ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ (Τ.Ε.Ι.) ΠΑΤΡΑΣ – ΠΑΡΑΡΤΗΜΑ ΑΜΑΛΙΑΔΑΣ

ΣΧΟΛΗ ΔΙΟΙΚΗΣΗΣ ΚΑΙ ΟΙΚΟΝΟΜΙΑΣ

ΤΜΗΜΑ ΕΦΑΡΜΟΓΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΣΤΗ ΔΙΟΙΚΗΣΗ ΚΑΙ ΣΤΗΝ ΟΙΚΟΝΟΜΙΑ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

# ΑΞΙΟΛΟΓΗΣΗ ΤΟΥ HIBERNATE, ΜΙΑΣ ΠΛΑΤΦΟΡΜΑΣ ΛΟΓΙΣΜΙΚΟΥ ΣΥΣΧΕΤΙΣΗΣ ΑΝΤΙΚΕΙΜΕΝΩΝ ΚΑΙ ΣΧΕΣΙΑΚΩΝ ΔΕΔΟΜΕΝΩΝ

# AN EVALUATION OF HIBERNATE, AN OBJECT-RELATIONAL MAPPING PERSISTENCE FRAMEWORK

ΟΝΟΜΑΤΕΠΩΝΥΜΟ ΣΠΟΥΔΑΣΤΗ: ΠΑΠΑΝΙΚΟΛΑΟΥ ΖΗΣΗΣ

ΕΠΟΠΤΕΥΩΝ ΚΑΘΗΓΗΤΗΣ: ΧΟΧΟΛΗΣ ΔΙΟΝΥΣΙΟΣ

ΑΜΑΛΙΑΔΑ 2012

# ΠΙΝΑΚΑΣ ΠΕΡΙΕΧΟΜΕΝΩΝ

ΠΕΡΙΛΗΨΗ ..... iii

ABSTRACT ..... iv

ΕΙΣΑΓΩΓΗ ..... v

ΚΕΦΑΛΑΙΟ 1$^{Ο}$ - ΕΙΣΑΓΩΓΙΚΑ ΚΑΙ ΥΠΟΒΑΘΡΟ ..... 1

1.1 Αμεταβλητότητα Αντικειμένων (Object Persistence) ..... 1

1.2 Σειριοποίηση αντικειμένων Java (Java Object Serialization) ..... 2

1.3 Αντικειμενοστραφής συσχέτιση και το περιεχόμενό της ..... 4

1.3.1 Συσχέτιση ιδιοτήτων των αντικειμένων σε στήλες πίνακα ..... 6

1.3.2 Συσχετίζοντας τη δομή της κληρονομικότητας αντικειμένου ..... 7

1.3.3 Αντιστοίχιση Σχέσης Αντικειμένων ..... 9

1.3.3.1 Τύποι σχέσεων ..... 9

1.3.3.2 Εφαρμογή σχέσεων στην αντικειμενοστρέφεια ..... 11

1.3.3.3 Εφαρμογή σχέσεων στις σχεσιακές βάσεις δεδομένων ..... 12

1.3.3.4 Υλοποίηση αντικειμενοστραφών συσχετίσεων (O/R Mapping ) ..... 14

1.3.3.4.1 JDBC ..... 14

1.3.3.4.2 Πλαίσια αμεταβλητότητας (Persistence Frameworks) ..... 16

1.3.3.4.2.1 Σύγκριση μεταξύ των Enterprise JavaBean (EJB) και Container Managed Persistence (CMP) ..... 18

1.3.3.4.2.2 Hibernate ..... 20

1.4 Πλεονεκτήματα του Hibernate ..... 22

1.5 Αρχιτεκτονική πολλαπλών επιπέδων (Multi-tier Architecture) ..... 24

ΚΕΦΑΛΑΙΟ 2$^{Ο}$ - ΣΥΓΚΡΙΣΗ ΤΩΝ JDBC ΚΑΙ HIBERNATE ..... 25

2.1 Σύγκριση απόδοσης (Performance comparison) ..... 25

2.1.1 JDBC ..... 27

2.1.2 Hibernate ..... 29
2.1.3 Σύγκριση απόδοσης της εντολής Insert (Insert Performance Comparison) ..... 30
2.1.4 Σύγκριση απόδοσης της εντολής Select (Select Performance Comparison) ..... 36
2.1.5 Σύγκριση Απόδοσης της εντολής Διαγραφή (Delete Performance Comparison) .. 40
2.1.6 Διάγραμμα της Βάσης Δεδομένων (Database Diagram) ..... 44
2.1.7 Συμπεράσματα ..... 45
2.2 Σύγκριση ευελιξίας και απλότητας (Flexibility and Simplicity Comparison) ..... 46
2.2.1 JDBC και Hibernate ..... 46
2.2.1.1 Συντήρηση (Maintenance) ..... 46
2.2.1.2 PreparedStatement και ResultSet ..... 48
2.2.1.3 Έλεγχος Συναλλαγής (Transaction Control) ..... 50
2.2.1.4 Συμπέρασμα ..... 51
ΚΕΦΑΛΑΙΟ 3$^{Ο}$ - ΣΥΜΠΕΡΑΣΜΑΤΑ ΚΑΙ ΜΕΛΛΟΝΤΙΚΕΣ ΒΕΛΤΙΩΣΕΙΣ ..... 52
ΑΝΑΦΟΡΕΣ - ΠΗΓΕΣ ..... 54
ΛΙΣΤΑ ΣΥΝΤΟΜΕΥΣΕΩΝ ..... 56
ΛΙΣΤΑ ΓΡΑΦΗΜΑΤΩΝ ..... 57
ΛΙΣΤΑ ΣΧΗΜΑΤΩΝ ..... 58
ΛΙΣΤΑ ΠΙΝΑΚΩΝ ..... 59
ΠΑΡΑΡΤΗΜΑ ..... 60
Hibernate performance test code ..... 60
JDBC performance test code ..... 64
Client Code for comparison Hibernate and JDBC ..... 68
Cache Configuration file for Hibernate ..... 69

# ΠΕΡΙΛΗΨΗ

Σ’ αυτήν την εργασία ασχολούμαστε με ένα σχετικά καινούριο πλαίσιο λογισμικού αμεταβλητότητας, το Hibernate. Το Hibernate είναι μία “open source” πλατφόρμα λογισμικού συσχέτισης αντικειμένων και σχεσιακών δεδομένων. Περιλαμβάνει κάποιες σημαντικές διαφορές από τη JDBC (Java Database Connectivity) και το Enterprise JavaBean. Βασιζόμενοι σε πειράματα σ’ αυτήν την εργασία, έχουμε μια πρόχειρη εκτίμηση απόδοσης μεταξύ JDBC και Hibernate. Μέσω ανάλυσης, συγκρίσεων και δοκιμών, βρήκαμε ότι το Hibernate είναι ένα καλύτερο πλαίσιο λογισμικού αμεταβλητότητας από την εκτέλεση του Enterprise JavaBean ή απευθείας του JDBC (Java Database Connectivity). Βρήκαμε επίσης ότι εισάγοντας το Hibernate στο Enterprise JavaBean είναι μια καλύτερη προσέγγιση για περιβάλλοντα επιχειρησιακών εφαρμογών από την άποψη της περαιτέρω ανάλυσης και εκτελέσεων, το οποίο όχι μόνο κάνει σωστή χρήση των πλεονεκτημάτων του Enterprise JavaBean όπως η διαχείριση συναλλαγών (transaction handling), αλλά επίσης χρησιμοποιεί και πλεονεκτήματα του Hibernate όπως η ευελιξία και η καλή απόδοση.

# ABSTRACT

In this thesis we introduced a new persistence framework, Hibernate. Hibernate is an open source object-relational mapping framework. It includes some important differences from JDBC (Java Database Connectivity) and Enterprise JavaBean. Based on experiments in this thesis, we have a rough performance estimate between JDBC and Hibernate. Through analysis, comparisons and experiments, we found that Hibernate is a better persistence framework that implementing Enterprise JavaBean or straight JDBC (Java Database Connectivity). We also found that introducing Hibernate into Enterprise JavaBean is a better approach for enterprise application environments in terms of further analysis and implementations, which not only makes good use of Enterprise JavaBean advantages such us transaction handling, but also uses Hibernate advantages such us flexibility and good performance.

# ΕΙΣΑΓΩΓΗ

Στις σύγχρονες εφαρμογές, η αμεταβλητότητα δεν είναι ένα απλό θέμα αποθήκευσης και επαναφοράς καταστάσεων. Ένας μηχανισμός αμεταβλητότητας ευθύνεται για την οργάνωση και τη δομή αμετάβλητων δεδομένων. Θα πρέπει να διαχειρίζεται ταυτόχρονη πρόσβαση σ' αυτά τα δεδομένα και να διασφαλίζει την ακεραιότητά τους. Κυρίως, θα έπρεπε να παρέχει ένα μηχανισμό που θα ταξινομεί, θα εξετάζει και θα αθροίζει δεδομένα. Στις περισσότερες επιχειρησιακές εφαρμογές, χρησιμοποιούνται ευρέως οι σχεσιακές βάσεις δεδομένων καθώς οι ευέλικτες και ανθεκτικές σχεσιακές βάσεις δεδομένων μπορούν να ικανοποιήσουν τις παραπάνω απαιτήσεις αμεταβλητότητας.

Η ανάπτυξη λογισμικού έχει γίνει όλο και πιο πολύπλοκη με την πάροδο των χρόνων. Ένας από τους κύριους λόγους που η αντικειμενοστραφής τεχνολογία είναι τόσο δημοφιλής στην ανάπτυξη λογισμικού είναι ότι βοηθάει τον προγραμματιστή να αντιμετωπίσει αυτήν την αυξανόμενη πολυπλοκότητα. Η αντικειμενοστραφής τεχνολογία είναι ισχυρή όταν οντότητες στον επαγγελματικό τομέα μοντελοποιούνται χρησιμοποιώντας καθαρά αντικειμενοστραφείς έννοιες όπως *η κληρονομικότητα, ο πολυμορφισμός και η ενσωμάτωση*.

Δυστυχώς, υπάρχουν τεχνικές δυσκολίες (impedance mismatch) [1] στη συνεργασία μεταξύ τεχνολογίας αντικειμενοστραφών βάσεων δεδομένων και τεχνολογίας σχεσιακών βάσεων δεδομένων. Στο αντικειμενοστραφές παράδειγμα, μετακινούμαστε από αντικείμενο σε αντικείμενο ακολουθώντας σχέσεις οι οποίες υλοποιούνται ως αναφορές σε αντικείμενα. Ενώ στο σχεσιακό παράδειγμα, οι λειτουργίες ορίζονται ταξινομημένες και οι σχέσεις υλοποιούνται χρησιμοποιώντας πρωτεύοντα κλειδιά και δευτερεύοντα κλειδιά ως συμβολικές αναφορές. Αφού οι υποκείμενες τεχνολογίες είναι διαφορετικές, οι δύο τεχνολογίες δε συμβαδίζουν απόλυτα, κάτι το οποίο κάνει τη συνεργασία αντικειμενοστραφούς λογισμικού και σχεσιακής βάσης δεδομένων πολύπλοκη, δυσκίνητη και ακριβή για την ανάπτυξη επιχειρηματικών εφαρμογών.

Φυσικά, μια άλλη προσέγγιση είναι η χρήση μιας αντικειμενοστραφής βάσης δεδομένων (OODBMS), η οποία μειώνει τη δυσκολία της προσπάθειας να

αντιστοιχίσουμε το ένα παράδειγμα με το άλλο. Παρόλο που αυτή η προσέγγιση έχει να προσφέρει πολλά, τα περισσότερα αντικειμενοστραφή συστήματα στην πραγματικότητα χρησιμοποιούν ακόμα σχεσιακές βάσεις δεδομένων. Πιστεύουμε ότι αυτή η κατάσταση θα συνεχιστεί μιας και οι σχεσιακές βάσεις δεδομένων χρησιμοποιούνται ευρέως.

Για να ξεπεραστούν αυτές οι τεχνικές δυσκολίες στη συνεργασία, υπάρχουν αρκετές δημοφιλείς προσεγγίσεις για τα αμετάβλητα δεδομένα στις σχεσιακές βάσεις δεδομένων με Java, όπως η *Java object serialization, η JDBC, η Enterprise JavaBeans - Container Managed Persistence (EJB - CMP)* [10] και τα *Persistence Frameworks*. Οι διαφορετικές αυτές λύσεις έχουν και διαφορετικά πλεονεκτήματα και μειονεκτήματα. Για παράδειγμα, κάποιες έχουν καλή απόδοση ανάπτυξης αλλά μικρή απόδοση εκτέλεσης, ενώ άλλες έχουν καλή απόδοση εκτέλεσης αλλά μικρή απόδοση ανάπτυξης. Είναι πολύ σημαντικό να βρεθεί η κατάλληλη προσέγγιση που να ικανοποιεί τις απαιτήσεις της επιχειρηματικής εφαρμογής όπως η απόδοση, ο εκτελεστικός χειρισμός, η επεκτασιμότητα και η ευελιξία. Για το λόγο αυτό, η ανάλυση και η εύρεση της κατάλληλης λύσης για ένα περιβάλλον επιχειρηματικής εφαρμογής και η υλοποίηση αυτής της λύσης είναι οι στόχοι αυτής της διατριβής.

# ΚΕΦΑΛΑΙΟ 1$^{Ο}$ – ΕΙΣΑΓΩΓΙΚΑ ΚΑΙ ΥΠΟΒΑΘΡΟ

## 1.1 **Αμεταβλητότητα Αντικειμένων** (Object Persistence)

Στην αντικειμενοστρέφεια, ένα αντικείμενο είναι ένα στιγμιότυπο μιας κλάσης. Κάθε αντικείμενο έχει τις ιδιότητές τους και τις μεθόδους του (ή χαρακτηριστικά και συμπεριφορά). Ένα σύνολο αντικειμένων μοντελοποιεί τον επιχειρηματικό τομέα και σχηματίζει το πρότυπο μοντέλο των αντικειμένων της εφαρμογής. Το πρότυπο μοντέλο των αντικειμένων περιλαμβάνει αντικείμενα που εκπροσωπούν τις πρωτεύουσες ιδιότητες και τις διαθέσιμες μεθόδους για την εφαρμογή καθώς αντιπροσωπεύουν έννοιες κατανοητές στον κόσμο. Είναι σύνηθες αυτά τα κύρια αντικείμενα – όπως "Λογαριασμός", "Εταιρία", "Άτομο", "Εργαζόμενος", "Εργασία", κτλ. – να απαιτείται να αποθηκεύονται μεταξύ διαφορετικών εκτελέσεων της επιχειρηματικής εφαρμογής και να χρησιμοποιούνται από κοινού από διαφορετικούς χρήστες. Η χωρητικότητα των αντικειμένων πέρα από τον χρόνο ζωής του Java Virtual Machine (JVM), στον οποίο δημιουργήθηκαν, ονομάζεται *Αμεταβλητότητα Αντικειμένων* (***Object Persistence).*** Η κατάσταση του αντικειμένου μπορεί να αποθηκευτεί στο δίσκο και ένα αντικείμενο με την ίδια κατάσταση μπορεί να δημιουργηθεί ξανά σε κάποια μελλοντική στιγμή. Από την άλλη, κάποια αντικείμενα δεν παραμένουν αμετάβλητα. Ένα παροδικό αντικείμενο έχει περιορισμένο χρόνο ζωής που ορίζεται από την αρχή της ύπαρξης της διαδικασίας που το δημιούργησε. Οι εφαρμογές  Java περιέχουν ένα μείγμα αμετάβλητων και παροδικών αντικειμένων.

## 1.2 **Σειριοποίηση αντικειμένων** Java (Java Object Serialization)

Η σειριοποίηση αντικειμένων Java είναι ένας εύκολος τρόπος αναπαράστασης ενός διαγράμματος διασύνδεσης αντικειμένων. Όταν ταξινομούμε ένα διάγραμμα αντικειμένων, μετατρέπουμε το διάγραμμα σε μια αλληλουχία από byte. Με αυτήν την αλληλουχία μπορούμε μετά να κάνουμε αυτά που θέλουμε, όπως να προωθήσουμε δεδομένα στο δίκτυο (έτσι το Java RMI περνάει παραμέτρους στο δίκτυο), ή μπορούμε να αποθηκεύσουμε την αλληλουχία στο χώρο αποθήκευσης σε μια μορφή όπως ένα σύστημα αρχείου ή μια βάση δεδομένων. Παρόλα αυτά, για εξελιγμένη αμεταβλητότητα, η σειριοποίηση αντικειμένων είναι ανεπαρκής από πολλές απόψεις.

Στην σειριοποίηση αντικειμένων Java (Java Object Serialization), ένα ταξινομημένο διάγραμμα διασυνδεδεμένων αντικειμένων μπορεί μόνο να προσεγγιστεί ως σύνολο. Είναι αδύνατο να ανακτηθεί οποιοδήποτε μέρος από τα δεδομένα της αλληλουχίας χωρίς αποσειριοποίηση της συνολικής αλληλουχίας. Η αλληλουχία byte που προκύπτει θεωρείται μη δομημένης μορφής, ακατάλληλη για αυθαίρετα ερωτήματα ή συσσωμάτωση. Είναι αδύνατον να υπάρξει πρόσβαση ή ενημέρωση ακόμα κι ενός μεμονωμένου αντικειμένου ή υπογραφήματος ανεξάρτητα. Η φόρτωση και η αντικατάσταση ενός ολόκληρου διαγράμματος αντικειμένων σε κάθε συναλλαγή δεν είναι βεβαίως επιλογή για συστήματα που σχεδιάστηκαν να υποστηρίζουν υψηλή σύμπτυξη. Δηλαδή, η σειριοποίηση δεν αρκεί για επιχειρηματικές εφαρμογές υψηλής σύμπτυξης. Είναι η μοναδική επιλογή ως κατάλληλος μηχανισμός αμεταβλητότητας για απλές εφαρμογές.

Γενικά, θέτοντας ένα ερώτημα σε αποθηκευμένα αντικείμενα χρησιμοποιώντας σειριοποίηση αντικειμένων είναι κάτι ακριβό και δυσκίνητο. Η υποβολή αυθαίρετων ερωτημάτων για επιχειρησιακά δεδομένα είναι μια απόλυτη αναγκαιότητα για τις επιχειρηματικές εφαρμογές, γεγονός που κάνει την απλή σειριοποίηση αντικειμένων ακατάλληλη για μόνιμη αποθήκευση. Μέχρι στιγμής, η σειριοποίηση αντικειμένων χρησιμοποιείται καλύτερα σε περιορισμένη έκταση – για επικοινωνίες δικτύων και απλή αμεταβλητότητα. Για παράδειγμα, έστω ότι αποθηκεύουμε χίλια σειριοποιημένα αντικείμενα τραπεζικών λογαριασμών σε μια βάση δεδομένων. Το κάνουμε με σειριοποίηση των αντικειμένων με Java μετατρέποντας τα αντικείμενα στη δυαδική τους αναπαράσταση κι έπειτα αποθηκεύοντας τα bytes στο δίσκο. Τώρα αν χρειαστεί να ανακτήσουμε όλους τους τραπεζικούς λογαριασμού που έχουν υπόλοιπο άνω των 5000€,

τότε πρέπει να φορτώσουμε κάθε σειριοποιημένο τραπεζικό λογαριασμό από το δίσκο, να κατασκευάσουμε το αντίστοιχο αντικείμενο, και μετά να εκτελέσουμε ένα ερώτημα στο αντικείμενο για να προσδιορίσουμε αν το υπόλοιπο είναι άνω των 5000€. Το παραπάνω ερώτημα είναι πολύ απλό αλλά η επεξεργασία είναι δυσκίνητη και αναποτελεσματική. Έτσι, είναι μη πρακτικό να εφαρμόζουμε σειριοποίηση των αντικειμένων με Java για πολύπλοκες επιχειρηματικές εφαρμογές.

## 1.3 Αντικειμενοστραφής συσχέτιση και το περιεχόμενό της

Ένας άλλος δημοφιλής τρόπος αποθήκευσης αντικειμένων Java είναι η χρήση μιας παραδοσιακής σχεσιακής βάσης δεδομένων, όπως η Oracle ή ο Microsoft SQL Server. Αντί για σειριοποίηση κάθε αντικειμένου, αναλύουμε κάθε αντικείμενο στα συστατικά του μέρη και αποθηκεύουμε κάθε κομμάτι ξεχωριστά. Για παράδειγμα, για ένα αντικείμενο τραπεζικού λογαριασμού, το ID του τραπεζικού λογαριασμού μπορεί να αποθηκευτεί σε ένα πεδίο μιας σχεσιακής βάσης δεδομένων και το υπόλοιπο του τραπεζικού λογαριασμού σε ένα άλλο πεδίο. Όταν αποθηκεύουμε το Java αντικείμενο, θα μπορούσαμε να χρησιμοποιήσουμε JDBC/SQL για να αντιστοιχίσουμε το αντικείμενο στη βάση δεδομένων. Θα μπορούσαμε να δημιουργήσουμε ένα αντικείμενο από αυτήν την κλάση, να ανακτήσουμε τα αντίστοιχα δεδομένα από τη βάση δεδομένων, και μετά να συμπληρώσουμε το πεδίο του στιγμιότυπου του αντικειμένου με τα φορτωμένα σχεσιακά δεδομένα.

Αυτή η αντιστοίχιση αντικειμένων σε σχεσιακές βάσεις δεδομένων είναι μια τεχνολογία που ονομάζεται αντικειμενοστραφής αντιστοίχιση (O/R Mapping). Είναι η συμπεριφορά της μετατροπής ενός αντικειμένου αποθηκευμένου στη μνήμη σε σχεσιακά δεδομένα και το αντίστροφο. Μια αντικειμενοστραφής αντιστοίχιση (O/R Mapping) μπορεί να αντιστοιχίζει αντικείμενα σε οποιοδήποτε σχήμα σχεσιακών βάσεων δεδομένων. Μια απλή αντικειμενοστραφής αντιστοίχιση θα αντιστοίχιζε απλώς μια κλάση Java σε έναν SQL ορισμένο πίνακα. Ένα στιγμιότυπο αυτής της κλάσης θα αντιστοιχούσε σε ένα ταξινομημένο σύνολο περιεχομένων στον πίνακα.

Η αντικειμενοστραφής αντιστοίχιση είναι ένας πολύ πιο ισχυρός μηχανισμός για αμετάβλητα αντικείμενα από την απλή σειριοποίηση αντικειμένων. Αναλύοντας τα αντικείμενα Java σαν σχεσιακά αντικείμενα, θα μπορούσαμε να θέσουμε αυθαίρετα ερωτήματα για τις πληροφορίες που θέλουμε. Για παράδειγμα, μπορούμε να αναζητήσουμε σε όλα τα αρχεία βάσεων δεδομένων που έχουν ένα υπόλοιπο λογαριασμού μεγαλύτερο του 5000€ και να φορτώσουμε τα αντικείμενα που ικανοποιούν αυτό το ερώτημα. Πιο σύνθετα ερωτήματα είναι επίσης εφικτά. Αφού δεν είναι σειριοποιημένη, θα μπορούσαμε να ρωτήσουμε τη βάση δεδομένων απευθείας χρησιμοποιώντας SQL, που είναι χρήσιμη για αποσφαλμάτωση ή έλεγχο. Υπάρχουν τρεις εκδοχές σχετικά με το περιεχόμενο της αντικειμενοστραφής αντιστοίχισης: *Αντιστοίχιση*

*χαρακτηριστικών του αντικειμένου σε στήλες Πίνακα, Αντιστοίχιση δομής κληρονομικότητας του αντικειμένου και Αντιστοίχιση σχέσεων αντικειμένων.*

### 1.3.1 Συσχέτιση ιδιοτήτων των αντικειμένων σε στήλες πίνακα

Η εύκολη συσχέτιση είναι μια συσχέτιση μίας μόνο ιδιότητας με μία μόνο στήλη. Είναι ακόμα πιο απλό όταν το κάθε ένα έχει τον ίδιο βασικό τύπο, για παράδειγμα, αν είναι και τα δύο σειριακού τύπου (string), η ιδιότητα είναι string και η στήλη είναι χαρακτήρας (char), ή η ιδιότητα είναι αριθμός (number) και η στήλη είναι δεκαδικός (float).

Είναι απαραίτητο να γνωρίζουμε ότι μια ιδιότητα θα μπορούσε να συσχετίζεται από καμία έως περισσότερες στήλες σε μια σχεσιακή βάση δεδομένων. Επίσης, δεν είναι όλες οι ιδιότητες αμετάβλητες., κάποιες χρησιμοποιούνται για προσωρινούς υπολογισμούς. Για παράδειγμα, ένα αντικείμενο τραπεζικού λογαριασμού μπορεί να έχει μια ιδιότητα ***sumOfBalance*** που είναι απαραίτητη για την εφαρμογή αλλά δε χρειάζεται να αποθηκευθεί στη βάση δεδομένων γιατί υπολογίζεται για κάποιο λόγο στην εφαρμογή. Σε ένα πιο σύνθετο παράδειγμα, κάποιες ιδιότητες ενός αντικειμένου είναι αντικείμενα από μόνα τους, ένα αντικείμενο Άτομο (***Person)*** έχει σαν ιδιότητα ένα αντικείμενο Διεύθυνση (***Address)*** – αυτό πραγματικά αντανακλά μια σχέση ανάμεσα στις δύο κλάσεις που πιθανώς να χρειαζόταν να αντιστοιχηθούν, και οι ιδιότητες αυτής καθεαυτής της κλάσης Διεύθυνση (***Address)*** θα χρειαστούν επίσης να αντιστοιχηθούν.

### 1.3.2 Συσχετίζοντας τη δομή της κληρονομικότητας αντικειμένου

Αφού οι σχεσιακές βάσεις δεδομένων δεν υποστηρίζουν από τη φύση τους την κληρονομικότητα, ο προγραμματιστής αναγκάζεται να συσχετίσει τη δομή της κληρονομικότητας του αντικειμένου με ένα γράφημα αντικειμένου σε ένα γράφημα δεδομένων. Σ' αυτό το κομμάτι, θα συζητήσουμε τρεις βασικές λύσεις για συσχέτιση κληρονομικότητας με μία σχεσιακή βάση δεδομένων.

1. **Συσχέτιση ιεραρχίας σε έναν μόνο πίνακα.** Συσχέτιση μίας ολόκληρης κλάσης ιεραρχίας σε έναν πίνακα, όπου όλες οι ιδιότητες από όλες τις κλάσεις στην ιεραρχία είναι αποθηκευμένες. Το πλεονέκτημα αυτής της προσέγγισης είναι η απλότητα. Η καταγραφή κατά παραγγελία (Ad hoc reporting) είναι επίσης πολύ εύκολη μ' αυτήν την προσέγγιση επειδή όλα τα δεδομένα που χρειάζονται βρίσκονται σε έναν πίνακα. Το μειονέκτημα είναι ότι, αν μια καινούρια ιδιότητα προστίθεται κάθε λίγα λεπτά οπουδήποτε στην ιεραρχία της κλάσης, μια καινούρια ιδιότητα πρέπει να προστεθεί στον πίνακα. Αυτό αυξάνει τη σύζευξη εντός της ιεραρχίας της κλάσης. – Αν γίνει οποιοδήποτε λάθος όταν προστίθεται μία ιδιότητα, θα μπορούσε να επηρεάσει όλες τις κλάσεις στην ιεραρχία και όχι μόνο τις υποκλάσεις της κλάσης που απέκτησε την καινούρια ιδιότητα. Επίσης, σπαταλά πολύ χώρο στη βάση δεδομένων [12] και περιέχει και πολλές μηδενικές τιμές.
2. **Συσχέτιση κάθε συγκεκριμένης - συμπαγούς κλάσης με δικό της πίνακα.** Κάθε πίνακας περιέχει και τις ιδιότητες και τις κληρονομημένες ιδιότητες της κλάσης που εκπροσωπεί. Το κύριο πλεονέκτημα αυτής της προσέγγισης είναι το ότι είναι ακόμα σχετικά εύκολο να γίνει καταγραφή κατά παραγγελία (ad hoc reporting) καθώς όλα τα δεδομένα που χρειάζονται σχετικά με μία μόνο κλάση είναι αποθηκευμένα σε μόνο έναν πίνακα. Παρόλα αυτά, υπάρχουν αρκετά μειονεκτήματα. Όταν τροποποιούμε μια κλάση χρειάζεται να τροποποιήσουμε τον πίνακά της και τον πίνακα κάθε μίας υποκλάσης της, κάτι που κάνει τη συντήρηση της ακεραιότητας των δεδομένων δύσκολη.
3. **Συσχέτιση κάθε κλάσης με δικό της πίνακα.** Δημιουργείται ένας πίνακας ανά κλάση, οι ιδιότητες του οποίου είναι οι ταυτότητες των αντικειμένων (Objects identifiers (OIDs)) και οι ιδιότητες που είναι συγκεκριμένα αυτής της κλάσης. Το κύριο πλεονέκτημα αυτής της προσέγγισης είναι ότι συμμορφώνεται στο μέγιστο

με τις αρχές τις αντικειμενοστρέφειας. Είναι πολύ εύκολο να τροποποιήσουμε υπερκλάσεις και να προσθέσουμε νέες υποκλάσεις καθώς χρειάζεται απλώς να τροποποιήσουμε - προσθέσουμε έναν πίνακα. Υπάρχουν αρκετά μειονεκτήματα σ' αυτήν την προσέγγιση. Πρώτον, υπάρχουν πολλοί πίνακες στη βάση δεδομένων, ένας για κάθε κλάση. Δεύτερον, διαρκεί περισσότερο η ανάγνωση και η εγγραφή δεδομένων χρησιμοποιώντας αυτήν την τεχνική επειδή απαιτείται πρόσβαση σε πολλαπλούς πίνακες. Αυτό το πρόβλημα μπορεί να σμικρυνθεί αν οργανώσουμε έξυπνα τη βάση δεδομένων τοποθετώντας κάθε πίνακα που είναι σε μια κλάση ιεραρχίας σε διαφορετικό φυσικό επίπεδο δίσκου (physical disk-drive platter). Τρίτον, η κατά παραγγελία καταγραφή (Ad hoc reporting) στη βάση δεδομένων είναι δύσκολη, εκτός αν προσθέσουμε προβολές (views) ώστε να προσομοιώσουμε τους επιθυμητούς πίνακες. [12] Τέταρτον, πρέπει να διατηρηθεί αναφορική ακεραιότητα ανάμεσα σε κλάσεις όπως η υπερκλάση και η υποκλάση.

Εν ολίγοις, καμία από τις παραπάνω μεθόδους - στρατηγικές συσχέτισης δεν είναι ιδανική για όλες τις περιπτώσεις. Για να γίνει σωστή χρήση των πλεονεκτημάτων τους, μπορούμε να συνδυάσουμε και τις τρεις μεθόδους. – έναν πίνακα για κάθε ιεραρχία, έναν πίνακα για κάθε συμπαγή - συγκεκριμένη κλάση, και έναν πίνακα για κάθε κλάση – σε κάθε δεδομένη εφαρμογή.

### 1.3.3 Αντιστοίχιση Σχέσης Αντικειμένων

Επιπροσθέτως για την συσχέτιση ιδιοκτησίας και της κληρονομικότητας, χρειάζεται να κατανοήσουμε την τέχνη της αντιστοίχισης σχέσεων. Υπάρχουν τρεις τύποι σχέσεων μεταξύ αντικειμένων: *η πρόσμιξη (aggregation), η σύνθεση (composition) και ο σύνδεσμος (association).* Η πρόσμιξη αντιπροσωπεύει την έννοια ότι ένα αντικείμενο μπορεί να δημιουργείται από άλλα αντικείμενα. Η σύνθεση είναι μια ισχυρότερη μορφή πρόσμιξης, εφαρμοσμένη συνήθως σε αντικείμενα που αντιπροσωπεύουν φυσικά στοιχεία όπως το ότι η μηχανή είναι μέρος ενός αυτοκινήτου. Ο σύνδεσμος χρησιμοποιείται για τη μοντελοποίηση άλλων τύπων σχέσεων μεταξύ αντικειμένων, όπως η σχέση μεταξύ οδηγού και αυτοκινήτου. Αυτοί οι τρεις τύποι σχέσεων μεταξύ αντικειμένων εστιάζουν στους επιχειρηματικούς κανόνες που περιέχουν πιθανές τιμές στη βάση δεδομένων. Έτσι είναι πολύ πιο πιθανό να συγκριθούν με καταστάσεις του πραγματικού κόσμου. Για το λόγο αυτό, από την άποψη της συσχέτισης, θα μπορούσαμε να μεταχειριστούμε με τον ίδιο τρόπο αυτούς τους τρεις τύπους σχέσεων – αντιστοιχίζονται με τον ίδιο τρόπο παρόλο που υπάρχουν μικρές διαφορές σχετικά με την αναφορική ακεραιότητα (referential integrity) [2].

#### 1.3.3.1 Τύποι σχέσεων

Υπάρχουν δύο απόψεις των σχέσεων μεταξύ αντικειμένων που πρέπει να λάβουμε υπόψη. Η πρώτη άποψη βασίζεται στην πολλαπλότητα και περιλαμβάνει τρεις τύπους:

1. **Σχέσεις ένα-προς-ένα.** Αυτή είναι μια σχέση στην οποία κάθε μία από τις πολλαπλότητές της είναι ένα, ένα τέτοιο παράδειγμα είναι η σχέση *έχει* ανάμεσα στο *σχολείο* και το *διευθυντή*. Ένα *σχολείο* έχει ένα και μόνο ένα *διευθυντή* και ένας *διευθυντής* δουλεύει σε ένα *σχολείο*. Οι σχέσεις ένα-προς-ένα είναι σχετικά σπάνιες σε σχεσιακή βάση δεδομένων.

2. **Σχέσεις ένα-προς-πολλά.** Αναφέρεται επίσης ως πολλά-προς-ένα σχέση, συναντάται όταν η μία πολλαπλότητα είναι ένα και η άλλη είναι μεγαλύτερη του ένα. Ένα παράδειγμα είναι η σχέση *σπουδάζει σε* ανάμεσα στο *φοιτητή* και το *σχολείο*. Ένας *φοιτητής* σπουδάζει σε ένα *σχολείο* και οποιοδήποτε *σχολείο* έχει έναν ή περισσότερους *φοιτητές* που σπουδάζουν σ' αυτό.

3. **Σχέσεις πολλά-προς-πολλά.** Αυτή είναι μια σχέση όπου και οι δύο πολλαπλότητες είναι μεγαλύτερες του ένα, ένα παράδειγμα της οποίας είναι η σχέση *ανάθεση* ανάμεσα στο *φοιτητή* και το *Εργασία.* Ο *φοιτητής* έχει ανάθεση μιας ή περισσότερων *εργασιών* και κάθε *εργασία* δίνεται ως ανάθεση σε έναν ή περισσότερους *φοιτητές*.

Η δεύτερη άποψη βασίζεται στην κατεύθυνση και περιέχει δύο τύπους, μονόδρομες σχέσεις και αμφίδρομες σχέσεις.

1. **Μονόδρομες σχέσεις.** Μια μονόδρομη σχέση είναι η σχέση στην οποία ένα αντικείμενο γνωρίζει για το αντικείμενο με το οποίο συνδέεται αλλά το άλλο αντικείμενο δε γνωρίζει για το αρχικό αντικείμενο. Ένα τέτοιο παράδειγμα είναι μια σχέση ανάμεσα σε έναν *ληξίαρχο* και στις *φόρμες υποβολής*. Ο *ληξίαρχος* δε γνωρίζει ποιος έχει *φόρμες υποβολής*, αλλά οι *φόρμες υποβολής* πρέπει να σταλούν πίσω στον *ληξίαρχο*. Δηλαδή, οι *φόρμες υποβολής* πρέπει να γνωρίζουν για τον *ληξίαρχο*.

2. **Αμφίδρομες σχέσεις.** Μια αμφίδρομη σχέση υπάρχει όταν τα αντικείμενα και στα δύο άκρα της σχέσης γνωρίζονται μεταξύ τους, της οποίας παράδειγμα είναι η σχέση *ανάθεση* ανάμεσα στο *φοιτητή* και την *εργασία.* Το αντικείμενο *φοιτητής* ξέρει ποια *εργασία* έχει πάρει ως *ανάθεση* και τα αντικείμενα *εργασίες* γνωρίζουν ποιοί *φοιτητές* τις έχουν αναλάβει.

Μια άποψη για τις τεχνικές δυσκολίες στη συνεργασία (impedance mismatch) ανάμεσα στην τεχνολογία αντικειμένων (object technology) και την τεχνολογία σχέσεων (relational technology) είναι ότι η τεχνολογία σχέσεων εσκεμμένα δεν υποστηρίζει την έννοια των μονόδρομων σχέσεων – στις σχεσιακές βάσεις δεδομένων όλοι οι συσχετισμοί είναι αμφίδρομοι.

#### 1.3.3.2 **Εφαρμογή σχέσεων στην αντικειμενοστρέφεια**

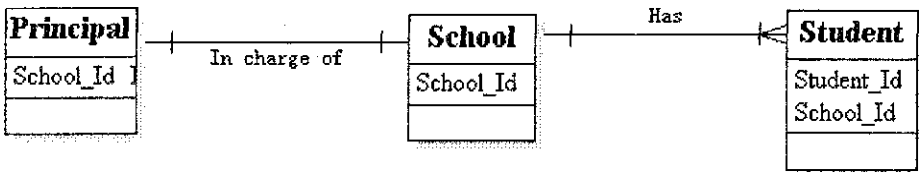
Οι σχέσεις στα σχεδιαγράμματα αντικειμένων υλοποιούνται από ένα συνδυασμό αναφορών στα αντικείμενα και στις λειτουργίες. Όταν η πολλαπλότητα είναι ένα, η σχέση υλοποιείται με μια αναφορά σε ένα αντικείμενο, μία λειτουργία λήψης πληροφοριών (getter operation), και μια λειτουργία εγγραφής πληροφοριών (setter operation). Για παράδειγμα, η σχέση ενός *Φοιτητή* που σπουδάζει σε ένα *σχολείο* υλοποιείται από την κλάση *Φοιτητής* μέσω του συνδυασμού της ιδιότητας *Σχολείο*, της λειτουργίας getΣχολείο() που επιστρέφει την τιμή του *Σχολείο*, και της λειτουργίας setΣχολείο() που γράφει την τιμή της ιδιότητας *Σχολείο*. Οι ιδιότητες και οι λειτουργίες που απαιτούνται για να υλοποιηθεί μια σχέση συχνά αναφέρονται ως "σκαλωσιά" (scaffolding).

Όταν η πολλαπλότητα είναι πολλά, η σχέση υλοποιείται μέσω μιας συλλεκτικής ιδιότητας, όπως μια ***Arraylist*** ή μια ***HashSet*** στη Java, και λειτουργίες χειρισμού αυτής της συλλογής. Για παράδειγμα η κλάση *Σχολείο* υλοποιεί μια ***HashSet*** ιδιότητα με όνομα Φοιτητές, getΦοιτητές() για να πάρει την τιμή, setΦοιτητές() για να γράψει την τιμή, addΦοιτητές() για να προσθέσει ένα άτομο στη HashSet, και removeΦοιτητές() για να αφαιρέσει ένα άτομο από τη HashSet.

Όταν μια σχέση είναι μονόδρομη, ο κώδικας υλοποιείται μόνο από το αντικείμενο που γνωρίζει για το άλλο αντικείμενο. Για παράδειγμα, στη μονόδρομη σχέση ανάμεσα στο *Διευθυντής* και το *Σχολείο*, μόνο η κλάση *Διευθυντής* εφαρμόζει το σύνδεσμο. Αμφίδρομοι σύνδεσμοι, από την άλλη, υλοποιούνται και από τις δύο κλάσεις, όπως η πολλά-προς-πολλά σχέση ανάμεσα στο *Φοιτητή* και την *Εργασία*.

### 1.3.3.3 **Εφαρμογή σχέσεων στις σχεσιακές βάσεις δεδομένων**

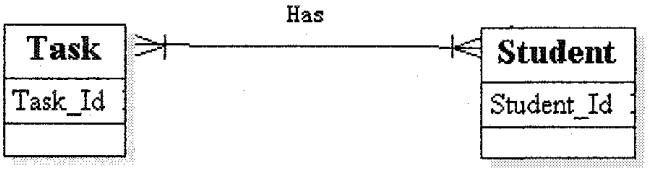
Οι σχέσεις στις σχεσιακές βάσεις δεδομένων διατηρούνται μέσω της χρήσης των ξένων κλειδιών. Ένα ξένο κλειδί είναι ένα χαρακτηριστικό των δεδομένων που υπάρχει σε έναν πίνακα που μπορεί να αποτελεί μέρος ή συμπίπτει με το κλειδί από έναν άλλο πίνακα. Τα ξένα κλειδιά μας επιτρέπουν να συσχετίσουμε ένα αρχείο σε έναν πίνακα με ένα αρχείο σε έναν άλλο. Για την υλοποίηση ένα-προς-ένα και ένα-προς-πολλά σχέσεων πρέπει να συμπεριλάβουμε το κλειδί του ενός πίνακα στον άλλο πίνακα. Ας δούμε ένα παράδειγμα.




Σχήμα 1. Σχέση των Principal, School και Student

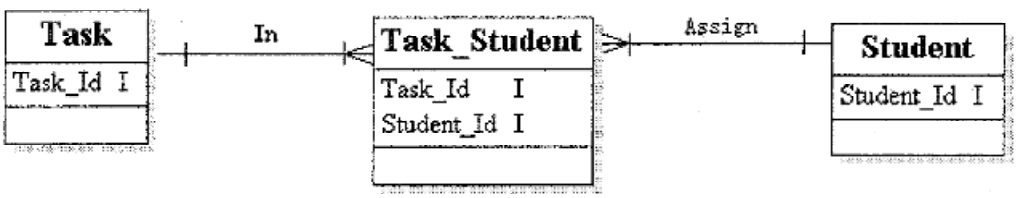
Στο παραπάνω σχήμα, υπάρχουν τρεις πίνακες, τα πρωτεύοντα κλειδιά τους και τα ξένα κλειδιά χρησιμοποιούνται για να εφαρμόσουν τις σχέσεις ανάμεσά τους. Πρώτον, έχουμε μια ένα-προς-ένα σχέση ανάμεσα στο *School* και στο *Principal.* Για να υλοποιήσουμε αυτή τη σχέση, ο *School* και ο *Principal* θα μπορούσαν να χρησιμοποιήσουν το ίδιο πρωτεύον κλειδί *School_Id.* Δεύτερον, υλοποιούμε την πολλά-προς-ένα σχέση (αναφερόμενη επίσης ως ένα-προς-πολλά σχέση) ανάμεσα στο *School* και το *Student.* Το *School* έχει το πρωτεύον κλειδί *School_Id* και το *Student* έχει το πρωτεύον κλειδί *Student_Id.* Θα μπορούσαμε να προσθέσουμε το ξένο κλειδί *School_Id* στο *Student* μιας και ο *Student* ήταν στην πλευρά των πολλών της σχέσης.

Για την υλοποίηση πολλά-προς-πολλά σχέσεων χρειάζεται να εισάγουμε την έννοια ενός συνδετικού πίνακα – ένας πίνακας του οποίου ο σκοπός είναι να διατηρήσει τη σχέση μεταξύ δύο ή περισσότερων πινάκων σε μια σχεσιακή βάση δεδομένων. Στο Σχήμα 2, υπάρχει μια πολλά-προς-πολλά σχέση ανάμεσα στο *Task* και το *Student.* Στο Σχήμα 3, θα μπορούσαμε να δούμε πώς να χρησιμοποιήσουμε ένα συνδετικό πίνακα για να υλοποιήσουμε μια πολλά-προς-πολλά σχέση. Στις σχεσιακές βάσεις δεδομένων, οι ιδιότητες που περιλαμβάνονται σε έναν συνδετικό πίνακα είναι συνήθως ο συνδυασμός των κλειδιών των πινάκων που συμμετέχουν στη σχέση.




Σχήμα 2. Πολλά-προς-πολλά σχέση ανάμεσα στα Task και Student

Το πλεονέκτημα αυτής της προσέγγισης είναι ότι όλοι οι πίνακες αντιμετωπίζονται το ίδιο από το επίπεδο αμεταβλητότητας (persistence layer), απλοποιώντας την υλοποίησή του. Επιπλέον, υπάρχει επίσης η δυνατότητα μία ή περισσότερες στήλες να προστεθούν στο συνδετικό πίνακα. Για παράδειγμα, μια στήλη security στο *Task_Student* αντιπροσωπεύει τα δικαιώματα πρόσβασης ασφαλείας για κάποιες προδιαγραφές. [12]




Σχήμα 3. Εκτέλεση μιας σχέσης πολλά-προς-πολλά σε μια σχεσιακή βάση δεδομένων

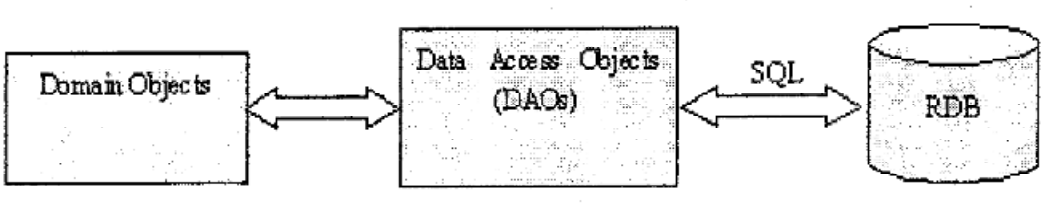
#### 1.3.3.4 **Υλοποίηση αντικειμενοστραφών συσχετίσεων** (O/R Mapping )

Μόλις οριστεί μια συσχέτιση πρέπει να την υλοποιήσουμε. Υπάρχουν δύο κύριες στρατηγικές για να γίνει αυτό, που ονομάζονται *JDBC* και *Πλαίσια Αμεταβλητότητας (Persistence Frameworks).*

##### 1.3.3.4.1 JDBC

Η Java Database Connectivity (JDBC) παρέχει τους προγραμματιστές Java μια πρότυπη επιφάνεια διεπαφής εφαρμογών (API) που χρησιμοποιείται για πρόσβαση σε σχεσιακές βάσεις δεδομένων, ανεξάρτητα από τον οδηγό (driver) και το προϊόν της βάσης δεδομένων. Το JDBC standard ορίστηκε από την Sun Microsystems, επιτρέποντας σε μεμονωμένους προγραμματιστές να υλοποιήσουν και να επεκτείνουν το πρότυπο με τους δικούς τους JDBC drivers. Η JDBC API παρέχει στις εφαρμογές Java υψηλού επιπέδου πρόσβαση στα περισσότερα συστήματα βάσεων δεδομένων, μέσω της Δομημένης Γλώσσας Αναζήτησης (Structured Query Language), SQL.

Η χρήση JDBC είναι ένας απευθείας και εύκολος τρόπος υλοποίησης αμεταβλητότητας αντικειμένων. Η JDBC είναι μάλλον η πιο συχνή προσέγγιση επειδή είναι απλή και δίνει στους προγραμματιστές πλήρη έλεγχο σχετικά με το πώς το επιχειρησιακό τους αντικείμενο θα αλληλεπιδράσει με τη βάση δεδομένων. Εξαιτίας της απλότητάς της αυτή είναι μια καλή προσέγγιση για να έχουμε στην αρχή μιας εργασίας (project) όταν οι απαιτήσεις για πρόσβαση στη βάση δεδομένων είναι αρκετά απλές. Παρόλα αυτά, όταν οι ανάγκες πρόσβασης στη βάση δεδομένων γίνονται πιο πολύπλοκες, αυτή η κωδικοποιημένη με το χέρι JDBC δεν μπορεί να δουλέψει σωστά γιατί η SQL περιπλέκεται.. Τότε, η JDBC εισάγει το πρότυπο Αντικείμενα Πρόσβασης Βάσεων Δεδομένων (Data Access Objects (DAO) Pattern). Το DAO είναι ένα από τα J2EE πρότυπα Πυρήνα (Core Patterns) [6]. Το DAO διαχειρίζεται τη σύνδεση με την πηγή των δεδομένων για να αποκτήσει, να αποθηκεύσει δεδομένα και να ενσωματώσει τη λογική πρόσβασης στη βάση δεδομένων που απαιτείται από τα επιχειρησιακά αντικείμενα. Έτσι οι επιχειρησιακές κλάσεις δεν είναι απευθείας συνδεδεμένες με τη βάση δεδομένων. Τα επιχειρησιακά αντικείμενα που χρησιμοποιούν τις DAO κλάσεις δε χρειάζεται να γνωρίζουν για καμία SQL δήλωση ή τα χαμηλού επιπέδου APIs που προέρχονται από τη βάση δεδομένων.



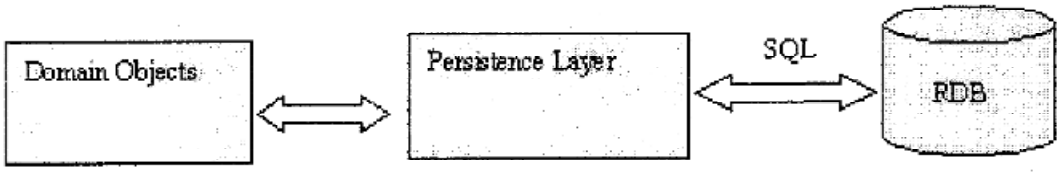

Σχήμα 4. Διάγραμμα DAO και JDBC

Παρόλα αυτά, αυτή η λύση του γραμμένου με το χέρι κώδικα, η γραφή SQL και τα στενά συνδεδεμένα DAO αντικείμενα σε μια μεγάλη εφαρμογή, μπορεί να είναι δύσκολη να συντηρηθεί, ειδικά όταν υποστηρίζονται πολλαπλές βάσεις δεδομένων. Αυτή η δουλειά συνήθως καταλήγει στην απορρόφηση μεγάλου μέρους της προσπάθειας ανάπτυξης κώδικα. Ευτυχώς, όταν συμβαίνει αλλαγή των προδιαγραφών ή των βαθμίδων της βάσης δεδομένων, αυτή η λύση του γραμμένου με το χέρι κώδικα, πάντα απαιτεί περισσότερη προσοχή και προσπάθεια συντήρησης. Για παράδειγμα, όταν αλλάζουν οι προδιαγραφές και απαιτείται αλλαγή των ονομάτων των πινάκων ή των στηλών, τότε κάθε SQL δήλωση που περιλαμβάνει τα ονόματα των πινάκων ή των στηλών πρέπει να τροποποιηθεί. Αν η στήλη και ο πίνακας εμφανίζονται συχνά στις δηλώσεις SQL, η συντήρηση θα γίνει δύσκολη. Επίσης, δε θα μπορούσαμε να ελέγξουμε αν οι τροποποιημένες δηλώσεις SQL είναι σωστές μιας και ο μεταγλωττιστής (compiler) Java δεν μπορεί να ελέγξει δηλώσεις SQL, κάτι που θα προκαλέσει πιθανά προβλήματα στον έλεγχο.

#### 1.3.3.4.2 **Πλαίσια αμεταβλητότητας** (Persistence Frameworks)

Ένα πλαίσιο αμεταβλητότητας, αναφερόμενο συχνά και ως επίπεδο αμεταβλητότητας, ενσωματώνει πλήρως πρόσβαση στη βάση δεδομένων για τα επιχειρησιακά αντικείμενα. Αντί να γράψουμε απευθείας κώδικα για να υλοποιήσουμε τη λογική που απαιτείται για την πρόσβαση στη βάση δεδομένων, ορίζουμε απλά meta δεδομένα (meta data) που αντιπροσωπεύουν την αντιστοίχιση. Έτσι, αν η κλάση Άτομο αντιστοιχεί στον πίνακα Άτομα, τα meta δεδομένα θα αντιπροσώπευαν αυτήν την αντιστοίχιση. Τα meta δεδομένα αντιπροσωπεύουν τις αντιστοιχίσεις όλων των επιχειρησιακών αντικειμένων και τους συνδέσμους μεταξύ τους. Βασισμένο σε αυτά τα meta δεδομένα, το πλαίσιο αμεταβλητότητας (persistence framework) θα δημιουργούσε τον κώδικα πρόσβασης στη βάση δεδομένων (SQL) που είναι απαραίτητος για να γίνουν τα επιχειρησιακά αντικείμενα αμετάβλητα. Αυτός ο κώδικας είτε δημιουργείται δυναμικά κατά την εκτέλεση είτε μπορεί να δημιουργηθεί στατικά με τη μορφή αντικειμένων πρόσβασης δεδομένων που εντάσσονται μετά στην εφαρμογή. Η πρώτη προσέγγιση δίνει μεγαλύτερη ευελιξία ενώ η δεύτερη έχει μεγαλύτερη απόδοση. [13]

Ένα πλαίσιο αμεταβλητότητας έχει κάποια βασικά χαρακτηριστικά. Για παράδειγμα, υποστηρίζει βασικές λειτουργίες για αντικείμενα, όπως δημιουργία, ανάγνωση, ενημέρωση, διαγραφή (CRUD functionalities) όπως και βασικό έλεγχο συναλλαγής και συγχρονισμού [3]. Το πλαίσιο αμεταβλητότητας διαβάζει τα meta δεδομένα αντιστοίχισης στη μνήμη και δημιουργεί από αυτά μια συλλογή αντικειμένων αντιστοίχισης (map objects). Χρησιμοποιεί αυτά τα αντικείμενα αντιστοίχισης για τη δημιουργία του κώδικα που χρειάζεται για πρόσβαση στη βάση δεδομένων. Αυτά τα αντικείμενα αντιστοίχισης συχνά συγκροτούν μέσα στο πλαίσιο αμεταβλητότητας, αυτό που αναφέρεται ως Μηχανή Παραγωγής SQL (SQL Generation Engine). Στην περίπτωση της δυναμικής παραγωγής το πλαίσιο αμεταβλητότητας βάζει τα αντικείμενα αντιστοίχισης στη μνήμη cache και συνεργάζεται μαζί τους κάθε φορά που υπάρχει πρόσβαση στη βάση δεδομένων. Για στατική παραγωγή αντικειμένων απαραίτητων για την εισαγωγή δεδομένων, το πλαίσιο αμεταβλητότητας απαιτεί μόνο τα αντικείμενα αντιστοίχισης τη στιγμή παραγωγής κώδικα παρόλο που προφανώς θα τα χρειαστεί κατά την εκτέλεση της εφαρμογής. [13]




Σχήμα 5. Διάγραμμα Βαθμίδας Αμεταβλητότητας

Το Σχήμα 5 δείχνει τη βαθμίδα αμεταβλητότητας (persistence layer) που συσχετίζει αντικείμενα στις σχεσιακές βάσεις δεδομένων. Σ' αυτήν την προσέγγιση, απλές αλλαγές στο σχεσιακό σχήμα δεν επηρεάζουν τον αντικειμενοστραφή κώδικα. Το πλεονέκτημα αυτής της προσέγγισης είναι ότι οι προγραμματιστές των εφαρμογών δε χρειάζεται να γνωρίζουν κάτι για το σχήμα της σχεσιακής βάσης δεδομένων. Δίνει τη δυνατότητα στους προγραμματιστές Java να χρησιμοποιήσουν αμετάβλητα αντικείμενα Java σε μία αποθήκη δεδομένων που εκτελεί συναλλαγές χωρίς την ανάγκη ρητής διαχείρισης της αποθήκευσης και επαναφοράς μεμονωμένων πεδίων. Στην πραγματικότητα, δε γνωρίζουν καν ότι τα αντικείμενά τους αποθηκεύονται σε μια σχεσιακή βάση δεδομένων. Θα παρουσιάσουμε δύο είδη βαθμίδων αμεταβλητότητας: *την Enterprise JavaBean* και *το Hibernate.*

#### 1.3.3.4.2.1 **Σύγκριση μεταξύ των** Enterprise JavaBean (EJB) **και** Container Managed Persistence (CMP)

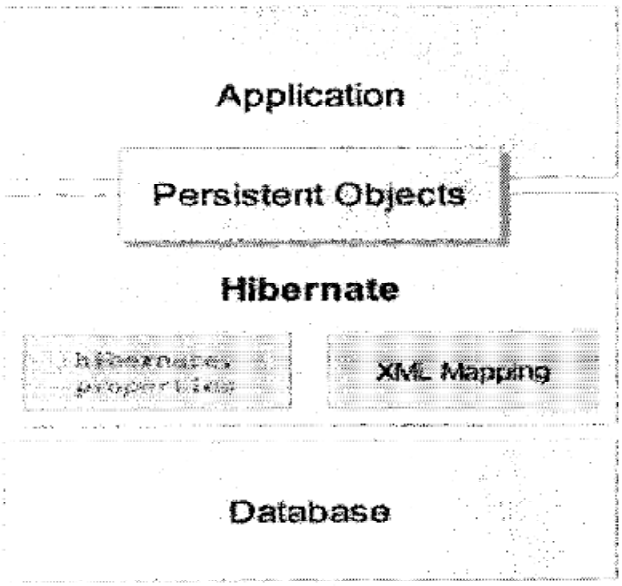
Η Enterprise JavaBean (EJB) είναι μια server-side αρχιτεκτονική συστατικών (component architecture) που απλοποιεί τη διαδικασία ανάπτυξης μίας επιχειρησιακής κλάσης κατανεμημένων συστατικών εφαρμογών (component applications) σε Java. Με τη χρήση της, ένας προγραμματιστής θα μπορούσε να γράψει κλιμακωτές, αξιόπιστες και ασφαλείς εφαρμογές χωρίς τη γραφή πολύπλοκων κατανεμημένων πλαισίων συστατικών (component frameworks). Ο προγραμματιστής μπορεί να συγκεντρωθεί στον προγραμματισμό της επιχειρηματικής λογικής. Για παράδειγμα, ο προγραμματιστής της επιχείρησης δε χρειάζεται πλέον να αναπτύξει κώδικα που χειρίζεται *συναλλαγματική συμπεριφορά (transactiona behavior), ασφάλεια (security), συγκέντρωση συνδέσεων (connection pooling), πρόσβαση από μακριά (remotability), ή ροές εκτέλεσης (threading),* επειδή η αρχιτεκτονική αναθέτει αυτή τη δουλειά στον πωλητή του Διακομιστή Εφαρμογών (Application Server).

Ένα από τα κύρια οφέλη της EJB είναι η δυνατότητα δημιουργίας αντικειμένων bean (entity beans). Τα αντικείμενα bean είναι αμετάβλητα αντικείμενα που μπορούν να αποθηκευθούν σε μόνιμη αποθήκη - μνήμη. Ο προγραμματιστής μπορεί να μοντελοποιήσει τα βασικά και θεμελιώδη δεδομένα της επιχείρησης ως αντικείμενα bean. Η Container Managed Persistence (CMP) είναι το μέρος του μοντέλου συστατικών J2EE που παρέχει υποστήριξη της αμεταβλητότητας των αντικειμένων για EJB περιέκτες (Containers). Χρησιμοποιώντας Container Managed Persistence (CMP), ο EJB περιέκτης ευθύνεται για την αποθήκευση της κατάστασης του αντικειμένου bean. Δηλαδή, ο προγραμματιστής δεν απαιτείται να εφαρμόζει στο αντικείμενο bean καμία λογική αμεταβλητότητας (όπως JDBC/SQL), και ο EJB περιέκτης εκτελεί λειτουργίες αποθήκευσης για τον προγραμματιστή. Επειδή είναι διαχειριζόμενη από τον περιέκτη, η εκτέλεση είναι ανεξάρτητη από την πηγή των δεδομένων. Ο σκοπός της CMP είναι να παρέχει έναν πρότυπο μηχανισμό για την εφαρμογή αμετάβλητων επιχειρησιακών συστατικών. Η CMP παρέχει κατανεμημένη, συναλλαγματική, ασφαλή πρόσβαση σε αμετάβλητα δεδομένα, με ένα εγγυημένο φορητό interface. Βασίζεται σε ένα λειτουργικό, παροχής/λήψης μοντέλο πρόσβασης δεδομένων. Αλλά δεν υποστηρίζει αδιάβλητη αμεταβλητότητα στιγμιαίας μεταβλητής Java, επειδή η CMP πρέπει να εφαρμόσει διαφορετικό interface; όπως τα Home Interface, Remote Interface. Εδώ, δε συζητάμε για ένα άλλου είδους αντικείμενο bean: Bean Managed Persistence (EJB - BMP), επειδή

χρησιμοποιώντας τη Bean Managed Persistence (BMP), οι προγραμματιστές πρέπει να γράψουν τη λογική αμεταβλητότητας. Το Αντικείμενο Bean ευθύνεται απόλυτα για την αποθήκευση της δικής του κατάστασης χρησιμοποιώντας JDBC/SQL και ο περιέκτης (Container) δε χρειάζεται να κάνει καμία κλήση προς τη βάση δεδομένων. Σ' αυτήν την περίπτωση, η αμεταβλητότητα πρέπει να είναι ενσωματωμένη (hard-coded) στο αντικείμενο bean μέσω του εκτελεστή bean. Η BMP διαφέρει από το Πλαίσιο Αμεταβλητότητας που συζητήθηκε σ' αυτήν την εργασία. Επίσης, η απόδοση της BMP είναι χειρότερη από αυτήν της CMP [11]. Για το λόγο αυτό, δε θα συζητηθεί η BMP σ' αυτή την εργασία. Να σημειωθεί ότι το αντικείμενο bean που συζητείται παρακάτω σημαίνει μόνο container managed persistence (CMP).

#### 1.3.3.4.2.2 Hibernate

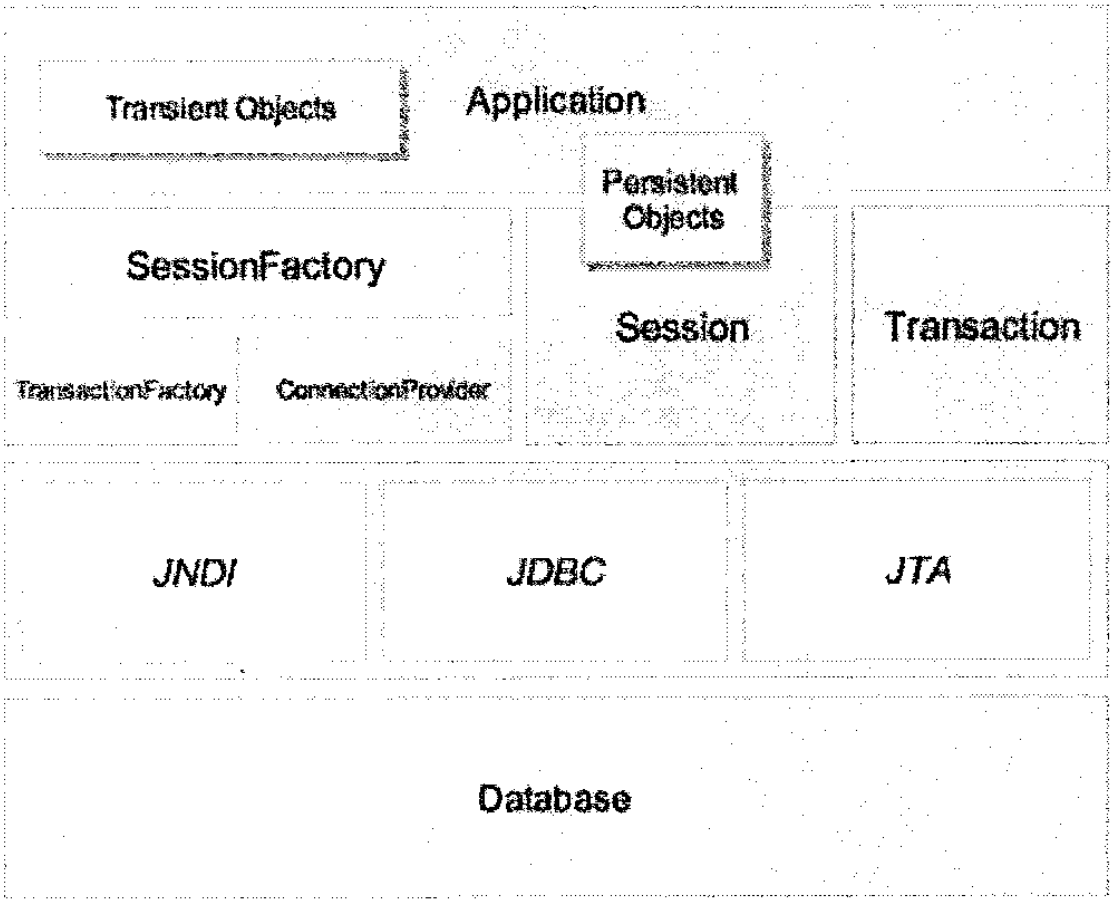
Το Hibernate είναι ένα εξαιρετικό πλαίσιο αμεταβλητότητας που παρέχει διαφανή (transparent) αμεταβλητότητα για κλάσεις Java. Υποστηρίζει *συνδέσμους* (***association),*** *κληρονομικότητα* ***(inheritance),*** *πολυμορφισμό* ***(polymorphism),*** *σύνθεση* ***(composition),*** *και τις πρότυπες κλάσεις της* ***Java*** *συλλογής.* Αντί για χειρωνακτική εκτέλεση interface και την παραγωγή κώδικα με JDK κατά τον έλεγχο (compilation), το Hibernate χρησιμοποιεί είδωλο κατά την εκτέλεση (run-time reflection) [8], εφαρμοσμένο με CGLIB [14] στην εκκίνηση του συστήματος για τη δημιουργία SQL και proxy εκτελέσεων [5]. Η βιβλιοθήκη παραγωγής κώδικα (CGLIB Code Generation Library) χρησιμοποιείται για την επέκταση κλάσεων Java και εφαρμόζει interfaces κατά την εκτέλεση. Η Γλώσσα Ερωτημάτων Hibernate (Hibernate Query Language HQL) δίνει την εντύπωση στον προγραμματιστή ότι δουλεύει με μια αντικειμενοστραφή βάση δεδομένων.




Σχήμα 6. Μια υψηλού επιπέδου άποψη της αρχιτεκτονικής Hibernate

Στο Hibernate, τα αμετάβλητα αντικείμενα ορίζονται σε ένα έγγραφο συσχέτισης, που εξυπηρετεί στην περιγραφή των αμετάβλητων πεδίων και σχέσεων, όπως και κάθε υποκλάση ή proxy του αμετάβλητου αντικειμένου, που περιγράφεται χρησιμοποιώντας XML. Τα έγγραφα συσχέτισης συντάσσονται στο χρόνο εκκίνησης της εφαρμογής και παρέχουν στο πλαίσιο απαραίτητες πληροφορίες για μια κλάση. Επιπλέον, χρησιμοποιούνται σε υποστηρικτικές λειτουργίες, όπως παραγωγή του σχήματος της

βάσης δεδομένων ή τη δημιουργία Java αρχείων προέλευσης (Java source files). Από την καταρτισμένη συλλογή εγγράφων συσχέτισης δημιουργείται ένα *SessionFactory*. Το *SessionFactory* παρέχει το μηχανισμό διαχείρισης αμετάβλητων κλάσεων, το *Session* interface. Η *Session* κλάση παρέχει το interface ανάμεσα στην αποθήκευση αμετάβλητων δεδομένων και την εφαρμογή. Το *Session* interface εμπεριέχει μια σύνδεση JDBC, η οποία μπορεί να είναι διαχειριζόμενη από το χρήστη ή να ελέγχεται από το Hibernate, και προορίζεται για χρήση από μία μόνο ροή εκτέλεσης, μετά κλείνει και απορρίπτεται. Το παρακάτω σχήμα δείχνει την επισκόπηση της αρχιτεκτονικής Hibernate.




Σχήμα 7. Αρχιτεκτονική Hibernate κατά την εκτέλεση

## 1.4 **Πλεονεκτήματα του** Hibernate

Υπάρχουν πολλά πλαίσια αμεταβλητότητας. Κάθε πλαίσιο αμεταβλητότητας έχει τα δικά του διαφορετικά χαρακτηριστικά. Οι λόγοι που επιλέξαμε το Hibernate παρατίθενται παρακάτω:

1. Το Hibernate είναι τώρα η πιο δημοφιλής ORM λύση για Java.
2. Το Hibernate μας επιτρέπει να αναπτύξουμε αμετάβλητα αντικείμενα ακολουθώντας κοινά ιδιώματα Java - συμπεριλαμβανομένων πλαισίων όπως, *συνδέσμου* (*association), κληρονομικότητας (inheritance), πολυμορφισμού (polymorphism), σύνθεσης (composition)* και *Java συλλογές (collections).*
3. Είναι δυνατά μοντέλα αντικειμένων που είναι λεπτομερή και υποστηρίζονται σθεναρά.
4. Το Hibernate απορρίπτει τη χρήση κώδικα που δημιουργείται την ώρα της κατασκευής και την επεξεργασία κώδικα byte (bytecode processing). Αντ' αυτού, χρησιμοποιούνται ανανακλάσεις και παραγωγή κώδικα byte κατά την εκτέλεση και κατά την εκκίνηση του συστήματος γίνεται και εκκίνηση της SQL. Αυτή η απόφαση εξασφαλίζει ότι το Hibernate δεν επιδρά στην IDE αποσφαλμάτωση (debugging) και στην αυξητική μετάφραση κώδικα (incremental compile).
5. Το Hibernate είναι μια ελαφριά ORM που ενσωματώθηκε από τη JDBC. Έτσι είναι απλό και μπορεί να αποφύγει τα πολύπλοκα θέματα της έντονης ενσωμάτωσης, όπως η δυνατότητα απομακρυσμένης πρόσβασης (remotability) στη CMP.
6. Η Γλώσσα Ερωτημάτων Hibernate (*Hibernate Query Language),* σχεδιασμένη ως μια "μηδαμινή" αντικειμενοστραφής επέκταση στην SQL, παρέχει μια κομψή γέφυρα ανάμεσα στον κόσμο των αντικειμένων και το σχεσιακό κόσμο.
7. Το Hibernate υποστηρίζεται από δημοφιλή εργαλεία ανάπτυξης ανοιχτού κώδικα (open source) συμπεριλαμβανομένων των XDoclet, Middlegen και AndroMDA. Έτσι, meta δεδομένα θα μπορούσαν να δημιουργηθούν αυτόματα από το σχήμα της βάσης δεδομένων ή τα αντικείμενα Java. Από την άλλη, τα αντικείμενα java θα μπορούσαν επίσης να δημιουργηθούν από meta δεδομένα, που υποστηρίζουν OOAD και μοντέλο καθοδηγούμενο από δεδομένα.
8. Το Hibernate είναι δωρεάν και είναι ένα πλαίσιο ανοιχτού κώδικα (open source). Το Lesser General Public License (LGPL) [16] είναι επαρκώς ευέλικτο για να επιτρέπει τη χρήση του Hibernate και σε έργα ανοιχτού κώδικα και σε εμπορικά

έργα. Έτσι θα μπορούσαμε να μελετήσουμε των πηγαίο κώδικα του Hibernate και να προσθέσουμε επιπλέον λειτουργικότητα αν χρειαστεί. Στην ιστοσελίδα JavaWorld [15], υπάρχουν μόνο τρία προϊόντα που ανταγωνίζονται ως το καλύτερο εργαλείο πρόσβασης δεδομένων Java (Best Java Data Access Tool):

A. CocoBase Enterprise O/R 4.5, Thought Inc.
B. Hibernate 4.0, που φιλοξενείται από το SourceForge.net
C. Oracle 9iAS TopLink, Oracle

Ανάμεσα σ' αυτά τα τρία εξαιρετικά προϊόντα, μόνο το Hibernate είναι δωρεάν και ανοιχτού κώδικα (open source).

9. Υποστηρίζονται όλα τα μεγάλα συστήματα διαχείρισης σχεσιακών βάσεων: Oracle, DB2, MySQL, PostgreSQL, Sybase, SAP DB, HypersonicSQL, Microsoft SQL Server, Informix, FrontBase, Ingres, Progress, Mckoi SQL, Pointbase και Interbase.
10. Ευέλικτη αντιστοίχιση σχέσεων περιλαμβάνει αντιστοίχιση ένα-προς-ένα, αντιστοίχιση ένα-προς-πολλά, αντιστοίχιση πολλά-προς-πολλά και αντιστοίχιση πολλαπλών κλάσεων σε έναν πίνακα. Επίσης, το Hibernate υποστηρίζει λειτουργίες ομαδοποίησης και πρόσθεσης.
11. Πλήρη και λεπτομερή βοηθητικά αρχεία υποστηρίζουν τη χρήση του Hibernate, το οποίο επιτρέπει στους ανθρώπους να κατανοήσουν και να εφαρμόσουν το Hibernate εύκολα.

## 1.5 **Αρχιτεκτονική πολλαπλών επιπέδων** (Multi-tier Architecture)

Μια κατανεμημένη εφαρμογή είναι ένα σύστημα που περιλαμβάνει την εκτέλεση διαφορετικών προγραμμάτων που τρέχουν σε διαφορετικούς εξυπηρετητές (host computers). Η αρχιτεκτονική της κατανεμημένης εφαρμογής είναι ένα σχεδιάγραμμα των διαφορετικών προγραμμάτων. Αυτό το σχεδιάγραμμα περιλαμβάνει τις τοποθεσίες των διαφορετικών προγραμμάτων, τις ευθύνες των διαφορετικών προγραμμάτων και ούτω καθεξής. Η λογική των επιπέδων παρέχει ένα βολικό τρόπο ομαδοποίησης αρχιτεκτονιών. Αν μια εφαρμογή τρέχει απλά σε έναν μόνο υπολογιστή, είναι η αρχιτεκτονική του ενός επιπέδου. Αλλιώς, η εφαρμογή είναι μια αρχιτεκτονική πολλαπλών επιπέδων η οποία θα μπορούσε να είναι δύο επιπέδων, τριών επιπέδων ή n επιπέδων. Η αρχιτεκτονική πολλαπλών επιπέδων έχει πολλά πλεονεκτήματα όπως επαναχρησιμότητα (reusability), δυνατότητα διαβάθμισης (scalability), χαμηλά κόστη συντήρησης και μεγαλύτερη ευελιξία. Έτσι, η αρχιτεκτονική πολλαπλών επιπέδων, ειδικά αυτή των τριών επιπέδων ή των n επιπέδων, είναι κατάλληλη για επιχειρησιακό περιβάλλον εφαρμογών.

# ΚΕΦΑΛΑΙΟ 2ο - ΣΥΓΚΡΙΣΗ ΤΩΝ JDBC ΚΑΙ HIBERNATE

## 2.1 **Σύγκριση απόδοσης** (Performance comparison)

Σ' αυτό το κεφάλαιο, θα παρουσιάσουμε συγκρίσεις απόδοσης μεταξύ των JDBC και Hibernate. Παρόλο που γνωρίζουμε ότι το Hibernate είναι σίγουρα πιο αργό από το JDBC αφού το Hibernate είναι η ελαφριά αφηρημένη ενσωμάτωση από το JDBC, θα έπρεπε να έχουμε μια πρόχειρη εκτίμηση της διαφοράς απόδοσης μεταξύ των JDBC και Hibernate. Και το Hibernate και το JDBC χρησιμοποιούν κλήση τοπικής διαδικασίας αντί για κλήση απομακρυσμένης διαδικασίας (RPC Remote Procedure Calling). Αλλά το CMP χρησιμοποιεί επίκληση απομακρυσμένης μεθόδου (RMI Remote Method Invocation), το οποίο θα οδηγούσε σε επιβάρυνση. Σε μια εφαρμογή J2EE, οι clients (εφαρμογές, σελίδες JavaServer, Servlets, τεχνολογίες JavaBeans) έχουν πρόσβαση στα αντικείμενα beans μέσω των απομακρυσμένων τους διεπαφών. Έτσι, κάθε επίκληση προγράμματος-πελάτη δυνητικά περνά μέσω στελεχών και σκελετών δικτύου ακόμα κι αν το πρόγραμμα-πελάτης και το επιχειρησιακό αντικείμενο bean (enterprise bean) είναι στην ίδια εικονική μηχανή Java, λειτουργικό σύστημα, ή μηχάνημα [9]. Γι' αυτό, δεν είναι απαραίτητο να συγκρίνουμε την απόδοση του CMP με το JDBC και το Hibernate.

Για να μετρήσουμε χρονικά διαστήματα, χρησιμοποιήσαμε τη μέθοδο Java: long System.currentTimeMillis(), η οποία επιστρέφει τον αριθμό των milliseconds που πέρασαν από την 1η Ιανουαρίου 1970. Εφαρμογές διαφορετικών εικονικών μηχανισμών Java (JVM) μπορούν να παρέχουν διαφορετικά επίπεδα ακρίβειας.

Για να εκτιμήσουμε την απόδοση, αναπτύξαμε ένα απλό πρόγραμμα Java. Κατά τη διάρκεια των δοκιμών, αποκτήσαμε τη μέση τιμή βασισμένοι σε 10 επαναλήψεις. Επιπλέον, τα μεγέθη των κλάσεων Hibernate είναι πολύ μεγαλύτερα από αυτά των JDBC κλάσεων, το οποίο θα έχει σαν αποτέλεσμα στην πρώτη επανάληψη οι κλάσεις να έχουν διαφορετικό χρόνο φόρτωσης. Μετά την πρώτη επανάληψη, όλες οι απαραίτητες κλάσεις θα έχουν φορτωθεί στην εικονική μηχανή Java (Java Virtual Machine - JVM). Αφού μας ενδιαφέρει μόνο η απόδοση της βάσης δεδομένων κατά την εκτέλεση, αποφεύγουμε κάθε απόκλιση που προκαλείται από το διαφορετικό χρόνο φόρτωσης, απορρίπτοντας την τιμή

της πρώτης επανάληψης. Έτσι, μπορούμε να πάρουμε ένα πιο ακριβές αποτέλεσμα της δοκιμής.

Κλείσαμε το σύνολο το συνδέσεων των βάσεων δεδομένων στο JDBC και το Hibernate έτσι ώστε να μπορέσουμε να αποκλείσουμε πιθανή απόκλιση της απόδοσης που θα μπορούσε να προκληθεί από το σύνολο των συνδέσεων των βάσεων δεδομένων. Σε κάθε τμήμα, διαφορετική εκτέλεση θα μας έδινε διαφορετικά αποτελέσματα απόδοσης. Για να πάρουμε λογικά αποτελέσματα σύγκρισης, χρησιμοποιούμε την κατάλληλη μέθοδο σε κάθε τμήμα. Παρακάτω, δείχνουμε την εφαρμογή μας λεπτομερώς.

Τέλος, αυτό που πρέπει να σημειωθεί είναι ότι δε λάβαμε υπόψη το συγχρονισμό και άλλες πολύπλοκες δοκιμές γιατί η απόδοση της δοκιμής είναι ένα πολύ πολύπλοκο θέμα. Μερικές φορές, η απόδοση της δοκιμής γίνεται χρησιμοποιώντας ειδικό υλικό (hardware) και λογισμικό (software). Εδώ, απλά δοκιμάζουμε την απόδοση κατά την ώρα εκτέλεσης και δίνουμε μια πρόχειρη εκτίμηση της διαφοράς μεταξύ τους.

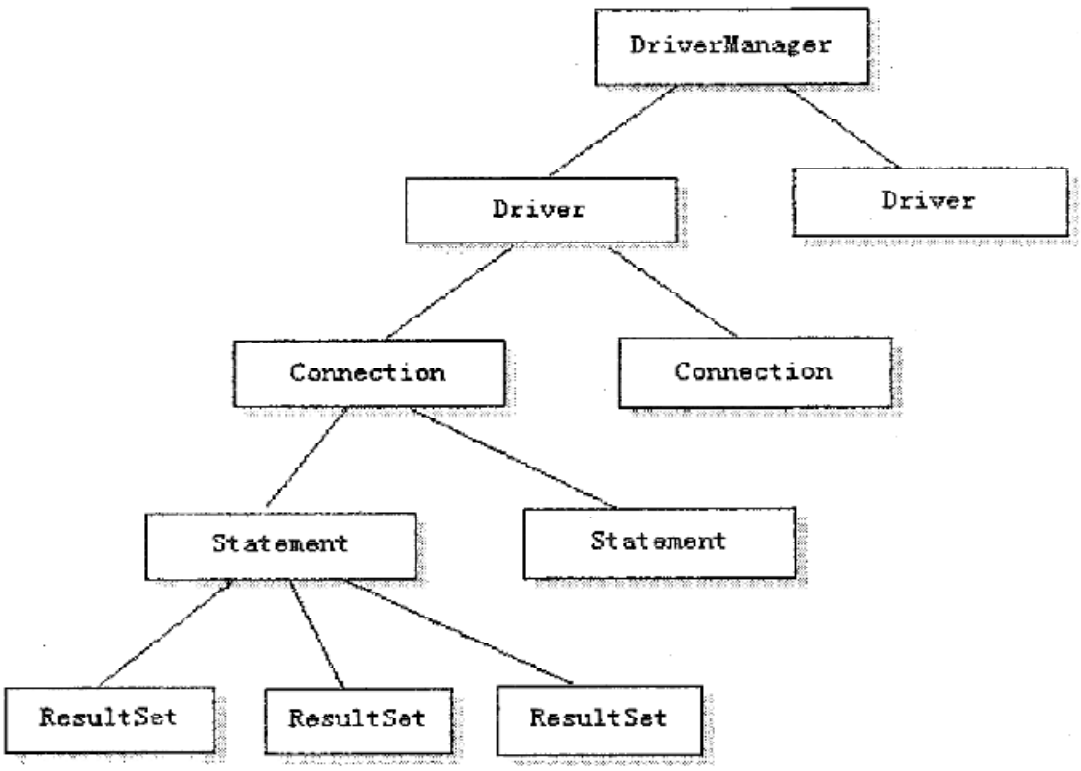
Ρυθμίσεις Hardware και Software

| Item | Content |
|---|---|
| CPU | INTEL CORE 2 DUO |
| Memory | 8 GB |
| Hard Disk | 1 T |
| Operating System | MS Windows XP |
| Database | MS SQL Server 2005 |
| Persistence Framework | Hibernate 4.0 |
| JDBC Driver | Official JDBC Driver For MS SQL Server |

Πίνακας 1. Ρυθμίσεις δοκιμής για Απόδοση του JDBC και του Hibernate

### 2.1.1 JDBC

Το ακόλουθο διάγραμμα είναι ένα JDBC διάγραμμα καταρράκτη.




Σχήμα 8. JDBC διάγραμμα καταρράκτη

1. Μια διεύθυνση δικτύου (URL) περνάει στη μέθοδο `getConnection` του `DriverManager`, που με τη σειρά του εντοπίζει ένα `Driver`.
2. Με ένα `Driver`, μπορούμε να αποκτήσουμε μια `Connection`.
3. Με μια `Connection`, μπορούμε να δημιουργήσουμε μια `Statement`.
4. Όταν μια `Statement` εκτελείται με μια μέθοδο `executeQuery`, μπορεί να επιστραφεί ένα `ResultSet`.

Στη JDBC, ένα αντικείμενο `statement` θα αναλύσει και θα μεταφράσει δηλώσεις SQL κάθε φορά όταν οι δηλώσεις SQL εκτελούνται. Αφού πρόκειται να εκτελέσουμε τις ίδιες SQL δηλώσεις πολλές φορές για να ελέγξουμε την απόδοση, αποφασίσαμε να επιλέξουμε αντικείμενο έτοιμης δήλωσης (`preparedStatement` object) αντί για αντικείμενο δήλωσης (`statement` object). Μια έτοιμη δήλωση (`preparedStatement`) είναι μια προ-μεταφρασμένη δήλωση SQL που είναι πιο

αποδοτική από την κλήση της ίδιας SQL δήλωσης ξανά και ξανά. Η κλάση `preparedStatement` επεκτείνει την κλάση `Statement` μα προσθέσει τη δυνατότητα να θέτει παραμέτρους εντός μιας δήλωσης. Όταν δηλώνουμε ένα αντικείμενο `preparedStatement`, χρησιμοποιούμε το χαρακτήρα ερωτηματικό (?) για να κρατήσει τη θέση για την παράμετρο που θα έρθει. Όταν περνά η παράμετρος στη δήλωση, υποδεικνύουμε σε ποια θέση αναφερόμαστε μέσω της σειριακής της θέσης στη δήλωση. [4]

Προκαθορισμένα, όταν δημιουργείται μια σύνδεση (`Connection`), η σύνδεση (`Connection`) είναι στη μέθοδο συναλλαγής αυτόματης καταχώρησης (Auto-Commit transaction mode). Αυτό σημαίνει ότι κάθε ξεχωριστή SQL δήλωση αντιμετωπίζεται σαν μια συναλλαγή και θα καταχωρηθεί αυτόματα ακριβώς μετά την εκτέλεσή της, το οποίο δε βολεύει για την εκτέλεση των ίδιων δηλώσεων SQL πολλές φορές. Για να βελτιωθεί η αποδοτικότητα, απενεργοποιούμε αυτή τη μέθοδο αυτόματης καταχώρησης και καταχωρούμε μια συναλλαγή μόνο μια φορά, μετά την εκτέλεση όλων των δηλώσεων SQL.

Παρομοίως, για να βελτιωθεί η αποδοτικότητα, χρησιμοποιούμε τις μαζικής επεξεργασίας (batch process) μεθόδους `addBatch(String)` και `executeBatch ()` επειδή υπάρχουν λειτουργίες μαζικής ενημέρωσης (batch update) και μαζικής εισαγωγής (batch insert) στη σύγκρισή μας. Μια μαζική ενημέρωση είναι μια ομάδα πολλαπλών δηλώσεων ενημέρωσης που τίθεται στη βάση δεδομένων για επεξεργασία ως σύνολο. Η αποστολή πολλαπλών δηλώσεων ενημέρωσης στη βάση δεδομένων μαζί σαν ομάδα, μπορεί να είναι πολύ πιο αποδοτική από την αποστολή κάθε μιας δήλωσης ενημέρωσης χωριστά. [4]

### 2.1.2 Hibernate

Υπάρχουν δύο μέθοδοι αναζήτησης στο Hibernate: *query.find()* και *query.iterate().* Όπως και το JDBC, καλώντας την αναζήτηση μέσω της μεθόδου *query.find()* θα επέστρεφε ένα πολύ μεγάλο JDBC resultSet. Όσο για την *query.iterate(),* αν περιμένουμε η αναζήτησή μας να επιστρέψει έναν πολύ μεγάλο αριθμό αντικειμένων, αλλά δεν ενδέχεται να τα χρησιμοποιήσουμε όλα, ίσως είχαμε καλύτερη απόδοση με την *iterate()* μέθοδο, η οποία επιστρέφει ένα *java.util.Iterator.* Το iterator θα φορτώνει αντικείμενα κατά απαίτηση, χρησιμοποιώντας τους δείκτες που επεστράφησαν από μια αρχική SQL αναζήτηση [5]. Μια άλλη διαφορά ανάμεσα στις δύο παραπάνω μεθόδους αναζήτησης, είναι ότι η μέθοδος *query.iterate()* μπορεί να χρησιμοποιήσει την cache μνήμη για να βελτιώσει την απόδοση της αναζήτησης, ενώ η μέθοδος *query.find()* δεν μπορεί να χρησιμοποιήσει την cache. Αφού η JDBC δεν μπορεί να χρησιμοποιήσει την cache, και αφού αναζητούμε ένα μεγάλο αριθμό αντικειμένων και τα χρησιμοποιούμε όλα για σύγκριση, διαλέξαμε τη μέθοδο *query.find()* αντί για την *query.iterate()* για τη σύγκριση απόδοσης αναζήτησης μεταξύ JDBC και Hibernate.

### 2.1.3 **Σύγκριση απόδοσης της εντολής** Insert (Insert Performance Comparison)

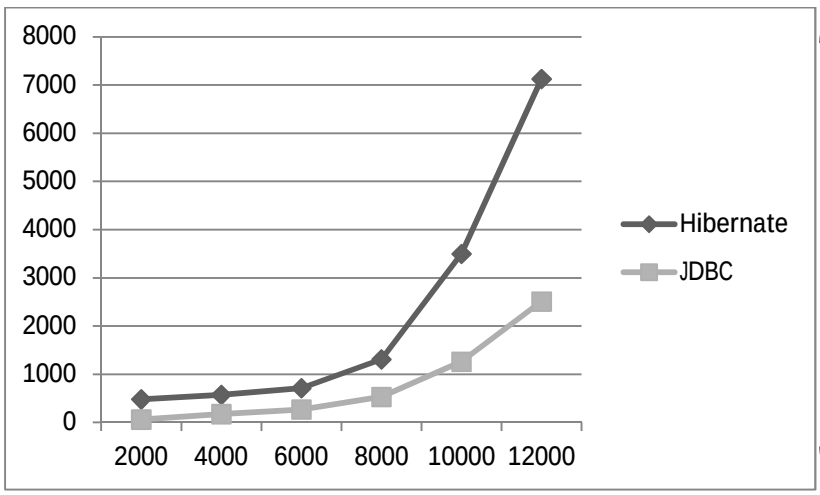
Για την παρακολούθηση των αλλαγών στις πηγές του συστήματος κατά τη διάρκεια της εκτέλεσης, χρησιμοποιούμε προϊόν της εταιρίας λογισμικού Quest Software: το spotlight, που παρουσιάζει τη δραστηριότητα πραγματικού χρόνου όλων των κρίσιμων συστατικών του SQL Server.

Unit: MS

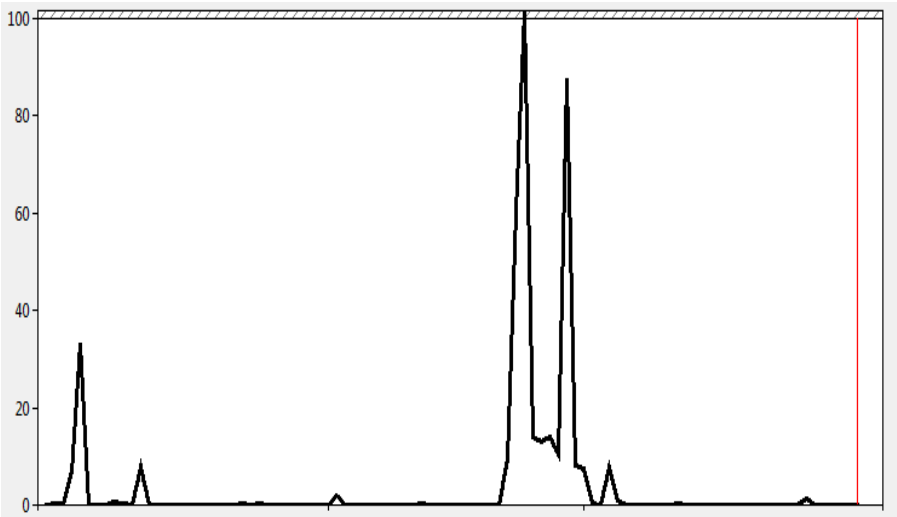
| INSERT * (10 times) in msec, 3 times average | Hibernate | JDBC |
|---|---|---|
| **1000** | 483 | 62 |
| **3000** | 576 | 174 |
| **5000** | 714 | 271 |
| **10000** | 1311 | 532 |
| **25000** | 3499 | 1262 |
| **50000** | 7126 | 2510 |

Πίνακας 2. Σύγκριση απόδοσης της εντολής Insert, μεταξύ του JDBC και του Hibernate

***Παράθεση γραφημάτων:***

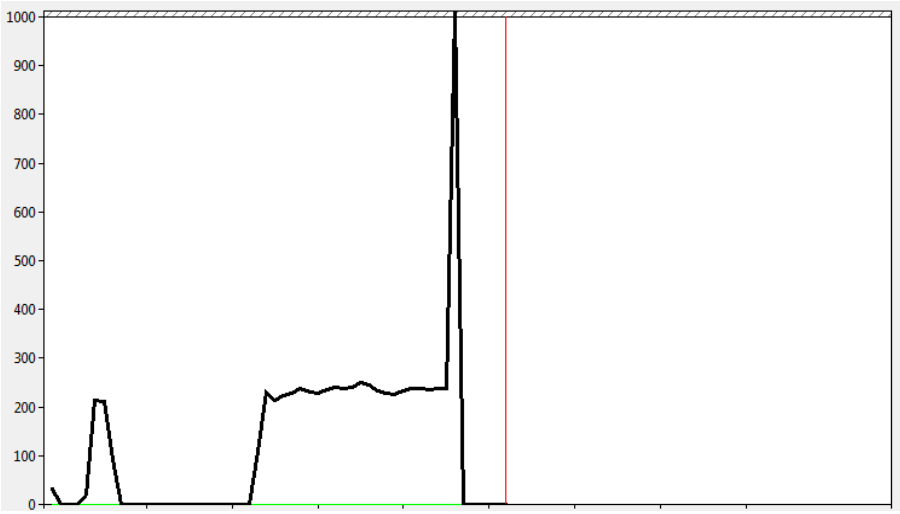



Γράφημα 1. Γράφημα χρόνου εκτέλεσης (σε msec) για το Hibernate και το JDBC σύμφωνα με τον αριθμό δημιουργούμενων στοιχείων.

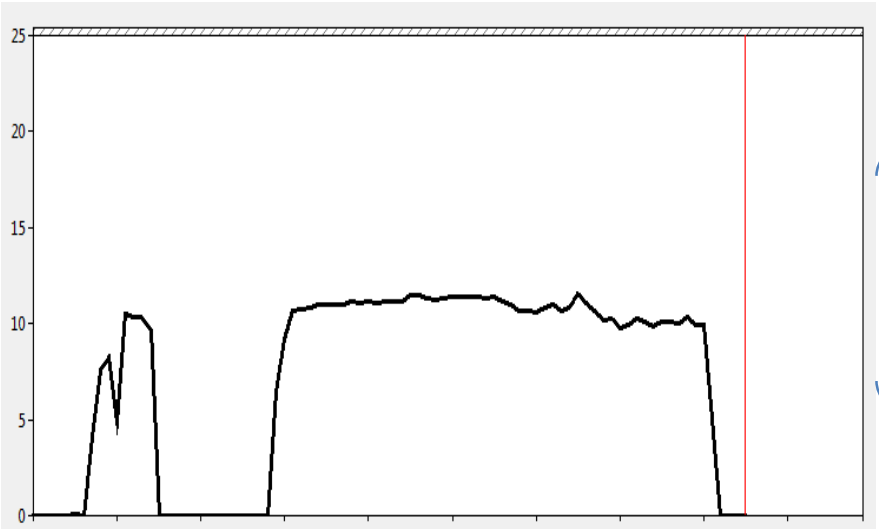



*Γράφημα 2.*
*INSERT*
*Hibernate*
*Logical I/O*

Λογική I/O σύγκριση μεταξύ Hibernate και JDBC όταν εισάγουμε 50000 αρχεία (το Hibernate είναι σε χρονική περίοδο από 11:05 μέχρι 11:07 και στο JDBC οι μονάδες του κάθετου πίνακα είναι απόλυτος αριθμός σε εκατοντάδες).

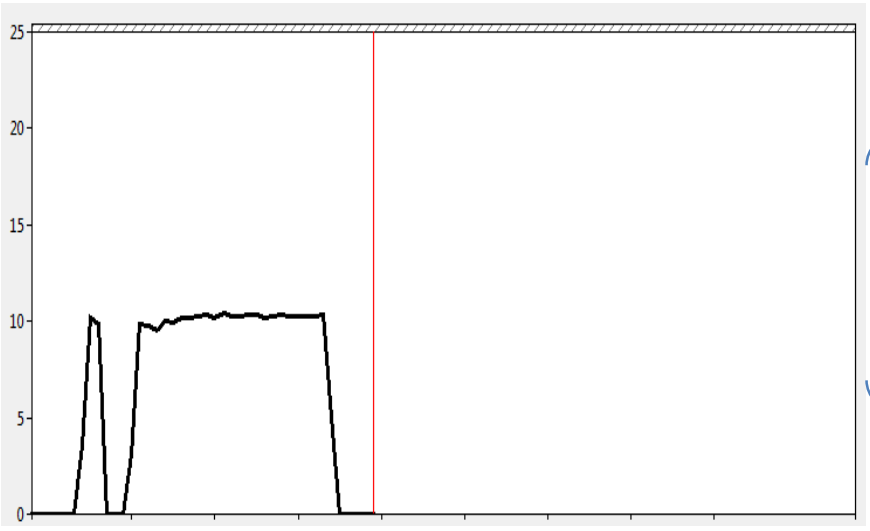



*Γράφημα 3.*
*INSERT*
*JDBC*
*Logical I/O*

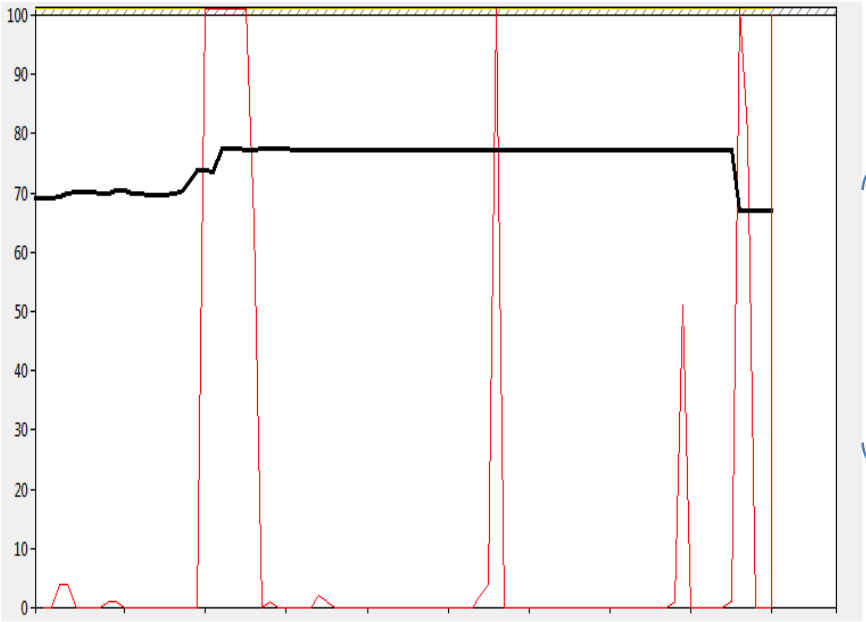



*Γράφημα 4. INSERT Hibernate CPU Usage*

Σύγκριση Κεντρικής Μονάδας Επεξεργασίας (CPU) μεταξύ Hibernate και JDBC όταν εισάγουμε 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 11:05 μέχρι 11:07 και η μονάδα που χρησιμοποιείται στους κάθετους άξονες των διαγραμμάτων είναι (%)).

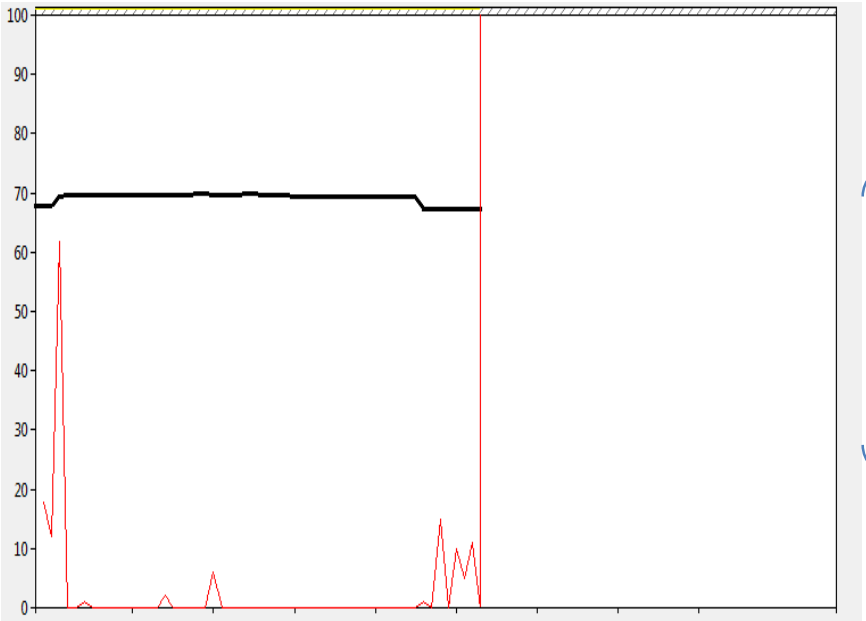



*Γράφημα 5. INSERT JDBC CPU Usage*

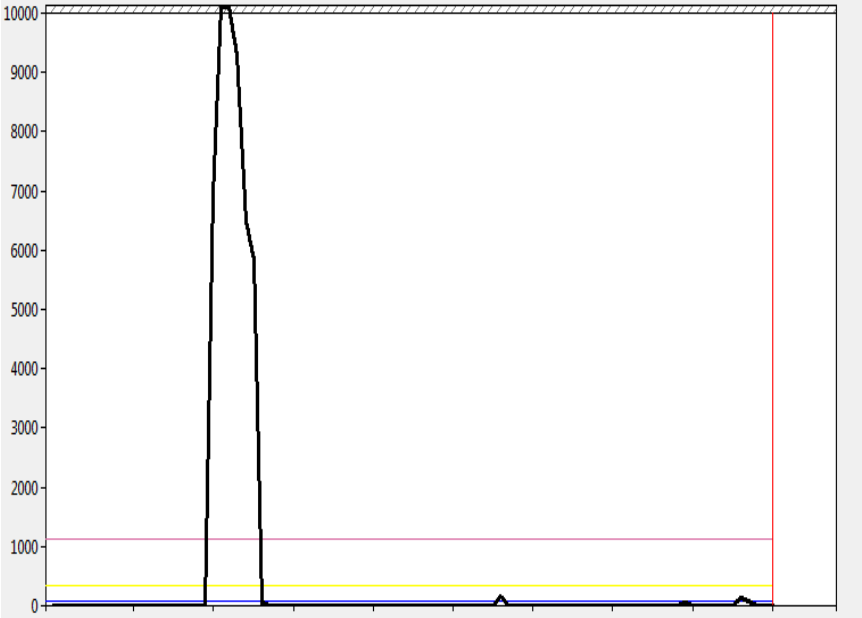



*Γράφημα 6. INSERT Hibernate Memory Usage*

Σύγκριση Μνήμης μεταξύ Hibernate και JDBC όταν εισάγουμε 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 11:05 μέχρι 11:07 και οι κάθετοι άξονες μας δίνουν τις επί τοις εκατό (%) τιμές της συνολικής χρησιμοποιούμενης μνήμης του συστήματος).

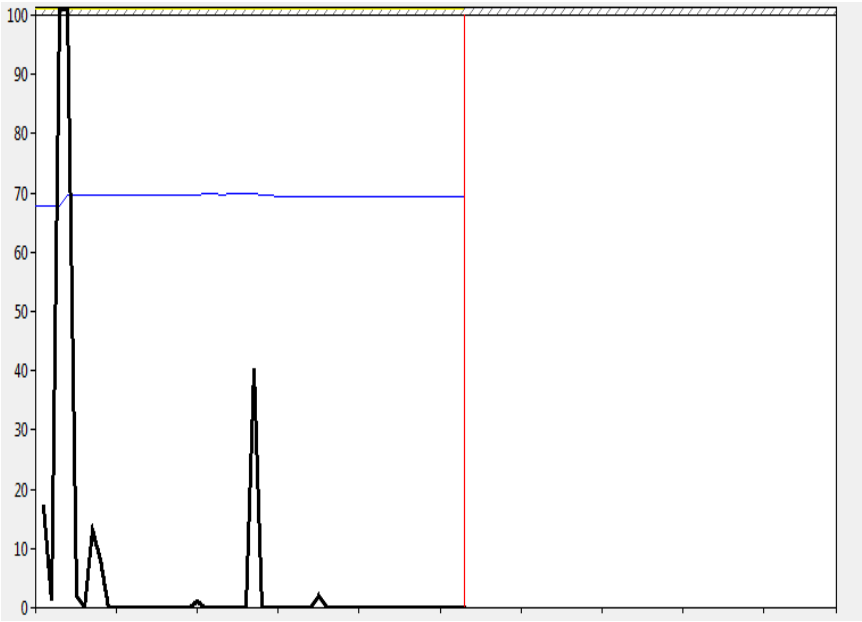



*Γράφημα 7. INSERT JDBC Memory Usage*




*Γράφημα 8. INSERT Hibernate Pages per Second*

Σύγκριση Page Rate μεταξύ Hibernate και JDBC όταν εισάγουμε 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 11:05 μέχρι 11:07 και ο κάθετος άξονας μετρά τις σελίδες ανά δευτερόλεπτο σε απόλυτο αριθμό).




*Γράφημα 9. INSERT JDBC Pages per Second*

Από τον πίνακα και τα σχήματα παραπάνω, βρήκαμε ότι δεν υπάρχει καμία σημαντική διαφορά απόδοσης ανάμεσα στο Hibernate και τη JDBC κατά τη δημιουργία αρχείων, παρόλο που υπάρχουν κάποιες διαφορές στην page rate μνήμη και τα logical I/O συγκριτικά διαγράμματα Ο λόγος για αυτές τις διαφορές είναι ότι, κατά τη διάρκεια των λειτουργιών δημιουργίας, το Hibernate χρειάζεται πολύ περισσότερη μνήμη για την κατασκευή νέων αμετάβλητων αντικειμένων, ενώ η JDBC δε χρειάζεται να κάνει αυτή τη δουλειά.

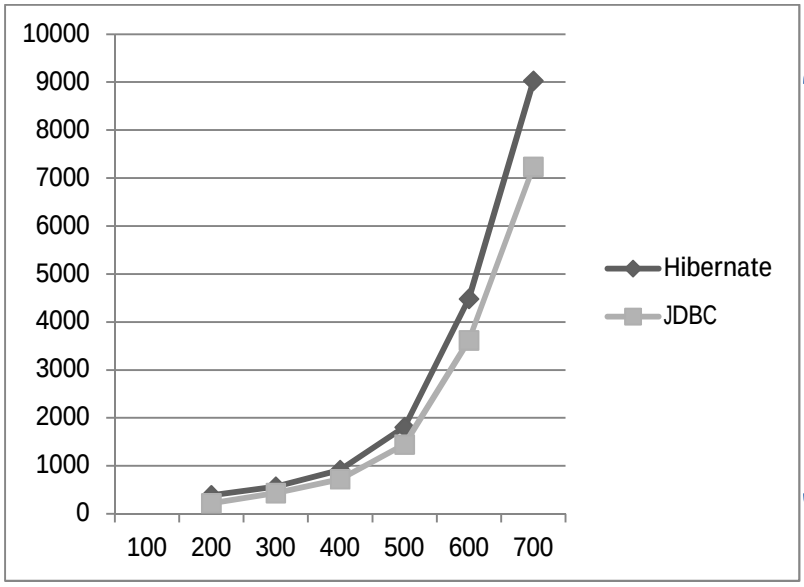
### 2.1.4 **Σύγκριση απόδοσης της εντολής** Select (Select Performance Comparison)

Unit: MS

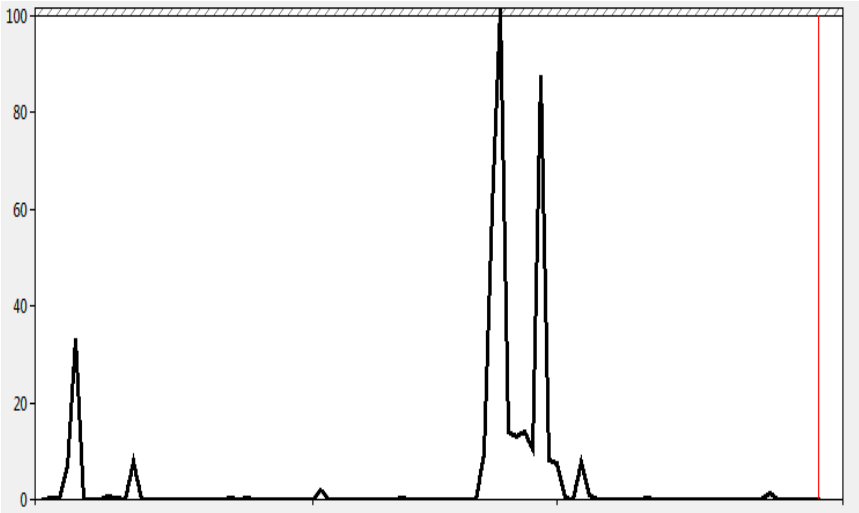
| SELECT * (100 times) in msec, 3 times average | Hibernate | JDBC |
|---|---|---|
| **1000** | 387 | 225 |
| **3000** | 569 | 435 |
| **5000** | 917 | 725 |
| **10000** | 1809 | 1448 |
| **25000** | 4484 | 3619 |
| **50000** | 9031 | 7239 |

Πίνακας 3. Σύγκριση Απόδοσης της εντολής Select, μεταξύ του JDBC και του Hibernate

***Παράθεση γραφημάτων:***

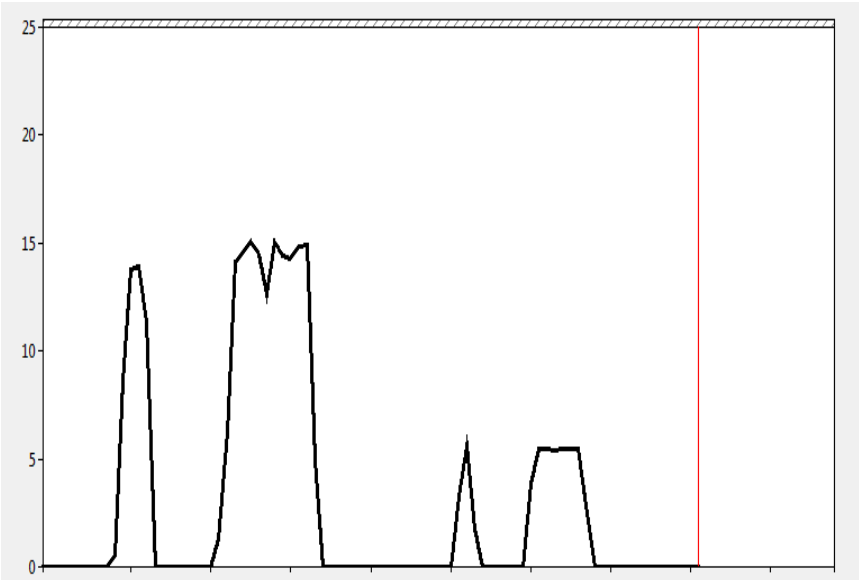



Γράφημα 10. Γράφημα χρόνου εκτέλεσης (σε msec) για το Hibernate και το JDBC σύμφωνα με τον αριθμό των επιλεγμένων στοιχείων.



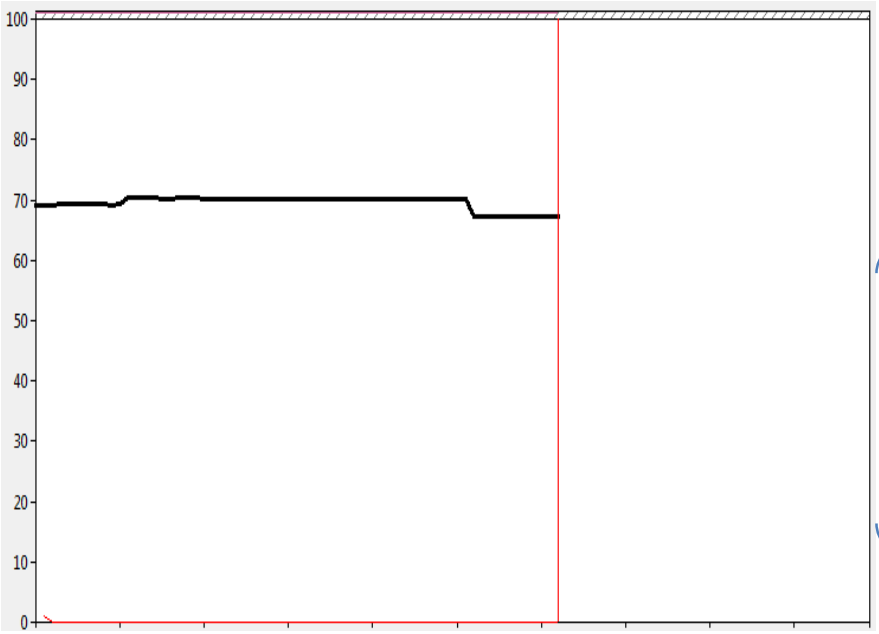

*Γράφημα 11. SELECT Hibernate and JDBC Logical I/O*

Σύγκριση Logical I/O μεταξύ Hibernate και JDBC όταν επιλέγονται 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 23:15 μέχρι 23:16 στην αρχή του διαγράμματος μιας και έχουν εκτελεστεί πρώτα αυτά του Hibernate (5000 και μετά 50000 εγγραφές) και μετά του JDBC (επίσης 5000 και 50000 εγγραφές) και οι μονάδες του κάθετου πίνακα είναι απόλυτος αριθμός σε εικοσάδες).



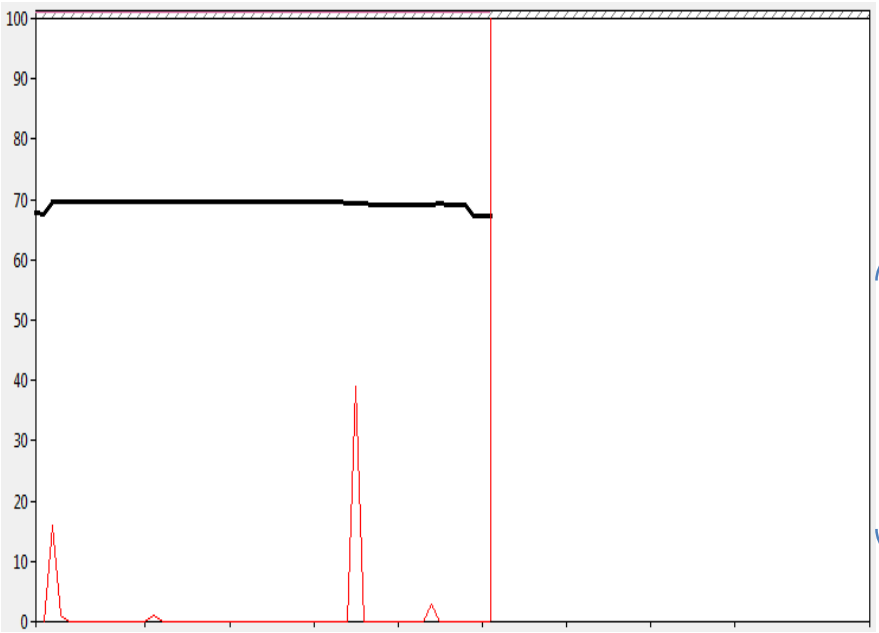

*Γράφημα 12. SELECT Hibernate and JDBC CPU Usage*

Σύγκριση Κεντρικής Μονάδας Επεξεργασίας (CPU) μεταξύ Hibernate και JDBC όταν επιλέγονται 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 23:15 μέχρι 23:16 στην αρχή του διαγράμματος μιας και έχουν εκτελεστεί πρώτα αυτά του Hibernate (5000 και μετά 50000 εγγραφές) και μετά του JDBC (επίσης 5000 και 50000 εγγραφές) και η μονάδα που χρησιμοποιείται στους κάθετους άξονες των διαγραμμάτων είναι (%)).

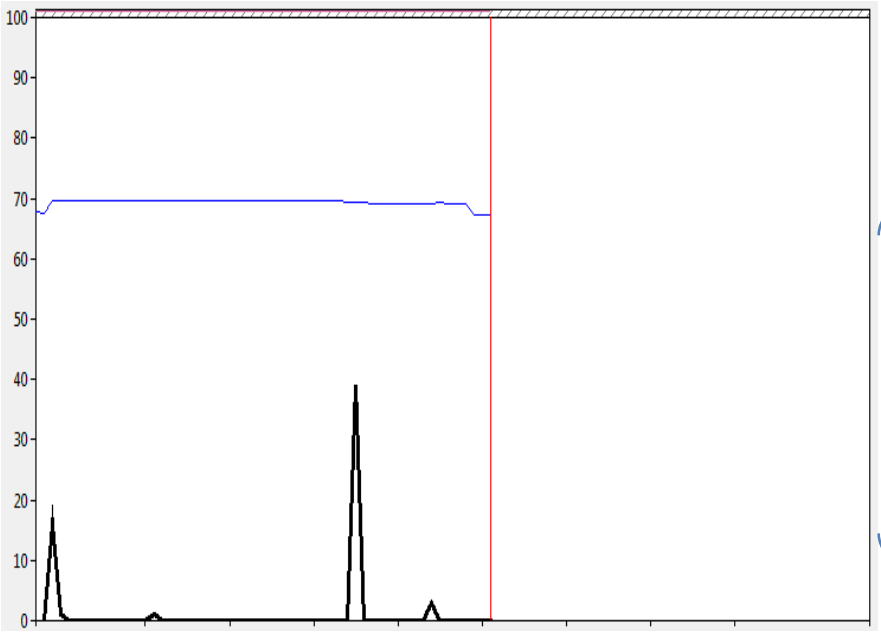



*Γράφημα 13. SELECT Hibernate Memory Usage*

Σύγκριση Μνήμης μεταξύ Hibernate και JDBC όταν επιλέγονται 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 23:15 μέχρι 23:16 και οι κάθετοι άξονες μας δίνουν τις επί τοις εκατό (%) τιμές της συνολικής χρησιμοποιούμενης μνήμης του συστήματος).




*Γράφημα 14. SELECT JDBC Memory Usage*




*Γράφημα 15. SELECT Hibernate and JDBC Pages per Second*

Σύγκριση Page Rate μεταξύ Hibernate και JDBC όταν επιλέγονται 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 23:15 μέχρι 23:16 στην αρχή του διαγράμματος μιας και έχουν εκτελεστεί πρώτα αυτά του Hibernate (5000 και μετά 50000 εγγραφές) και μετά του JDBC (επίσης 5000 και 50000 εγγραφές) και ο κάθετος άξονας μετρά τις σελίδες ανά δευτερόλεπτο σε απόλυτο αριθμό).

Από τον παραπάνω πίνακα και τα σχεδιαγράμματα, βρήκαμε ότι, όσο ο αριθμός των αρχείων που επιλέγονται αυξάνει, η διαφορά απόδοσης μεταξύ Hibernate και JDBC μεγαλώνει. Από τα παραπάνω σχήματα, βρήκαμε ότι η μεγάλη διαφορά είναι στο σχεδιάγραμμα σύγκρισης για τη χρήση της Κεντρικής Μονάδας Επεξεργασίας (CPU). Ο λόγος για τον οποίο υπάρχουν διαφορές είναι ότι κατά τη διάρκεια των λειτουργιών επιλογής, το Hibernate παίρνει το resultSet και χρειάζεται πολύ περισσότερη μνήμη για την κατασκευή αμετάβλητων αντικειμένων, ενώ η JDBC δεν έχει να κάνει αυτή τη δουλειά.

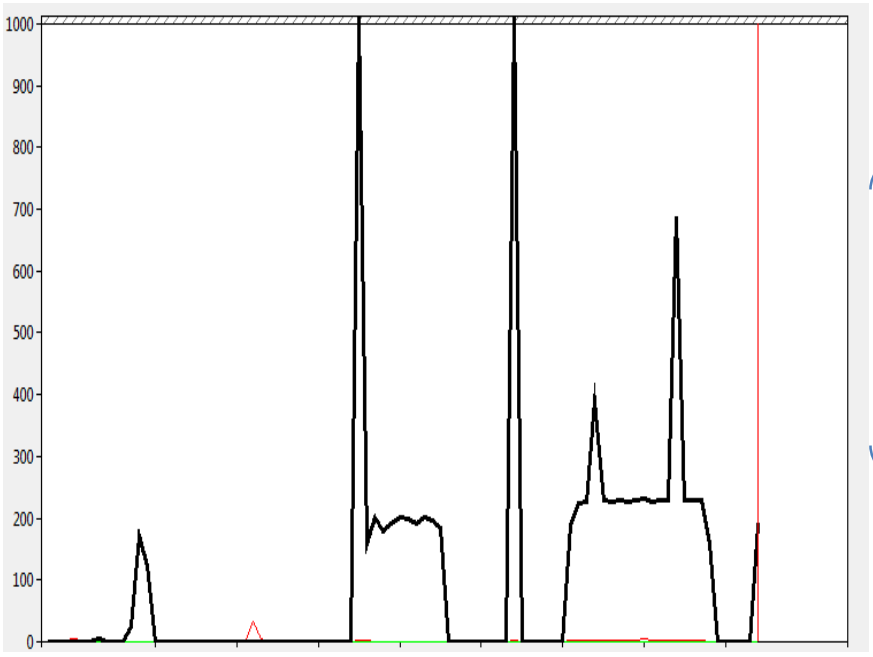
### 2.1.5 *Σύγκριση Απόδοσης της εντολής Διαγραφή (Delete Performance Comparison)*

Unit: MS

| DELETE * (1 time) in msec, 3 times average | Hibernate | JDBC |
|---|---|---|
| **1000** | 302 | 51 |
| **3000** | 394 | 151 |
| **5000** | 486 | 251 |
| **10000** | 841 | 506 |
| **25000** | 1807 | 1291 |
| **50000** | 3480 | 2501 |

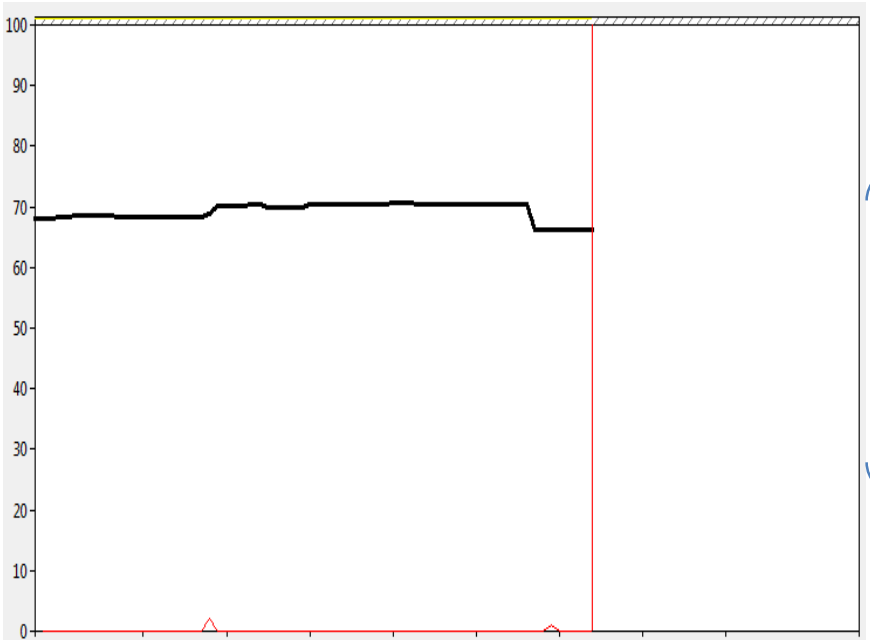
Πίνακας 4 Σύγκριση Απόδοσης της εντολής Delete, μεταξύ του JDBC και του Hibernate

***Παράθεση γραφημάτων:***



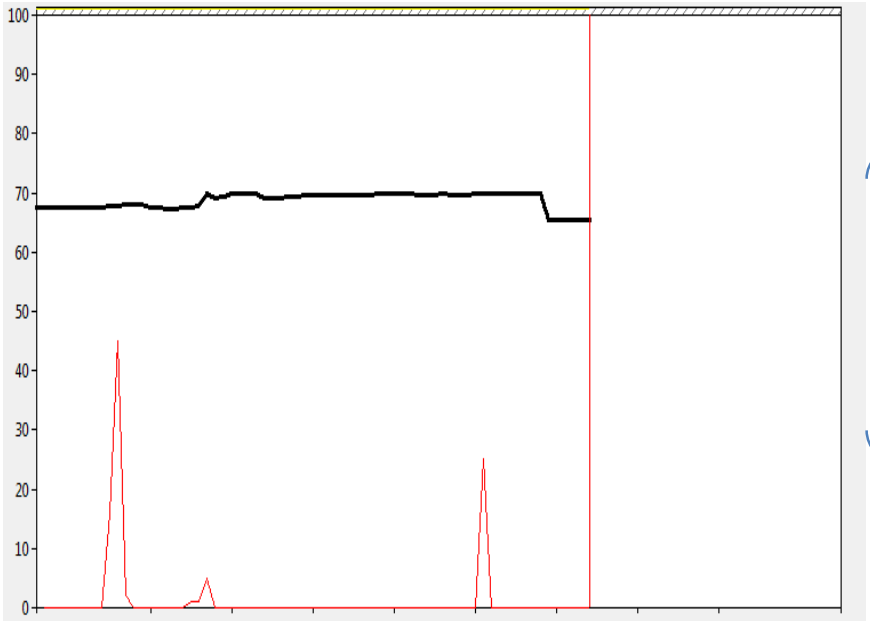

*Γράφημα 16. DELETE Hibernate and JDBC Logical I/O*

Σύγκριση Logical I/O μεταξύ Hibernate και JDBC όταν διαγράφονται 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 23:15 μέχρι 23:16 στην αρχή του διαγράμματος μιας και έχουν εκτελεστεί πρώτα αυτά του Hibernate (5000 και μετά 50000 εγγραφές) και μετά του JDBC (επίσης 5000 και 50000 εγγραφές) και οι μονάδες του κάθετου πίνακα είναι απόλυτος αριθμός σε εκατοντάδες).

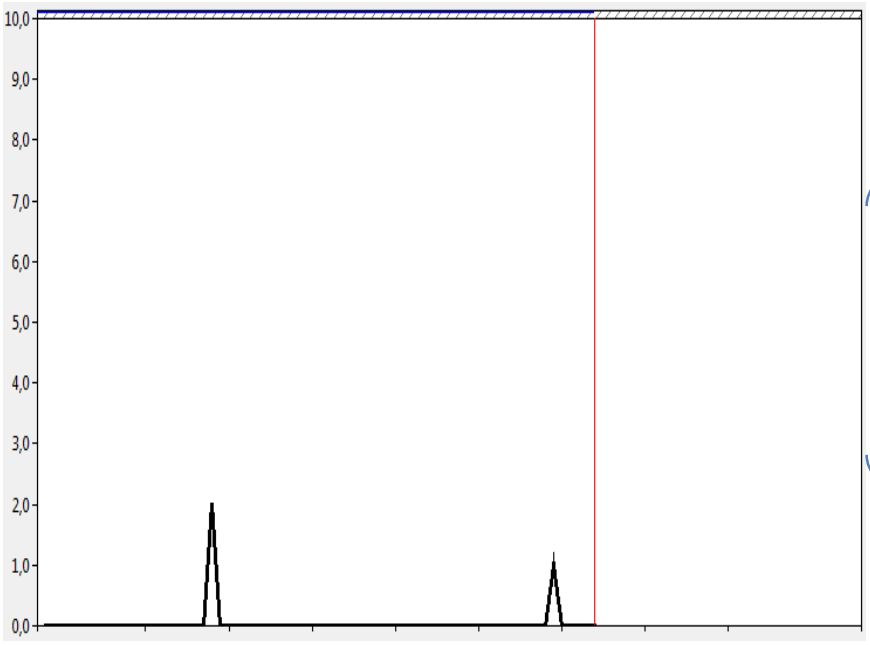



*Γράφημα 17. DELETE Hibernate Memory Usage*

Σύγκριση Μνήμης μεταξύ Hibernate και JDBC όταν διαγράφονται 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 23:27 μέχρι 23:30 και οι κάθετοι άξονες μας δίνουν τις επί τοις εκατό (%) τιμές της συνολικής χρησιμοποιούμενης μνήμης του συστήματος).

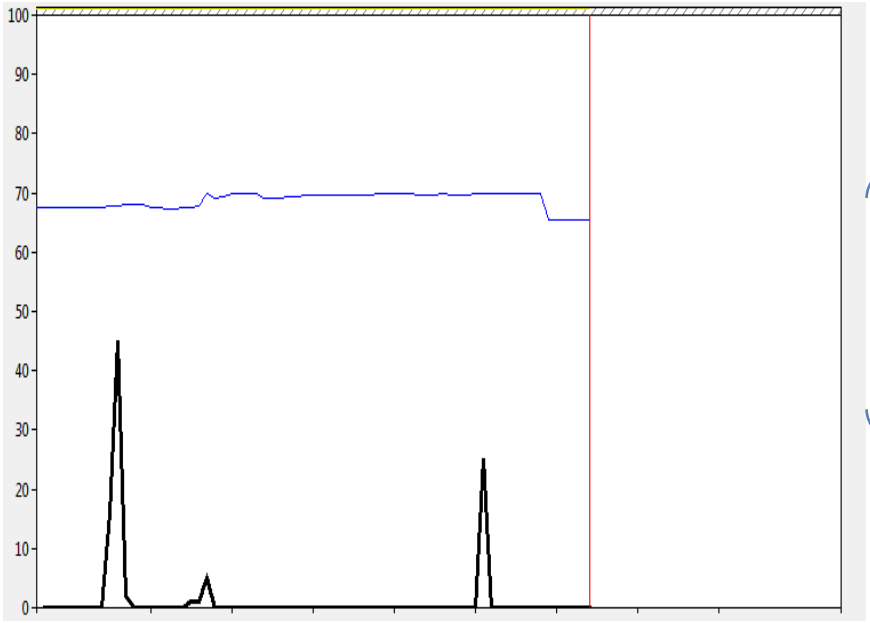



*Γράφημα 18. DELETE JDBC Memory Usage*

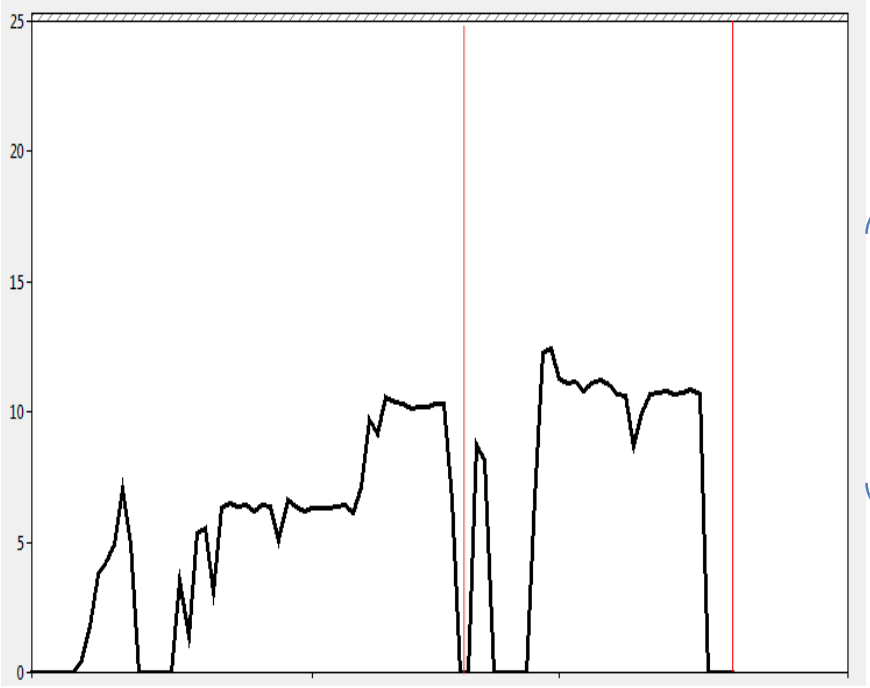



*Γράφημα 19. DELETE Hibernate Pages per Second*

Σύγκριση Page Rate μεταξύ Hibernate και JDBC όταν διαγράφουμε 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 23:27 μέχρι 23:30 και ο κάθετος άξονας μετρά τις σελίδες ανά δευτερόλεπτο σε απόλυτο αριθμό).




*Γράφημα 20. DELETE JDBC Pages per Second*



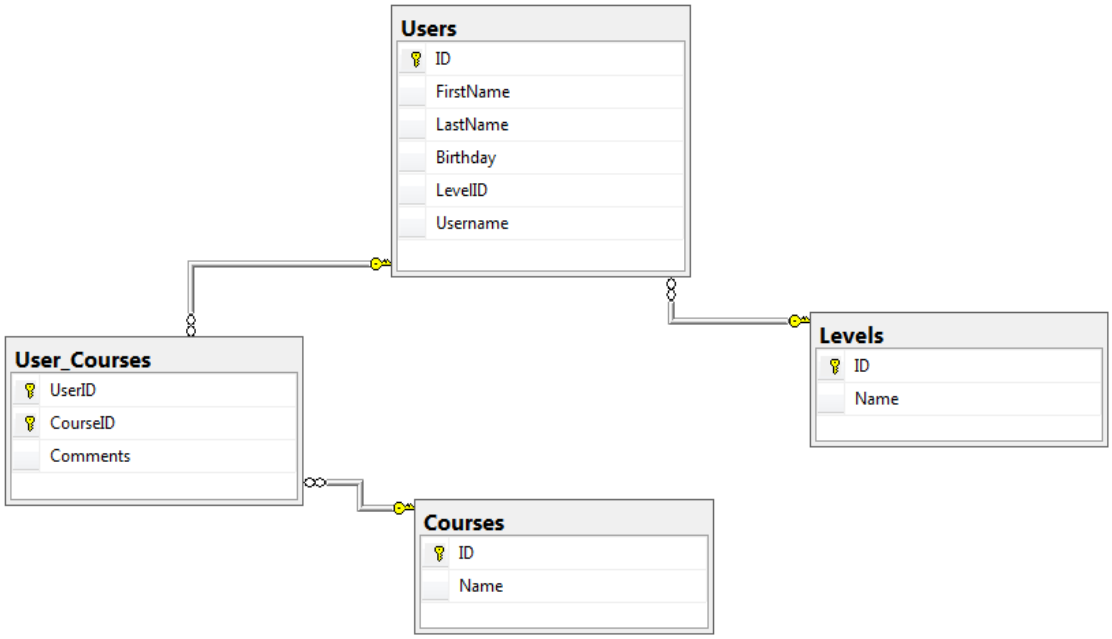

*Γράφημα 21. DELETE Hibernate and JDBC CPU Usage*

Σύγκριση Κεντρικής Μονάδας Επεξεργασίας (CPU) μεταξύ Hibernate και JDBC όταν διαγράφονται 50000 αρχεία (το Hibernate είναι στη χρονική περίοδο από 23:27 μέχρι 23:30 στην αρχή του διαγράμματος μιας και έχουν εκτελεστεί πρώτα αυτά του Hibernate (5000 και μετά 50000 εγγραφές) και μετά του JDBC (επίσης 5000 και 50000 εγγραφές) και η μονάδα που χρησιμοποιείται στους κάθετους άξονες των διαγραμμάτων είναι (%)).

Από τον πίνακα και τα σχήματα παραπάνω, βρήκαμε ότι υπάρχει μια σημαντική διαφορά στην απόδοση μεταξύ Hibernate και JDBC κατά τη διαγραφή αρχείων. Υπάρχουν κάποιες διαφορές στα διαγράμματα των Page Rate, της κεντρικής μονάδας επεξεργασίας (CPU), και το Logical I/O. Ο λόγος για τον οποίο υπάρχουν διαφορές είναι ότι, κατά τη διάρκεια των λειτουργιών διαγραφής, το Hibernate πρέπει να συγχρονίσει τα αμετάβλητα αντικείμενα που είναι στη μνήμη με τα δεδομένα της βάσης δεδομένων, το οποίο απαιτεί χρόνο, ενώ η JDBC δε χρειάζεται να κάνει αυτή τη δουλειά.

### 2.1.6 **Διάγραμμα της Βάσης Δεδομένων** (Database Diagram)

Ακολουθεί απεικόνιση του διαγράμματος της Βάσης Δεδομένων που χρησιμοποιήθηκε για τα παραδείγματα – πειράματα που πραγματοποιήθηκαν παραπάνω.




Σχήμα 9. Διάγραμμα της Βάσης Δεδομένων (DataBase Diagram)

### 2.1.7 **Συμπεράσματα**

Από τα παραπάνω αποτελέσματα απόδοσης, βρήκαμε ότι δεν υπάρχει σημαντική διαφορά στην εισαγωγή και στην επιλογή μεταξύ Hibernate και JDBC. Παρόλα αυτά, η απόδοση διαγραφής της JDBC είναι πολύ πιο γρήγορη από του Hibernate. Μέσω της ανάλυσης μεγεθών και δηλώσεων SQL παραγόμενων από το Hibernate, βρήκαμε ότι η μαζική διαγραφή στο Hibernate είναι διαφορετική από τη μαζική διαγραφή στη JDBC. Κατά τη διάρκεια της μαζικής διαγραφής στο Hibernate, το Hibernate παίρνει ένα αρχείο χρησιμοποιώντας SQL δήλωση ***select*** και μετά διαγράφει αυτό το αρχείο χρησιμοποιώντας SQL δήλωση ***delete***, και επαναλαμβάνει τις ίδιες λειτουργίες για κάθε αρχείο. Με άλλα λόγια, το Hibernate διαγράφει αρχεία ένα προς ένα. Ο λόγος που το Hibernate λειτουργεί έτσι είναι ότι το Hibernate πρέπει να συγχρονίσει τα δεδομένα εικονικής μηχανής Java (Java Virtual Machine - JVM) με τα δεδομένα της βάσης δεδομένων. Δεν παρέχουμε ενημέρωση σύγκρισης απόδοσης, γιατί το Hibernate δεν έχει λειτουργία μαζικής ενημέρωσης κι έτσι η διαδικασία ενημέρωσης του Hibernate θα ήταν η ίδια με τη διαδικασία διαγραφής του Hibernate, ψάχνοντας δεδομένα και ενημερώνοντας δεδομένα ένα προς ένα, το οποίο θα οδηγούσε σε δαπανηρή επιβάρυνση κατά τη διαχείριση μεγάλου αριθμού δεδομένων.

Εν συντομία, θα μπορούσαμε να καταλήξουμε στο παρακάτω συμπέρασμα με βάση την ανάλυση και τα αποτελέσματα των δοκιμών: η JDBC είναι η πιο αποδοτική προσέγγιση διαχείρισης βάσεων δεδομένων. Παρόλο που το Hibernate είναι πολύ πιο αργό από τη JDBC στη μαζική ενημέρωση και μαζική διαγραφή, η απόδοση της εισαγωγής και της επιλογής στο Hibernate είναι κοντά σε αυτήν της JDBC.

## 2.2 **Σύγκριση ευελιξίας και απλότητας** (Flexibility and Simplicity Comparison)

### 2.2.1 JDBC **και** Hibernate

Εδώ, θα συγκρίνουμε διαφορετικές πλευρές της JDBC και του Hibernate συμπεριλαμβανομένων της *συντήρησης (maintenance), preparedStatement, resultSet* και *του ελέγχου συναλλαγών (transaction control).*

#### 2.2.1.1 **Συντήρηση** (Maintenance)

Όπως γνωρίζουμε, αλλαγές μπορούν να συμβούν οποιαδήποτε στιγμή κατά τη διάρκεια ανάπτυξης λογισμικού. Μέχρι στιγμής, υπάρχουν πολλές προσεγγίσεις διαχείρισης αλλαγών στην ανάπτυξη λογισμικού. Όπως το πλαίσιο (framework), τα Πρότυπα Σχεδίασης (Design Patterns) [7], τα Πρότυπα Πυρήνα J2EE (J2EE Core Patterns) [6], το μοντέλο MVC (Model-View-Controller) και ούτω καθεξής. Αλλά η εφαρμογή JDBC σ' αυτές τις προσεγγίσεις έχει ακόμα ένα μεγάλο πρόβλημα αν θέλουμε να τροποποιήσουμε τα ονόματα των πεδίων, τα ονόματα πινάκων και να γυρίσουμε σε μια διαφορετική σχεσιακή βάση δεδομένων. Αφού η JDBC είναι γραμμένη με το χέρι, είναι στενά συνδεδεμένη με τη βάση δεδομένων. Οποιαδήποτε αλλαγή επιπέδου της βάσης δεδομένων που συμβαίνει θα έχει μεγάλο αντίκτυπο στον κώδικα JDBC. Επιπλέον, ακόμα και αν οι τροποποιήσεις στη JDBC τελειώσουν, κανείς δεν μπορεί να εγγυηθεί ότι έχουν γίνει σωστά γιατί είναι αδύνατο για το μεταφραστή Java να ελέγξει αν η δήλωση SQL στον κώδικα Java είναι σωστή κατά τη διάρκεια της μετάφρασης. Τα λάθη δεν εντοπίζονται παρά μόνο κατά την εκτέλεση. Για το λόγο αυτό, είναι σημαντικό οι λεπτομέρειες αμεταβλητότητας και τα θέματα αναπαράστασης εσωτερικών δεδομένων να απομακρύνονται από τη σχεσιακή βάση δεδομένων. Το Hibernate κάνει τις λεπτομέρειες αμεταβλητότητας αδιάλειπτες και έχει ένα απλό αντικειμενοστραφές API για να χειρίζεται την αποθήκευση δεδομένων. Έτσι, μέσω του Hibernate, ο προγραμματιστής της εφαρμογής δε χρειάζεται να ασχοληθεί με χαμηλού επιπέδου δομές μοντελοποίησης δεδομένων όπως γραμμές και στήλες και να χρειάζεται να τις μεταφράζει συνεχώς από τη μια μεριά στην άλλη. Αντ' αυτού, ο προγραμματιστής απελευθερώνεται από τη διαχείριση αυτών των λεπτομερειών χαμηλού επιπέδου για να συγκεντρωθεί σε θέματα που θα τον βοηθήσουν να παραδώσει την επιχειρησιακή εφαρμογή του. Ο προγραμματιστής της

εφαρμογής θα μπορούσε να χρησιμοποιήσει μια προσέγγιση άμεσης λειτουργίας (plug-and-play) όσον αφορά στην αποθήκευση δεδομένων και να αλλάξει παρόχους αποθηκευτικών μέσων και εκτελέσεις χωρίς να χρειάζεται να αλλάξει μια γραμμή από τον πηγαίο κώδικα της εφαρμογής. Από την άποψη αυτού του πλεονεκτήματος, το Hibernate μπορεί επίσης να επιλύσει το πρόβλημα φορητότητας αν χρειαστεί να μεταφερθούμε μεταξύ σχεσιακών βάσεων δεδομένων.

#### 2.2.1.2 PreparedStatement και ResultSet

Το Hibernate είναι μια ελαφριά περιληπτική ενσωμάτωση της JDBC. Στέλνει πάντα *preparedStatement* στη βάση δεδομένων. Έτσι, θα μπορούσε να εμποδίσει έναν προγραμματιστή από το να χρησιμοποιήσει κατά λάθος *preparedStatement* και *statement*. Εν τω μεταξύ, ακόμα κι αν ο προγραμματιστής χρησιμοποιεί *preparedStatement* αλλά δε χρησιμοποιεί *placehold* [4] ή ξεχάσει να το χρησιμοποιήσει, θα μπορούσαν πάλι να υπάρξουν κάποια προβλήματα απόδοσης. Η καλή χρήση της *preparedStatement* δεν είναι εύκολη στο περιβάλλον της επιχειρησιακής εφαρμογής. Όταν δημιουργούμε δηλώσεις SQL στην JDBC, ο όρος *where* είναι πάντα αβέβαιος εξαιτίας της πολύπλοκης επιχειρηματικής λογικής. Γι' αυτό, η σειρά της *placehold* στην *preparedStatement* είναι επίσης αβέβαιη. Παρόλα αυτά, μια *Placehold* στη JDBC είναι ευαίσθητη στη σειρά με την οποία εμφανίζονται στη συμβολοσειρά αναζήτησης (query string) όταν αντικαταστήσουμε την *placehold* με λεπτομερείς τιμές. Έτσι χρειάζεται να γράψουμε πολλούς σύνθετους *if-else* όρους για να βρούμε τη σωστή σειρά placehold. Για παράδειγμα,

```
String insert =
"select * from person where firstname=? And lastname=?"
try {
pstmt = conn.prepareStatement(insert);
pstmt.setString(1,    "firstnameValue");
pstmt.setString(2, "lastnameValue");
ResultSet rs = pstmt.executeQuery();
}
```

Έστω ότι υπάρχει ένα list box που έχει 10 δυνατότητες για επιλογή στο interface αναζήτησης από την πλευρά του πελάτη. Είναι εύκολο να διαχειριστούμε τέτοιου είδους αναζητήσεις. Αλλά θα χειροτερέψει αν υπάρχουν περισσότερα του ενός list box ή το list box υποστηρίζει πολλαπλή επιλογή. Πρέπει να είναι πολύ πολύπλοκος ο όρος *where* στη JDBC.

Παρόλα αυτά, αν χρησιμοποιήσουμε το Hibernate, μπορούμε να λύσουμε το πρόβλημα εύκολα σύμφωνα με την επιλογή του πελάτη. Για παράδειγμα,

```
String insert =
"select * from person where firstname=:first And lastname=:last"
 try {
pstmt  =  conn.prepareStatement(insert);
pstmt.setString("last",  "lastnameValue");
pstmt.setString("first",
"firstnameValue");
ResultSet rs = pstmt.executeQuery();
}
```

Επειδή η ***Placehold*** στο Hibernate δεν επηρεάζεται από τη σειρά με την οποία εμφανίζονται στη συμβολοσειρά αναζήτησης, όταν αντικαταστήσουμε τη ***placehold*** με λεπτομερή τιμή, δε χρειάζεται να ανησυχούμε καθόλου για τη σειρά της, και θα μπορούσαμε να δημιουργήσουμε εύκολα τη δήλωση SQL δυναμικά. Έτσι, παρέχεται πολύ μεγαλύτερη ευελιξία στη ***preparedStatement*** και τα σχετικά της θέματα από ότι στη JDBC.

Και το Hibernate και η JDBC θα μπορούσαν να επιστρέψουν ένα σύνολο αποτελεσμάτων από τη βάση δεδομένων. Αν το σύνολο των αποτελεσμάτων είναι τεράστιο, η επίδειξη όλου του συνόλου των αποτελεσμάτων στον πελάτη είναι άσχημη γιατί οδηγεί σε απάντηση μεγάλης διάρκειας προς τον πελάτη. Συνήθως, χρησιμοποιούμε τη μέθοδο πολλαπλών σελίδων (multi-page method) για να δείξουμε το σύνολο των αποτελεσμάτων ανά σελίδα. Όταν ο κόσμος αναζητά τον αριθμό σελίδας, η JDBC υπολογίζει το όριο του συνόλου των αποτελεσμάτων ανάλογα με τον αριθμό σελίδων που αναζητούνται και επιστρέφει στον πελάτη ένα μέρος από το σύνολο των αποτελεσμάτων. Αλλά το Hibernate απλοποιεί αυτή τη διαδικασία χρησιμοποιώντας ένα JDBC σύνολο αποτελεσμάτων με δυνατότητα μετακίνησης (scrollable result set) [4]. Μπορούμε να καθορίσουμε όρια στο σύνολο των αποτελεσμάτων (ο μέγιστος αριθμός γραμμών που θέλουμε να ανακτήσουμε και/ή πρώτη γραμμή που θέλουμε να ανακτήσουμε). Ακολουθεί δείγμα κώδικα:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(lO);
List cats = q.listQ;
```

#### 2.2.1.3 **Έλεγχος Συναλλαγής** (Transaction Control)

Όταν χρησιμοποιούμε το JDBC για την ενημέρωση των δεδομένων, η βάση δεδομένων θα κλειδώσει τα δεδομένα σύμφωνα με διαφορετικά επίπεδα απομόνωσης και με διαφορετικά κλειδώματα πολλαπλής διακριτότητας (granularity locks), όπως ορίζεται από το διαχειριστή της βάσης δεδομένων, η οποία υφίσταται επεξεργασία στη διαβάθμιση της βάσης δεδομένων. Ενώ το Hibernate χρησιμοποιεί αισιόδοξο κλείδωμα με εκδόσεις σε αδρανοποίηση (timestamps) ώστε να διατηρήσει την απομόνωση των επιχειρηματικών διαδικασιών κατά την διαβάθμιση της εφαρμογής και όχι στη διαβάθμιση της βάσης δεδομένων. Δηλαδή, όλη η ενημέρωση αντικειμένου είναι τελικά σε Java Virtual Machine (JVM). Κατά την εφαρμογή των αλλαγών στη βάση δεδομένων, το Hibernate, θα ελέγξει αν υπάρχουν άλλα threads που έχουν ενημερώσει το ίδιο αντικείμενο. Αυτή η διαδικασία ελέγχου εφαρμόζεται με τη χρήση έκδοσης ή timestamps ελέγχου. Οι αριθμοί έκδοσης ή τα timestamps ενημερώνονται αυτόματα από το Hibernate. Ως εκ τούτου, κατά τη διάρκεια μιας συναλλαγής του Hibernate, δεν κλειδώνει τη βάση δεδομένων μέχρι η πραγματική δέσμευση να συμβεί. Θα μπορούσε να διασφαλίσει ότι οι μακρόχρονες συναλλαγές δεν επηρεάζουν τις συναλλαγές άλλων threads. Φυσικά, αν υπάρχουν πάρα πολλές πράξεις ενημέρωσης σε μια συναλλαγή και η σύγκλιση συμβαίνει συχνά, το ποσοστό επιτυχίας της δέσμευσης θα μειωθεί. Φαίνεται ότι το Hibernate παρέχει πολύ μεγαλύτερη ευελιξία στο χειρισμό των συναλλαγών από ό, τι το JDBC.

#### 2.2.1.4 **Συμπέρασμα**

Από τα παραπάνω, έχουμε καταλήξει στο συμπέρασμα ότι το Hibernate είναι πιο ευέλικτο από ό, τι το JDBC. Το Hibernate είναι ένα αρκετά λεπτό περίβλημα στην κορυφή του JDBC, γεφυρώνοντας το χάσμα στην αποσαφήνιση των αντικείμενων Java. Αν και το JDBC είναι εύκολο στη χρήση, αυτό δεν σημαίνει ότι είναι κατάλληλο για την ανάπτυξη των εφαρμογών για επιχειρήσεις, λόγω της περιορισμένης ευελιξίας του. Επίσης, είναι δύσκολο να γίνει καλή χρήση του JDBC. Μάλιστα, η καλή χρήση του JDBC και η βελτιστοποίηση των επιδόσεων είναι πολύ περίπλοκα, καθώς περιλαμβάνει πολύ τεχνολογία JDBC και τεχνολογία βάσης δεδομένων. Για παράδειγμα, εσωτερικοί σύνδεσμοι, αριστερά/δεξιά εξωτερικοί σύνδεσμοι, ενημέρωση των batch και ούτω καθεξής. Όλες οι τεχνολογίες απαιτούν από τον προγραμματιστή να έχει μια βαθιά κατανόηση του JDBC και των βάσεων δεδομένων. Εν τω μεταξύ, σύνθετες δηλώσεις SQL σε κώδικα Java είναι δύσκολο να διατηρηθούν. Το Hibernate κάνει αυτά τα προβλήματα απλά. Το Hibernate επίσης δεν απαιτεί από τον προγραμματιστή να γνωρίζει το JDBC και τη βάση που χρησιμοποιείται. Ο προγραμματιστής χρησιμοποιεί μόνο την αντικειμενοστραφή μέθοδο για αμετάβλητα δεδομένα άμεσα και εύκολα.

# ΚΕΦΑΛΑΙΟ 3ο – ΣΥΜΠΕΡΑΣΜΑΤΑ ΚΑΙ ΜΕΛΛΟΝΤΙΚΕΣ ΒΕΛΤΙΩΣΕΙΣ

Στην παρούσα εργασία, παρουσιάζουμε ένα πλαίσιο λογισμικού αμεταβλητότητας: το Hibernate. Συνήθως, οι συγκρίσεις μεταξύ των συστημάτων είναι δύσκολες. Προσπαθούμε να συγκρίνουμε το Hibernate με το JDBC και να συζητήσουμε τα πλεονεκτήματα και τα μειονεκτήματά τους στην απόδοση, την απλότητα, την ευελιξία και ούτω καθεξής. Με βάση τα πειράματα σε αυτήν την εργασία, έχουμε μια κατά προσέγγιση εκτίμηση των επιδόσεων μεταξύ JDBC και Hibernate. Μέσα από τις συγκρίσεις, την ανάλυση και τις εφαρμογές σε αυτή την εργασία, εξάγουμε το συμπέρασμα ότι το Hibernate δεν είναι μόνο ένα καλύτερο πλαίσιο λογισμικού αμεταβλητότητας από την εφαρμογή Enterprise JavaBean ή το JDBC απ’ ευθείας, αλλά παρέχει και περισσότερες επιλογές για την ανάλυση και το σχεδιασμό λογισμικού, όπως ένα Object Oriented Analysis and Designed (OOAD) και με γνώμονα τα δεδομένα μοντέλο. Μπορούμε επίσης να συμπεράνουμε ότι η εισαγωγή του Hibernate στο EJB θα είναι μια καλύτερη λύση για τις εφαρμογές των επιχειρήσεων όσον αφορά την περαιτέρω ανάλυση και τις εφαρμογές, το οποίο όχι μόνο θα κάνει καλή χρήση των πλεονεκτημάτων του EJB, όπως η διεκπεραίωση υποθέσεων, αλλά θα βελτιώνει την απόδοση και θα μειώνει την πολυπλοκότητα των προβλημάτων στο EJB.

Ωστόσο, τίποτα δεν είναι τέλειο, το ίδιο το Hibernate έχει κάποια μειονεκτήματα. Δεν είναι κατάλληλο για ποσοτική ενημέρωση ή ποσοτική διαγραφή λόγω της αναποτελεσματικής απόδοσής του. Το Hibernate υποστηρίζει μόνο τα δεδομένα που δραστηριοποιούνται στον πίνακα και όχι τις διαδικασίες προβολής ή αποθήκευσης. Δηλαδή, δεν μπορεί να υποστηρίξει τις προηγμένες λειτουργίες στη βάση δεδομένων. Επιπλέον, παρόλο που έχει ευέλικτη σχεσιακή χαρτογράφηση, δεν υπάρχει πρώτης κλάσης σε πολλαπλών επιπέδων σχέση. Υπάρχουν δύο είδη μνήμης cache στην εφαρμογή του server, μία είναι η JCS (Java Caching Cystem) [17], και η άλλη είναι η EJB cache στον EJB Container. Έτσι πρέπει να εξετασθούν χωριστά. Θα μπορούσε να μειώσει την επικάλυψη μεταξύ των δύο cache και να εξοικονομήσει χώρο στη μνήμη, αν και οι δύο μπορούσαν να ενσωματωθούν μαζί.

Σε μελλοντική εργασία, θα μπορούσαμε να επεκτείνουμε την έρευνά μας στο περιβάλλον ταυτοχρονισμού. Θα μπορούσαμε επίσης να διεξάγουμε την ίδια δοκιμή σε διάφορα είδη διακομιστών βάσεων δεδομένων, υπό διαφορετικά λειτουργικά συστήματα. Συγκρίνοντας τις επιδόσεις με πιο πολύπλοκες σχέσεις χαρτογράφησης που θα μπορούσαν επίσης να γίνουν.

# ΑΝΑΦΟΡΕΣ – ΠΗΓΕΣ


[1] Scott W. Ambler, *The Object-Relational Impedance Mismatch*

http://www.agiledata.org/essays/impedanceMismatch.html (Last accessed Nov, 5, 2011)

[2] Scott W. Ambler, *Implementing Referential Integrity and Shared Business Logic*

http://www.agiledata.org/essays/referentialIntegrity.html (Last accessed Nov, 5, 2011)

[3] *Scott W. Ambler,* Concurrency Control

http://www.agiledata.org/essays/concurrencyControl.html (Last accessed Nov, 5, 2011)

[4] JDBC 2.0 Reference

http://javasun.com/docs/books/tutorial/idbc/jdbc2dot0/index.html (Last accessed Nov, 5, 2011)

[5] *Gavin King,* Hibernate2 Reference Documentation

http://www.hibernate.org/docs (Last accessed Nov, 5, 2011)

[6] J2EE Core Patterns Website

http://javasun.com/blueprints/corej2eepatterns (Last accessed Nov, 5, 2011)

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison Wesley. (October 1994)

[8] Dennis M. Sosnoski, *Java programming dynamics, Part 2: Introducing reflection*

http://www.ibm.com/developerWorks/java (Last accessed Nov, 5, 2011)

[9] EJB restriction Website

http://developer.javasun.com/developer/restrictedpatterns/AggregateEntity (Last accessed Nov, 5, 2011)

[10] J2EE Website

http://Javasun.com/j2ee (Last accessed Nov, 5, 2011)

[11] Ed Roman, Scott Ambler and Tyler Jewell, *Mastering Enterprise JavaBean, Second Edition*, Wiley Computer Publishing (2002)

[12] *Scott W. Ambler,* Mapping Objects To Relational Databases, An AmbySoft Inc. White Paper

SIGS Books/Cambridge University Press (1998)

[13] *Scott W. Ambler,* Encapsulating Database Access

http://www.agiledata.org/essays/implementationStrategies.html (Last accessed Nov, 5, 2011)

[14] Code Generation Library(cglib) Website

http://cglib.sourceforge.net/ (Last accessed Nov, 5, 2011)

[15] Java World Website

http://www.javaworld.com/javaworld/jw-05-2003/jw-0509-finalists.html (Last accessed Nov, 5, 2011)

[16] GNU Lesser General Public License Website

http://www.gnu.org/licenses/lgpl.html (Last accessed Nov, 5, 2011)

[17] Java Caching System(JCS) Website

http://jakarta.apache.org/turbine/jcs (Last accessed Nov, 5, 2011)

- *Java Persistence with Hibernate* by Christian Bauer, Gavin King, Manning Publications
- *Hibernate Search in Action* by Emmanuel Bernard; John Griffin, Manning Publications
- *Applied Enterprise Java Beans Technology* by Kevin Boone, Prentice Hall
- *http://www.hibernate.org*
- *http://javasun.com*

# ΛΙΣΤΑ ΣΥΝΤΟΜΟΓΡΑΦΙΩΝ

| | |
|---|---|
| BMP | Bean-Managed Persistence |
| CGLIB | Code Generation Library |
| CMP | Container-Managed Persistence |
| DAO | Data Access Object |
| EJB | Enterprise JavaBean |
| HQL | Hibernate Query Language |
| J2EE | Java 2 Platform, Enterprise Edition |
| JCS | Java Caching Cystem |
| JDBC | Java Database Connectivity |
| JNDI | Java Naming and Directory Interface |
| JTA | Java Transaction API |
| JVM | Java Virtual Machine |
| LGPL | Lesser General Public License |
| MVC | Model-View-Controller |
| OOAD | Object Oriented Analysis and Design |
| OODBMS | Object-Oriented Database Management Systems |
| O/R Mapping | Object-Relational Mapping |
| OID | Objects Identifier |
| RMI | Remote Method Invocation |
| RPC | Remote Procedure Calling |

# ΛΙΣΤΑ ΓΡΑΦΗΜΑΤΩΝ

*Γράφημα 1. Γράφημα χρόνου εκτέλεσης για το Hibernate και το JDBC σύμφωνα με τον αριθμό δημιουργούμενων στοιχείων.*
*Γράφημα 2. INSERT Hibernate Logical I/O*
*Γράφημα 3. INSERT JDBC Logical I/O*
*Γράφημα 4. INSERT Hibernate CPU Usage*
*Γράφημα 5. INSERT JDBC CPU Usage*
*Γράφημα 6. INSERT Hibernate Memory Usage*
*Γράφημα 7. INSERT JDBC Memory Usage*
*Γράφημα 8. INSERT Hibernate Pages per Second*
*Γράφημα 9. INSERT JDBC Pages per Second*
*Γράφημα 10. Γράφημα χρόνου εκτέλεσης για το Hibernate και το JDBC σύμφωνα με τον αριθμό των επιλεγμένων στοιχείων.*
*Γράφημα 11. SELECT Hibernate and JDBC Logical I/O*
*Γράφημα 12. SELECT Hibernate and JDBC CPU Usage*
*Γράφημα 13. SELECT Hibernate Memory Usage*
*Γράφημα 14. SELECT JDBC Memory Usage*
*Γράφημα 15. SELECT Hibernate and JDBC Pages per Second*
*Γράφημα 16. DELETE Hibernate and JDBC Logical I/O*
*Γράφημα 17. DELETE Hibernate Memory Usage*
*Γράφημα 18. DELETE JDBC Memory Usage*
*Γράφημα 19. DELETE Hibernate Pages per Second*
*Γράφημα 20. DELETE JDBC Pages per Second*
*Γράφημα 21. DELETE Hibernate and JDBC CPU Usage*

# ΛΙΣΤΑ ΣΧΗΜΑΤΩΝ

*Σχήμα 1. Σχέση των Principal, School και Student*

*Σχήμα 2. Πολλά-προς-πολλά σχέση ανάμεσα στα Task και Student*

*Σχήμα 3. Εκτέλεση μιας σχέσης πολλά-προς-πολλά σε μια σχεσιακή βάση δεδομένων*

*Σχήμα 4. Διάγραμμα DAO και JDBC*

*Σχήμα 5. Διάγραμμα Βαθμίδας Αμεταβλητότητας*

*Σχήμα 6. Μια υψηλού επιπέδου άποψη της αρχιτεκτονικής του Hibernate*

*Σχήμα 7. Αρχιτεκτονική του Hibernate κατά την εκτέλεση*

*Σχήμα 8. JDBC διάγραμμα καταρράκτη*

*Σχήμα 9. Διάγραμμα της Βάσης Δεδομένων (DataBase Diagram)*

# ΛΙΣΤΑ ΠΙΝΑΚΩΝ

*Πίνακας 1. Ρυθμίσεις δοκιμής για Απόδοση του JDBC και του Hibernate*

*Πίνακας 2. Σύγκριση απόδοσης της εντολής Insert, μεταξύ του JDBC και του Hibernate*

*Πίνακας 3. Σύγκριση Απόδοσης της εντολής Select, μεταξύ του JDBC και του Hibernate*

*Πίνακας 4. Σύγκριση Απόδοσης της εντολής Delete, μεταξύ του JDBC και του Hibernate*

# ΠΑΡΑΡΤΗΜΑ

## Hibernate performance test code

```
Package com.bruce;
import java.util.*;
Import net.sf.hibernate.*;
import org.apache.log4j.*;

public class TestUsersHibernate {
 //Add Class TestUsersHibernate to log4j
 Final private static Logger log = Logger.getLogger(TestUsersHibernate.class);
 //Test Hibernate insert performance
 public static long testInsert(int count) {
  SessionFactory sf = null;
  Session s = null;
  Transaction tx = null;
  Users c = null;
  long t0 = 0;            //Execution begin time
  long tl = 0;            //Execution end time
  try {
     sf = HibernateSessionFactory.getSessionFactory(); //get Hibernate SessionFactory
     s = sf.openSession();                              //get Hibernate Session
     System.out.println("Start Inserting Records with Hibernate...");
     t() = System.currentTimeMillis();
     tx = s.beginTransaction();                         //Begin the transaction
     //Detailed inserting process
     for (int i = 0; i < count; i++) {
        c = new User();
        c.setId(String.valueOf(System.currentTimeMillis() + i));
        c.setFirstName("A" + i);
        c.setLastName('A');
        c.setBirthday(1/1/00);
```

```
            c.setLevelID(0);
            c.setUsername('A');
            c.save(c);
        } //End of for-loop
        s.flush();
        tx.commit();        //Commit the transaction
        t1 = System.currentTimeMillis();
        System.out.println("Finished Inserting Records with Hibernate: " +
                    (t1 – t0) + "ms");
    }
    catch (HibernateException e) {
        log.error(e.getMessage());          //Log the error if error happens
    }
    finally {
      if (s != null) {
        try {
          s.close();                    //Close Hibernate Session when finish the transaction
        }
        catch (Exception e) {}
        }
      }
      return (t1-t0);          //Calculate the execution time and return this time
}                                       //End of testInsert Method

//Test Hibernate select performance
public static long testReadAll(int mycount) {
    SessionFactory sf = null;
    Session s = null;
    User c = null;
    long t0 = 0;            //Execution begin time
    long t1 = 0;            //Execution end time
    try {
        sf = HibernateSessionFactory.getSessionFactory(); //get Hibernate SessionFactory
        s = sf.openSession();                                   //get Hibernate Session
```

```
        System.out.println("Start Reading Records with Hibernate...");
        t0= System.currentTimeMillis();
        //Detailed select process
        Query q = s.createQuery("select user from User as user"); //Use HQL to query
        List 1 = q.list();
        for (int i=0; i< l.size(); i++) {
           c = (User) 1.get(i);
        }
        t1 = System.currentTimeMillis();
        System.out.println("Finished Reading Records with Hibernate: " + (t1 – t0) +
                           "ms");
        }
        catch (HibernateException e) {
           log.error(e.getMessage());      //Log the error if error happens
        }
        finally {
           if (s != null) {
                try {
                   s.close();                //Close Hibernate Session
                }
                catch (Exception e) {}
                }
        }
        return (t1-t0);                      //Calculate the execution time and return this time
}                                            //End of testReadAll Method

//Test Hibernate delete performance
public static long deleteAll(int mycount) {
    SessionFactory sf = null;
    Session s = null;
    Transaction tx = null;
    Users c = null;
```

```
    long t0 = 0;          //Execution begin time
    long t1 = 0;          //Execution end time
    try {
      sf = HibernateSessionFactory.getSessionFactoryO; //get Hibernate SessionFactory
      s = sf.openSession();                            //get Hibernate Session
      tx = s.beginTransaction();                       //Begin the transaction
      System.out.println("Start deleting Records with Hibernate...");
      t0= System.currentTimeMillis();
        s.delete("from Users");                        //delete process
        s.flush();
        tx.commit();                                   //Commit the transaction
        t1 = System.currentTimeMillis();
        System.out.println("Finished Deleting Records with Hibernate: " + (t1 – t0) +
                           "ms");
    }
    catch (HibernateException e) {
        log.error(e.getMessage());              //Log the error if error happens
    }
    finally {
        if (s != null) {
            try {
                s.close();              //Close Hibernate Session when finish the transaction
            }

            catch (Exception e) {}

            }

    }
    return (t1-t0);                     //Calculate the execution time and return this time
    }                                   //End of deleteAll Method
}                                       //End of Class TestCatHibernate
```

## JDBC performance test code

```
package com.bruce;
import java.sql.*;
import org.apache.log4j.*;

public class TestUsers {
 //Add Class TestUsers to log4j
 final private static Logger log = Logger.getLogger(TestUsers.class);
 //Test JDBC insert performance
 public static long testInsert(int count) {
     Connection conn = null;
     PreparedStatement pstmt = null;
     long t0 = 0;          //Execution begin time
     long t1 = 0;          //Execution end time
     try {
          Class.forFirstName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
          conn = DriverManager.getConnection(

                  jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=test;SelectMet
                  hod=Cursor;user=sa;password=sa");
          System.out.println("Start Inserting Records with JDBC...");
          t0 = System.currentTimeMillis();
           conn.setAutoCommit(false);          //set automatic-commit false
           //Detailed insert process
           pstmt = conn.prepareStatement(
                   "insert into users(id,fisrtname,lastname,birthday,leveled,username)
                   values(?,?,?,?)");
           for (int i = 0; i < count; i++) {
               pstmt.setString(1, String.valueOf(System.currentTimeMillis() + i));
               pstmt.setString(2, "A" + i);
               pstmt.setString(3, "A");
               pstmt.setFloat(1/1/00);
```

```
            pstmt.setFloat(0);
            pstmt.setString('A');
            pstmt.addBatch();
        }
        pstmt.executeBatch();
        pstmt.close();
        conn.commit();          // Commit the transaction
        t1 = System.currentTimeMillis();
        System.out.println("Finished Creating Records with JDBC: " + (tl - tO) + "ms");
 }
 catch (Exception e) {
        log.error(e.getMessage());     //Log the error if error happens
 }
 finally {
   if (conn != null) {
        try {
        conn.close();              //Close connection
        }
        catch (Exception e) {}
        }
 }
 return (t1-t0);                   //Calculate the execution time and return this time
 }                                 //End of testCreate Method

//Test JDBC select performance
public static long testSelectAll(int mycount) {
   Connection conn = null;
   PreparedStatement pstmt = null;
   long t0 = 0;          //Execution begin time
   long t1 = 0;          //Execution end time
   try {
        Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
        conn = DriverManager.getConnection(
```

```
            "jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=test;SelectMeth
            od=Cursor;user=sa;password=sa");
    System.out.println("Start Selecting Records with JDBC...");
    t0 = System.currentTimeMillis();
    //Detailed select process
    pstmt = conn.prepareStatement("select * from users");
    ResultSet rset = pstmt.executeQueryO;
    while (rset.next()) {
          String id = rset.getString(l);
          String firstname = rset.getString(2);
          String lastname = rset.getString(3);
          float birthday = rset.getFloat(4);
          float levelid = rset.getFloat(5);
          String username = rset.getString(6);
       }
rset.close();
pstmt.close();
t1 = System.currentTimeMillis();
  System.out.println("Finished Selecting Records with JDBC: " + (t1 – t0) + "ms");
  }
  catch (Exception e) {
       log.error(e.getMessage());          //Log the error if error happens
  }
  finally {
      if (conn != null) {
          try {
       conn.close();                      //Close connection
          }
          catch (Exception e) {}
          }
      }
  return (t1-t0);                        //Calculate the execution time and return this time
  }                                      //End of testSelectAll Method
```

```
//Test JDBC delete performance
public static long deleteAll(int mycount) {
    Connection conn = null;
    PreparedStatement pstmt = null;
    long t0 = 0;          //Execution begin time
    long t1 = 0;          //Execution end time
    try {
        Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
        conn = DriverManager.getConnection(
                "jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=test;SelectMethod
                =Cursor;user=sa;password=sa");
        System.out.println("Start Deleting Records with JDBC...");
        t0 = System.currentTimeMillis();
        //Detailed delete process
        pstmt = conn.prepareStatement("delete from user");
        pstmt.execute();
        pstmt.close();
        t1 = System.currentTimeMillis();
        System.out.println("Finished Deleting Records with JDBC: " + (t1 – t0) + "ms");
  }
  catch (Exception e) {
      log.error(e.getMessage());       //Log the error if error happens
  }
  finally {
      if (conn != null) {
        try {
            conn.close();        //Close connection
        }
      catch (Exception e) {}
      }
  }
  return (t1-t0);                    //Calculate the execution time and return this time
  }                                   //End of deleteAll Method
}                                     //End of Class TestUsers
```

## Client Code for comparison Hibernate and JDBC

```
package com.bruce;

import net.sf.hibernate.*;
import org.apache.log4j.*;

public class Main {
   //Add Class Main to log4j
   final private static Logger log = Logger.getLogger(Main.class);

   public static void main(String[] args) throws Exception {
        long s = 0;
        //loop 11 times and drop the first loop
        for(int i=l;i<12;i++){
             long t = TestUsers.testInsert(Integer.parseInt(args[0]));
             if(i!=l){
                  s = s + t;
             }
        } //End of for-loop
        log.info("Average time is " + s/10);
       } //End of main Method
}
```

## Cache Configuration file for Hibernate

File name: cache.ccf

```
# DEFAULT CACHE REGION (memory cache)
jcs.default=DC
jcs.default.cacheattributes=org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=1000
jcs.default.cacheattributes.MemoryCacheName=org.apache.jcs.engine.memory.lru.LRU
MemoryCache
jcs.default.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.default.elementattributes.IsEternal=false
jcs.default.elementattributes.MaxLifeSeconds=240
jcs default.elementattributes.IdleTime= 1800
jcs.default.elementattributes.IsSpool=true
jcs.default.elementattributes.IsRemote=false
jcs.default.elementattributes.IsLateral=false

# System CACHE REGION
#jcs.system.groupIdCache=DC
#jcs.system.groupIdCache.cacheattributes=org.apache.jcs.engine.CompositeCacheAttribu
tes
#jcs.system.groupIdCache.cacheattributes.MaxObjects=10000
#jcs.system.groupIdCache.cacheattributes.MemoryCacheName=org.apache.jcs.engine.
memory.lru.LRUMemoryCache

#Auxiliary CACHE (disk cache)
#jcs.auxiliary.DC=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
#jcs.auxiliary.DC.attributes=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttr
ibutes
#jcs.auxiliary.DC.attributes.DiskPath=cache
```