίας εισόδου /εξόδου (non-blocking *I/O*) που της παρέχουν απόδοση και έναν εύκολο τρόπο δημιουργίας κλιμακωτών διαδικτυακών εφαρμογών πραγματικού χρόνου (real-time). Η nodeJS χρησιμοποιεί την *V8 GOOGLE JavaSript* μηχανή για την εκτέλεση του κώδικα της και ο κώδικας της είναι βασισμένος στα *callbacks* τα οποία επιτρέπουν ασύγχρονη επικοινωνία με δίκτυα, συστήματα αρχείων, βάσεις δεδομένων κ.ά. Δημιουργήθηκε από τον Ryah Dahl το 2009 και από τότε έχει αναπτύξει μια πολύ μεγάλη κοινότητα.

Παρακάτω δίνεται ένα παράδειγμα ασύγχρονου κώδικα με την χρήση callbacks. Στον πίνακα φαίνεται η εκτέλεση ενός I/O σε βάση δεδομένων που συνήθως χρησιμοποιείται από πολλές διαδικτυακές εφαρμογές.

Blocking I/O

```
// Many web-applications have code like this

var result =db.query(“select * From T*);

    //use result
```

Πίνακας 8. Blocking I/O

Σε πολλές περιπτώσεις η εκτέλεση του παραπάνω κώδικα μπλοκάρει την όλη διαδικασία περιμένοντας από την βάση να επιστρέψει το αποτέλεσμα.

Αντίστοιχα ο παραπάνω κώδικας γραμμένος σε nodeJS με χρήση ασύγχρονου κώδικα δίνεται στον πίνακα #.

Non blocking I/O

```
 query('select ..', function callback(result) {
        //use result

  callback(result);
});
```

Πίνακας 9. Non blocking I/O

Με τον παραπάνω κώδικα επιτυγχάνεται η ασύγχρονη επικοινωνία με την βάση. Η συνάρτηση callback καλείται όταν το query είναι έτοιμο και έτσι δεν μπλοκάρεται η διαδικασία, συνεχίζοντας στις επόμενες γραμμές κώδικα. Μια διεργασία node δεν στηρίζεται στην πολυνηματικότητα (multi-thread) αλλά εκτελείται σε μια event-loop μονονηματικά (single-thread). Αυτό υλοποιείται μέσω των συναρτήσεων callback και η κεντρική ιδέα είναι ότι καμία συνάρτηση δεν πρέπει να κάνει αμέσως I/O [31]. Παρακάτω δίνεται ακόμη ένα παράδειγμα ψευδό κώδικα για την επικοινωνία με τον δίσκο και στις δύο μορφές.

Blocking code

```
  Var  contents  =  fs.readFileSync('/etc/hosts');      //  1.  Read  file
fromsystem
 console.log(contents);                            // 2.  print contents
 console.log('Doing Something else');              // 3. do something else
```

Πίνακας 10. Blocking code

Non blocking code

```
fs.readFile('/etc/hosts'), function(err, contents) {// 1. Read file from
system
console.log(contents);      // 3. whenever you complete(callback), print
contents
});
console.log('Doing Something else');   // 2. do something else
```

Πίνακας 11. non blocking code

Η nodejs χρησιμοποιείται για περιπτώσεις όπως :

- WebSocket Server, like chat servers
- Fast File Upload Client
- Add Servers
- Real time data applications

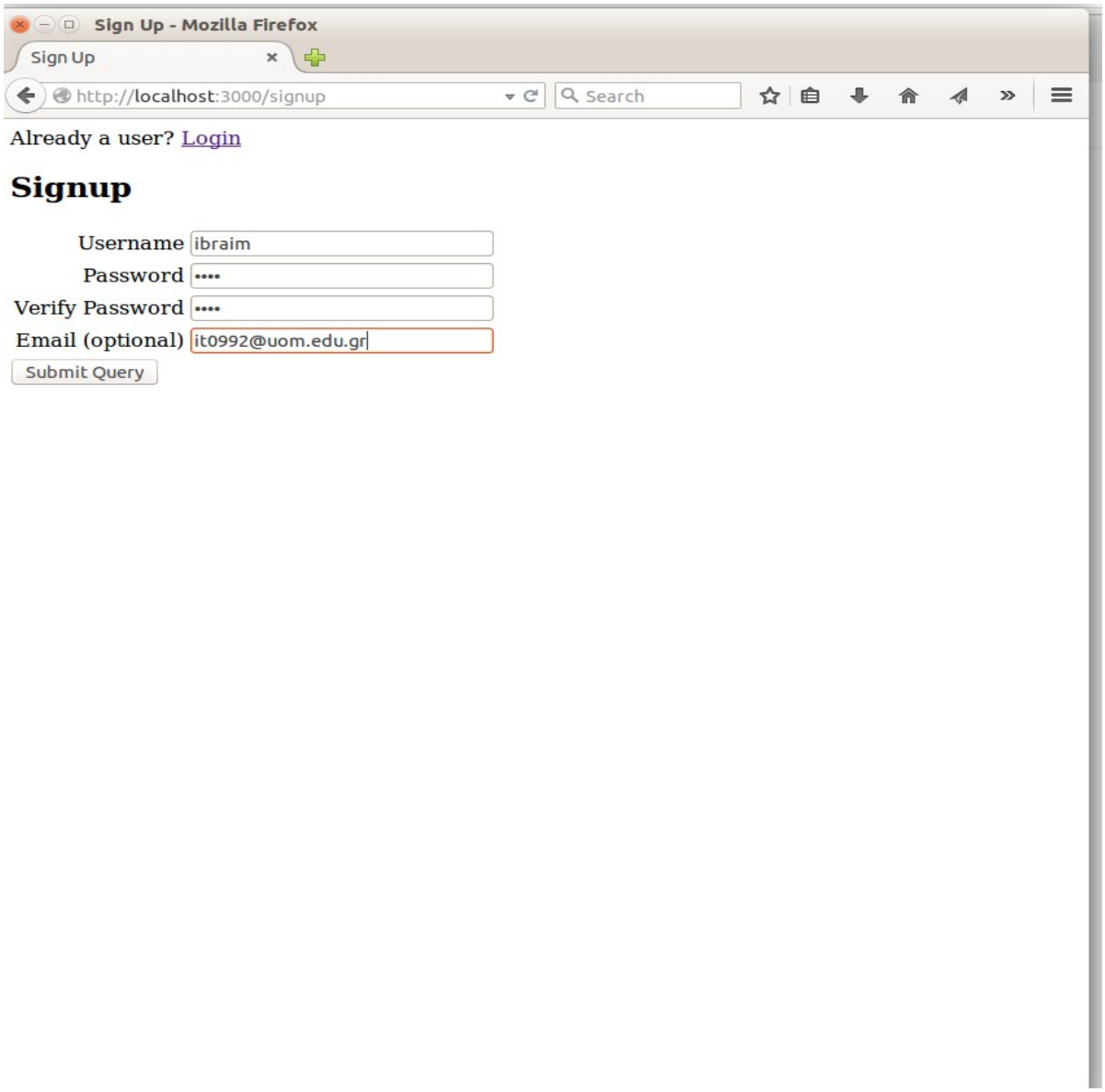
MongoDB

Για της ανάγκες αποθήκευσης και διαχείρισης των δεδομένων του blog χρησιμοποιήθηκε η μη σχεσιακή βάση δεδομένων της MongoDB, η οποία είναι κατάλληλη για την αποθήκευση τέτοιου είδους πληροφορίας. Η MongoDB, η οποία αναλύθηκε διεξοδικά στα παραπάνω κεφάλαια καθώς αποτελούσε το αντικείμενο της πτυχιακής, χρησιμοποιείται περισσότερο σε περιπτώσεις όπως:

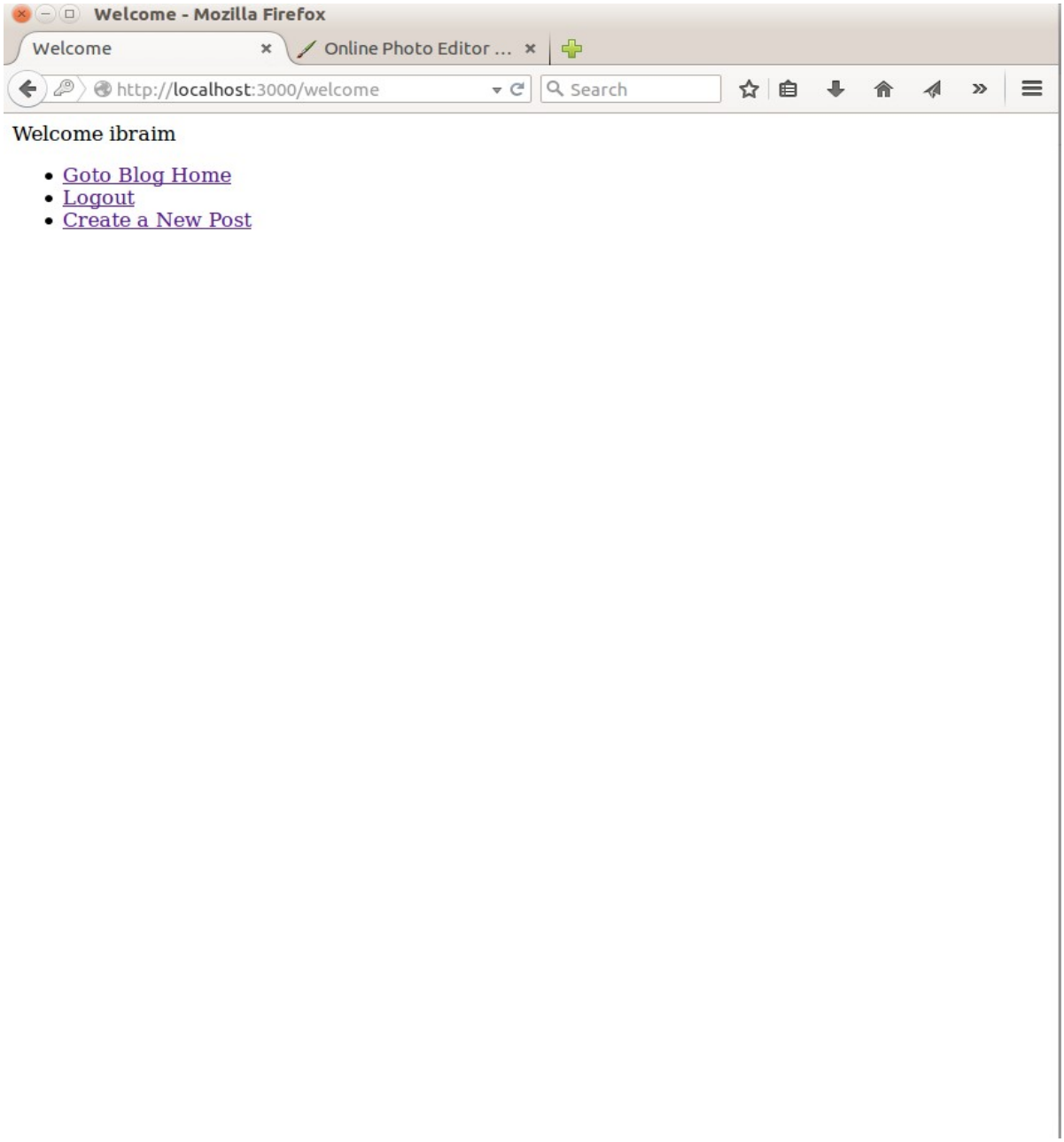
- Storing Log Data
- Pre-Aggregated Reports
- Hierarchical Aggregation
- Product Catalog
- Inventory Management
- Category Hierarchy
- Metadata and asset Management
- Storing comments
- Content management systems

4.2 Περιβάλλον εφαρμογής

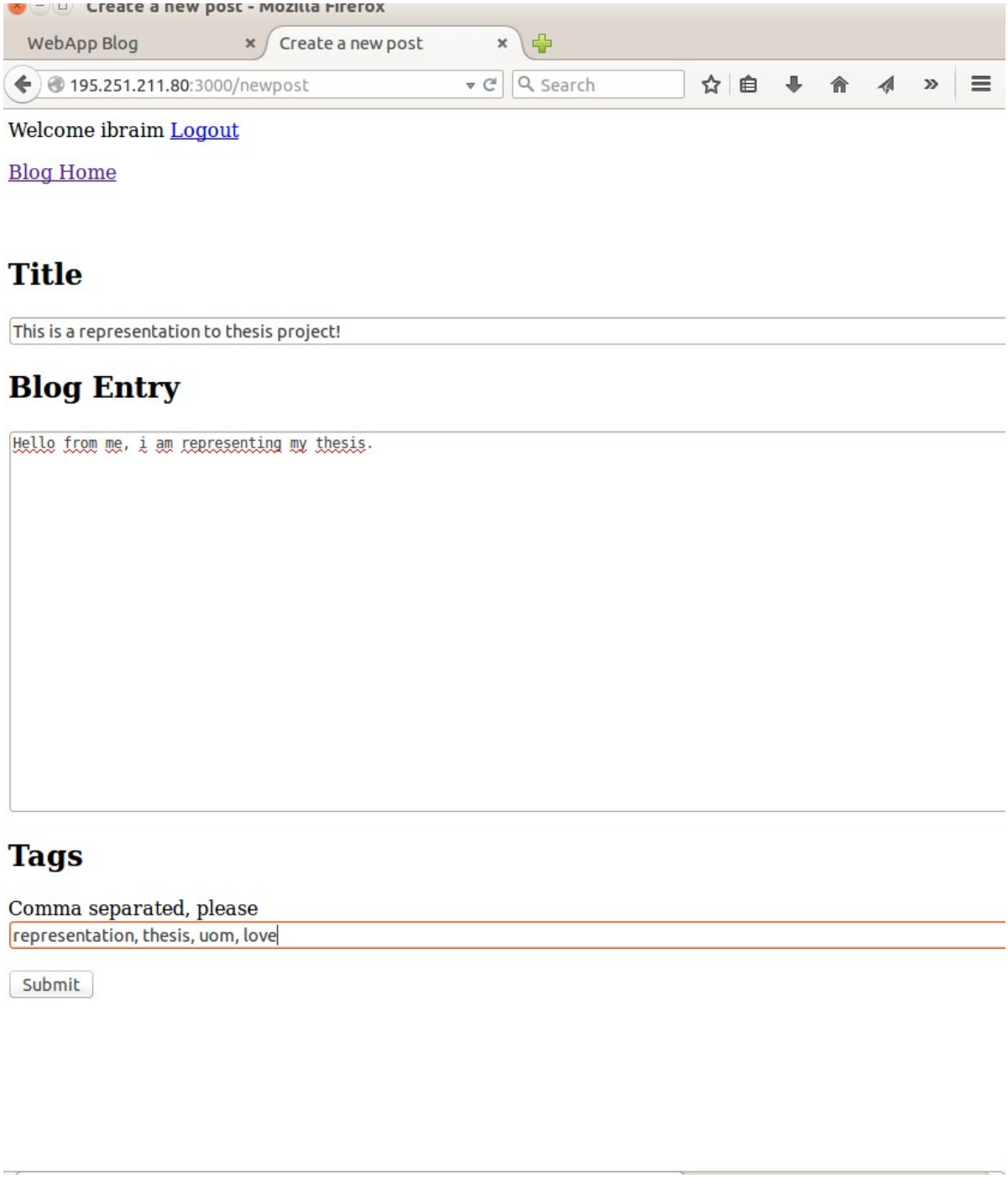
Σε αυτήν την ενότητα θα παρουσιαστεί το περιβάλλον του blog. Το σύστημα επιτρέπει την δημιουργία χρηστών (register) και την δημιουργία ενός post (New Post) εφόσον είναι συνδεδεμένος ο χρήστης. Ένα post αποτελείται από *title*, *body*, *tags* και *comments*. Επίσης υπάρχει η δυνατότητα δημιουργίας σχολίων (comments) από χρήστες είτε εγγεγραμμένους είτε ως επισκέπτες σε κάθε post. Τέλος, δίνεται η δυνατότητα αναζήτησης ενός post με βάση των τίτλο του (search) και η δυνατότητα εμφάνισης των ίδιων post με βάση το tag που επιλέγει ο χρήστης. Παρακάτω παρουσιάζονται οι αντίστοιχες σελίδες του blog με screenshot για κάθε σελίδα.



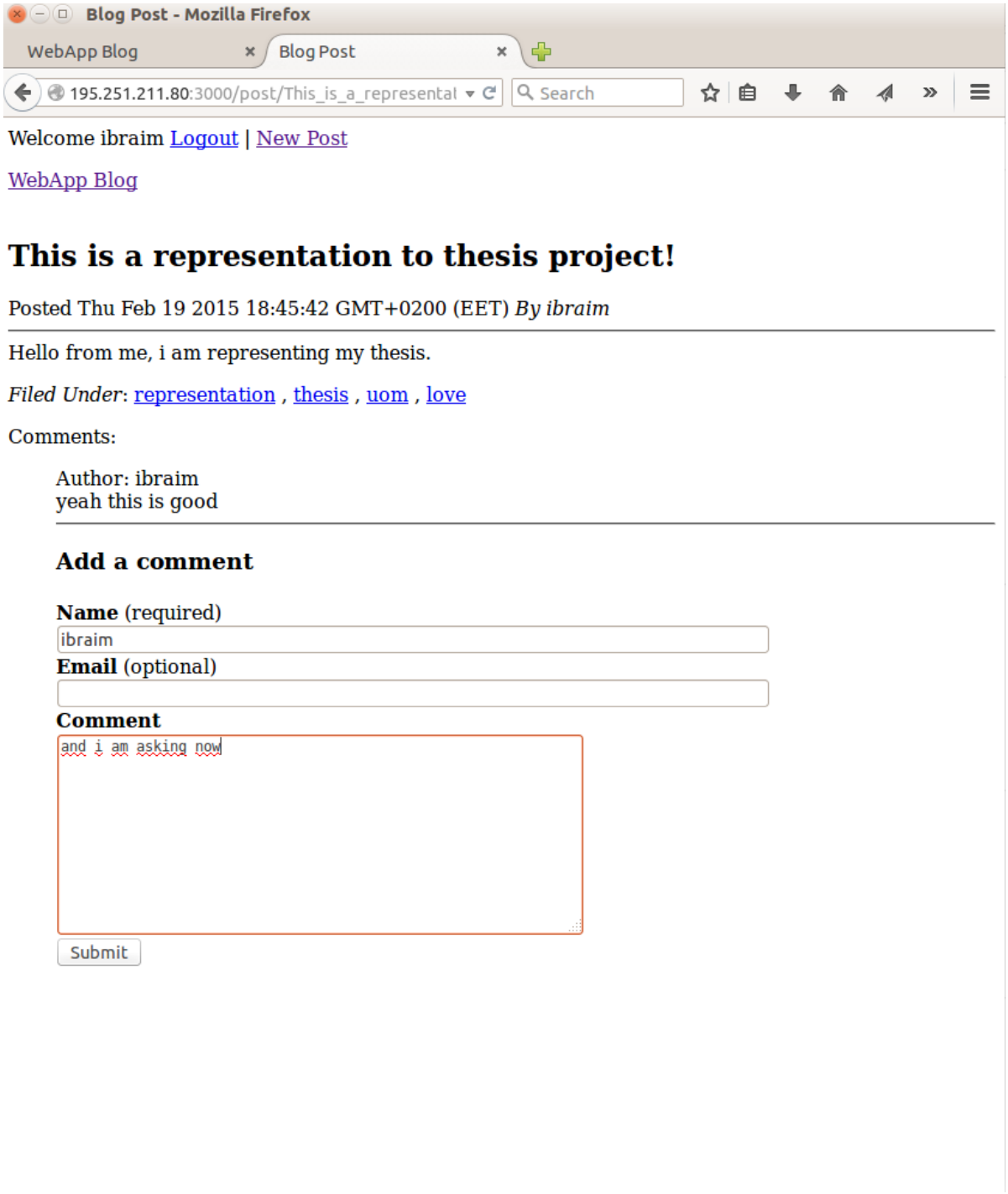
Εικόνα 35. *Δημιουργία χρήστη*



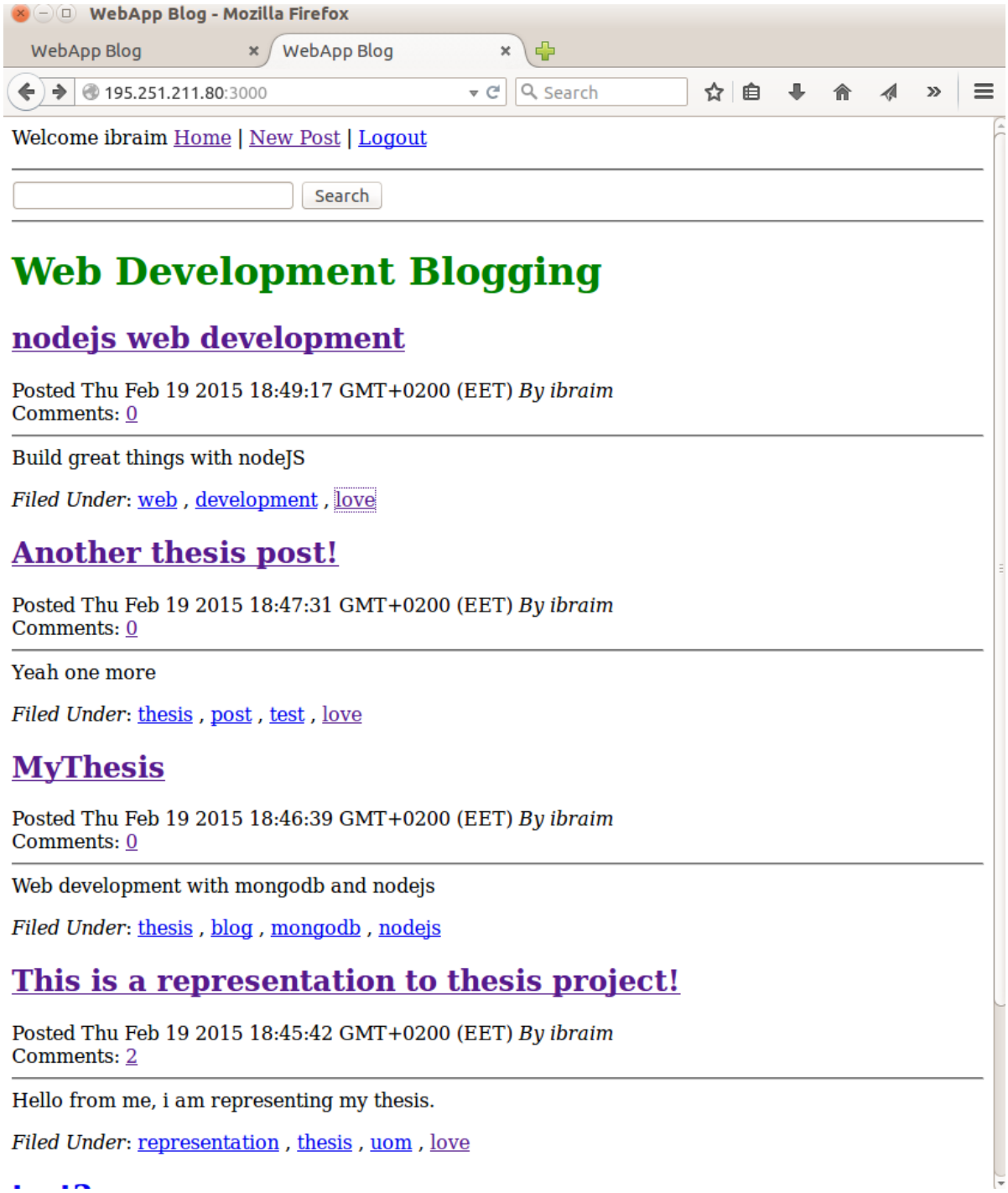
Εικόνα 36. *Σελίδα μετά την επιτυχή δημιουργία χρήστη*



Εικόνα 37. *Σελίδα δημιουργίας νέου post*



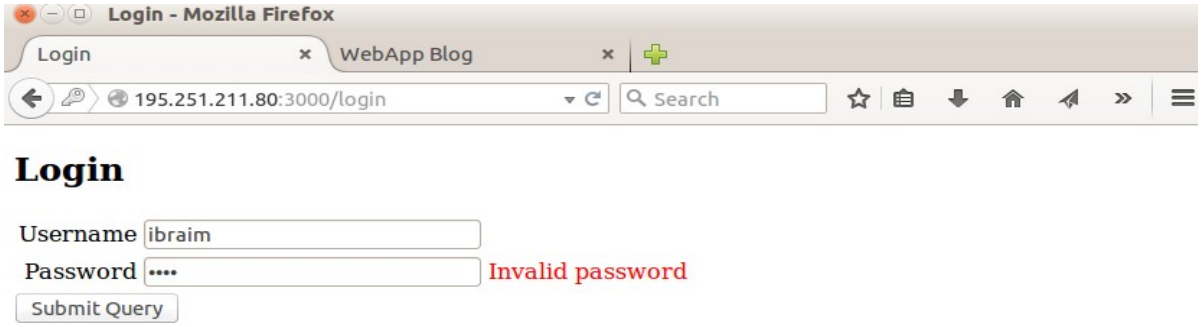
Εικόνα 38. *Δημιουργία comment στο ίδιο post*



Εικόνα 39. *Αρχική σελίδα εφαρμογής*

Εικόνα 40. *Αποτελέσματα εφαρμογής μετά την αναζήτηση του όρου 'nodejs'*

Εικόνα 41. *Αποτελέσματα αναζήτησης εφαρμογής με βάση το tag 'love'*



Εικόνα 42*. Σελίδα σύνδεσης χρήστη, αποτυχία σύνδεσης λόγου λάθους κωδικού*

Βιβλιογραφία

[1] Big Data Analytics: A Literature Review Paper Nada Elgendy and Ahmed Elragal

[2] TechAmerica: Demystifying Big Data: A Practical Guide to Transforming the Business of Government. In: TechAmerica Reports, pp. 1–40 (2012)]

[3] mongodb.com/big-data-explained

[4] mongodb.com/customers/city-of-chicago

[5] Cuzzocrea, A., Song, I., Davis, K.C.: Analytics over Large-Scale Multidimensional Data: The Big Data Revolution! In: Proceedings of the ACM International Workshop on Data Warehousing and OLAP, pp. 101–104 (2011)

[6] docs.mongodb.org/ecosystem/use-cases/hadoop/

[7] mongodb.com/nosql-explained

[8] nosql-database.org/

[9] en.wikipedia.org/wiki/Relational_database_management_system

[10] en.wikipedia.org/wiki/ACID

[11] Data management in cloud environments: NoSQL and NewSQL data stores

[12] en.wikipedia.org/wiki/CAP_theorem

[13] db-engines.com

[14] NoSQL Database: New Era of Databases for Big Data Analytics - Classification, Characteristics and Comparison A B M Moniruzzaman, Syed Akhter Hossain

[15] json.org

[16] http://bsonspec.org/

[17] docs.mongodb.org/manual/core/gridfs/

[18] docs.mongodb.org/manual/reference/database-references/#document-references

[19] docs.mongodb.org/manual/core/data-model-operations/#record-atomicity

[20] docs.mongodb.org/manual/reference/operator/query/

[21] docs.mongodb.org/manual/tutorial/query-documents/

[22] docs.mongodb.org/manual/indexes/

[23] docs.mongodb.org/manual/core/sharded-cluster-components/

[24] docs.mongodb.org/manual/core/sharded-cluster-config-servers/

[25] docs.mongodb.org/manual/core/sharded-cluster-query-router/

[26] docs.mongodb.org/manual/core/storage/

[27] docs.mongodb.org/manual/faq/diagnostics/#faq-memory

[28] docs.mongodb.org/manual/core/journaling/

[29] docs.mongodb.org/manual/core/replication-introduction/

[30] docs.mongodb.org/manual/core/sharding-introduction/

[31] github.com/joyent/node

Παράρτημα Α: Κώδικας υλοποίησης εφαρμογής (blog)

Server file

app.js

```
var express = require ('express')
  , app = express () // Web framework MVC 'sinatra-like' to handle routing requests
  , cons = require ('consolidate') // Templating library adapter for Express
  , MongoClient = require ('mongodb').MongoClient // Driver for connecting to MongoDB
  , routes = require ('./routes'); // Routes for our application


// Connection url: path/port/databaseName
var url = 'mongodb://localhost:27017/blogI';
// Connect to the server
MongoClient.connect (url, function (err, db) {
    "use strict";

    if (err) throw err;

    console.log ("Connection to Mongo sucessfull!")

    // Register our templating engine
    app.engine ('html', cons.swig);
    app.set ('view engine', 'html');
    app.set ('views', __dirname + '/views');

    // Express middleware to populate 'req.cookies' so we can access cookies
    app.use (express.cookieParser ());

    // Express middleware to populate 'req.body' so we can access POST variables
    app.use (express.json ());
    app.use (express.urlencoded ());

    // Application routes
    routes (app, db);

    app.listen (3000);
    console.log ('Express server listening on port 3000 @ UOM Services');
});
```

Πίνακας 12. app.js

Database objects

posts.js

```
/* The PostsDAO must be constructed with a connected database object */
// Post Database Object
function PostsDAO (db) {
   "use strict";

   if (false === (this instanceof PostsDAO)) {
      console.log ('PostsDAO constructor called without "new" operator');
      return new PostsDAO (db);
   }

   //Get from mongo collection posts and assign it to a variable named posts
   var posts = db.collection ("posts");


   // Function to search a user posts with title
   // db.posts.ensureIndex ({'title': 'text'})

   this.searchPost = function (title, searchPosts) {
      "use strict";

      console.log ("Searching for a post with title "  + title + "");

      posts.find ({"$text": {"$search": title }}).sort ('date', -1).toArray (function (err, items)
{
         "use strict";

         if (err) return (err, null);

         console.log ("Searched "  + items.length + " posts with this title");

         searchPosts (err, items);
      });
   }


   this.getPosts = function (skipnum, limitnum, foundPosts) {
      "use strict";

   // Mongo
   // Find all posts, sorted by date DESC, limit by num
   // Indexing in mongo shell, db.posts.ensureIndex ({'date':-1})

      posts.find ().sort ('date', -1).skip (skipnum).limit (limitnum).toArray (function (err,
items) {
         "use strict";

         if (err) return (err, null);
```

```
        console.log ("Found " + items.length + " posts");



        foundPosts (err, items);
    });
}

this.getPostByTitle = function (title, foundPostByTitle) {
    "use strict";

    // Mongo
    // Find one post by Title

    posts.findOne ({'title': title}, function (err, post) {
        "use strict";

        if (err) return callback (err, null);

        console.log ("Found post with the same title");

        foundPostByTitle (err, post);
    });

}

    //Function that inserts a post
this.insertEntry = function (title, body, tags, author, insertedPost) {
    "use strict";
    console.log ("inserting blog entry" + title + body);


    // fix up the permalink to not include whitespace
    var permalink = title.replace ( /\s/g, '_' );
    permalink = permalink.replace ( /\W/g, '' );
    // Build a new post, schema-like for a post
    var post = {"title": title,
            "author": author,
            "body":  body,
            "permalink":  permalink,
            "tags":  tags,
            "comments":  [],
            "date": new Date ()}

    // Now insert post in database
posts.insert (post, function (err, result) {
        "use strict";

        // If err throw err, callback
        if (err) return (err, null);
```

```
        console.log ("Inserted new post");
        //Insert post if no err, and return permalink
        insertedPost (err, permalink);
    });

}


this.getPostsByTag = function (tag, num, foundTags) {
    "use strict";

        // Mongo
        // Find all posts by given tag, sorted by date DESC, limit by num
        // Indexing in mongo shell related to performance
        // db.posts.ensureIndex ( { 'tags':1,'date':-1})
    posts.find ({ tags : tag }).sort ('date', -1).limit (num).toArray (function (err, items) {
        "use strict";

        if (err) return (err, null);

        console.log ("Found " + items.length + " posts");

        foundTags (err, items);
    });
}

this.getPostByPermalink = function (permalink, foundPostByPermalink) {
    "use strict";

    // Mongo
    // Find one post by permalink
    // Indexing in mongo shell, db.posts.ensureIndex ({ permalink : 1 } )
    posts.findOne ({'permalink': permalink}, function (err, post) {
        "use strict";

        if (err) return callback (err, null);

        foundPostByPermalink (err, post);
    });
}

// Function that adds comment, updating the array Comments
this.addComment = function (permalink, name, email, body, addComment) {
    "use strict";

    var comment = {'author': name, 'body': body}

    if (email != "") {
```

```
            comment['email'] = email
        }
            // Mongo
            // Now add the comment using update
            posts.update (
                { 'permalink' : permalink},
13                  { '$push': { 'comments' : comment}},
                function (err, numModified) {

            "use strict";

            if (err) return (err, null);


            addComment (err, numModified);
        });

    }
}

module.exports.PostsDAO = PostsDAO;  // Exporting database object Posts
```

Πίνακας 13. posts.js

users.js

```
var bcrypt = require ('bcrypt-nodejs');

/* The UsersDAO must be constructed with a connected database object */

function UsersDAO (db) {
   "use strict";


   if (false === (this instanceof UsersDAO)) {
      console.log ('Warning: UsersDAO constructor called without "new" operator');
      return new UsersDAO (db);
   }

   //  Get from mongo collection 'users' and assign it to a variable named users
   var users = db.collection ("users");

   //Function that adds a user to database
   this.addUser = function (username, password, email, insertUser) {
      "use strict";

      // Generate password hash
      var salt = bcrypt.genSaltSync ();
      var password_hash = bcrypt.hashSync (password, salt);

      // Create user document, relative schema for a user
      var user = {'_id': username, 'password': password_hash};

      // Add email if set
      if (email != "") {
         user['email'] = email;
      }


         // Mongo
         // Insert user in database
         users.insert ( user, function ( err, result) {
                        "use strict";

                        if (!err) {
                                console.log ("Inserted new user");
                                return insertUser (null, result[0]);
                        }

                        return insertUser (err, null);
                });

   }
```

```
    this.validateLogin = function (username, password, callback) {
        "use strict";

        // Callback to pass to MongoDB that validates a user document
        function validateUserDoc (err, user) {
            "use strict";

            if (err) return callback (err, null);

            if (user) {
                if (bcrypt.compareSync (password, user.password)) {
                    callback (null, user);
                }
                else {
                    var invalid_password_error = new Error ("Invalid password");
                    // Set an extra field so we can distinguish this from a db error
                    invalid_password_error.invalid_password = true;
                    callback (invalid_password_error, null);
                }
            }
            else {
                var no_such_user_error = new Error ("User: " + user + " does not exist");
                // Set an extra field so we can distinguish this from a db error
                no_such_user_error.no_such_user = true;
                callback (no_such_user_error, null);
            }

        }

        // Mongo
        // Find user in database and validate him
        users.findOne ({ '_id' : username }, validateUserDoc);

    }
}

module.exports.UsersDAO = UsersDAO;
```

Πίνακας 14. users.js

sessions.js

```
var crypto = require ('crypto');

/* The SessionsDAO must be constructed with a connected database object */
function SessionsDAO (db) {
    "use strict";


    if (false === (this instanceof SessionsDAO)) {
        console.log ('Warning: SessionsDAO constructor called without "new" operator');
        return new SessionsDAO (db);
    }

    // Get from mongo collection 'sessions' and assign it to a variable named sessions
    var sessions = db.collection ("sessions");

    this.startSession = function (username, startSess) {
        "use strict";

        // Generate session id
        var current_date = (new Date ()).valueOf ().toString ();
        var random = Math.random ().toString ();
        var session_id = crypto.createHash ('sha1').update (current_date + random).digest
('hex');

        // Create session document, schema for session
        var session = {'username': username, '_id': session_id}

            // Mongo
        // Insert session document
        sessions.insert (session, function (err, result) {
            "use strict";
            startSess (err, session_id);
        });
    }

    this.endSession = function (session_id, removeSession) {
        "use strict";
        // Mongo
        // Remove session document
        sessions.remove ({ '_id' : session_id }, function (err) {
            "use strict";
            removeSession (err);
        });
    }
    this.getUsername = function (session_id, getUser) {
        "use strict";

        if (!session_id) {
```

```
            getUser (Error ("Session not set"), null);
            return;
        }

            // Mongo
            // Find one session by session_id
        sessions.findOne ({ '_id' : session_id }, function (err, session) {
            "use strict";

            if (err) return callback (err, null);

            if (!session) {
                getUser (new Error ("Session: " + session + " does not exist"), null);
                return;
            }

            getUser (null, session.username);
        });
    }
}

module.exports.SessionsDAO = SessionsDAO;
```

Πίνακας 15. sessions.js

Routes files

contentHandler.js

```
var PostsDAO = require ('../posts').PostsDAO //Posts database object
  , sanitize = require ('validator').sanitize; // Helper to sanitize form input

/* The ContentHandler must be constructed with a connected db */
function ContentHandler (db) {
    "use strict";

    var posts = new PostsDAO (db);


    this.displayMainPage = function (req, res, next) {
        "use strict";


        posts.getPosts (0, 10, function (err, results) {
            "use strict";

            if (err) return next (err);

            return res.render ('blog_template', {
                title: 'blog homepage',
                username: req.username,
                myposts: results,
            });
        });
    }

    // Under development
    this.displayNextPage = function (req, res, next) {
        "use strict";

        var page = req.params.page;
        var num = 0;

        for (page>1; page<6; page++) {
            num +=  5;
        }

        posts.getPosts (num, 5, function (err, results) {
            "use strict";

            if (err) return next (err);

            return res.render ('blog_template', {
                title: 'blog homepage',
                username: req.username,
                myposts: results,
                pages : page,
```

```
        });
    });
}



this.searchPosts = function (req, res, next) {
    "use strict";

    var title = req.body.term;

    posts.searchPost (title, function (err, results) {
        "use strict";

        if (err) return next (err);

        if (results == 0) return res.redirect ("/post_not_found");


        return res.render ('blog_template', {
            username: req.username,
            myposts: results
        });

    });
}

this.displayMainPageByTag = function (req, res, next) {
    "use strict";

    var tag = req.params.tag;

    posts.getPostsByTag (tag, 10, function found (err, results) {
        "use strict";

        if (err) return next (err);

        return res.render ('blog_template', {
            title: 'blog homepage',
            username: req.username,
            myposts: results
        });
    });
}

this.displayPostByPermalink = function (req, res, next) {
    "use strict";

    var permalink = req.params.permalink;
```

```
        posts.getPostByPermalink (permalink, function (err, post) {
            "use strict";

            if (err) return next (err);

            if (!post) return res.redirect ("/post_not_found");

            // init comment form fields for additional comment
            var comment = {'name': req.username, 'body': "", 'email': ""}

            return res.render ('entry_template', {
                title: 'blog post',
                username: req.username,
                post: post,
                comment: comment,
                errors: ""
            });
        });
    }

    this.handleNewComment = function (req, res, next) {
        "use strict";
        var name = req.body.commentName;
        var email = req.body.commentEmail;
        var body = req.body.commentBody;
        var permalink = req.body.permalink;

        // Put to  the comment the actual user name if found
        if (req.username) {
            name = req.username;
        }

        if (!name || !body) {
            // user did not fill in enough information

            posts.getPostByPermalink (permalink, function (err, post) {
                "use strict";

                if (err) return next (err);

                if (!post) return res.redirect ("/post_not_found");

                // init comment form fields for additional comment
                var comment = {'name': name, 'body': "", 'email': ""}

                var errors = "Post must contain your name and an actual comment."
                return res.render ('entry_template', {
                    title: 'blog post',
                    username: req.username,
                    post: post,
```

```
                comment: comment,
                errors: errors
            });
        });

        return;
    }

    // even if there is no logged in user, we can still post a comment
    posts.addComment (permalink, name, email, body, function (err, updated) {
        "use strict";

        if (err) return next (err);

        if (updated == 0) return res.redirect ("/post_not_found");

        return res.redirect ("/post/" + permalink);
    });
}

this.displayPostNotFound = function (req, res, next) {
    "use strict";
    return res.send ('Sorry, post not found', 404);
}

this.displayNewPostPage = function (req, res, next) {
    "use strict";

    if (!req.username) return res.redirect ("/login");

    return res.render ('newpost_template', {
        subject: "",
        body: "",
        errors: "",
        tags: "",
        username: req.username
    });
}

function extract_tags (tags) {
    "use strict";

    var cleaned = [];

    var tags_array = tags.split (',');

    for (var i = 0; i < tags_array.length; i++) {
        if ( (cleaned.indexOf (tags_array[i]) == -1) && tags_array[i] != "") {
            cleaned.push (tags_array[i].replace (/\s/g,''));
        }
```

```
        }

        return cleaned
    }

    this.handleNewPost = function (req, res, next) {
        "use strict";

        var title = req.body.subject
        var entry = req.body.body
        var tags = req.body.tags

        if (!req.username) return res.redirect ("/signup");

        if (!title || !entry) {
            var errors = "Post must contain a title and blog entry";
            return res.render ("newpost_template", {subject:title, username:req.username,
body:post, tags:tags, errors:errors});
        }

        var tags_array = extract_tags (tags)

        // looks like a good entry, insert it escaped
        var escaped_post = sanitize (entry).escape ();

        // substitute some <br> for the paragraph breaks
        var formatted_post = escaped_post.replace (/\r?\n/g,'<br>');

              //Chech if there is already a post with the same title

       posts.getPostByTitle (title, function (err, post) {
        "use sctrict";

        if (err) return next (err);

        // if no same title  post
        if (!post)      //Insert the post
            posts.insertEntry (title, formatted_post, tags_array, req.username, function (err,
permalink) {
                  "use strict";

                if (err) return next (err);

                      // now redirect to the blog permalink
                      return res.redirect ("/post/" + permalink)
        });
            // else give error
            else {
                  var errors = "Post with title " +title+ " exists, please choose another title for
your post";
```

```
                return res.render ("newpost_template", {subject:"", username:req.username,
body:entry, tags:tags, errors:errors});
            }
        });
    }
}

module.exports = ContentHandler;
```

Πίνακας 16. content.js

sessionHandler.js

```
var UsersDAO = require ('../users').UsersDAO
  , SessionsDAO = require ('../sessions').SessionsDAO;

/* The SessionHandler must be constructed with a connected db */
function SessionHandler (db) {
    "use strict";

    var users = new UsersDAO (db);
    var sessions = new SessionsDAO (db);

    this.isLoggedInMiddleware = function (req, res, next) {
        var session_id = req.cookies.session;
        sessions.getUsername (session_id, function (err, username) {
            "use strict";

            if (!err && username) {
                req.username = username;
            }
            return next ();
        });
    }

    this.displayLoginPage = function (req, res, next) {
        "use strict";
        return res.render ("login", {username:"", password:"", login_error:""})
    }

    this.handleLoginRequest = function (req, res, next) {
        "use strict";

        // Request user input
        var username = req.body.username;
        var password = req.body.password;


        console.log ("user submitted username: " + username + " pass: " + password);


        users.validateLogin (username, password, function (err, user) {
            "use strict";

            if (err) {
                if (err.no_such_user) {
                    return res.render ("login", {username:username, password:"", login_error:"No
such user"});
                }
                else if (err.invalid_password) {
                    return res.render ("login", {username:username, password:"",
```

```
login_error:"Invalid password"});
            }
            else {
                // Some other kind of error
                return next (err);
            }
        }

        sessions.startSession (user['_id'], function (err, session_id) {
            "use strict";

            if (err) return next (err);

            res.cookie ('session', session_id);
            return res.redirect ('/welcome');
        });
    });
}

this.displayLogoutPage = function (req, res, next) {
    "use strict";

    var session_id = req.cookies.session;
    sessions.endSession (session_id, function (err) {
        "use strict";

        // Even if the user wasn't logged in, redirect to home
        res.cookie ('session', '');
        return res.redirect ('/');
    });
}

this.displaySignupPage =  function (req, res, next) {
    "use strict";
    res.render ("signup", {username:"", password:"",
                           password_error:"",
                           email:"", username_error:"", email_error:"",
                           verify_error :""});
}

function validateSignup (username, password, verify, email, errors) {
    "use strict";
    var USER_RE = /^[a-zA-Z0-9_-]{3,20}$/;
    var PASS_RE = /^.{3,20}$/;
    var EMAIL_RE = /^[\S]+@[\S]+\.[\S]+$/;

    errors['username_error'] = "";
    errors['password_error'] = "";
    errors['verify_error'] = "";
    errors['email_error'] = "";
```

```
        if (!USER_RE.test (username)) {
            errors['username_error'] = "invalid username. try just letters and numbers";
            return false;
        }
        if (!PASS_RE.test (password)) {
            errors['password_error'] = "invalid password.";
            return false;
        }
        if (password != verify) {
            errors['verify_error'] = "password must match";
            return false;
        }
        if (email != "") {
            if (!EMAIL_RE.test (email)) {
                errors['email_error'] = "invalid email address";
                return false;
            }
        }
        return true;
    }

    this.handleSignup = function (req, res, next) {
        "use strict";

        var email = req.body.email
        var username = req.body.username
        var password = req.body.password
        var verify = req.body.verify

        // set these up in case we have an error case
        var errors = {'username': username, 'email': email}
        if (validateSignup (username, password, verify, email, errors)) {
            users.addUser (username, password, email, function (err, user) {
                "use strict";

                if (err) {
                    // this was a duplicate error, mongodb error code
                    if (err.code == '11000') {
                        errors['username_error'] = "Username already in use. Please choose
another";
                        return res.render ("signup", errors);
                    }
                    // this was a different error
                    else {
                        return next (err);
                    }
                }

                sessions.startSession (user['_id'], function (err, session_id) {
```

```
                "use strict";

                if (err) return next (err);

                res.cookie ('session', session_id);
                return res.redirect ('/welcome');
            });
        });
    }
    else {
        console.log ("user did not validate");
        return res.render ("signup", errors);
    }
}

this.displayWelcomePage = function (req, res, next) {
    "use strict";

    if (!req.username) {
        console.log ("Welcome : can't identify user...redirecting to signup");
        return res.redirect ("/signup");
    }

    return res.render ("welcome", {'username':req.username})
}
}

module.exports = SessionHandler;
```

Πίνακας 17. session.js

error.js

```
// Error handling middleware

exports.errorHandler = function (err, req, res, next) {
    "use strict";
    console.error (err.message);
    console.error (err.stack);
    res.status (500);
    res.render ('error_template', { error: err });
}
```

Πίνακας 18. error.js

index.js

```
var SessionHandler = require ('./session')
  , ContentHandler = require ('./content')
  , ErrorHandler = require ('./error').errorHandler;

module.exports = exports = function (app, db) {

    // New session and content objects
    var sessionHandler = new SessionHandler (db);
    var contentHandler = new ContentHandler (db);

    // Middleware to see if a user is logged in
    app.use (sessionHandler.isLoggedInMiddleware);

    // The main page of the blog
    app.get ('/', contentHandler.displayMainPage);

    app.get ('/page/:page', contentHandler.displayNextPage);

    // The main page of the blog, filtered by tag
    app.get ('/tag/:tag', contentHandler.displayMainPageByTag);

    // Search for a post
    app.post ('/search', contentHandler.searchPosts);


    // A single post, which can be commented on
    app.get ("/post/:permalink", contentHandler.displayPostByPermalink);
    app.post ('/newcomment', contentHandler.handleNewComment);
    app.get ("/post_not_found", contentHandler.displayPostNotFound);

    // Displays the form allowing a user to add a new post. Only works for logged in users
    app.get ('/newpost', contentHandler.displayNewPostPage);
    app.post ('/newpost', contentHandler.handleNewPost);

    // Login form
    app.get ('/login', sessionHandler.displayLoginPage);
    app.post ('/login', sessionHandler.handleLoginRequest);

    // Logout page
    app.get ('/logout', sessionHandler.displayLogoutPage);

    // Welcome page
    app.get ("/welcome", sessionHandler.displayWelcomePage);

    // Signup form
    app.get ('/signup', sessionHandler.displaySignupPage);
    app.post ('/signup', sessionHandler.handleSignup);
```

```
    // Error handling middleware
    app.use (ErrorHandler);
}
```

Πίνακας 19. index.js

Views

blog_template.html

```
<!DOCTYPE html>
<html>
<head>
<title>WebApp Blog</title>
</head>
<body>
<style type="text/css">
     h1 {text-align: left}
     h1 {color: green}
   </style>


{% if username %}
Welcome {{username}}        <a href="/">Home</a> | <a href="/newpost">New Post</a>
| <a href="/logout">Logout</a><p>
{% else %}
You are not logged in!        <a href="/">Home</a> | <a href="/login">Login</a>
| <a href="/signup">Sign Up</a><p>
{% endif %}

<hr>
<form action="/search" method="POST">
<input type="input" name="term"  autocorrect="off" autocapitalize="off"  >
<input type="submit" value="Search">
<hr>

<h1>Web Development Blogging</h1>


{% for post in myposts %}
<h2><a href="/post/{{post['permalink']}}">{{post['title']}}</a></h2>
Posted {{post['date']}} <i>By {{post['author']}}</i><br>
Comments:
<a href="/post/{{post['permalink']}}">{{post['comments']|length}}</a>
<hr>
{% autoescape false %}
{{post['body']}}
{% endautoescape %}
<p>
<p>
<em>Filed Under</em>:
{% for tag in post.tags %}
   {% if loop.first %}
      <a href="/tag/{{tag}}">{{tag}}</a>
   {% else %}
      , <a href="/tag/{{tag}}">{{tag}}</a>
   {% endif %}
{% endfor %}
```

```
{% endfor %}
<p>
{% for page in pages %}
<h2><a href="/page/{{page}}">{{page}}</a>
{% endfor %}
</body>
</html>
```

Πίνακας 20. blog_template.html

entry_template.html

```
<!doctype HTML>
<html
<head>
<title>
Blog Post
</title>

</head>
<body>
{% if username %}
Welcome {{username}}        <a href="/logout">Logout</a> | <a href="/newpost">New
Post</a><p>
{% else %}
You are not logged in!        <a href="/login">Login</a> | <a href="/signup">Sign
Up</a><p>
{% endif %}
<a href="/">WebApp Blog</a><br><br>

<h2>{{post['title']}}</h2>
Posted {{post['date']}}<i> By {{post['author']}}</i><br>
<hr>
{% autoescape false %}
{{post['body']}}
{% endautoescape %}
<p>
<em>Filed Under</em>:
{% for tag in post.tags %}
   {% if loop.first %}
      <a href="/tag/{{tag}}">{{tag}}</a>
   {% else %}
      , <a href="/tag/{{tag}}">{{tag}}</a>
   {% endif %}
{% endfor %}
<p>
Comments:
<ul>
{% for comment in post.comments %}
Author: {{comment['author']}}<br>
{{comment['body']}}<br>
<hr>
{% endfor %}
<h3>Add a comment</h3>
<form action="/newcomment" method="POST">
<input type="hidden" name="permalink", value="{{post['permalink']}}">
<h4>{{errors}}</h4>
<b>Name</b> (required)<br>
<input type="text" name="commentName" size="60" value="{{comment['name']}}"><br>
<b>Email</b> (optional)<br>
```

```
<input type="text" name="commentEmail" size="60" value="{{comment['email']}}"><br>
<b>Comment</b><br>
<textarea name="commentBody" cols="60"
rows="10">{{comment['body']}}</textarea><br>
<input type="submit" value="Submit">
</ul>
</body>
</html>
```

Πίνακας 21. entry_template.html

error_template.html

```
<!doctype HTML>
<html>
<head>
<title>Internal Error</title>
</head>
<body>

Oops..<br>
{{error}}
</body>
</html>
```

Πίνακας 22. error_template.html

login.html

```
<!DOCTYPE html>

<html>
 <head>
  <title>Login</title>
  <style type="text/css">
   .label {text-align: right}
   .error {color: red}
  </style>

 </head>

 <body>
  <h2>Login</h2>
  <form method="post">
   <table>
    <tr>
     <td class="label">
      Username
     </td>
     <td>
      <input type="text" name="username" value="{{username}}">
     </td>
     <td class="error">
     </td>
    </tr>

    <tr>
     <td class="label">
      Password
     </td>
     <td>
      <input type="password" name="password" value="">
     </td>
     <td class="error">
         {{login_error}}

     </td>
    </tr>

   </table>

   <input type="submit">
  </form>
 </body>

</html>
```

Πίνακας 23. login.html

newpost_template.html

```
<!doctype HTML>
<html>
<head>
<title>Create a new post</title>
<style type="text/css">
    .errors {color: red}
</style>
</head>
<body>

{% if username %}
Welcome {{username}}        <a href="/logout">Logout</a><p>
{% else %}
You are not logged in!        <a href="/login">Login</a> | <a href="/signup">Sign
Up</a><p>
{% endif %}
<a href="/">Blog Home</a><br><br>

<form action="/newpost" method="POST">
<table>
          <tr>
     <td class="errors">
                    {{errors}}
    </td>
  </tr>
</table>
<h2>Title</h2>
<input type="text" name="subject" size="120" value="{{subject}}"><br>
<h2>Blog Entry<h2>
<textarea name="body" cols="120" rows="20">{{body}}</textarea><br>
<h2>Tags</h2>
Comma separated, please<br>
<input type="text" name="tags" size="120" value="{{tags}}"><br>
<p>
<input type="submit" value="Submit">

</body>
</html>
```

Πίνακας 24. newpost_template.html

signup.htlm

```
<!DOCTYPE html>

<html>
 <head>
  <title>Sign Up</title>
  <style type="text/css">
   .label {text-align: right}
   .error {color: red}
  </style>

 </head>

 <body>
  Already a user? <a href="/login">Login</a><p>
  <h2>Signup</h2>
  <form method="post">
   <table>
    <tr>
     <td class="label">
      Username
     </td>
     <td>
      <input type="text" name="username" value="{{username}}">
     </td>
     <td class="error">
         {{username_error}}

     </td>
    </tr>

    <tr>
     <td class="label">
      Password
     </td>
     <td>
      <input type="password" name="password" value="">
     </td>
     <td class="error">
         {{password_error}}

     </td>
    </tr>

    <tr>
     <td class="label">
      Verify Password
     </td>
     <td>
```

```
          <input type="password" name="verify" value="">
        </td>
        <td class="error">
            {{verify_error}}

        </td>
      </tr>

      <tr>
        <td class="label">
          Email (optional)
        </td>
        <td>
          <input type="text" name="email" value="{{email}}">
        </td>
        <td class="error">
            {{email_error}}

        </td>
      </tr>
    </table>

    <input type="submit">
  </form>
 </body>

</html>
```

Πίνακας 25. signup.html

welcome.html

```
<!DOCTYPE html>

<html>
  <head>
    <title>Welcome</title>
    <style type="text/css">
      .label {text-align: right}
      .error {color: red}
    </style>

  </head>

  <body>
    Welcome {{username}}
<p>
<ul>
<li><a href="/">Goto Blog Home</a></li>
<li>
<a href="/logout">Logout</a>
</li>
<li>
<a href="/newpost">Create a New Post</a>
</li>

</body>
</html>
```

Πίνακας 26. welcome.html