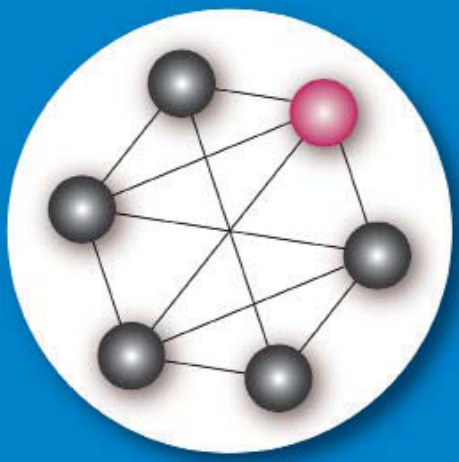


DBTechNet

DBTech VET

SQL Transactions

Θεωρία και Ασκήσεις
Πρακτικής Εφαρμογής

Στην ελληνική

Lifelong
Learning
Programme

This publication has been developed in the framework of the project DBTech VET Teachers (DBTech VET). Code: 2012-1-FI1-LEO05-09365.

DBTech VET is Leonardo da Vinci Multilateral Transfer of Innovation project, funded by the European Commission and project partners.

www.DBTechNet.org
DBTech VET

SQL Transactions – Theory and Hands-on Exercises
Version 1.1 of the first edition 2013

Authors: Martti Laiho, Dimitris A. Dervos, Kari Silpiö
Production: DBTech VET Teachers project

ISBN 978-952-93-2420-0 (paperback)
ISBN 978-952-93-2421-7 (PDF)

Περιεχόμενα

Μέρος 1: SQL Συναλλαγή – Λογική Μονάδα Εργασίας

1.1 Εισαγωγή στις συναλλαγές

Στην καθημερινή ζωή, οι άνθρωποι διεκπεραιώνουν/διεξάγουν διαφορετικά είδη επιχειρηματικών συναλλαγών, όπως: αγορά προϊόντων, κράτηση ταξιδιών, αλλαγή ή ακύρωση παραγγελιών, αγορά εισιτηρίων για συναυλίες, πληρωμή ενοικίου και λογαριασμών ηλεκτρικού ρεύματος, τιμολόγια ασφάλισης/ασφαλιστικής κάλυψης/ασφαλιστήρια, κλπ. Οι συναλλαγές δεν αφορούν μόνο σε υπολογιστές, φυσικά. Κάθε είδος της ανθρώπινης δραστηριότητας που περιλαμβάνει μία λογική μονάδα εργασίας, που (σημαίνει ότι) είτε πρέπει να εκτελεστεί στο σύνολό της ή να ακυρωθεί στο σύνολό της αποτελεί μια συναλλαγή.

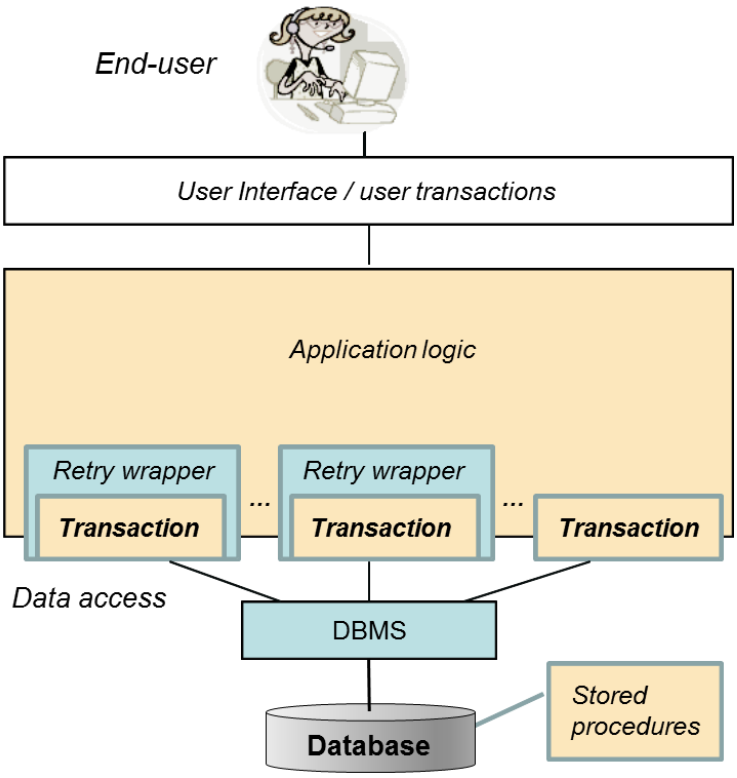
Σχεδόν όλα τα πληροφοριακά συστήματα χρησιμοποιούν τις υπηρεσίες κάποιου Συστήματος Διαχείρισης Βάσεων Δεδομένων (ΣΔΒΔ) για την αποθήκευση και ανάκτηση δεδομένων. Τα σύγχρονα ΣΔΒΔ είναι τεχνικά εξελιγμένα εξασφαλίζοντας την ακεραιότητα των δεδομένων στις βάσεις δεδομένων τους και παρέχουν γρήγορη πρόσβαση στα δεδομένα ακόμα και από πολλούς ταυτόχρονους χρήστες. Επίσης, προσφέρουν αξιόπιστες υπηρεσίες στις εφαρμογές για τη διαχείριση της ανθεκτικότητας των δεδομένων (persistency of data), αλλά μόνο αν οι εφαρμογές χρησιμοποιούν αυτές τις αξιόπιστες υπηρεσίες κατάλληλα. Αυτό εξασφαλίζεται αν υλοποιηθούν τα τμήματα της αρχιτεκτονικής λογισμικού για τη **πρόσβαση στα δεδομένα** χρησιμοποιώντας **συναλλαγές βάσεων δεδομένων**

Η ακατάλληλη χρήση των συναλλαγών στις εφαρμογές λογισμικού, μπορεί να οδηγήσει (α) να χαθούν παραγγελίες ή πληρωμές πελατών και αποστολές προϊόντων (στην περίπτωση ενός e-shop), (β) σε αποτυχίες ή διπλοκρατήσεις στην κράτηση θέσεων για τρένα ή αεροπλάνα, (γ) σε χαμένες καταχωρήσεις κλήσεων σε κέντρα αντιμετώπισης καταστάσεων έκτακτης ανάγκης, κτλ. Τέτοια περιστατικά συμβαίνουν συχνά, αλλά συνήθως οι υπεύθυνοι τα αποκρύπτουν από το κοινό. Ο στόχος του DBTech VET είναι να ορίσει ένα πλαίσιο βέλτιστων πρακτικών και μεθοδολογιών για την αποφυγή παρόμοιων άτυχων περιστατικών

Οι συναλλαγές, όσον αφορά τη διαχείριση του περιεχομένου μίας βάσης δεδομένων, είναι τμήματα (μονάδες) εργασιών πρόσβασης στα δεδομένα που έχουν τη δυνατότητα (σε περίπτωση αποτυχίας) να επανέλθουν στην αρχική τους κατάσταση. Επίσης, περιλαμβάνουν τρόπους επαναφοράς όλης της βάσης δεδομένων σε περίπτωση κατάρρευσης/πτώσης του συστήματος, όπως παρουσιάζεται στο Παράρτημα 3. Τέλος, προσφέρουν βασικές γνώσεις για τη διαχείριση του ταυτοχρονισμού σε περιβάλλοντα πολυχρησίας.

1.2 Έννοιες Πελάτη/Διακομιστή σε Περιβάλλον SQL

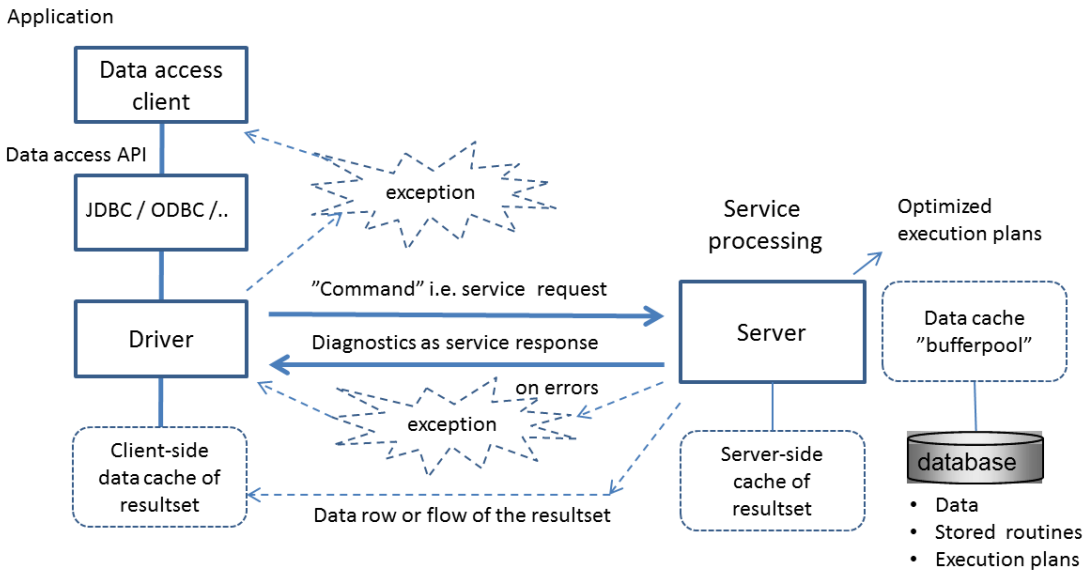
Σε αυτή τη διδακτική παρουσίαση, εστιάζουμε στην πρόσβαση των δεδομένων με χρήση **SQL συναλλαγών** εκτελώντας τον SQL κώδικα διαδραστικά (μέσα από το περιβάλλον ενός ΣΔΒΔ), και γνωρίζοντας ότι η πρόσβαση δεδομένων μέσα από προγραμματιστικό περιβάλλον ακολουθεί ένα ελαφρά διαφορετικό μοντέλο, το οποίο παρουσιάζουμε μέσω ενός παραδείγματος στο Παράρτημα 2.



Εικόνα 1 Η θέση των SQL συναλλαγών σε μία εφαρμογή

Η Εικόνα 1.1 παρουσιάζει μία απλουστευμένη άποψη της αρχιτεκτονικής μίας κλασικής εφαρμογής Βάσεων Δεδομένων, τοποθετώντας τις συναλλαγές βάσεων δεδομένων σε ξεχωριστό επίπεδο από το επίπεδο διεπαφής χρήστη. Από την οπτική γωνία του τελικού χρήστη, η εφαρμογή εξυπηρετεί τις περιπτώσεις χρήσης υλοποιώντας τες ως **συναλλαγές χρηστών**. Μία απλή συναλλαγή χρήστη μπορεί να περιλαμβάνει πολλαπλές SQL συναλλαγές, κάποιες από τις οποίες περιλαμβάνουν συναλλαγές ανάκτησης, και συνήθως η τελευταία συναλλαγή στη σειρά ενημερώνει τα περιεχόμενα της βάσης δεδομένων. **Οι ρουτίνες επανεκτέλεσης** αποτελούν το μέσο για την εφαρμογή προγραμματιστικών ενεργειών επανεκτέλεσης, στην περίπτωση αποτυχίας του ταυτοχρονισμού στις SQL συναλλαγές.

Για να κατανοήσουμε σωστά τις SQL συναλλαγές, θα πρέπει να συμφωνήσουμε σε μερικές βασικές έννοιες σχετικές με την επικοινωνία πελάτη διακομιστή. Μία εφαρμογή για να αποκτήσει πρόσβαση σε μία βάση δεδομένων, θα πρέπει πρώτα να οριστεί μία σύνδεση της εφαρμογής με τη βάση δεδομένων, η οποία θα ορίσει το πλαίσιο μίας ***SQL συνεδρίας***. Για απλότητα, μία SQL συνεδρία θεωρείται ότι περιλαμβάνει τον ***SQL πελάτη***, και ο διακομιστής της βάσης δεδομένων περιέχει τον ***SQL διακομιστή***. Από την πλευρά του διακομιστή της βάσης δεδομένων, η εφαρμογή χρησιμοποιεί υπηρεσίες βάσεων δεδομένων, σε μοντέλο πελάτη/διακομιστή, περνώντας **SQL εντολές** ως παραμέτρους σε συναρτήσεις/μεθόδους μέσω ενός ειδικού API (προγραμματιστική διεπαφή)[1] για την πρόσβαση σε δεδομένα. Ανεξάρτητα από τη διεπαφή που θα χρησιμοποιηθεί από τον πελάτη για την πρόσβαση στα δεδομένα, το «λογικό επίπεδο» επικοινωνίας με τον διακομιστή στηρίζεται στην SQL, και η αξιόπιστη πρόσβαση στα δεδομένα υλοποιείται με την ορθή χρήση των SQL συναλλαγών.



Εικόνα 1.2 Επεξήγηση επεξεργασίας SQL εντολών

Στην Εικόνα 1.2 εξηγείται το "κυκλικό ταξίδι", του κύκλου επεξεργασίας μιας εντολής SQL, η οποία ξεκινά από τον πελάτη, ως ***αίτημα* εξυπηρέτησης** προς τον διακομιστή χρησιμοποιώντας ένα ενδιάμεσο λογισμικό και υπηρεσίες δικτύου, επεξεργάζεται στον διακομιστή και τέλος επιστρέφεται η απάντηση στην αίτηση. Η εντολή SQL μπορεί να περιλαμβάνει μία ή πολλές ***SQL δηλώσεις*** (statements). Οι SQL δηλώσεις, που αποτελούν την εντολή, μεταγλωττίζονται, αναλύονται, με βάση τα μεταδεδομένα της βάσης δεδομένων, βελτιώνονται και τελικά εκτελούνται. Για να βελτιωθεί η υποβάθμιση των επιδόσεων λόγω των αργών υπηρεσιών ανάγνωσης/εγγραφής από/προς τον δίσκο, ο διακομιστής διατηρεί όλες τις τελευταίες εγγραφές στην RAM και όλη η επεξεργασία των δεδομένων γίνεται εκεί.

Η εκτέλεση μίας εισαγόμενης SQL εντολής στον διακομιστή είναι *ατομική* με την έννοια ότι θα πρέπει ολόκληρη η SQL εντολή (όλες οι SQL δηλώσεις της) να εκτελεστεί επιτυχώς αλλιώς όλη η εντολή θα αναιρεθεί (rolled back). Ως απόκριση στην SQL εντολή (αίτηση από τον πελάτη) ο διακομιστής στέλνει διαγνωστικά μηνύματα αναφέροντας την επιτυχημένη ή αποτυχημένη εκτέλεση της εντολής. Η

[1] όπως ODBC, JDBC, ADO.NET, LINQ, κτλ., το οποίο εξαρτάται από τη γλώσσα προγραμματισμού που χρησιμοποιείτε, όπως C++, C#, Java, PHP, κτλ

αποτυχημένη εκτέλεση μίας εντολής εμφανίζεται στον πελάτη ως μία σειρά εξαιρέσεων. Ωστόσο, είναι σημαντικό να κατανοήσουμε ότι οι SQL δηλώσεις όπως UPDATE ή DELETE θεωρείται ότι εκτελούνται επιτυχημένα ακόμη και αν δεν βρεθούν εγγραφές σχετικές με αυτές που αναφέρονται στις SQL δηλώσεις. Από την πλευρά της εφαρμογής αυτές οι περιπτώσεις θα μπορούσαν να παρουσιασθούν ως αποτυχημένες, αφού δεν έχουν αποτέλεσμα, όμως από την πλευρά εκτέλεσης της εντολής παρουσιάζονται ως πετυχημένες αφού δεν οδηγούν σε σφάλμα. Συνεπώς, στην εφαρμογή θα πρέπει να προβλεφθεί προσεκτικός έλεγχος των διαγνωστικών μηνυμάτων που επιστρέφονται από τον διακομιστή, ώστε να καθορισθεί ο αριθμός των εγγραφών που επηρεάστηκαν από την τρέχουσα πράξη (λειτουργία).

Στην περίπτωση μιας SELECT δήλωσης, ο παραγόμενος πίνακας αποτελεσμάτων (resultset) θα ανακτάται γραμμή-γραμμή στην πλευρά του πελάτη· οι γραμμές του πίνακα αποτελεσμάτων είτε ανακτώνται απευθείας από τον διακομιστή, μέσω δικτύου, ή ανακτώνται από την γρήγορη μνήμη (cache) του πελάτη.

1.3 Συναλλαγές SQL

Όταν στην αρχιτεκτονική (εφαρμογής) λογισμικού έχει σχεδιασθεί η εκτέλεση μίας σειράς SQL εντολών ως μία, τότε οι εντολές πρέπει να ομαδοποιηθούν ως μία λογική μονάδα εργασίας (ΛΜΕ) η οποία ονομάζεται ***SQL συναλλαγή***, και ολοκληρώνεται είτε με μία εντολή **COMMIT** ή με μία εντολή **ROLLBACK**. Η πρώτη (COMMIT) επικυρώνει όλες τις αλλαγές που έκανε η συναλλαγή στη βάση δεδομένων, ενώ η τελευταία (ROLLBACK) τις αναιρεί. Το πλεονέκτημα της εντολής ROLLBACK είναι ότι όταν οι εντολές μιας συναλλαγής δεν μπορούν να ολοκληρωθούν, τότε δεν χρειάζεται να εκτελεστούν μία σειρά από αντίστροφες ενέργειες για να αναιρέσουν μία μία τις εντολές, αλλά η συνολική συναλλαγή αναιρείται απλά με την εντολή ROLLBACK, η οποία πάντα εκτελείται χωρίς προβλήματα. Ανολοκλήρωτες συναλλαγές στο τέλος του προγράμματος ή σε κατάρρευση συστήματος, αυτόματα αναιρούνται (rollback) από το σύστημα. Επίσης, στις περιπτώσεις συγκρούσεων ταυτοχρονισμού, κάποια προϊόντα ΣΔΒΔ αυτόματα θα αναιρέσουν μία συναλλαγή, όπως εξηγείται παρακάτω.

Το πλεονέκτημα της εντολής ROLLBACK (περίπτωση προτύπου - standard SQL) είναι ότι όταν η εφαρμογή λογισμικού όπως υλοποιείται στο σώμα της συναλλαγής κατά την εκτέλεσή της δεν μπορεί να ολοκληρωθεί τότε δεν υπάρχει λόγος (για την εφαρμογή λογισμικού ή για τον προγραμματιστή/χρήστη) να διεξάγει/εκτελέσει μια σειρά αντίστροφων πράξεων/λειτουργιών εντολή-εντολή. Αντί αυτού, όλες οι αλλαγές που έγιναν ήδη στη βάση δεδομένων από τη συναλλαγή που δεν ολοκληρώθηκε αναιρούνται απλά με την επεξεργασία μίας εντολής ROLLBACK που εγγυάται πάντοτε την επιτυχή εκτέλεση της συναλλαγής. Οι ενεργές συναλλαγές (για παράδειγμα οι μη επικυρωμένες) σε περίπτωση κατάρρευσης /πτώσης συστήματος αυτόματα αναιρούνται (rolled back) όταν το σύστημα επαναρχίζει την κανονική λειτουργία του.

> **Σημείωση:** Σύμφωνα με το ISO SQLπρότυπο όπως υλοποιείται για παράδειγμα στα προϊόντα DB2 και Oracle, κάθε εντολή SQL (command) στην αρχή μιας SQL συνεδρίας ή μία εντολή που ακολουθεί το τέλος μιας συναλλαγής ξεκινά αυτόματα μια νέα SQL συναλλαγή. Στην περίπτωση αυτή λέμε ότι έχουμε μία έμμεση έναρξη/αρχή μιας SQL συναλλαγής.

Μερικά προϊόντα ΣΔΒΔ, για παράδειγμα, SQL Server, MySQL/InnoDB, PostgreSQL και Pyrrho λειτουργούν από προεπιλογή στη λειτουργία **AUTOCOMMIT**. Αυτό σημαίνει ότι η εκτέλεση κάθε SQL εντολής θα επικυρώνεται αυτόματα και τα αποτελέσματα/αλλαγές που έγιναν στη βάση δεδομένων με την εντολή δεν θα μπορεί να αναιρεθούν. Έτσι στην περίπτωση σφαλμάτων απαιτείται η εκτέλεση αντίστροφων λειτουργιών από την εφαρμογή για τη λογική μονάδα εργασίας, η οποία όμως θα μπορούσε να είναι αδύνατη μετά από λειτουργίες ταυτόχρονων SQL-πελατών (clients). Επίσης, στην περίπτωση διακοπής της σύνδεσης η βάση δεδομένων θα μπορούσε να βρεθεί/αφεθεί σε ασυνεπή κατάσταση. Στη συνέχεια, αν θέλουμε να χρησιμοποιήσουμε πραγματική λογική συναλλαγής, θα πρέπει κάθε συναλλαγή να ξεκινάει με κάποια συγκεκριμένη εντολή εκκίνησης, όπως BEGIN WORK, BEGIN TRANSACTION, ή START TRANSACTION, ανάλογα με το προϊόν ΣΔΒΔ που χρησιμοποιείται.

Σημείωση: Στη MySQL/InnoDB μία νέα SQL συνεδρία μπορεί να αλλάξει τρόπο χρήσης των συναλλαγών είτε με έμμεσο ή ρητό τρόπο χρησιμοποιώντας τη δήλωση

SET AUTOCOMMIT = { 0 | 1 }

όπου 0 σημαίνει έμμεση χρήση των συναλλαγών, και 1 σημαίνει AUTOCOMMIT λειτουργία

Κάποια προϊόντα ΣΔΒΔ, όπως η Oracle, κάνουν έμμεση επικύρωση των συναλλαγών και σε κάθε DDL εντολή (CREATE, ALTER ή DROP κάποιου αντικειμένου της Βάσης Δεδομένων όπως TABLE, INDEX, VIEW, κτλ).

Στον SQL Server, όλο το στιγμιότυπο μιας βάσης δεδομένων[2], συμπεριλαμβανομένων των βάσεων δεδομένων της, μπορεί να ρυθμιστεί να χρησιμοποιεί έμμεσα συναλλαγές, και μία νέα SQL συνεδρία (σύνδεση) μπορεί να τεθεί σε έμμεση χρήση συναλλαγών και να ορίζεται η λειτουργία AUTOCOMMIT με την ακόλουθη μορφή δηλώσεων

SET IMPLICIT_TRANSACTIONS { ON | OFF }

Σημείωση: Ορισμένα βοηθητικά προγράμματα, όπως ο επεξεργαστής γραμμής εντολών (CLP) της IBM DB2, και ορισμένες διασυνδέσεις πρόσβασης δεδομένων, όπως ODBC και JDBC, λειτουργούν από προεπιλογή στη λειτουργία autocommit.
Για παράδειγμα, στο JDBC API, κάθε συναλλαγή χρειάζεται να ξεκινήσει από την ακόλουθη κλήση μεθόδου του αντικειμένου σύνδεσης

<connection>.setAutoCommit(false);

[2] Στιγμιότυπο της Βάσης Δεδομένων συνήθως ονομάζεται ως διακομιστής Βάσης Δεδομένων. Ανάλογα με το προϊόν ΣΔΒΔ, ένα στιγμιότυπο μπορεί να περιέχει πολλές βάσεις δεδομένων.

Αντί για μια απλή ακολουθία εντολών διαχείρισης των δεδομένων, ορισμένες SQL συναλλαγές μπορεί να περιλαμβάνουν πολύπλοκη προγραμματιστική λογική. Σε τέτοιες περιπτώσεις, η λογική της συναλλαγής θα κάνει κατάλληλες επιλογές κατά το χρόνο εκτέλεσης, ανάλογα με τις πληροφορίες που ανακτώνται από τη βάση δεδομένων. Ακόμα και τότε, η συναλλαγή SQL μπορεί να θεωρηθεί ως μία αδιαίρετη "Λογική Μονάδα Εργασίας» (ΛΜΕ), που είτε επικυρώνεται ή αναιρείται. Ωστόσο, μία αποτυχία στην συναλλαγή συνήθως δεν δημιουργεί αυτόματα ROLLBACK, αλλά θα πρέπει να διαγνωσθεί από την εφαρμογή η αποτυχία (βλ. «Η διάγνωση των SQL σφαλμάτων» παρακάτω) και η εφαρμογή είναι υπεύθυνη για να υλοποιήσει το ROLLBACK

1.4 Λογική Συναλλαγής

Ας θεωρήσουμε τον παρακάτω σχεσιακό πίνακα των τραπεζικών λογαριασμών:

```
CREATE TABLE Accounts (
acctId  INTEGER NOT NULL PRIMARY KEY,
balance DECIMAL(11,2) CHECK (balance >= 0.00)
);
```

Ένα τυπικό παράδειγμα SQL συναλλαγής είναι η μεταφορά ενός ορισμένου ποσού (π.χ. 100 ευρώ) από έναν λογαριασμό σε έναν άλλο

```
BEGIN TRANSACTION;
UPDATE Accounts SET balance = balance - 100 WHERE acctId = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctId = 202;
COMMIT;
```

Αν το σύστημα αποτύχει, ή η σύνδεση δικτύου πελάτη-διακομιστή καταρρεύσει μετά την εκτέλεση της πρώτης εντολής UPDATE, το πρωτόκολλο συναλλαγής εγγυάται ότι δεν θα χαθούν χρήματα από τον λογαριασμού με αριθμό «101», δεδομένου ότι, η συναλλαγή θα πρέπει να αναιρεθεί.
Ωστόσο, η συναλλαγή απέχει πολύ από να θεωρηθεί αξιόπιστη, όπως φαίνεται στις παρακάτω δύο περιπτώσεις:

a) Στην περίπτωση που ο ένας από τους δύο τραπεζικούς λογαριασμούς δεν υπάρχει, οι εντολές UPDATE θα εκτελεστούν κανονικά από την άποψη της SQL και θα «πετύχουν». Γι' αυτό, θα πρέπει να ελεγχθούν τα διαθέσιμα SQL διαγνωστικά μηνύματα και επίσης ο αριθμός των εγγραφών που έχουν επηρεαστεί από κάθε μία από τις δύο εντολές UPDATE.

b) Στην περίπτωση που η πρώτη εντολή UPDATE αποτύχει εξαιτίας αρνητικού υπολοίπου στον λογαριασμού με αριθμό «101» (επομένως: παραβιάζοντας το αντίστοιχο περιορισμό CHECK), στη συνέχεια, η εκτέλεση της δεύτερης εντολής UPDATE θα οδηγήσει σε μια λογικά εσφαλμένη κατάσταση στη βάση δεδομένων

Από αυτό το απλό παράδειγμα, αντιλαμβανόμαστε ότι οι προγραμματιστές εφαρμογών θα πρέπει να γνωρίζουν τον τρόπο που κάθε προϊόν ΣΔΒΔ συμπεριφέρεται, και πως επιστρέφονται τα SQL διαγνωστικά από το API που χρησιμοποιείται για τη διασύνδεση των δεδομένων. Ακόμα και τότε, υπάρχουν πολλά περισσότερα που πρέπει να μάθουν και μια σειρά από λειτουργίες ρύθμισης της βάσης δεδομένων που πρέπει να διεξαχθούν.

1.5 Η διάγνωση των SQL σφαλμάτων

Αντί για μια απλή ακολουθία εντολών διαχείρισης των δεδομένων, ορισμένες SQL συναλλαγές μπορεί να περιλαμβάνουν πολύπλοκη προγραμματιστική λογική. Σε τέτοιες περιπτώσεις, η λογική της συναλλαγής θα κάνει κατάλληλες επιλογές κατά το χρόνο εκτέλεσης, ανάλογα με τις πληροφορίες που ανακτώνται από τη βάση δεδομένων. Ακόμα και τότε, η συναλλαγή SQL μπορεί να θεωρηθεί ως μία αδιαίρετη "Λογική Μονάδα Εργασίας» (ΛΜΕ), που είτε επικυρώνεται ή αναιρείται. Ωστόσο, μία αποτυχία στην συναλλαγή συνήθως δεν δημιουργεί αυτόματα ROLLBACK, αλλά θα πρέπει να διαγνωσθεί από την εφαρμογή η αποτυχία (βλ. «Η διάγνωση των SQL σφαλμάτων " παρακάτω) και η εφαρμογή είναι υπεύθυνη για να υλοποιήσει το ROLLBACK

Για αυτό το λόγο, στο παλαιότερο ISO πρότυπο SQL-89 ορίστηκε ο ειδικός ακέραιος διαγνωστικός δείκτης SQLCODE, ο οποίος παίρνει την τιμή 0 όταν μία SQL εντολή εκτελείται πετυχημένα, την τιμή 100 όταν δεν επιστρέφονται αποτελέσματα, και όλες οι υπόλοιπες τιμές αντιστοιχούν σε συγκεκριμένα μηνύματα τα οποία επεξηγούνται στα αντίστοιχα εγχειρίδια αναφοράς κάθε προϊόντος ΣΔΒΔ. Οι θετικές τιμές δείχνουν προειδοποιήσεις ενώ οι αρνητικές τιμές τα λάθη.

Στο ISO SQL-92, το SQLCODE εγκαταλείπεται και ορίζεται ο νέος αλφαριθμητικός δείκτης SQLSTATE, ο οποίος αποτελείται από μια σειρά από πέντε χαρακτήρες, εκ των οποίων οι δύο πρώτοι χαρακτήρες αποτελούν την κατηγορία SQL σφάλματος και προειδοποίησης, και οι τελευταίοι τρεις χαρακτήρες κωδικοποιούν πιο συγκεκριμένα λάθη ή προειδοποιήσεις. Έτσι, η επιτυχημένη εκτέλεση μιας SQL εντολής συμβολίζεται με πέντε μηδενικά ("00000"). Εκατοντάδες άλλες τιμές έχουν τυποποιηθεί (π.χ. για SQL παραβιάσεις περιορισμών), αλλά ένας μεγάλος αριθμός ορίζεται ειδικά σε κάθε προϊόν ΣΔΒΔ. Οι SQLSTATE τιμές που ξεκινούν με το "40" δείχνουν μια αποτυχημένη συναλλαγή, για παράδειγμα λόγω συγκρούσεων ταυτοχρονισμού, σφάλματος σε μια αποθηκευμένη διαδικασία, χαμένη σύνδεση, ή πρόβλημα στον διακομιστή.

Το X/Open group, για να παρέχει καλύτερη διαγνωστική πληροφορία/πληροφόρηση στις εφαρμογές πελάτη για ότι συμβαίνει στην πλευρά του διακομιστή, επέκτεινε τη γλώσσα SQL με τη δήλωση GET DIAGNOSTICS η οποία μπορεί να χρησιμοποιηθεί για να πάρει λεπτομερέστερες πληροφορίες και να εκτελεστεί επαναληπτικά για την πλοήγηση σε διαγνωστικές εγγραφές αναφέροντας πολλαπλά σφάλματα ή προειδοποιήσεις. Η επέκταση περιλαμβάνεται στο πρότυπο ISO SQL από την SQL:1999 αλλά έχει υλοποιηθεί μόνο τμηματικά/μερικά σε προϊόντα ΣΔΒΔ, για παράδειγμα DB2, Mimer, and MySQL 5.6. Το επόμενο παράδειγμα σε MySQL 5.6 παρουσιάζει την ιδέα της ανάγνωσης διαγνωστικής πληροφορίας

```
GET DIAGNOSTICS @rowcount = ROW_COUNT;
GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE,
                @sqlcode  = MYSQL_ERRNO ;
SELECT @sqlstate, @sqlcode, @rowcount;
```

Κάποιες υλοποιήσεις της γλώσσας SQL με διαδικαστικά χαρακτηριστικά παρέχουν διαγνωστικούς δείκτες σε ειδικούς καταχωρητές ή σε συναρτήσεις της γλώσσας. Στην Transact-SQL (ονομάζεται επίσης ως T-SQL) του MS SQL Server, ορισμένοι διαγνωστικοί δείκτες είναι διαθέσιμοι σε μεταβλητές που ξεκινούν με «@@», όπως

το @@ERROR για τον μητρικό κωδικό σφάλματος, ή @@ROWCOUNT για τον αριθμό των εγγραφών που επεξεργάστηκαν στην τελευταία επεξεργασία.

Από την έκδοση του SQL Server 2005, η Transact-SQL υποστηρίζει επίσης τη δομή ελέγχου try-catch, η οποία υιοθετήθηκε από τις σύγχρονες γλώσσες προγραμματισμού.:

Στη γηγενή (native) IBM DB2 SQL γλώσσα, τα διαγνωστικά αναγνωριστικά SQLCODE και SQLSTATE του πρότυπου ISO SQL είναι διαθέσιμα στην προγραμματιστική/διαδικαστική επέκταση της γλώσσας για αποθηκευμένες διαδικασίες, όπως παρακάτω:

```
<SQL statement>
IF (SQLSTATE <> '00000') THEN
    <error handling>
END IF;
```

Στον κώδικα που περιέχεται ανάμεσα στα BEGIN-END της Oracle PL/SQL γλώσσας, ο χειρισμός του λάθος (ή εξαίρεσης) κωδικοποιείται στο κάτω τμήμα του κώδικα με τη βοήθεια της οδηγίας EXCEPTION ως εξής:

```
BEGIN
    <processing>
EXCEPTION
WHEN <exception name> THEN
    <exception handling>;
...
WHEN OTHERS THEN
    err_code := sqlcode;
    err_text := sqlerrm;
    <exception handling>;
END;
```

Τα πρώτα διαγνωστικά αρχεία που σχετίζονται με αντίστοιχες υλοποιήσεις ανήκουν στο ODBC και το JDBC με τα SQLExceptions και SQLWarnings. Το JDBC API της γλώσσας Java, μόλις βρίσκει κάποιο SQL λάθος ορίζει SQL εξαιρέσεις. Αυτές θα πρέπει υποχρεωτικά να «πιαστούν» με χρήση της δομής ελέγχου «try-catch» στην λογική της εφαρμογής ως εξής:

```
... throws SQLexception {
...
try {
    ...
    <JDBC statement(s)>
}
catch (SQLException ex) {
    <exception handling>
}
```

Στην περίπτωση JDBC, παρέχεται το διαγνωστικό στοιχείο (diagnostic item) *rowcount* δηλαδή αριθμός επεξεργασθέντων γραμμών που επιστρέφεται από τις μεθόδους εκτέλεσης

1.6 Πρακτική εξάσκηση στο εργαστήριο

Συμβουλή:
Μην πιστεύετε ό,τι ακούτε/διαβάζετε σε σχέση με το βαθμό στον οποίο υποστηρίζουν τις συναλλαγές τα διάφορα συστήματα DBMS! Για να είστε σε θέση να αναπτύσσετε αξιόπιστες εφαρμογές, πρέπει εσείς οι ίδιοι να πειραματιστείτε και να αποδείξετε τη λειτουργικότητα των κάθε είδους υπηρεσιών/λειτουργιών που υποστηρίζει το DBMS που χρησιμοποιείτε. Υπάρχουν διαφορές μεταξύ των διαφόρων προϊόντων DBMS όσον αφορά στον τρόπο με τον οποίο αυτά υλοποιούν και υποστηρίζουν ακόμη και τις πλέον βασικές των περιπτώσεων υπηρεσιών/λειτουργιών που έχουν να κάνουν με συναλλαγές στην SQL.

Στο Παράρτημα 1, παρουσιάζονται ασκήσεις ρητών και έμμεσων SQL συναλλαγών., εντολές COMMIT και ROLLBACK, και λογική συναλλαγών (transaction logic) με χρήση MS SQL Server, αλλά θα πρέπει να τις δοκιμάσετε για να επαληθεύσετε συμπεριφορά του SQL Server όπως παρουσιάστηκε στις υποενότητες A1.1 – A1.2. Για την εξάσκησή σας μπορείτε να κατεβάσετε χωρίς χρέωση την SQL Server Express από τον ιστότοπο της Microsoft.

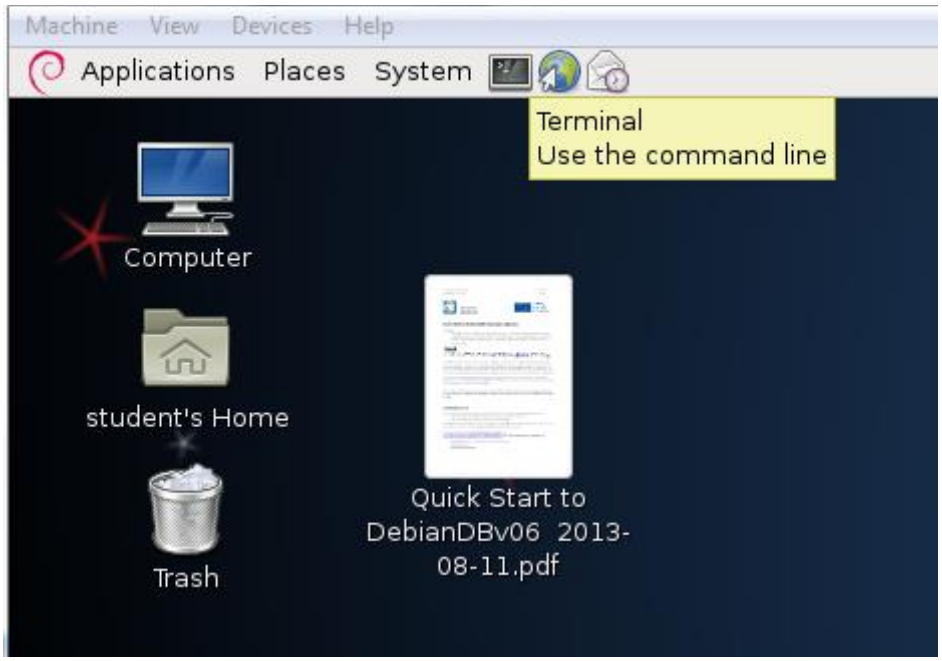
Στο πρώτο μέρος της πρακτικής εξάσκησης στο εργαστήριο του μαθήματος ο εκπαιδευόμενος εξοικειώνεται με τη χρήση του ελεύθερου εικονικού εργαστηρίου βάσεων δεδομένων DebianDB του DBTechNet's

Καλώς ήρθατε σε ένα «ταξίδι μυστηρίου» στον κόσμο των SQL συναλλαγών χρησιμοποιώντας MySQL. Παλαιότερα η προκαθορισμένη μηχανή της MySQL δεν υποστήριζε συναλλαγές αλλά από τη διανομή MySQL 5.1 προκαθορισμένη μηχανή είναι η InnoDB που υποστηρίζει συναλλαγές. Ακόμη και τώρα κάποιες υπηρεσίες μπορεί να παράγουν παράδοξα αποτελέσματα.

Σημείωση: Η σειρά πειραμάτων ακολουθεί το ίδιο μοτίβο για όλα τα ΣΔΒΔ στο DebianDB και υπάρχει στα αρχεία εντολών (script files) του τύπου Appendix1_<dbms>.txt που είναι αποθηκευμένα στην περιοχή “**Transactions”** **(**directory) του χρήστη “student”.

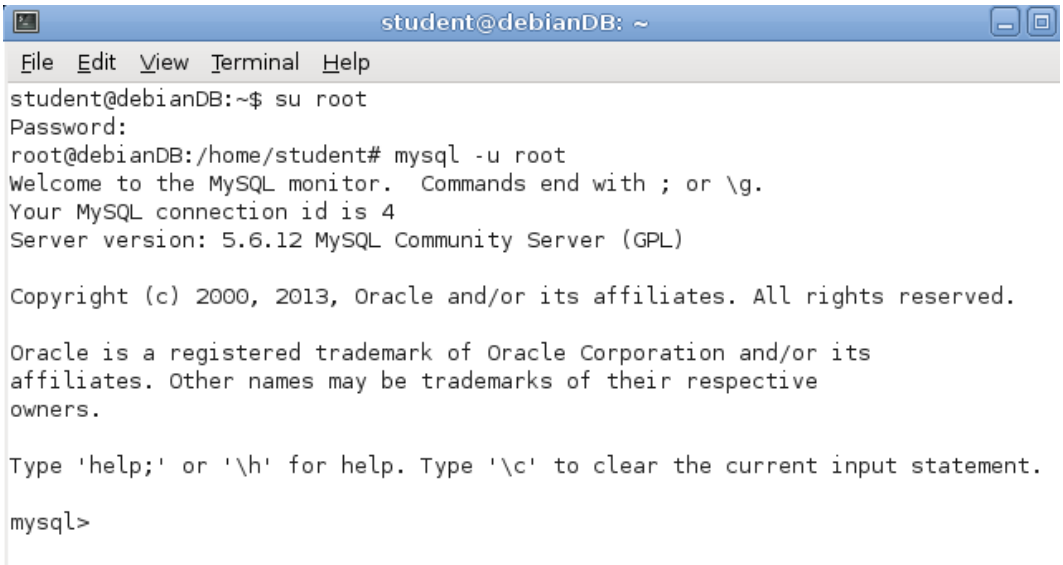
Υποτίθεται ότι ο αναγνώστης ήδη μελέτησε το DBTech VET έγγραφο με τίτλο «Γρήγορος Οδηγός για το Εργαστήριο DebianDB», ένα έγγραφο που εξηγεί την εγκατάσταση και την αρχική παραμετροποίηση που απαιτείται ώστε το DebianDB να λειτουργήσει σε ένα εικονικό περιβάλλον (συνήθως το VirtualBox της Oracle).

Μόλις το DebianDB είναι έτοιμο για χρήση είναι χρήσιμο να παρατηρήσουμε ότι είναι προκαθορισμένο ότι ο χρήστης συνδέεται με (username, password) = (**student**, **password**). Για να δημιουργήσει βάση δεδομένων ο χρήστης πρέπει να συνδεθεί χρησιμοποιώντας το λογαριασμό (username, password) = (**root**, **P4ssw0rd**), όπως εξηγείται στο έγγραφο «Γρήγορος Οδηγός για το Εργαστήριο DebianDB». Αυτό γίνεται κάνοντας κλικ στο top menu bar της Εικονικής Μηχανής (Εικόνα 1.4) .



Εικόνα 1.4 Η επιλογή (το εικονίδιο) "Terminal / Use the command line" του μενού επιλογής της Εικονικής Μηχανής (Virtual Machine)

Στη συνέχεια δίδονται οι ακόλουθες εντολές Linux ενώ στο παράθυρο του τερματικού/της γραμμής εντολών ξεκινά το πρόγραμμα του **mysql** πελάτη.



Εικόνα 1.5 Έναρξη SQL συνεδρίας από χρήστη 'root'

Η επόμενη SQL command δημιουργεί τη βάση δεδομένων "TestDB".

```
-- ---------------------------------
CREATE DATABASE TestDB;
-- ---------------------------------
```

Για να εκχωρήσουμε δικαίωμα πρόσβασης και όλα τα πιθανά δικαιώματα στη βάση TestDB απαιτείται ο χρήστης "root" να πληκτρολογήσει την επόμενη εντολή:

```
-- ----------------------------------
GRANT ALL ON TestDB.* TO 'student'@'localhost';
-- ----------------------------------
```

Τώρα τερματίζουμε το χρήστη "root", να κάνουμε EXIT από τη MySQL συνεδρία και επιστρέφουμε στη σύνοδο του "student", με τις επόμενες δύο εντολές:

```
-- ----------------------------------
EXIT;
exit
```

> **Συμβουλή**: Αν κατά τη διάρκεια του πειραματισμού όλα πήγαν καλά η οθόνη του DebianDB γίνεται μαύρη και ζητά συνθηματικό για να γίνει ενεργή και πάλι. Το απαιτούμενο συνθηματικό είναι για το λογαριασμό "student": Student1

Τώρα ο προκαθορισμένος χρήστης "student" μπορεί να ξεκινήσει MySQL πελάτη και να έχει πρόσβαση στη βάση TestDB ως εξής:

```
-- ----------------------------------
mysql
use TestDB;
-- ----------------------------------
```

Είναι η αρχή μιας νέας MySQL συνεδρίας.

Άσκηση 1.1

Θα δημιουργήσουμε νέο πίνακα με όνομα "T" με τρεις στήλες, id (τύπου integer, πρωτεύον κλειδί), s (τύπου character string με μήκος μέχρι 40 χαρακτήρες), και si (τύπου small integer):

```
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(40), si SMALLINT);
```

Μετά από κάθε SQL εντολή ο MySQL πελάτης δείχνει κάποια διαγνωστικά της εκτέλεσης.

Για να επιβεβαιώσουμε τη δημιουργία του πίνακα και τη δομή του μπορούμε να χρησιμοποιήσουμε την εντολή DESCRIBE:

```
DESCRIBE T;
```

> **Σημείωση:** Η γραφή εντολών SQL στο προϊόν MySQL δεν εξαρτάται από τη χρήση κεφαλαίων-πεζών γραμμάτων με εξαίρεση τα ονόματα πινάκων και βάσεων δεδομένων. Αυτό σημαίνει ότι οι εντολές "Describe T", "describe T", "create TaBle T ...", είναι ορθές αλλά οι εντολές "use testDB", και "describe t" αφορούν βάση δεδομένων και πίνακα διαφορετικά από τα χρησιμοποιούμενα στο εργαστήριο αυτό.

Θα προσθέσουμε/εισάγουμε γραμμές στο νεοδημιουργηθέντα πίνακα:

```
-- ----------------------------------
INSERT INTO T (id, s) VALUES (1, 'first');
INSERT INTO T (id, s) VALUES (2, 'second');
```

```
INSERT INTO T (id, s) VALUES (3, 'third');
SELECT * FROM T ;
-- ---------------------------------
```

Η εντολή "SELECT * FROM T" επιβεβαιώνει ότι προστέθηκαν οι τρεις νέες γραμμές στον πίνακα (παρατηρήστε ότι τιμές NULL καταχωρήθηκαν στη στήλη "si".

> **Σημείωση:** Πάντοτε βεβαιωθείτε ότι πληκτρολογήσατε το χαρακτήρα ";" στο τέλος κάθε εντολής και στη συνέχεια πατήστε το πλήκτρο "enter".

Ανακαλώντας στη μνήμη ότι ειπώθηκε μέχρι στιγμής για τις SQL συναλλαγές προσπαθήστε να ακυρώσετε/ανακαλέσετε την τρέχουσα συναλλαγή δίδοντας την εντολή:

```
-- ---------------------------------
ROLLBACK;
SELECT * FROM T ;
-- ---------------------------------
```

Φαίνεται να εκτελείται αλλά δίδοντας εκ νέου εντολή "SELECT * FROM T" διαπιστώνετε ότι διατηρήθηκε το περιεχόμενο των γραμμμών του πίνακα. Απρόσμενο …

Η πηγή της έκπληξης έχει όνομα: "AUTOCOMMIT". Η MySQL ξεκινά σε AUTOCOMMIT μορφή λειτουργίας (mode) όπου κάθε συναλλαγή απαιτείται να ξεκινά με εντολή "START TRANSACTION", και μετά το τέλος της συναλλαγής η MySQL επιστρέφει σε AUTOCOMMIT μορφή λειτουργίας (mode). Για να επαληθεύσετε τα παραπάνω εκτελέστε το ακόλουθο σύνολο SQL εντολών.

```
-- ---------------------------------
START TRANSACTION;
INSERT INTO T (id, s) VALUES (4, 'fourth');
SELECT * FROM T ;
ROLLBACK;

SELECT * FROM T;
-- ---------------------------------
```

Ερώτηση

- Συγκρίνατε τα παραγόμενα αποτελέσματα εκτελώντας τις παραπάνω δύο εντολές SELECT * FROM T

Άσκηση 1.2

Εκτελέστε τις παρακάτω εντολές:

```
-------------------------------------
INSERT INTO T (id, s) VALUES (5, 'fifth');
ROLLBACK;
SELECT * FROM T;
-------------------------------------
```

Ερωτήσεις

- Ποιο το σύνολο αποτελεσμάτων μετά την εκτέλεση της παραπάνω εντολής SELECT * FROM T;
- Ποια τα συμπεράσματά σας αναφορικά με πιθανούς περιορισμούς στη χρήση της εντολής START TRANSACTION στη μηχανή MySQL/InnoDB;

Άσκηση 1.3

Η μορφή λειτουργίας AUTOCOMMIT (mode) της συνεδρίας ακυρώνεται/αναιρείται χρησιμοποιώντας εντολή "SET AUTOCOMMIT":

```
-------------------------------------
SET AUTOCOMMIT = 0;
-------------------------------------
```

Όλες οι γραμμές του πίνακα εκτός από μία διαγράφονται:

```
-------------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;
-------------------------------------
```

Εισαγωγή νέων γραμμών:

```
-------------------------------------
INSERT INTO T (id, s) VALUES (6, 'sixth');
INSERT INTO T (id, s) VALUES (7, 'seventh');
SELECT * FROM T;
-------------------------------------
```

… και εντολή ROLLBACK:

```
-------------------------------------
ROLLBACK;
SELECT * FROM T;
-------------------------------------
```

Ερώτηση

- Αναφέρατε πλεονέκτημα-μειονέκτημα της χρήσης εντολής "SET TRANSACTION" συγκριτικά με τη χρήση

Σημείωση: Στο περιβάλλον Linux και στην περίπτωση του MySQL πελάτη μπορείτε να χρησιμοποιήσετε άνω και κάτω βέλος του πληκτρολογίου για να μετακινηθείτε εμπρός και πίσω στο κείμενο των προτάσεων (των εντολών) που έχετε ήδη εισάγει. Αυτό δεν είναι πάντοτε δυνατόν στα υπόλοιπα περιβάλλοντα ΣΔΒΔ που είναι προεγκατεστημένα **στο** DebianDB.

Σημείωση: Δύο συνεχόμενοι χαρακτήρες "-" στη γλώσσα SQL δηλώνουν ότι η υπόλοιπη γραμμή της εντολής είναι σχόλιο, δηλαδή το κείμενο που ακολουθεί δύο συνεχόμενες παύλες και μέχρι τον επόμενο χαρακτήρα "enter" αγνοείται από τον MySQL parser.

Άσκηση 1.4

```
-- Initializing only in case you want to repeat the exercise 1.4
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
DROP TABLE T2;  --
COMMIT;
```

Γνωρίζετε ήδη ότι κάποιες εντολές SQL ανήκουν στην (υπο)Γλώσσα Ορισμού Δεδομένων -Data Definition Language (DDL)- και κάποιες στην (υπο)Γλώσσα Διαχείρισης Δεδομένων -Data Manipulation Language (DML)-. Παραδείγματα εντολών του πρώτου τύπου είναι οι εντολές CREATE TABLE, CREATE INDEX και DROP TABLE, ενώ παραδείγματα εντολών δευτέρου τύπου (DML εντολές) είναι εντολές όπως SELECT, INSERT και DELETE. Έχοντας αυτά στο μυαλό μας/κατά νουν είναι σημαντικό να εξετάσουμε παραπέρα το «εύρος» («βεληνεκές») της εντολής ROLLBACK στη συνέχεια:

```
SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (9, 'will this be committed?');
CREATE TABLE T2 (id INT);
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;

SELECT * FROM T;  -- What has happened to T ?
SELECT * FROM T2; -- What has happened to T2 ?
-- Compare this with SELECT from a missing table as follows:
SELECT * FROM T3; -- assuming that we have not created table T3

SHOW TABLES;
DROP TABLE T2;
COMMIT;
```

Ερώτηση

- Ποιο συμπέρασμα βγάλατε;

Άσκηση 1.5

Αρχικοποιήστε το περιεχόμενο του πίνακα T:

```
-----------------------------------
SET AUTOCOMMIT=0;
DELETE FROM T WHERE id > 1;
COMMIT;
SELECT * FROM T;
COMMIT;
-----------------------------------
```

Τώρα θα εξετάσετε αν ένα λάθος εκκινεί αυτόματη την εκτέλεση εντολής ROLLBACK στο περιβάλλον MySQL:

```
SET AUTOCOMMIT=0;
INSERT INTO T (id, s) VALUES (2, 'Error test starts here');
-- division by zero should fail
SELECT (1/0) AS dummy FROM DUAL;
-- Now update a non-existing row
UPDATE T SET s = 'foo' WHERE id = 9999 ;
-- and delete an non-existing row
DELETE FROM T WHERE id = 7777 ;
--
INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
INSERT INTO T (id, s)
     VALUES (3, 'How about inserting too long of a string value?');
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
SHOW ERRORS;
SHOW WARNINGS;
INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
SELECT * FROM T;
COMMIT;

DELETE FROM T WHERE id > 1;
SELECT * FROM T;
COMMIT;
-----------------------------------
```

Ερωτήσεις

a) Τι διαπιστώσατε για το αυτόματο rollback σε περίπτωση SQL λαθών στη MySQL;
b) Είναι η διαίρεση με το μηδέν λάθος;
c) Αντιδρά η MySQL σε περίπτωση υπερχείλισης;
d) Τι μαθαίνετε από τα παρακάτω αποτελέσματα:

```
-----------------------------------
mysql> UPDATE T SET s = 'foo' WHERE id = 9999 ;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate');
ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'
-----------------------------------
```

Στην περίπτωση αυτή η τιμή "23000" που μας επιστρέφει/δείχνει ο MySQL πελάτης είναι η τιμή της τυποποιημένης μεταβλητής SQLSTATE που ορίζεται στο πρότυπο της SQL και (η τιμή αυτή) σημειώνει την παραβίαση περιορισμού πρωτεύοντος

κλειδιού και η τιμή 1062 είναι ο αντίστοιχος κωδικός σφάλματος του προϊόντος MySQL.

Τα διαγνωστικά για την αποτυχία της εντολής INSERT στο παραπάνω παράδειγμά μας μπορούν να προσπελαστούν με την SQL εντολή GET DIAGNOSTICS μια νέα εντολή της MySQL version 5.6.

```
mysql> GET DIAGNOSTICS @rowcount = ROW_COUNT;
Query OK, 0 rows affected (0.00 sec)

mysql> GET DIAGNOSTICS CONDITION 1 @sqlstate = RETURNED_SQLSTATE,
   ->                      @sqlcode = MYSQL_ERRNO ;
Query OK, 0 rows affected (0.00 sec)
```

Μεταβλητές που αρχίζουν με "@" είναι τοπικές μεταβλητές, χωρίς τύπο δεδομένων, στην SQL language της MySQL Χρησιμοποιούμε τις μεταβλητές αυτές στην άσκησή μας για προσομοιώσουμε το επίπεδο της εφαρμογής ενώ στο σε αυτό βιβλίο κυρίως εργαζόμαστε στο επίπεδο της γλώσσας SQL. Σε APIs πρόσβασης δεδομένων οι τιμές του δείκτη διαγνωστικών (diagnostic indicator) μπορούν να διαβάζονται απ' ευθείας σε μεταβλητές υποδοχής (host) αλλά στο παράδειγμά μας βλέπετε πως μπορούμε επίσης να διαβάζουμε τις τιμές από τοπικές μεταβλητές.

```
mysql> SELECT @sqlstate, @sqlcode, @rowcount;
+-----------+----------+-----------+
| @sqlstate | @sqlcode | @rowcount |
+-----------+----------+-----------+
| 23000     |     1062 |        -1 |
+-----------+----------+-----------+
1 row in set (0.00 sec)
```

Άσκηση 1.6

```
DROP TABLE Accounts;
SET AUTOCOMMIT=0;
```

Η τρέχουσα διανομή της MySQL δεν υποστηρίζει τη σύνταξη περιορισμού **CHECK σε επίπεδο στήλης** που χρησιμοποιούμε σε άλλα προϊόντα ΣΔΒΔ για την άσκηση αυτή, ως εξής:.

```
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL CONSTRAINT unloanable_account CHECK (balance >= 0)
);
```

Δέχεται τη σύνταξη περιορισμού **CHECK σε επίπεδο γραμμής,** ως εξής:

```
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL ,
CONSTRAINT unloanable_account CHECK (balance >= 0)
) ENGINE = InnoDB;
```

Αλλά ακόμη και αν δέχεται τη σύνταξη στην πραγματικότητα δε χρησιμοπιεί τον περιορισμό CHECK όπψς θα δείτε πειραματιζόμενοι με την ακόλουθη δοκιμή που θα αποτύχει:

```
INSERT INTO Accounts (acctID, balance) VALUES (100,-1000);
SELECT * FROM Accounts;
ROLLBACK;
```

Σημείωση: Ο περιορισμός CHECK διατηρήθηκε για τη σύγκριση των πειραμάτων με άλλα προϊόντα. Όλα τα προϊόντα παρουσιάζουν κάποια προβλήματα και οι προγραμματιστές εφαρμογών πρέπει να τα αντιμετωπίζουν. Το πρόβλημα της μη υποστήριξης του περιορισμού CHECKs μπορεί να επιλυθεί με τη δημιουργία SQL triggers. Οι ενδιαφερόμενοι αναγνώστες μπορούν να βρουν παραδείγματα στο αρχείο εντολών (script file) AdvTopics_MySQL.txt.

```
-- Let's load also contents for our test:
SET AUTOCOMMIT=0;
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
SELECT * FROM Accounts;
COMMIT;

-- A. Let's try the bank transfer
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;
SELECT * FROM Accounts;
ROLLBACK;

-- B. Let's test that the CHECK constraint actually works:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
SELECT * FROM Accounts ;
ROLLBACK;
```

Η επόμενη προς έλεγχο συναλλαγή προσπαθεί να μεταφέρει 500 ευρώ από τον τραπεζικό λογαριασμό 101 σε έναν ανύπαρκτο λογαριασμό με acctID = 777:

```
-- C. Updating a non-existent bank account 777:
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;
SELECT * FROM Accounts ;
ROLLBACK;
```

Ερωτήσεις

a) Οι δύο εντολές UPDATE εκτελούνται παρά το γεγονός ότι η δεύτερη αντιστοιχεί στην απαίτηση ενημέρωσης ανύπαρκτης γραμμής του πίνακα Accounts;

b) Αν η εντολή ROLLBACK στα παραδείγματα συναλλαγής B και C αντικατασταθεί με εντολή COMMIT τότε εξετάστε αν η συναλλαγή θα εκτελεστεί με επιτυχία οριστικοποιώντας τα αποτελέσματά της στη βάση δεδομένων.

c) Ποιους δείκτες διαγνωστικών της MySQL θα μπορούσε να χρησιμοποιήσει η εφαρμογή για να ανιχνεύσει προβλήματα στις παραπάνω συναλλαγές;

Άσκηση 1.7 Η SQL συναλλαγή ως Μονάδα Επαναφοράς

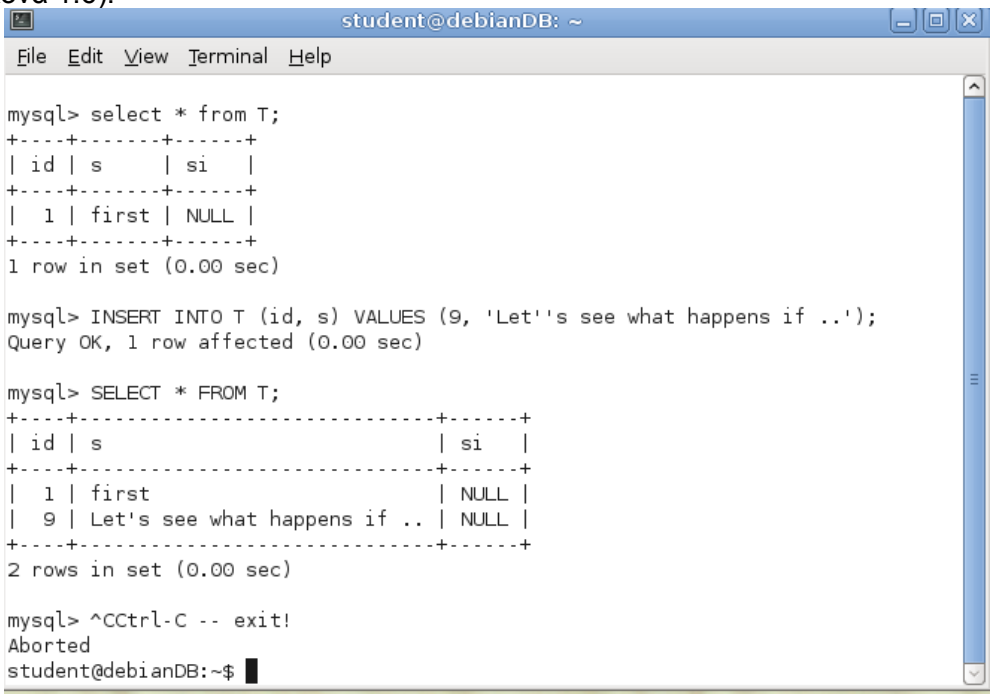
Στη συνέχεια θα πειραματισθείτε με την ιδιότητα της «Μονάδας Επαναφοράς» των SQL συναλλαγών στην περίπτωση με επικυρωμένων συναλλαγών. Για δοκιμή θα ξεκινήσετε μία συναλλαγή, στη συνέχεια θα τερματίσετε απότομα την SQL σύνδεση του MySQL πελάτη και έπειτα θα επανασυνδεθείτε με τη βάση δεδομένων για να δείτε αν οι αλλαγές που έγιναν από τις μη επικυρωμένες συναλλαγές υπάρχουν στη βάση.

Σημειώστε πως μια ενεργή (δηλαδή μη επικυρωμένη) συναλλαγή επηρεάζεται από απότομο τερματισμό σύνδεσης, μια συνηθισμένη περίπτωση στις διαδικτυακές εφαρμογές:

Πρώτα, μια νέα γραμμή εισάγεται στον πίνακα T:

```
-------------------------------------
SET AUTOCOMMIT = 0;
INSERT INTO T (id, s) VALUES (9, 'Let"s see what happens if ..');
SELECT * FROM T;
-------------------------------------
```

Έπειτα η συνεδρία του πελάτη διακόπτεται βίαια με "Control C" (Ctrl-C) εντολή (Εικόνα 1.6):



Εικόνα 1.6 Προσομοίωση κατάστασης πτώσης του ΣΔΒΔ

Στη συνέχεια κλείνετε το παράθυρο του DebianDB τερματικού, ανοίγετε ένα νέο και ξεκινάτε μια νέα MySQL συνεδρία με την TestDB βάση δεδομένων:

```
mysql
USE TestDB;
SET AUTOCOMMIT = 0;
SELECT * FROM T;
COMMIT;
EXIT;
```

Ερώτηση

- Κάποιο σχόλιο για το περιεχόμενο της γραμμής του πίνακα Τ:

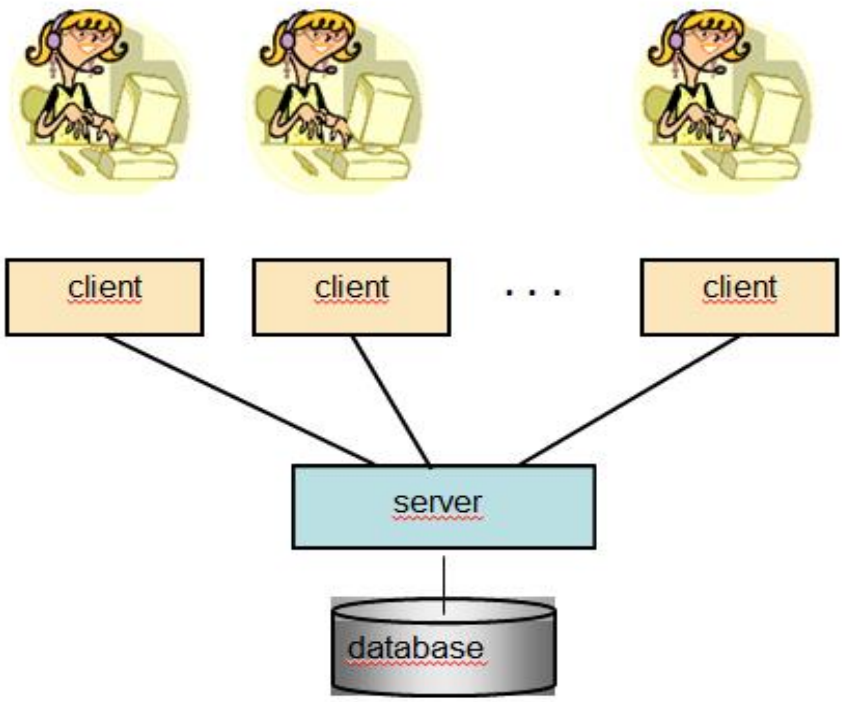
Σημείωση: Όλες οι αλλαγές στη βάση δεδομένων μπορούν να προσπελαστούν/ιχνηλατηθούν/υπάρχουν στο **ιστορικό συναλλαγών** (**transaction log)** της βάσης. Στο Παράρτημα 3 επεξηγείται πως οι διακομιστές βάσεις δεδομένων χρησιμοποιούν τα αρχεία αυτά για να επαναφέρουν τη βάση στο περιεχόμενο που οριστικοποιήθηκε από τη τελευταία επικυρωθείσα συναλλαγή πριν την πτώση του συστήματος. Η άσκηση 1.7 θα μπορούσε να επεκταθεί για τη δοκιμή πτώσης συστήματος αν αντί της κατάργησης μόνο του MySQL πελάτη ακυρωθεί και η διαδικασία **mysqld του διακομιστή, ως εξής:**

```
student@debianDB:~$ su root
Password:
root@debianDB:/home/student# ps -e | grep mysqld
 1092 ?        00:00:00 mysqld_debianDB
 1095 ?        00:00:00 mysqld_safe
 1465 ?        00:00:01 mysqld
root@debianDB:/home/student# kill 1465
```

Μέρος 2: Ταυτόχρονες συναλλαγές

Συμβουλή:
Μην πιστεύετε ο,τι ακούτε/διαβάζετε σε σχέση με το βαθμό στον οποίο υποστηρίζουν τις συναλλαγές τα διάφορα συστήματα DBMS! Για να είστε σε θέση να αναπτύσσετε αξιόπιστες εφαρμογές, πρέπει εσείς οι ίδιοι να πειραματιστείτε και να αποδείξετε τη λειτουργικότητα των κάθε είδους υπηρεσιών/λειτουργιών που υποστηρίζει το DBMS που χρησιμοποιείτε. Υπάρχουν διαφορές μεταξύ των διαφόρων προϊόντων DBMS όσον αφορά στον τρόπο με τον οποίο αυτά υλοποιούν και υποστηρίζουν ακόμη και τις πλέον βασικές των περιπτώσεων υπηρεσιών/λειτουργιών που έχουν να κάνουν με συναλλαγές στην SQL.

Μία εφαρμογή η οποία λειτουργεί χωρίς πρόβλημα σε μονοχρηστικό υπολογιστικό περιβάλλον μπορεί να είναι προβληματική όταν λειτουργεί παράλληλα με αριθμό εφαρμογών πελάτη (δηλαδή: με επιπλέον λειτουργικά στιγμιότυπα της ίδιας ή και άλλων εφαρμογών) όπως το πολυχρηστικό περιβάλλον της Εικόνας 2.1



Εικόνα 2.1 Πρόσβαση στη βάση δεδομένων από πολλαπλούς πελάτες/χρήστες (πολυχρηστικό περιβάλλον)

2.1 Προβλήματα ταυτοχρονισμού – Διακινδύνευση της αξιοπιστίας

Όταν το προϊόν DBMS που χρησιμοποιούμε δεν υποστηρίζει τις κατάλληλες υπηρεσίες ελέγχου του ταυτοχρονισμού, ή τις υποστηρίζει μεν όμως εμείς δεν γνωρίζουμε πως να τις αξιοποιούμε, υπάρχει κίνδυνος καταστροφής του **περιεχομένου** της βάσης δεδομένων, ή/και της πληροφορίας η οποία προκύπτει στο **αποτέλεσμα της επεξεργασίας** των αιτημάτων που υποβάλλονται στη βάση. Σε κάθε περίπτωση, μιλάμε για **μη αξιόπιστη λειτουργικότητα** της εφαρμογής.

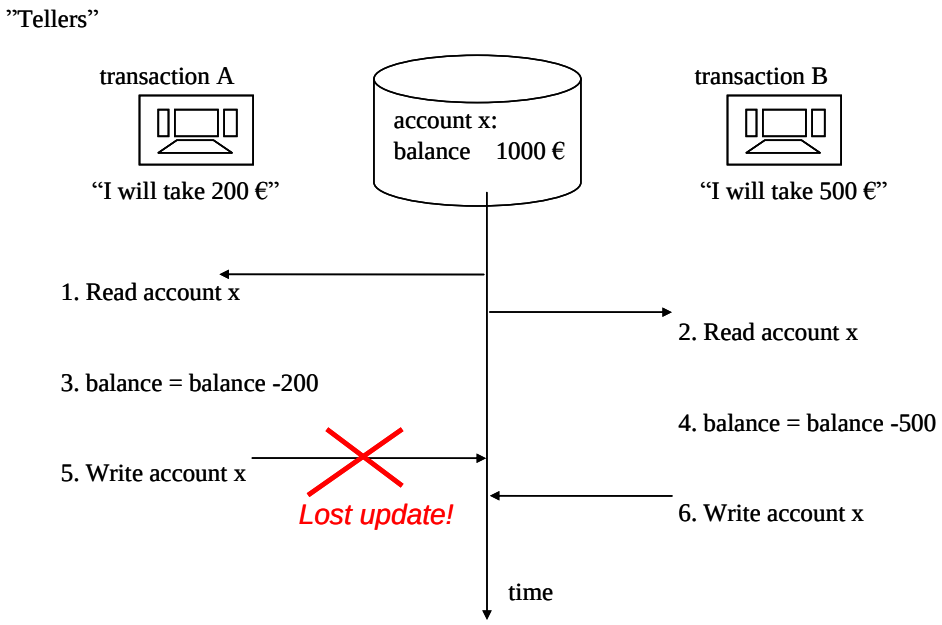
Στα επόμενα, εξετάζονται τυπικές περιπτώσεις προβλημάτων (ανωμαλιών) ταυτοχρονισμού:

1. το πρόβλημα της χαμένης ενημέρωσης (lost update)
2. το πρόβλημα της πρόχειρης ανάγνωσης (dirty read), δηλαδή η ανάγνωση δεδομένων η εγκυρότητα των οποίων δεν έχει ακόμη επικυρωθεί από τις ταυτόχρονα εκτελούμενες συναλλαγές που τα έχουν καταχωρήσει στη βάση
3. το πρόβλημα της μη-επαναλήψιμης ανάγνωσης (non-repeatable read), δηλαδή περιπτώσεις όπου διαδοχικές αναγνώσεις με το ίδιο κριτήριο αναζήτησης δεν επιστρέφουν τις ίδιες πλειάδες/γραμμές στο αποτέλεσμα
4. το πρόβλημα ανάγνωσης φαντάσματος (phantom read problem), δηλαδή, κατά την εκτέλεση της συναλλαγής μερικές από τις πλειάδες οι οποίες θα έπρεπε να συνυπολογιστούν στην επεξεργασία εξαιρούνται γιατί η συναλλαγή δεν αισθάνεται την ύπαρξή τους

και παρουσιάζονται τρόποι με τους οποίους τα παραπάνω μπορούν να αντιμετωπισθούν στο πλαίσιο του πρότυπου ISO SQL στα υπάρχοντα περιβάλλοντα/προϊόντα DBMS.

2.1.1 Το πρόβλημα της χαμένης ενημέρωσης (lost update)

Ο C.J. Date έχει παρουσιάσει το παράδειγμα της Εικόνας 2.2 όπου δύο χρήστες διαφορετικών τραπεζικών ATM προβαίνουν σε αναλήψεις χρημάτων από τον ίδιο τραπεζικό λογαριασμό που αρχικά διαθέτει 1000 ευρώ.



Εικόνα 2.2 Παράδειγμα προβλήματος χαμένης ενημέρωσης

Χωρίς την ύπαρξη ελέγχου στον ταυτοχρονισμό, η ενέργεια write των 800 ευρώ της συναλλαγής Α του βήματος 5 χάνεται στο βήμα 6 όπου η συναλλαγή Β καταχωρεί ως υπόλοιπο τα 500 ευρώ που έχει υπολογίσει στα 'τυφλά', χωρίς να λαμβάνει υπ' όψιν τα της λειτουργίας της συναλλαγής Α. Αν αυτό επιτρέπονταν να γίνει πριν από την ολοκλήρωση και την επικύρωση των μεταβολών που έχει προκαλέσει η συναλλαγή Α, τότε θα προέκυπτε το πρόβλημα της χαμένης ενημέρωσης. Ευτυχώς, τα σύγχρονα DBMS εφαρμόζουν αυτόματα κάποιο βαθμό ελέγχου του ταυτοχρονισμού στην εκτέλεση των συναλλαγών, ελέγχου που αποτρέπει την τροποποίηση

εγγραφών από ενέργειες write συναλλαγών οι οποίες εκτελούνται ταυτόχρονα με συναλλαγή η οποία έχει επικαιροποιήσει τις εν λόγω εγγραφές και είναι ακόμη σε εξέλιξη.

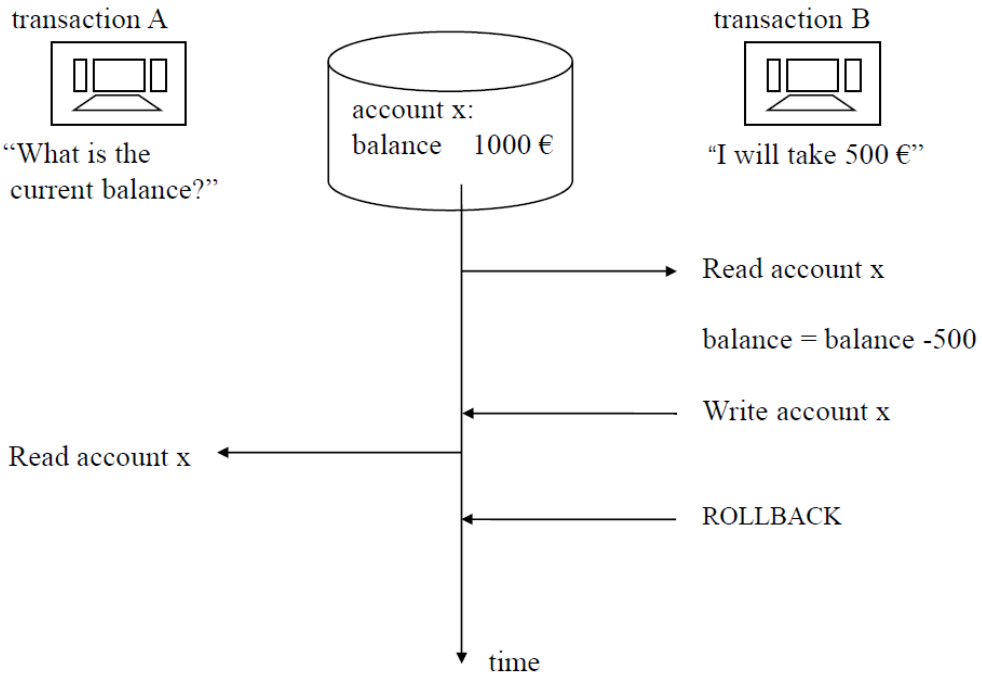
Αν για την υλοποίηση της παραπάνω επεξεργασίας γίνει χρήση ενεργειών τύπου 'SELECT … UPDATE' και με την επιπλέον ύπαρξη σχήματος κλειδαριών για τον έλεγχο του ταυτοχρονισμού, τότε αντί για το πρόβλημα της χαμένης ενημέρωσης θα προκύψει αδιέξοδη παύση (DEADLOCK, εξετάζεται στη συνέχεια). Όταν συμβεί και ανιχνευτεί ως τέτοια η τελευταία, η συναλλαγή B θα αναιρεθεί (αυτόματα ή όχι) από το DBMS ώστε να μπορέσει να ολοκληρώσει η συναλλαγή A.

Αν τα παραπάνω υλοποιηθούν κάνοντας χρήση των λεγόμενων ενεργειών ευαίσθητης επικαιροποίησης (δηλαδή, επικαιροποίηση που λαμβάνει υπ' όψιν την τρέχουσα τιμή του πεδίου που επικαιροποιείται), όπως η

```
UPDATE Accounts SET balance = balance – 200
WHERE acctID = 100;
```

με παράλληλη προστασία μέσω μηχανισμού κλειδώματος (πρόκειται να εξεταστεί στη συνέχεια), τότε η επεξεργασία της Εικόνας 2.2 προχωρά απρόσκοπτα, χωρίς προβλήματα όσον αφορά στην ενημέρωση του περιεχομένου της βάσης εκ μέρους των δύο ταυτόχρονων συναλλαγών.

2.1.2 Το πρόβλημα της πρόχειρης (dirty) ανάγνωσης



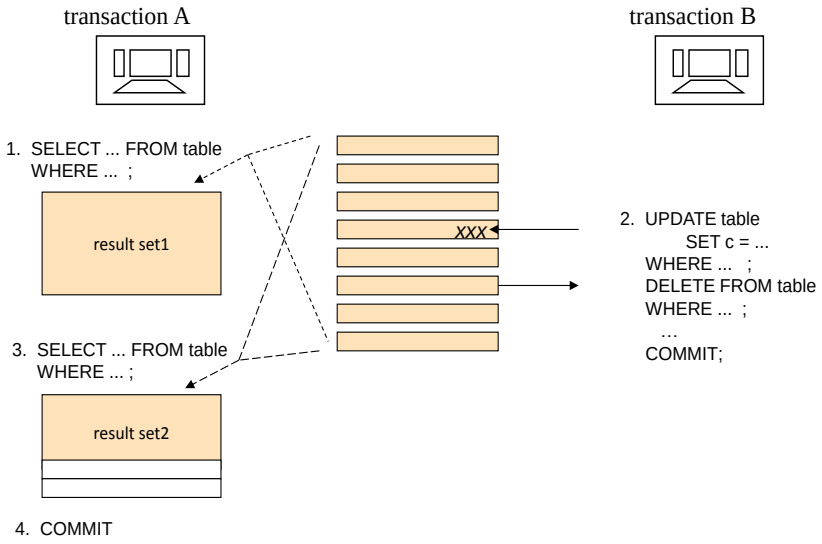
Εικόνα 2.3 Παράδειγμα προβλήματος πρόχειρης ανάγνωσης

Το πρόβλημα της πρόχειρης ανάγνωσης παρουσιάζεται στην Εικόνα 2.3 και έχει να κάνει με την ανάληψη του ρίσκου εκ μέρους της συναλλαγής A να αναγνώσει και μη επικυρωμένα δεδομένα τα οποία έχουν καταχωρηθεί από άλλες, ταυτόχρονα εκτελούμενες, συναλλαγές οι οποίες είναι ακόμη σε εξέλιξη. Αυτό διότι οι τελευταίες υπόκεινται στο ενδεχόμενο της αναίρεσής τους (rollback), οπότε τα μη επικυρωμένα

δεδομένα τους δεν είναι αξιόπιστα. Εξαιτίας του γεγονότος αυτού, συναλλαγές οι οποίες επιλέγεται να αναλάβουν το εν λόγω ρίσκο δεν πρέπει να προβαίνουν στην ενημέρωση (και την καταστροφή, λόγω της αναξιοπιστίας τους) του περιεχομένου της βάσης δεδομένων. Χρειάζεται να λαμβάνεται σωστά υπ'όψιν το γεγονός ότι η χρήση αναξιόπιστων δεδομένων είναι παρακινδυνευμένη και μπορεί να οδηγήσει σε λάθος ενέργειες και αποφάσεις.

2.1.3 Το πρόβλημα της μη-επαναλήψιμης (non-repeatable) ανάγνωσης

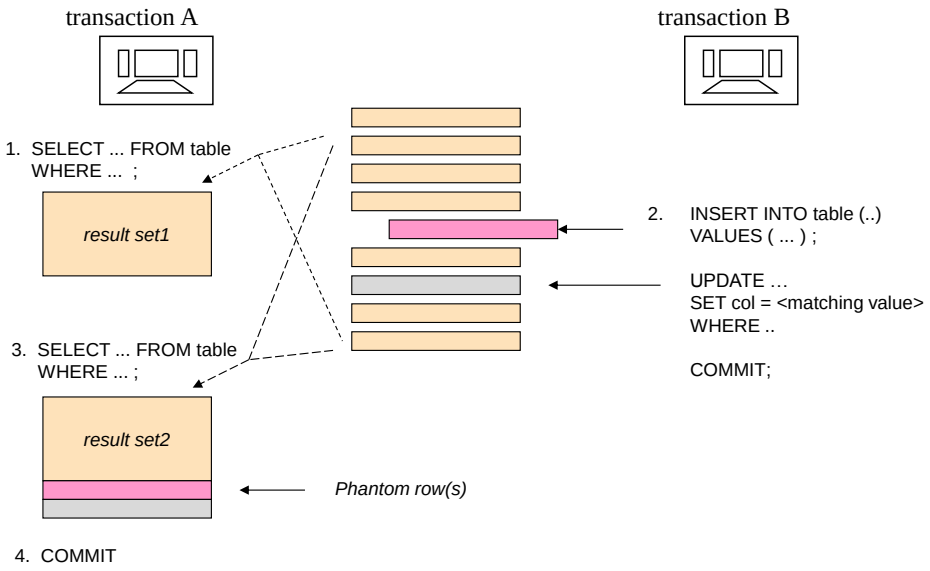
Το πρόβλημα της μη-επαναλήψιμης ανάγνωσης παρουσιάζεται στην Εικόνα 2.4 και έχει να κάνει με το ότι τα σύνολα των πλειάδων που ανακτώντα στο αποτέλεσμα της επεξεργασίας ενός αιτήματος δεν είναι εγγυημένα σταθερά στο πλαίσιο μιας συναλλαγής. Ισοδύναμα, με την επανάληψη στην εκτέλεση ενός αιτήματος μερικές από τις πλειάδες που είχαν ανακτηθεί αρχικά δεν ανακτώνται πλέον. Δεν αποκλείεται επίσης η περίπτωση να εμφανιστούν επιπλέον (νέες) πλειάδες στο αποτέλεσμα της δεύτερης εκτέλεσης σε σχέση με την πρώτη εκτέλεση του αιτήματος.



Εικόνα 2.4 Παράδειγμα προβλήματος μη-επαναλήψιμης ανάγνωσης

2.1.4 Το πρόβλημα της ανάγνωσης φαντάσματος (phantom read)

Το πρόβλημα του φαντάσματος παρουσιάζεται στην Εικόνα 2.5 και έχει να κάνει με το γεγονός ότι τα αποτελέσματα στην έξοδο της επεξεργασίας αιτημάτων μπορεί να συμπεριλαμβάνουν νέες πλειάδες κατά την επανάληψη της εκτέλεσης κάποιων από τα αιτήματα. Νέες πλειάδες οι οποίες είτε έχουν μόλις εισαχθεί/προκύψει ως τέτοιες στη βάση, είτε αντιστοιχούν σε πλειάδες που προϋπήρχαν στη βάση όμως έχουν περιεχόμενο το οποίο στο μεταξύ έχει επικαιροποιηθεί και ως τέτοιο ικανοποιεί το κριτήριο αναζήτησης του κάθε ενός αιτήματος που (επανα)εκτελείται.



Εικόνα 2.5 Παράδειγμα προβλήματος μη-επαναλήψιμης ανάγνωσης

2.2 Ιδανική συναλλαγή, ιδιότητες ACID

Οι ιδιότητες ACID παρουσιάστηκαν για πρώτη φορά με το όνομα ‘η αρχή ACID’ (ACID principle) από τους Theo Härder και Andreas Reuter με άρθρο τους στο περιοδικό ACM Computing Surveys το 1983. Μέσω της εν λόγω αρχής, οι συγγραφείς του άρθρου ορίζουν το ιδανικό για την αξιόπιστη εκτέλεση των συναλλαγών SQL σε πολυχρηστικό υπολογιστικό περιβάλλον. Πρόκειται για ακροστιχίδα που συνιστούν τα αρχικά των επόμενων τεσσάρων ιδιοτήτων των συναλλαγών (στα Αγγλικά):

Ατομική (**A**tomic)

Οι επιμέρους ενέργειες μιας συναλλαγής χρειάζεται να εκτελούνται με ‘ατομικό’ τρόπο (όλες ή καμία). Ισοδύναμα, η συναλλαγή είτε επιτυγχάνει, οπότε όλες οι επιμέρους ενέργειές της ολοκληρώνουν και τα αποτελέσματά τους επικυρώνονται στη βάση δεδομένων, είτε αποτυγχάνει και ακυρώνεται, οπότε η όποια επίδραση επιμέρους ενέργειάς της στη βάση δεδομένων αναιρείται.

Συνεπής (**C**onsistent)

Με την εκτέλεση του συνόλου των επιμέρους ενεργειών της συναλλαγής, η βάση δεδομένων μεταβαίνει από μία συνεπή (consistent) κατάσταση σε άλλη, επίσης συνεπή κατάσταση. Τουλάχιστον κατά τη στιγμή της επικύρωσης (commit) των μεταβολών που έχει επιφέρει η συναλλαγή, το περιεχόμενο της βάσης δεδομένων δεν θα παραβιάζει κανέναν από τους περιορισμούς που αφορούν στην ακεραιότητα των καταχωρημένων δεδομένων (κύρια κλειδιά, μοναδικά κλειδιά, ξένα κλειδιά, ελέγχους).
Τα περισσότερα των προϊόντων DBMS επιβάλλουν/ελέγχουν τους περιορισμούς ακεραιότητας των δεδομένων με την ολοκλήρωση της κάθε μίας ενέργειας ενημέρωσης των τελευταίων. Στην περίπτωσή μας, ‘συνέπεια’ κατά την εκτέλεση μιας συναλλαγής σημαίνει την απαίτηση όπως η λογική της εφαρμογής λειτουργεί σωστά και έχει επαρκώς

	δοκιμαστεί για αυτό (ορθά διαμορφωμένη συναλλαγή, well-formed transaction), συμπεριλαμβανομένου και του τρόπου με τον οποίο διαχειρίζεται τις εξαιρέσεις (exception handling).
Απομονωμένη (**I**solated)	Οι Härder και Reuter απαίτησαν όπως "…οτιδήποτε συμβαίνει εσωτερικά σε μία συναλλαγή δεν πρέπει σε καμία περίπτωση να είναι αισθητό από τρίτες συναλλαγές που εκτελούνται ταυτόχρονα με την υπό εξέταση συναλλαγή". Στα περισσότερα των σύγχρονων προϊόντων DBMS η εν λόγω απαίτηση έχει αμβλυνθεί, όπως εξηγείται και στο κείμενο των M. Laiho και F. Laux "On SQL Concurrency Technologies - for Application Developers"(http://www.dbtechnet.org/papers/SQL_ConcurrencyTechnologies.pdf), παρόλα αυτά χρειάζεται να λαμβάνεται υπ' όψιν κατά το σχεδιασμό και την ανάπτυξη του κώδικα των εφαρμογών βάσεων δεδομένων. Τα σύγχρονα προϊόντα DBMS χρησιμοποιούν ένα ευρύ φάσμα τεχνολογιών ελέγχου του ταυτοχρονισμού κατά την εκτέλεση των συναλλαγών, σε τρόπο ώστε οι τελευταίες να προστατεύονται από τυχόν παρενέργειες που σχετίζονται με την ταυτόχρονη εκτέλεση τρίτων συναλλαγών. Όσοι αναπτύσσουν κώδικα εφαρμογών χρειάζεται να γνωρίζουν καλά και να χρησιμοποιούν σωστά αυτού του είδους τις υπηρεσίες εκ μέρους του DBMS.
Ανθεκτική (**D**urable)	Δεδομένα που έχει μεταβάλει και προλάβει να επικυρώσει (commit) συναλλαγή επιβιώνουν στους δίσκους των όποιων τυχόν βλαβών του συστήματος που μπορεί να συμβούν στην πορεία.

Το νόημα της αρχής ACID είναι ότι συναλλαγές οι οποίες δεν πληρούν το σύνολο των ιδιοτήτων ACID δεν επικυρώνουν τις μεταβολές που προκαλούν στα δεδομένα της βάσης και ακυρώνονται, είτε από την εφαρμογή είτε από αυτόν καθαυτό τον εξυπηρετητή των βάσεων δεδομένων (DBMS).

2.3 Επίπεδα απομόνωσης

Η ιδιότητα της απομόνωσης στην αρχή ACID συνιστά πρόκληση. Ανάλογα με τους μηχανισμούς επιβολής ελέγχου του ταυτοχρονισμού που χρησιμοποιούνται, μπορεί να προκαλέσει συγκρούσεις μεταξύ ταυτόχρονα εκτελούμενων συναλλαγών και μεγάλους χρόνους αναμονής που έχουν ως συνέπεια καθυστερήσεις στην παραγωγική λειτουργία της βάσης δεδομένων.

Το πρότυπο ISO SQL δεν ορίζει τον τρόπο με τον οποίο θα εφαρμόζεται στην πράξη ο έλεγχος του ταυτοχρονισμού. Αντί να κάνει αυτό, βασίζεται στην απαλοιφή των προβληματικών καταστάσεων (ανωμαλιών κατά τη λειτουργία) που έχουν αναφερθεί. Κατά το πρότυπο, τα επίπεδα απομόνωσης που αποτρέπουν στην πράξη την εμφάνιση αυτών των ανωμαλιών είναι αυτά του Πίνακα 2.1 και περιγράφονται εν συντομία στον Πίνακα 2.2 όσον αφορά στο είδος των ενεργειών ανάγνωσης (read) που επιτρέπεται να εκτελεί το κάθε ένα από αυτά. Μερικά λιγότερο και άλλα περισσότερο περιοριστικά όσον αφορά στο βαθμό της απομόνωσης που επιτυγχάνουν, τα επίπεδα αυτά συνεπάγονται πιθανές αρνητικές επιπτώσεις επί των επιδόσεων του συστήματος.

Αξίζει να σημειωθεί το γεγονός ότι τα επίπεδα απομόνωσης δεν έχουν να κάνουν με περιορισμούς στις ενέργειες εγγραφής (write) δεδομένων. Οι τελευταίες ελέγχονται συνήθως με τη χρήση κλειδαριών μέσω των οποίων, όσο μία συναλλαγή βρίσκεται ακόμη σε εξέλιξη, εγγραφές που αυτή έχει επιτελέσει με επιτυχία, προστατεύονται και δεν είναι δυνατόν να τροποποιηθούν από τρίτες, ταυτόχρονα εκτελούμενες, συναλλαγές. Τροποποιήσεις στις εν λόγω εγγραφές μπορούν να γίνουν μόνον μετά την επιτυχή ολοκλήρωση της συναλλαγής η οποία τις έχει προκαλέσει.

Πίνακας 2.1 Τα επίπεδα απομόνωσης ISO SQL και ανωμαλίες που αυτά αντιμετωπίζουν

Ανώμαλη λειτουργία / Επίπεδο απομόνωσης:	Χαμένη ενημέρωση	Πρόχειρη ανάγνωση	Μη επαναλαμβανόμενη ανάγνωση	Ανάγνωση φαντάσματος
READ UNCOMMITTED	ΟΧΙ	ΙΣΩΣ	ΙΣΩΣ	ΙΣΩΣ
READ COMMITTED	ΟΧΙ	ΟΧΙ	ΙΣΩΣ	ΙΣΩΣ
REPEATABLE READ	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΙΣΩΣ
SERIALIZABLE	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ

Πίνακας 2.2 Επεξήγηση των επιπέδων απομόνωσης ISO SQL (και IBM DB2)

ISO SQL	IBM DB2	Υποστηριζόμενη απομόνωση
Ανάγνωση μη επικυρωμένων (Read Uncommitted)	UR	Επιτρέπεται η πρόχειρη ανάγνωση δεδομένων που έχουν επικαιροποιηθεί από τρίτες ταυτόχρονες συναλλαγές οι οποίες βρίσκονται ακόμη σε εξέλιξη (προφανώς) χωρίς να έχουν επικυρώσει την εικόνα των μεταβολών τους στη βάση
Ανάγνωση επικυρωμένων (Read Committed)	CS (CC)	Δεν επιτρέπεται η ανάγνωση μη επικυρωμένων μεταβολών των ταυτόχρονων συναλλαγών στη βάση. Η Oracle και η IBM DB2 (από την έκδοση 9.7 και μετά) επιτρέπουν την ανάγνωση της πλέον πρόσφατα επικαιροποιημένης έκδοσης των δεδομένων. Ο λόγος για το επίπεδο των τρεχουσών επικυρώσεων (currently committed, CC) της IBM DB2. Άλλα προϊόντα DBMS επιτρέπουν την επικαιροποίηση της εικόνας της βάσης που αισθάνεται η εφαρμογή/πελάτης μόνον μετά τη στιγμή της ολοκλήρωσης και της επικύρωσης των μεταβολών της κάθε μίας συναλλαγής που αυτή εκτελεί.
Επαναλήψιμη ανάγνωση (Repeatable Read)	RS	Επιτρέπεται η ανάγνωση μόνον εκείνων των δεδομένων των οποίων η επικαιροποίηση είχε επικυρωθεί όταν ξεκίνησε η συναλλαγή. Ισοδύναμα, οι ενέργειες ανάγνωσης των εν λόγω δεδομένων μπορούν να επαναληφθούν πολλές φορές στη διάρκεια της συναλλαγής και να οδηγούν πάντα στο ίδιο αποτέλεσμα, ακόμη και αν κάποια από τα δεδομένα στις πλειάδες/γραμμές του τελευταίου μεταβάλλονται με ενέργειες UPDATE ή DELETE επί των αντίστοιχων πινάκων που κάνουν άλλες, ταυτόχρονα εκτελούμενες, συναλλαγές.
Σειριοποιήσιμο (Serializable)	RR	Επιτρέπεται η ανάγνωση μόνον εκείνων των δεδομένων των οποίων η επικαιροποίηση είχε επικυρωθεί όταν ξεκίνησε η συναλλαγή. Ισοδύναμα, οι ενέργειες ανάγνωσης των εν λόγω δεδομένων μπορούν να επαναληφθούν πολλές φορές στη διάρκεια της συναλλαγής και να οδηγούν πάντα στο ίδιο αποτέλεσμα, ακόμη και αν κάποια από τα δεδομένα στις πλειάδες/γραμμές του τελευταίου μεταβάλλονται με ενέργειες INSERT, UPDATE ή DELETE επί των αντίστοιχων πινάκων που κάνουν άλλες, ταυτόχρονα εκτελούμενες, συναλλαγές.

Σημείωση-1 Σημειώνεται η διαφορά στον ονοματισμό των επιπέδων απομόνωσης ανάμεσα στο πρότυπο ISO SQL και στην IBM DB2. Η δεύτερη ξεκίνησε ορίζοντας μόνον δύο επίπεδα απομόνωσης: το επίπεδο του σταθερού δρομέα (cursor stability, CS), και το επίπεδο της επαναλήψιμης ανάγνωσης (repeatable read, RR). Αυτές τις αρχικές ονομασίες, η IBM DB2 δεν τις άλλαξε ακόμη και όταν ορίστηκαν τα τέσσερα επίπεδα ISO SQL στη συνέχεια. Αυτό είχε ως αποτέλεσμα την ύπαρξη απόκλισης στη σημασιολογία του ονοματισμού που εφαρμόζει η IBM σε σχέση με το πρότυπο: το RR της πρώτης αντιστοιχίζεται όχι στο RR αλλά στο Serializable του προτύπου.

Σημείωση-2 Επιπλέον των όσων ορίζει το πρότυπο ISO SQL, η Oracle και ο MS SQL Server υποστηρίζουν ένα επιπλέον επίπεδο απομόνωσης, αυτό της **στιγμιαίας εικόνας** (snapshot isolation level). Στη συγκεκριμένη περίπτωση, η συναλλαγή έχει πρόσβαση μόνο στη στιγμιαία εικόνα που είχε το περιεχόμενο της βάσης με τα δεδομένα τα οποία είχαν επικυρωθεί τη στιγμή που ξεκίνησε να εκτελείται η συναλλαγή. Ισοδύναμα, η υπό εξέταση συναλλαγή δεν έχει πρόσβαση σε τυχόν επικαιροποιήσεις του περιεχομένου της βάσης οι οποίες έγιναν ή γίνονται από τρίτες συναλλαγές που ξεκίνησαν να εκτελούνται αργότερα από τη χρονική στιγμή που ξεκίνησε να εκτελείται η συναλλαγή ή/και εξελίσσονται παράλληλα με την τελευταία. Στην περίπτωση της Oracle, το εν λόγω επίπεδο απομόνωσης ονομάζεται SERIALIZABLE.

Στη συνέχεια πρόκειται να εξεταστούν τα επίπεδα απομόνωσης που υποστηρίζουν τα διάφορα προϊόντα DBMS που θεωρούνται στο παρόν μάθημα, μάλιστα: στο επίπεδο της εφαρμογής τους. Ανάλογα με το προϊόν DBMS, το επίπεδο απομόνωσης συνιστά προεπιλογή στο επίπεδο της βάσης δεδομένων που χρησιμοποιείται, ή στο επίπεδο της συνεδρίας (session) SQL στο ξεκίνημα μια συναλλαγής, ή ακόμη προεπιλογή σε επίπεδο πρότασης σε κώδικα SQL ή επεξεργασίας πίνακα. Ορθή πρακτική, σε συμφωνία με το πρότυπο ISO SQL, συνιστά ο καθορισμός του επιπέδου απομόνωσης στο ξεκίνημα της κάθε μίας συναλλαγής, ανάλογα με τις απαιτήσεις για επεξεργασία που συνεπάγεται η συγκεκριμένη συναλλαγή. Ακολουθώντας το πρότυπο ISO SQL, η Oracle και ο MS SQL Server χρησιμοποιούν την ακόλουθη σύνταξη κώδικα για τον καθορισμό του επιπέδου απομόνωσης μιας συναλλαγής:

```
SET TRANSACTION ISOLATION LEVEL <isolation level>
```

ενώ η IBM DB2 αποκλίνει, χρησιμοποιώντας την εξής σύνταξη κώδικα:

```
SET CURRENT ISOLATION = <isolation level>
```

Παρά τις υπάρχουσες διαφοροποιήσεις στη σύνταξη του σχετικού κώδικα και στον ονοματισμό που εφαρμόζουν τα διάφορα προϊόντα DBMS όταν αναφέρονται σε επίπεδα απομόνωσης, οι προγραμματιστικές διεπαφές (application programming interface, API) ODBC και JDBC εφαρμόζουν πιστά την ονοματολογία του πρότυπου ISO SQL. Για παράδειγμα, το JDBC ορίζει το επίπεδο απομόνωσης ως παράμετρο της μεθόδου setTransactionIsolation στο αντικείμενο που υλοποιεί τη σύνδεση, ως εξής:

```
<connection>.setTransactionIsolation(Connection.<transaction isolation>);
```

…όπου στη θέση του <transaction isolation> μπαίνει η δεσμευμένη λέξη που αντιστοιχεί στο επιθυμητό επίπεδο απομόνωσης, π.χ.

TRANSACTION_SERIALIZABLE για το σειριοποιήσιμο επίπεδο απομόνωσης. Στη συνέχεια, το επίπεδο απομόνωσης που έχει καθοριστεί σε επίπεδο κώδικα JDBC απεικονίζεται, μέσω του κατάλληλου οδηγού JDBC, στο αντίστοιχο επίπεδο απομόνωσης του προϊόντος DBMS που χρησιμοποιείται. Αν συμβεί να μην υπάρχει αντίστοιχο επίπεδο απομόνωσης στο προϊόν DBMS, τότε ο οδηγός JDBC παράγει μία εξαίρεση (SQLException) την οποία επικοινωνεί στο περιβάλλον της εφαρμογής/πελάτη.

2.4 Μηχανισμοί ελέγχου του ταυτοχρονισμού

Τα σύγχρονα προϊόντα DBMS χρησιμοποιούν κυρίως τους επόμενους τρεις μηχανισμούς ελέγχου του ταυτοχρονισμού που υλοποιούν στην πράξη τα επίπεδα απομόνωσης των συναλλαγών:

1. Σχήμα κλειδώματος με μεταβλητό βαθμό ευαισθησίας (multi-granular locking scheme[3], MGL), ή απλά: έλεγχος ταυτοχρονισμού με σχήμα κλειδώματος (locking scheme concurrency control, LSCC)
2. Έλεγχος ταυτοχρονισμού με πολλαπλές εκδόσεις (multi-versioning concurrency control, MVCC)
3. Αισιόδοξος έλεγχος ταυτοχρονισμού (optimistic concurrency control, OCC).

2.4.1 Έλεγχος ταυτοχρονισμού με σχήμα κλειδώματος (Locking Scheme Concurrency Control, LSCC)

Στον Πίνακα 2.3 ορίζεται το βασικό σχήμα κλειδώματος που εφαρμόζει αυτόματα η συνιστώσα (λογισμικό) του **διαχειριστή κλειδώματος** στο DBMS ώστε να διασφαλίζεται η ακεραιότητα των δεδομένων κατά τη διεκπεραίωση ενεργειών ανάγνωσης (read) και ενημέρωσής (write) τους από συναλλαγές που εκτελούνται ταυτόχρονα (δηλαδή, με χρονική επικάλυψη των επιμέρους ενεργειών τους). Από τη στιγμή που το σχήμα επιβάλλει ότι μόνον μία ενέργεια ενημέρωσης (write) θα εκτελείται την κάθε μία χρονική στιγμή σε κάθε πλειάδα/γραμμή δεδομένων, ο διαχειριστής κλειδώματος φροντίζει ώστε να τίθεται αποκλειστικού τύπου προτασία (**κλειδαριά X-lock**) στις πλειάδες στις οποίες βρίσκονται σε εξέλιξη ενέργειες όπως οι INSERT, UPDATE και DELETE. Κλειδαριά X-lock αποδίδεται και τίθεται σε ισχύ μόνον όταν δεν υφίσταται ήδη άλλη κλειδαριά (οποιουδήποτε τύπου) επί της πλειάδας/γραμμής ή επί των πλειάδων/γραμμών που πρόκειται να ενημερωθεί/ούν (βλέπε τον Πίνακα 2.3). Από τη στιγμή που ενεργοποιείται, **η (X-lock) κλειδαριά παραμένει σε ισχύ μέχρι την ολοκλήρωση (επικύρωση ή ακύρωση) της αντίστοιχης συναλλαγής**.

Ο διαχειριστής κλειδώματος (lock manager) διαφαλίζει την ακεραιότητα ενεργειών ανάγνωσης δεδομένων, όπως η ενέργεια SELECT, χρησιμοποιώντας κοινόχρηστες κλειδαριές (**S-locks**). Οι τελευταίες μπορούν να έχουν αποδοθεί ταυτόχρονα σε πολλές εφαρμογές-πελάτη των οποίων οι ενέργειες ανάγνωσης δεν επηρεάζουν

---

3 στη διεθνή βιβλιογραφία, ορισμένοι αποκαλούν συλλογικά τα σχήματα ελέγχου του ταυτοχρονισμού που κάνουν χρήση κλειδώματος ως ‘απαισιόδοξα σχήματα ελέγχου ταυτοχρονισμού (pessimistic concurrency control, PCC)’ και εκείνα που χρησιμοποιούν τεχνικές πολλαπλών εκδόσεων ως ‘αισιόδοξα (optimistic concurrency control, OCC)’, παρά το γεγονός ότι η σημασιολογία του ταυτοχρονισμού στο πραγματικό OCC είναι διαφορετική.

αρνητικά την εγκυρότητα των δεδομένων που διαβάζονται/ανακτώνται. Σε σχέση με το επίπεδο απομόνωσης στο οποίο εκτελείται μία συναλλαγή, μπορεί να ικανοποιούνται/διεκπεραιώνονται ενέργειες ανάγνωσης συναλλαγών για τις οποίες έχει οριστεί να εκτελούνται σε επίπεδο READ UNCOMMITTED: στην περίπτωση αυτή, ο διαχειριστής κλειδώματος δεν απαιτεί την ύπαρξη κλειδαριάς S-lock για να επιτρέψει την ανάγνωση των πλειάδων από τη συγκεκριμένη συναλλαγή. Σε κάθε άλλη περίπτωση επιπέδου απομόνωσης, η κλειδαριά S-lock συνιστά απαραίτητη προϋπόθεση για την ανάγνωση και τίθεται/ενεργοποιείται στη γραμμή, ή στις γραμμές, που πρόκειται να διαβαστεί/ούν μόνον όταν δεν συμβαίνει να έχει ήδη τεθεί κλειδαριά X-lock από άλλη (ταυτόχρονα εκτελούμενη) συναλλαγή.

Πίνακας 2.3 Συμβατότητα S- και X-τύπου κλειδαριών

Όταν μία συναλλαγή αιτείται να θέσει τον ακόλουθο τύπο κλειδαριάς σε πλειάδα…	…τη στιγμή που άλλη συναλλαγή έχει ήδη θέσει αυτόν τον τύπο κλειδαριάς στην ίδια πλειάδα		…όταν η εν λόγω πλειάδα δεν έχει κλειδωθεί από τρίτη συναλλαγή
	S-lock	**X-lock**	
S-lock	μπαίνει η νέα κλειδαριά	αναμονή για απελευθέρωση κλειδαριάς	μπαίνει η νέα κλειδαριά
X-lock	αναμονή για απελευθέρωση κλειδαριάς	αναμονή για απελευθέρωση κλειδαριάς	μπαίνει η νέα κλειδαριά

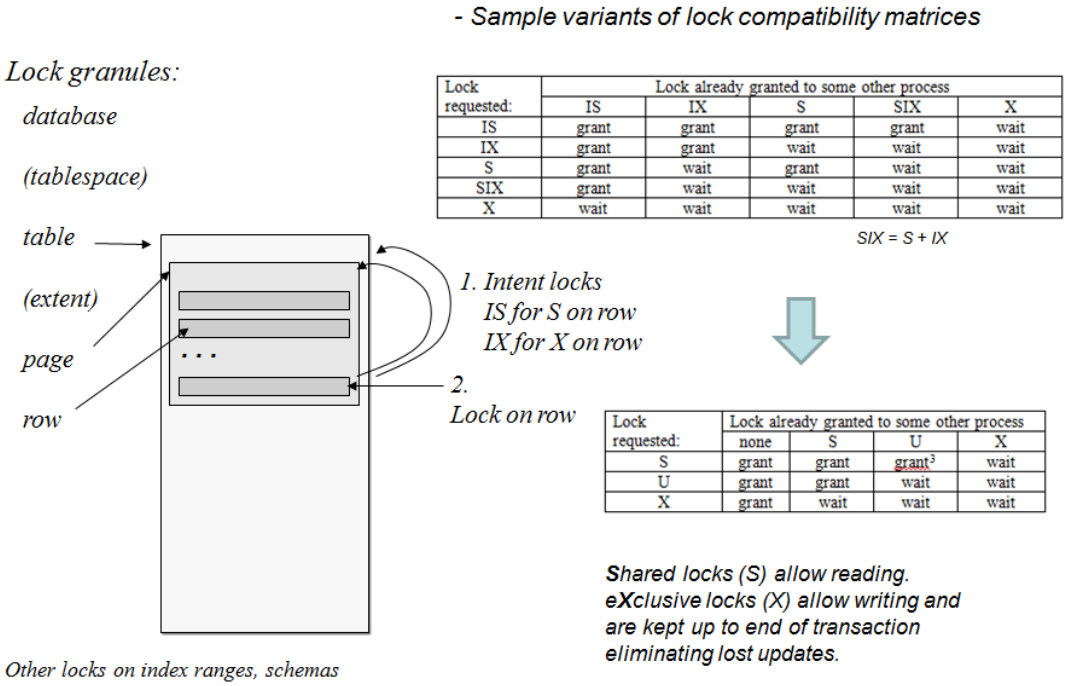
Στην περίπτωση που συναλλαγή εκτελείται σε επίπεδο απομόνωσης RED COMMITTED, η κάθε κλειδαριά S-lock που αυτή θέτει, αυτή απελευθερώνεται με την ενέργεια ανάγνωσης των δεδομένων. Αντίθετα, στην περίπτωση των επιπέδων απομόνωσης REPEATABLE READ και SERIALIZABLE, οι κλειδαριές S-lock διατηρούνται έως και την ολοκλήρωση (επικύρωση ή ακύρωση) των συναλλαγών που τις θέτουν. Σε κάθε περίπτωση, όλες οι κλειδαριές[4] απελευθερώνονται με την ολοκλήρωση της συναλλαγής που τις έχει θέσει, ανεξάρτητα από το είδος αυτής της ολοκλήρωσης (επικύρωση/COMMIT, ή ακύρωση/ROLLBACK).

Σημείωση Ορισμένα προϊόντα DBMS η σύνταξη SQL που χρησιμοποιείται επιτρέπει να δηλώνεται ρητά η εντολή LOCK TABLE που κλειδώνει ολόκληρο τον πίνακα. Και σε αυτήν την περίπτωση, η κλειδαριά που τίθεται σε επίπεδο πίνακα απελευθερώνεται πάντα αυτόματα όταν τερματίζει η συναλλαγή, ενώ όταν πρόκειται για S-lock κλειδαριά, η απελευθέρωσή της γίνεται νωρίτερα, με την ολοκλήρωση της διαδικασίας της ανάγνωσης, εφόσον η συναλλαγή εκτελείται σε επίπεδο απομόνωσης READ COMMITTED. Οι διάφορες παραλλαγές/διάλεκτοι της SQL δεν υποστηρίζουν τη ρητή δήλωση UNLOCK TABLE στη σύνταξη των εντολών, με εξαίρεση εκείνη του προϊόντος MySQL/InnoDB.

Στην πράξη, τα σχήματα κλειδώματος που εφαρμόζουν τα προϊόντα DBMS είναι πιο πολύπλοκα. Γίνεται χρήση κλειδαριών μεταβλητού βαθμού ευαισθησίας, όπως

4 εξαίρεση συνιστά η ύπαρξη ενεργούς δήλωσης WITH HOLD σε δρομέα επί τρέχουσας γραμμής σε μερικά προϊόντα, όπως η IBM DB2

κλειδαριών στο επίπεδο των: πλειάδας/γραμμής, σελίδας μνήμης, πίνακα, διαστήματος τιμών ευρετηρίου, σχεσιακού σχήματος, κλπ., μάλιστα: χρησιμοποιώντας μεθόδους κλειδώματος επιπλέον των ήδη γνωστών S-lock και X-lock. Στην πορεία της εξυπηρέτησης αιτημάτων για κλείδωμα γραμμής/ών, οι διαχειριστές κλειδώματος ξεκινούν θέτοντας κλειδαριές πρόθεσης (intent/intention locks) σε αντικείμενα μικρότερης της αιτούμενης ευαισθησίας, και με τον τρόπο αυτό μπορούν και ελέγχουν τη συμβατότητα των κλειδαριών που ενεργοποποιούνται κατά την εκτέλεση των συναλλαγών, σύμφωνα με το σχήμα **Κλειδώματος Μεταβλητής Ευαισθησίας** (**Multi-Granular Locking, MGL**) που παρουσιάζεται στην Εικόνα 2.6.

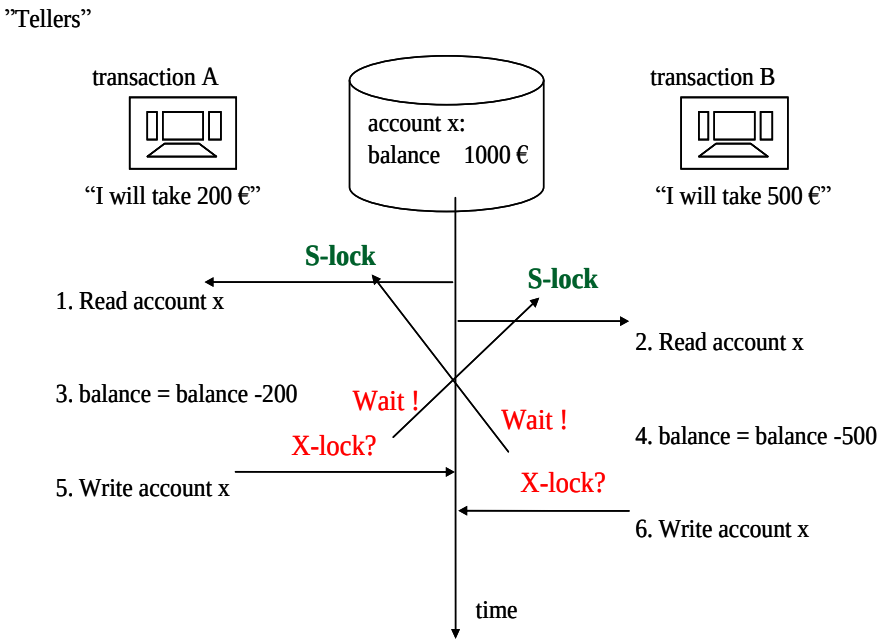


Lock requested:	Lock already granted to some other process				
	IS	IX	S	SIX	X
IS	grant	grant	grant	grant	wait
IX	grant	grant	wait	wait	wait
S	grant	wait	grant	wait	wait
SIX	grant	wait	wait	wait	wait
X	wait	wait	wait	wait	wait

Lock requested:	Lock already granted to some other process			
	none	S	U	X
S	grant	grant	grant[3]	wait
U	grant	grant	wait	wait
X	grant	wait	wait	wait

Εικόνα 2.6 Κλειδαριές διαφόρων επιπέδων ευαισθησίας και η μεταξύ τους συμβατότητα

Εφαρμόζοντας το πρωτόκολλο κλειδώματος αντιμετωπίζεται το πρόβλημα της χαμένης ενημέρωσης (lost update problem), όμως: στην περίπτωση όπου οι ανταγωνίστριες συναλλαγές εκτελούνται σε επίπεδο απομόνωσης που διατηρεί τις κλειδαριές S-lock μέχρι τέλους τους, τότε μπορεί να προκύψει το πρόβλημα που παρουσιάζεται στην Εικόνα 2.7. Πρόκειται για μία κατάσταση αδιέξοδης παύσης (deadlock), όπου οι συναλλαγές περιμένουν η μία την άλλη να ολοκληρώσει, εγκλωβισμένες σε έναν 'κύκλο δίχως τέλος'. Στο παρελθόν, αυτή η κατάταση, κάθε φορά που προέκυπτε, συνιστούσε κρίσιμο πρόβλημα για τη λειτουργία του DBMS. Σήμερα, τα σύγχρονα περιβάλλοντα DBMS είναι εξοπλισμένα με αλληλουχίες (threads) σχετικών ενεργειών οι οποίες τελούν εν υπνώσει και 'ξυπνούν' κάθε δύο (2) δευτερόλεπτα (κάτι που μπορεί και ρυθμίζεται στην πράξη), ελέγχοντας για τυχόν ύπαρξη συμβάντος αδιέξοδης παύσης. Ανιχνεύοντας την ύπαρξη του τελευταίου, εκτελείται το σύνολο της σχετικής αλληλουχίας ενεργειών, επιλέγοντας μία από τις συναλλαγές που εμπλέκονται στην αδιέξοδη παύση την οποία καθιστά θύμα και την αναιρεί (rollback) αυτόματα.



Εικόνα 2.7 LSCC και αδιέξοδη παύση

Η εφαρμογή-πελάτης, της οποίας η συναλλαγή καθίσταται θύμα της αντιμετώπισης της αδιέξοδης παύσης, γίνεται αποδέκτης σχετικού μηνύματος-εξαίρεσης ώστε να επιχειρήσει να επανα-εκτελέσει τη συναλλαγή μετά από παρέλευση ενός τυχαία οριζόμενου, πάντως μικρού, χρονικού διαστήματος. Τυπική περίπτωση προγραμματιστικής διαχείρισης αυτού του ζητήματος συνιστά ο κώδικας Java της ρουτίνας επανα-υποβολής (retry-wrapper) του παραδείγματος BankTransfer στο Παράρτημα 2.

Σημείωση Υπενθυμίζεται ότι κανένα DBMS δεν είναι σε θέση να επανα-εκκινήσει τη συναλλαγή-θύμα μίας περίπτωσης αδιέξοδης παύσης. Η επανα-εκκίνηση της συναλλαγής-θύματος συνιστά αρμοδιότητα του κώδικα της εφαρμογής, ή του εξυπηρετητή εφαρμογών (application server) όπου έχει εγκατασταθεί και λειτουργεί η αντίστοιχη συνιστώσα-πελάτης για πρόσβαση στα δεδομένα. Πρέπει να γίνει κατανοητό ότι ο εντοπισμός αδιέξοδης παύσης δεν συνιστά βλάβη για το σύστημα και ότι η ακύρωση της συναλλαγής-θύματος της αδιέξοδης παύσης συνιστά υπηρεσία του εξυπηρετητή προς τους πελάτες-εφαρμογές. Υπηρεσία που προσφέρεται σε τρόπο ώστε οι τελευταίοι να μπορούν συνεχίζουν την παραγωγική τους λειτουργία στην περίπτωση που προκύπτουν αδιέξοδες παύσεις στις οποίες εμπλέκονται συναλλαγές που (κάποιες στιγμές) αδυνατούν να συνεχίσουν να εκτελούνται ταυτόχρονα.

2.4.2 Έλεγχος ταυτοχρονισμού με πολλαπλές εκδόσεις (Multi-Versioning Concurrency Control, MVCC)

Κατά την εφαρμογή της τεχνικής MVCC, ο εξυπηρετητής συντηρεί ένα χρονολογικά ταξινομημένο ιστορικό με τις διαδοχικές εκδόσεις όλων των πλειάδων/γραμμών. Μέσω του ιστορικού αυτού, καθίσταται δυνατός ο εντοπισμός της επικυρωμένης ενημέρωσης κάθε γραμμής, όπως αυτή έχει τη στιγμή που ξεκινά κάθε μία από τις ταυτόχρονα εκτελούμενες συναλλαγές. Η MVCC δεν συνεπάγεται χρόνους αναμονής προς ανάγνωση δεδομένων από τη βάση και υποστηρίζει δύο (2) επίπεδα απομόνωσης: συναλλαγές που εκτελούνται σε επίπεδο απομόνωσης READ COMMITTED κάνουν χρήση του ιστορικού εκδόσεων και **ανακτούν την πλέον πρόσφατα επικυρωμένη έκδοση των γραμμών** που χρησιμοποιούν, ενώ συναλλαγές που εκτελούνται σε επίπεδο απομόνωσης SNAPSHOT **ανακτούν την πλέον πρόσφατα επικυρωμένη έκδοση των γραμμών όπως αυτή είχε κατά τη στιγμή της εκκίνησής τους** (εκτός και αν πρόκειται για γραμμές που έχουν επικαιροποιηθεί από αυτή καθαυτή τη συναλλαγή που τις διαβάζει σε επίπεδο απομόνωσης SNAPSHOT).

Σημειώνεται ότι συναλλαγή η οποία εκτελείται σε επίπεδο απομόνωσης SNAPSHOT δεν έχει πρόσβαση σε γραμμές-φαντάσματα και δεν μπορεί να αποκλείσει άλλες συναλλαγές από το να δημιουργούν γραμμές-φαντάσματα. Αντίθετα, στην περίπτωση του επίπεδου απομόνωσης SERIALIZABLE που υλοποιείται με την τεχνική LSCC/MGL, οι ταυτόχρονα εκτελούμενες συναλλαγές δεν μπορούν να δημιουργούν γραμμές-φαντάσματα στη βάση. Επιστρέφοντας στο επίπεδο απομόνωσης SNAPSHOT, η συναλλαγή συνεχίζει να επεξεργάζεται γραμμές-φαντασιώσεις στη στιγμιαία εικόνα της βάσης την οποία επεξεργάζεται, δηλαδή γραμμές οι οποίες έχουν στο μεταξύ διαγραφεί από τρίτες, ταυτόχρονα εκτελούμενες, συναλλαγές.

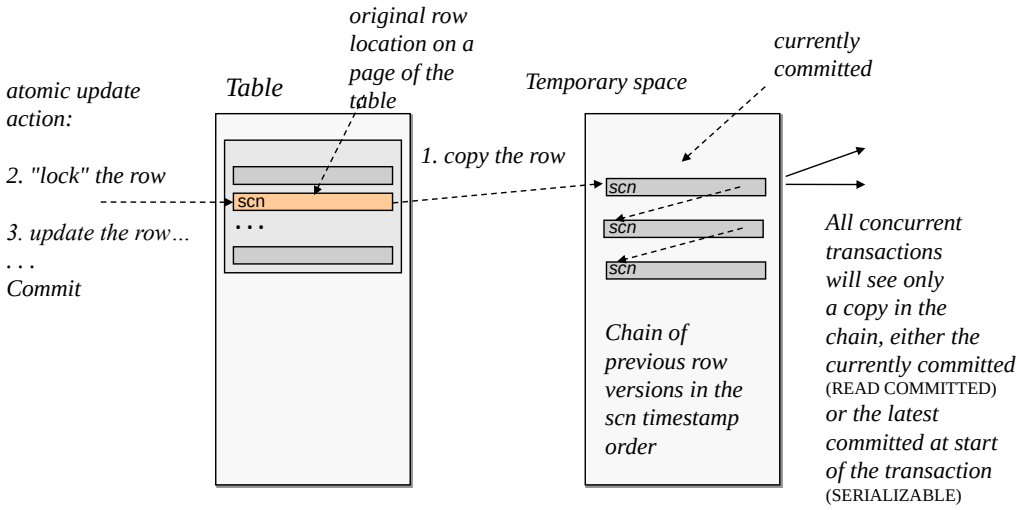
Ανεξάρτητα από το επίπεδο απομόνωσης που εφαρμόζεται, οι υλοποιήσεις της MVCC κάνουν χρήση ενός τύπου κλειδώματος για να αποτρέπουν ενέργειες ενημέρωσης των δεδομένων[5]. Απόπειρες ενημέρωσης γραμμών οι οποίες έχουν ήδη ενημερωθεί από άλλες ταυτόχρονες συναλλαγές προκαλούν μηνύματα λάθους του τύπου “Η στιγμιαία εικόνα που χρησιμοποιείτε δεν είναι πλέον επίκαιρη”.

Υλοποίηση της προσέγγισης MVCC

Παράδειγμα υλοποίησης της MVCC στην Oracle συνιστά η διαδικασία που παρουσιάζεται στην Εικόνα 2.8. Η Oracle χρησιμοποιεί το όνομα SERIALIZABLE για τον ονοματισμό του επίπεδου απομόνωσης SNAPSHOT.

---

5 Το προϊόν SolidDB υλοποιεί την τεχνική MVCC χωρίς τη χρήση κλειδατιών για τις ενέργιες επικαιροποίησης, απλά εφαρμόζοντας ένα σχήμα όπου η πρώτη συναλλαγή που επικαιροποιεί είναι αυτή που κερδίζει στην κούρσα του ταυτοχρονισμού.



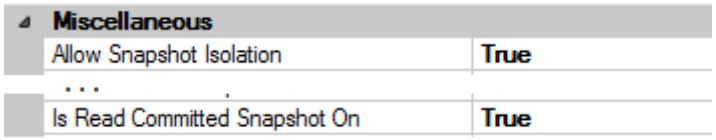
Εικόνα 2.8 Υλοποίηση MVCC με ιστορικό επικυρωμένων πλειάδων/γραμμών

Στην υλοποίηση της MVCC από την Oracle, η συναλλαγή η οποία πρώτη δημιουργεί, επικαιροποιεί, ή διαγράφει μία γραμμή ενεργοποιεί κάποιο είδος κλειδαριάς, σε τρόπο ώστε να είναι αυτή η οποία έχει την αποκλειστικότητα επί της συγκεκριμένης γραμμής σε σχέση με την όποιες τρίτες, ανταγωνίστριες συναλλαγές που εκτελούνται ταυτόχρονα. Οι τελευταίες οδηγούνται σε μία ουρά αναμονής για να επικαιροποιήσουν με τη σειρά τους την εν λόγω πλειάδα/γραμμή της βάσης. Οι ειδικού τύπου κλειδαριές υλοποιούνται με την επισύναψη διακριτών αριθμών μεταβολής συστήματος (System Change Numbers, SCN) στις πλειάδες/γραμμές που επικαιροποιούνται. Οι αριθμοί SCN αντανακλούν την αύξουσα αρίθμηση των συναλλαγών κατά την έναρξή τους. Η κάθε μία πλειάδα/γραμμή της βάσης παραμένει στην αποκλειστική διάθεση μιας ενεργού συναλλαγής όσο ο αριθμός SCN που της επισυνάπτεται αντιστοιχεί στον αύξοντα αριθμό που δηλώνει τη στιγμή έναρξης της τελευταίας. Όταν ο αριθμός SCN της γραμμής δεν αντιστοιχίζεται, πλέον, σε ενεργό συναλλαγή, η γραμμή είναι διαθέσιμη ώστε να 'κλειδωθεί' προς επικαιροποίηση από μία νέα συναλλαγή. Η διαχείριση των ταυτόχρονων ενεργειών ενημέρωσης μέσω κλειδαριών αφήνει ανοικτό το ενδεχόμενο να προκύπτουν αδιέξοδες παύσεις. Μόλις εντοπιστεί περίπτωση αδιέξοδης παύσης, η Oracle δεν ακολουθεί την κλασική συνταγή εντοπισμού μίας συναλλαγής-θύματος την οποία να αναιρέσει (rollback). Αντί αυτού, εντοπίζει την πλειάδα/γραμμή η οποία ενέχεται στην πρόκληση της αδιέξοδης παύσης, ενημερώνει άμεσα με μήνυμα-εξαίρεση τον πελάτη-εφαρμογή συναλλαγή του οποίου έχει 'κλειδώσει' τη συγκεκριμένη πλειάδα και αφήνει σε αυτόν την πρωτοβουλία να προχωρήσει διατυπώνοντας ρητά την εντολή ROLLBACK ώστε να αναιρεθεί η εν λόγω συναλλαγή προς αντιμετώπιση της αδιέξοδης παύσης.

Η τεχνική που εφαρμόζει η Oracle για τον έλεγχο του ταυτοχρονισμού μπορεί να θεωρηθεί ότι είναι υβριδικού τύπου καθώς, επιπλέον της χρήσης της τεχνικής MVCC για το έμμεσο κλείδωμα των πλειάδων που επικαιροποιούνται, υποστηρίζονται η ρητή διατύπωση εντολών "LOCK TABLE" και η ρητή διατύπωση της εντολής "SELECT … FOR UPDATE" η οποία κλειδώνει πλειάδες/γραμμές, εμποδίζοντας την εμφάνιση γραμμών-φαντασμάτων. Επιπλέον αυτών, η Oracle παρέχει τη δυνατότητα δήλωσης **Read Only** συναλλαγών.

Τα οφέλη της χρήσης της τεχνικής MVCC για τον έλεγχο του ταυτοχρονισμού έχουν εκτιμηθεί δέοντος και από την Microsoft. Από την έκδοση 2005 και μετά, ο MS SQL Server μπορεί να ρυθμίζεται ώστε να ενεργοποιούνται και να απενεργοποιούνται λειτουργίες MVCC, κατά βούληση. Πρόκειται για εντολές Transact-SQL οι οποίες

ρυθμίζουν κατάλληλα τις αντίστοιχες ιδιότητες της βάσης δεδομένων που παρουσιάζονται στην Εικόνα 2.9 (έκδοση 2012 και μεταγενέστερες).



Εικόνα 2.9 Ιδιότητες βάσης δεδομένων SQL Server 2012 που αφορούν σε Snapshots

Κλασική περίπτωση υβριδικού μηχανισμού ελέγχου του ταυτοχρονισμού συνιστά η υλοποίηση του τελευταίου στο προϊόν **MySQL/InnoDB** το οποίο υποστηρίζει τέσσερα (4) επίπεδα απομόνωσης όταν πρόκειται για ενέργειες ανάγνωσης δεδομένων:

- Ανάγνωσης μη επικυρωμένων (Read Uncommitted) χωρίς κλειδαριές και χωρίς τη χρήση εκδόσεων των πλειάδω/γραμμών της βάσης
- Ανάγνωσης επικυρωμένων (Read Committed) με τη χρήση MVCC (στην πράξη: ανάγνωση της πλέον πρόσφατα επικυρωμένης έκδοσης των πλειάδων/γραμμών της βάσης)
- Επαναλήψιμης ανάγνωσης (Repeatable Read) με τη χρήση MVCC
- Σειριοποιήσιμο (Serializable) με τη χρήση MGL/LSCC και κλειδαριών X-/S-locks που αποτρέπουν την ύπαρξη γραμμών-φαντασμάτων

2.4.3 Αισιόδοξος έλεγχος ταυτοχρονισμού (Optimistic Concurrency Control, OCC)

Στην πλέον γνήσια έκδοση του μηχανισμού OCC, όλες οι μεταβολές που προκαλούνται από την κάθε συναλλαγή διατηρούνται σε ιδιαίτερο χώρο και όχι στη βάση δεδομένων αυτή καθαυτή. Τα δεδομένα αυτού του ιδιαίτερου χώρου συγχρονίζονται με το περιεχόμενο της βάσης δεδομένων μόνον κατά τη φάση της επικύρωσης (COMMIT) της ολοκλήρωσης της εν λόγω συναλλαγής. Η συγκεκριμένη τεχνική ελέγχου ταυτοχρονισμού εφαρμόζεται αμιγώς στο προϊόν Pyrrho DBMS που έχει αναπτύξει το Πανεπιστήμιο της Δυτικής Σκωτίας (University of the West of Scotland). Το Pyrrho DBMS υποστηρίζει (έμμεσα, με αυτόματο τρόπο) μόνον ένα επίπεδο απομόνωσης: το σειριοποιήσιμο (SERIALIZABLE). Για περισσότερες πληροφορίες, ο ενδιαφερόμενος αναγνώστης μπορεί να επισκεφθεί τη διεύθυνση http://www.pyrrhodb.com.

2.4.4 Σύνοψη

Το πρότυπο ISO SQL έχει αναπτυχθεί από το Αμερικανικό Εθνικό Ινστιτούτο Προτύπων (American National Standards Institute, ANSI) και στην αρχική του έκδοση βασίζονταν σε μεγάλο βαθμό στη διάλεκτο SQL που υλοποιούσαν οι αρχικές εκδόσεις της IBM DB2. Για την ακρίβεια, η IBM δώρισε την εν λόγω έκδοση/διάλεκτο SQL στο ANSI. Η IBM DB2 υλοποιούσε από τότε έναν μηχανισμό ελέγχου του ταυτοχρονισμού που έκανε χρήση κλειδώματος μεταβλητού βαθμού ευαισθησίας (MGL), υποστηρίζοντας μόνον δύο (2) επίπεδα απόμόνωσης των συναλλαγών: το επίπεδο διασφάλισης δρομέα (Cursor Stability, CS), και το επίπεδο της επαναλήψιμης ανάγνωσης (Repetable Read, RR). Προφανώς, η εν λόγω κίνηση της IBM επηρέασε σε μεγάλο βαθμό τη σημασιολογία των νέων επίπεδων

απομόνωσης του σημερινού προτύπου ANSI/ISO SQL (βλέπε Πίνακα 2.2 στην ενότητα 2.3 στα προηγούμενα), επιπέδων απομόνωσης που μπορούν να εξηγούνται μέσω της χρήσης κοινόχρηστων κλειδαριών.

Το πρότυπο SQL δεν κάνει αναφορά και δεν θέτει περιορισμούς που να αφορούν στην υλοποίηση των μηχανισμών ελέγχου ταυτοχρονισμού. Αυτό οδηγεί σε διαφοροποιήσεις από σύστημα σε σύστημα, μάλιστα: και σε περιπτώσεις όπου το ίδιο όνομα επιπέδου απομόνωσης χρησιμοποιείται για να υποδηλώσει κάτι το ελαφρά διαφορετικό. Με αυτήν την έννοια, τα επίπεδα απομόνωσης που αναγράφονται με μπλε φόντο στον Πίνακα 2.4: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ και SERIALIZABLE πρεσβεύουν επίπεδα απομόνωσης όπως αυτά ορίζονται στη στάνταρ σημασιολογία του προτύπου SQL. Τα ίδια ονόματα αναγράφονται μέσα σε διπλά αυτάκια (“) στις στήλες κάποιων προϊόντων DBMS πρεσβεύοντας διαφορετική σημασιολογία από εκείνη του πρότυπου SQL (π.χ. το “serializable” σε μία από τις κυψέλες της στήλης του Oracle DBMS).

Το επίπεδο απομόνωσης “read latest committed” που αναγράφεται στον Πίνακα 2.4 συνιστά δική μας έμπνευση. Η τελευταία στοχεύει στην εξομάλυνση του γεγονότος της χρήσης διαφορετικών ονομάτων από διάφορα προϊόντα DBMS για την περίπτωση επιπέδου απομόνωσης όπου οι ενέργειες ανάγνωσης προχωρούν απρόσκοπτα, χωρίς τη χρήση κλειδαριών, με τη τιμή της πλειάδας/γραμμής που διαβάζεται να είναι είτε εκείνη της πλέον πρόσφατης ενημέρωσής της από την τρέχουσα συναλλαγή, είτε (αν δεν έχει επικυρωθεί ακόμη η μεταβολή του περιεχομένου της γραμμής από τρίτη, ταυτόχρονα εκτελούμενη, συναλλαγή) εκείνη της πλέον πρόσφατα επικυρωμένης έκδοσής της. Πρόκειται για τη σημασιολογία του μηχανισμού της στιγμιαίας εικόνας (snapshot) για την οποία υπάρχει γενικά σύγχυση, ώστε να χρειάζεται να διευκρινίζονται και να αναλύονται περαιτέρω οι σχετικές έννοιες που αφορούν στα αντίστοιχα επίπεδα απομόνωσης των συναλλαγών. Η παραδοχή της ύπαρξης του προβλήματος έχει διατυπωθεί και στο άρθρο “A Critique of ANSI SQL Isolation Levels” (Proc. ACM SIGMOD 95, pp. 1-10, San Jose CA, June 1995) των H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil και P. O’Neil, οι οποίοι έχουν συμμετάσχει στην ανάπτυξη του προτύπου ANSI SQL.

Ο μηχανισμός της στιγμιαίας εικόνας (snapshot) σχετίζεται με προβλήματα που έχουν να κάνουν με την μη συνεπή (consistent) κατάσταση του περιεχομένου της στιγμιαίας εικόνας όταν γίνονται ενέργειες ενημέρωσής του από ταυτόχρονες συναλλαγές. Για παράδειγμα, μπορεί κάποιος να εφαρμόσει τα της άσκησης 2.7 του εργαστηριακού μέρους που ακολουθεί σε διάφορα προϊόντα DBMS και να μελετήσει μερικά από τα προβλήματα που παρουσιάζουν οι στιγμιαίες εικόνες στην πράξη. Τελικά, η πλέον ασφαλής χρήση των τελευταίων είναι εκείνη που γίνεται για την παραγωγή αναφορών επί του περιεχομένου των βάσεων δεδομένων.

Ο Πίνακας 2.4 που ακολουθεί συνοψίζει επί των διαφοροποιήσεων που παρουσιάζουν τα διάφορα προϊόντα DBMS.

Πίνακας 2.4 Υποστήριξη των συναλλαγών κατά το πρότυπο ISO/SQL και στα διάφορα προϊόντα DBMS

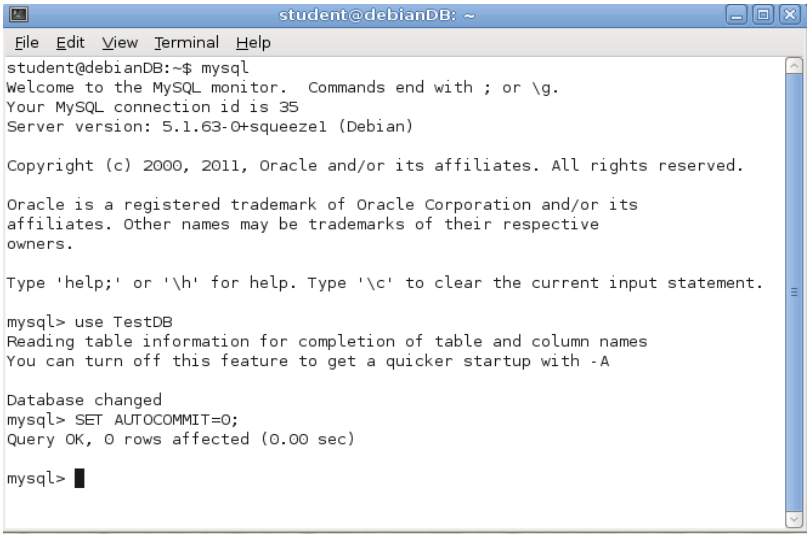
	ANSI/ISO SQL	DB2	Oracle	SQL SERVER	MySQL/InnoDB	PostgreSQL	Pyrrho
	SQL:2006	LUW 9.7	12g1	2012	5.6	9.2	4.8
autocommit (server-side)	n/a	n/a	n/a	yes	yes	yes	yes
Transaction Limits							
implicit start	yes	yes	yes	(configurable)	(configurable)		
explicit start				yes	yes	yes	yes
implicit commit on DDL	n/a	n/a	yes	n/a	yes	n/a	n/a
COMMIT	yes	yes	yes	yes	yes	yes	yes
COMMIT WORK	yes	yes		yes	yes	yes	n/a
ROLLBACK	yes	yes	yes	yes	yes	yes	yes
SAVEPOINT	yes	yes	yes	yes	yes	yes	n/a
ROLLBACK TO SAVEPOINT	yes	yes	yes	yes	yes	yes	n/a
RELEASE SAVEPOINT	yes	yes	yes	n/a	yes	yes	n/a
Isolation levels							
Read Uncommitted	yes	UR	n/a	yes	yes	n/a	n/a
"read latest committed"	n/a	CS (currently committed)	"read committed"	(configurable)	"read committed"	"read committed"	n/a
Read Committed	yes	CS	n/a	yes	n/a	n/a	n/a
Repeatable Read	yes	RS	n/a	yes	n/a	n/a	n/a
snapshot		n/a	"serializable"	(configurable)	"repeatable read"	"serializable"	"serializable"
Serializable	yes	RR	explicit locking	yes	yes	explicit locking	n/a
CC mechanism	n/a						
MGL (locking)		yes		yes	yes		
MVCC (versioning)			yes	(configurable)	yes	yes	
OCC							yes
Cursor processing							
- WITH HOLD	yes	yes		default			
- optimistic locking				yes			

Στιγμιαία εικόνα (snapshot) συνιστά η οπτική επί μιας συνεπούς (consistent) κατάστασης του περιεχομένου της βάσης τη στιγμή έναρξης μιας συναλλαγής. Αυτό το μοντέλο ταιριάζει απόλυτα σε περιπτώσεις συναλλαγών οι οποίες εκτελούν αποκλειστικά και μόνον ενέργειες ανάγνωσης (read only transactions). Η σημασιολογία της στιγμιαίας εικόνας δεν αποκλείει την περίπτωση ύπαρξης φαντασμάτων (phantoms), μόνο που τα τελευταία (ακόμη και αν υφίστανται) δεν συμπεριλαμβάνονται στην στιγμιαία εικόνα της βάσης δεδομένων. Στα προϊόντα DBMS που υλοποιούν το επίπεδο απομόνωσης ISO SQL SERIALIZABLE, η κατάλληλη χρήση κλειδαριών σε επίπεδο πίνακα αποτρέπει την ύπαρξη πλειάδων/γραμμών φαντασμάτων: τα προϊόντα Oracle και PostgreSQL, για παράδειγμα. Στην περίπτωση του μηχανισμού της στιγμιαίας εικόνας, όλα τα προϊόντα που τον χρησιμοποιούν στον έλεγχο ταυτοχρονισμού επιτρέπουν τη διεκπεραίωση ενεργειών SQL INSERT, ενώ υπάρχουν διαφοροποιήσεις από προϊόν σε προϊόν στο κατά πόσον επιτρέπονται οι υπόλοιπες ενέργειες ενημέρωσης των δεδομένων (SQL UPDATE και SQL DELETE).

Παρατηρώντας τα του Πίνακα 2.4, η IBM DB2 είναι το μόνο προϊόν DBMS από αυτά που είναι προεγκατεστημένα στην εικονική μηχανή DBTechNet Debian Linux που δεν χρησιμοποιεί τον μηχανισμό της στιγμιαίας εικόνας στην υλοποίηση των επιπέδων απομόνωσης των συναλλαγών.

2.5 Πρακτική εξάσκηση στο εργαστήριο

Ξεκινά μία νέα συνεδρία χρήστη της MySQL κατά τα γνωστά (Εικόνα 2.10):

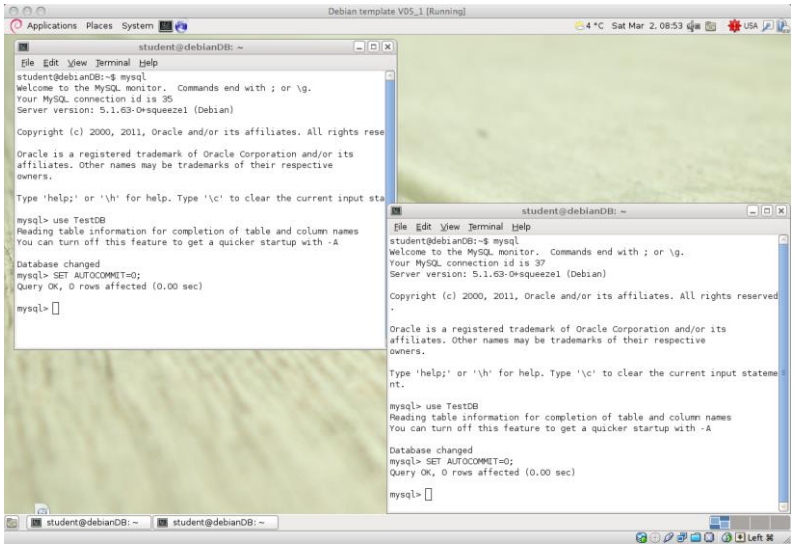


Εικόνα 2.10 Εκκίνηση νέας συνεδρίας χρήστη στη MySQL

ΑΣΚΗΣΗ 2.0

Για τους πειραματισμούς που πρόκειται να ακολουθήσουν και αφορούν στην ταυτόχρονη εκτέλεση συναλλαγών, χρειάζεται να δημιουργηθεί και δεύτερο παράθυρο με νέα συνεδρία MySQL, εφαρμόζοντας τα της Εικόνας 2.10. Ονομάζοντας Session A (= Συνεδρία Α) το παράθυρο/τερματικό που βρίσκεται στο αριστερό τμήμα της οθόνης και Session Β εκείνο του δεξιού τμήματος, προχωρούμε αποκαθιστώντας σύνδεση με τη βάση TestDB και απενεργοποιούμε το AUTOCOMMIT και στις δύο ‘παράλληλες’ συνεδρίες:

```
-----------------------------------
use TestDB
SET AUTOCOMMIT = 0;
-----------------------------------
```



Εικόνα 2.11 Ταυτόχρονες συνεδρίες MySQL στην εικονική μηχανή DebianDB

Όπως θα γίνει κατανοητό στη συνέχεια, συναλλαγές οι οποίες επενεργούν ταυτόχρονα στα ίδια δεδομένα μπορεί να παρεμποδίζουν η μία την άλλη. Για αυτόν τον λόγο οι συναλλαγές πρέπει να σχεδιάζονται ώστε να είναι όσον το δυνατόν πιο σύντομες και να διαρκούν όσο χρόνο χρειάζεται ώστε να επιτελούν το έργο που πρέπει να επιτελέσουν. Η ύπαρξη διαλόγου με τον χρήστη μέσα στη λογική της μπορεί να έχει καταστροφικές συνέπειες για τον χρόνο αναμονής κατά την παραγωγική λειτουργία της συναλλαγής SQL στην πράξη. Κατά συνέπεια, **είναι απόλυτα αναγκαίο να μην υπάρχει περίπτωση όπου μία συναλλαγή SQL να περνά τον έλεγχο στη χρηστική διεπαφή του συστήματος ενόσω αυτή βρίσκεται ακόμη σε εξέλιξη και δεν έχει ολοκληρώσει**.

Το τρέχον επίπεδο απομόνωσης ελέγχεται και μπορεί να παρουσιαστεί μέσω των μεταβλητών συστήματος και της εντολής SELECT ως εξής:

```
-------------------------------------
SELECT @@GLOBAL.tx_isolation, @@tx_isolation;
-------------------------------------
```

Η MySQL/InnoDB προεπιλέγει αυτόματα το επίπεδο απομόνωσης REPEATABLE READ, τόσο σε καθολικό (global) όσο και σε τοπικό επίπεδο της τρέχουσας συνεδρίας (session).

Για σιγουριά, γίνεται ακύρωση/διαγραφή και επανα-δημιουργία του πίνακα Accounts, σε τρόπο ώστε αυτός να καταχωρεί δύο γραμμές δεδομένων:

```
-------------------------------------
DROP TABLE Accounts;
CREATE TABLE Accounts (
acctID  INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL,
CONSTRAINT remains_nonnegative CHECK (balance >= 0)
);
SET AUTOCOMMIT = 0;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
```

Σύμφωνα με το πρότυπο ISO SQL, το επίπεδο απομόνωσης ορίζεται στο ξεκίνημα της συναλλαγής και δεν μπορεί να τροποποιηθεί όσο διαρκεί η τελευταία. Στην πράξη υπάρχουν διαφοροποιήσεις ατον τρόπο με τον οποίο υλοποιείται η συγκεκριμένη απαίτηση. Για τον λόγο αυτό, ελέγχεται και αποδεικνύεται στη συνέχεια το χρονικό σημείο κατά το οποίο μπορεί να ορισθεί ένα επίπεδο απομόνωσης στο περιβάλλον MySQL/InnoDB, κάνοντας χρήση συγκεκριμένων και επί τούτου ορισμένων συναλλαγών:

```
START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;

-- Then another try in different order:
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;
```

Σύμφωνα, πάλι, με το πρότυπο ISO SQL, όταν μία συναλλαγή εκτελείται με επίπεδο απομόνωσης READ UNCOMMITTED δεν επιτρέπεται να συνπεριλαμβάνει ενέργειες ενημέρωσης (write) των δεδομένων. Ας ελέγξουμε την ισχύ ή όχι του συγκεκριμένου περιορισμού στην περίπτωση της MySQL:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
DELETE FROM Accounts;
SELECT COUNT(*) FROM Accounts;
ROLLBACK;
```

Ερωτήσεις

1. Σε τι συμπεράσματα καταλήγετε έχοντας εκτελέσει/δοκιμάσει τα παραπάνω;

ΑΣΚΗΣΗ 2.1

Όπως εξηγήθηκε στα προηγούμενα, το **πρόβλημα της χαμένης ενημέρωσης** έχει να κάνει με την τροποποίηση από άλλη (ταυτόχρονα εκτελούμενη) συναλλαγή γραμμής της οποίας το περιεχόμενο έχει μόλις ενημερωθεί από την τρέχουσα συναλλαγή, μάλιστα: ΠΡΙΝ τον τερματισμό/ολοκλήρωση της τελευταίας. Το συγκεκριμένο πρόβλημα είναι αδύνατο να αναπαραχθεί σε οποιοδήποτε από τα σύγχρονα συστήματα DBMS λόγω των υφιστάμενων υπηρεσιών ελέγχου ταυτοχρονισμού. Παρόλα αυτά, μπορεί να υπάρξει περίπτωση ΑΠΡΟΣΕΚΤΟΥ/ΑΝΕΥΘΥΝΟΥ προγραμματισμού όπου μία ταυτόχρονα εκτελούμενη συναλλαγή, η οποία συνεχίζει και ΜΕΤΑ την ολοκλήρωση της τρέχουσας συναλλαγής, θα προχωρήσει και θα τροποποιήσει εκ νέου, χωρίς να ξανα-διαβάσει το υπό εξέταση αποτέλεσμα και χωρίς να έχει αντιληφθεί τη νέα του τιμή. Η περίπτωση αυτού του είδους της ενημέρωσης ονομάζεται "**Τυφλή Τροποποίηση**" ή "**Πρόχειρη Ενημέρωση**" και μπορεί εύκολα να αναπαραχθεί στο σύγχρονο DBMS.

Αναπαραγωγή της τυφλής τροποποίησης αποτελεί, για παράδειγμα, όταν μία εφαρμογή διαβάζει τιμές από τη βάση δεδομένων στο δίσκο, ενημερώνει τις τιμές αυτές στην κύρια μνήμη και στη συνέχεια γράφει/μεταφέρει τις νέες τιμές στο δίσκο. Στον Πίνακα 2.4 που ακολουθεί γίνεται προσομοίωση αυτής της περίπτωσης χρησιμοποιώντας τοπικές μεταβλητές (local variable) στην MySQL. Η τοπική μεταβλητή δεν είναι τίποτα άλλο από μία θέση στη μνήμης εργασίας (working memory) της τρέχουσας συνεδρίας. Διακρίνεται από το σύμβολο '@' που προηγείται στην αναγραφή του ονόματός της και μπορεί να ενσωματώνεται ως τέτοια στον κώδικα των εντολών SQL με την προϋπόθεση ότι της προσδίδονται/επισυνάπτονται πάντα διακριτές, απλές τιμές.

Το παράδειγμα/πείραμα του Πίνακα 2.4 έχει να κάνει με δύο ταυτόχρονα εκτελούμενες συνεδρίες (Α και Β, όπου τα επιμέρους της κάθε μίας βρίσκονται στην αντίστοιχη στήλη του πίνακα). Στην πρώτη στήλη (στα αριστερά) καταχωρείται αύξουσα αρίθμηση που σημαίνει τη σειρά με την οποία εκτελούνται οι ενέργειες και οι εντολές των δύο άλλων στηλών του πίνακα. Στόχο αποτελεί η προσομοίωση η περίπτωση της Εικόνας 2.2: η συναλλαγή της συνεδρίας Α πρόκειται να αποσύρει €200 από τον τραπεζικό λογαριασμό 101 και η συναλλαγή της συνεδρίας Β πρόκειται να αποσύρει €500 από τον ίδιο λογαριασμό και να ΤΡΟΠΟΠΟΙΗΣΕΙ ΕΚ ΝΕΟΥ (με τυφλή τροποποίηση) την τιμή του υπολοίπου του (εξαφανίζοντας με αυτόν τον τρόπο την ανάληψη των €200 από τη συναλλαγή Α). Κατά συνέπεια, θα προκύψει τελικό αποτέλεσμα ισοδύναμο εκείνου του προβλήματος της χαμένης ενημέρωσης.

Ξεκινούμε με την επαναφορά της αρχικής κατάστασης του περιεχομένου στον πίνακα Accounts, ώστε να μην υπάρξει επηρεασμός από τις τροποποιήσεις που έχουν γίνει στην προηγούμενη άσκηση:

```
-----------------------------------
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID, balance) VALUES (101, 1000);
INSERT INTO Accounts (acctID, balance) VALUES (202, 2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Πίνακας 2.4 Προσομοίωση του προβλήματος της τυφλής τροποποίησης με τη χρήση τοπικών μεταβλητών

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** -- Amount to be transfered by A SET @amountA = 200; SET @balanceA = 0; -- Init value SELECT balance INTO @balanceA FROM Accounts WHERE acctID = 101; SET @balanceA = @balanceA - @amountA; SELECT @balanceA;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** -- Amount to be transfered by B SET @amountB = 500; SET @balanceB = 0; -- Init value SELECT balance INTO @balanceB FROM Accounts WHERE acctID = 101; SET @balanceB = @balanceB - @amountB;
3	UPDATE Accounts SET balance = @balanceA WHERE acctID = 101;	
4		UPDATE Accounts SET balance = @balanceB WHERE acctID = 101;
5	SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**	
6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**

Σημείωση: Στο βήμα αριθμός 4 του Πίνακα 2.4 πρέπει κάποιος να έχει υπ'όψιν ότι στην MySQL ισχύει ως προεπιλογή χρονική διάρκεια ενενήντα (90) δευτερολέπτων στην ισχύ των κλειδαριών. Κατά συνέπεια, η συναλλαγή (πελάτης) της συνεδρίας (session) Α πρέπει να προχωρήσει στο βήμα 5 μέσα σε αυτό το προκαθορισμένο χρονικό διάστημα, αμέσως μετά το βήμα 4 της συναλλαγής/πελάτη Β. Αν υπάρξει καθυστέρηση πέρα από τα 90 δευτερόλεπτα στο βήμα 4 (της συνεδρίας Β) τότε θα πάψει να ισχύει η κλειδαριά που έχει θέσει η συναλλαγή της συνεδρίας Α στην εγγραφή με κωδικό 101 και η προσομοίωση θα αποτύχει.

Εξετάζοντας το σενάριο εκτέλεσης των συναλλαγών Α και Β του Πίνακα 2.4, στο βήμα 1 η Α ετοιμάζεται να αποσύρει €200 από τον τραπεζικό λογαριασμό (χωρίς να έχει προβεί ακόμη σε ενημέρωση της αντίστοιχης εγγραφής στη βάση), στο βήμα 2 η Β ετοιμάζεται με ανάλογο τρόπο να αποσύρει €500, στο βήμα 3 Α ενημερώνει τη βάση, στο βήμα 4 η Β επιχειρεί να ενημερώσει τη βάση όμως περιμένει να ξεκλειδώσει η Α την εγγραφή, στο βήμα 5 η Α ελέγχει το υπόλοιπο του τραπεζικού λογαριασμού και ολοκληρώνει (COMMIT, οπότε η Β ολοκληρώνει το δικό της βήμα 4) και στο βήμα 6 η Β ελέγχει το υπόλοιπο του τραπεζικού λογαριασμού και ολοκληρώνει (COMMIT).

Ερωτήσεις

1. Λειτούργησε το σύστημα με τον τρόπο που αναμένονταν να λειτουργήσει;
2. Υπάρχουν ενδείξεις απώλειας δεδομένων στην περίπτωση αυτή;

Σημείωση: Όλα τα προϊόντα DBMS υποστηρίζουν τον έλεγχο του ταυτοχρονισμού στην εκτέλεση των συναλλαγών. Για τον λόγο αυτό, το πρόβλημα της χαμένης ενημέρωσης δεν εμπίπτει σε κανένα από τα επίπεδα απομόνωσης των συναλλαγών. Παρόλα αυτά, δεν μπορεί να αποτραπεί η περίπτωση προχειρογραμμένου (λανθασμένου) προγραμματιστικού κώδικα όπου οι τυφλές τροποποιήσεις του περιεχομένου της βάσης έχουν εξίσου καταστροφικές επιπτώσεις στο τελευταίο με εκείνες του προβλήματος της χαμένης ενημέρωσης. Αυτό σημαίνει ότι το εν λόγω πρόβλημα είναι δυνατόν να προκύψει 'από την πίσω πόρτα', παρά την θωράκιση που προσφέρουν οι υπηρεσίες ελέγχου του ταυτοχρονισμού κατά την εκτέλεση των συναλλαγών εκ μέρους του DBMS!

ΑΣΚΗΣΗ 2.2α

Επαναλαμβάνεται το σενάριο της άσκησης 2.1, μόνο που τώρα γίνεται χρήση του επίπεδου απομόνωσης SERIALIZABLE που υλοποιείται με κλείδωμα μεταβλητού βαθμού ευαισθησίας (MGL) από την MySQL/InnoDB.

Επαναφέρεται το αρχικό στιγμιότυπο του πίνακα Accounts:

```
-----------------------------------
SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Πίνακας 2.5a Το σενάριο της άσκησης 2.1, τώρα με μακράς διάρκειας κοινόχρηστες κλειδαριές (S-Locks), δηλαδή: σε επίπεδο απομόνωσης SERIALIZABLE

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL SERIALIZABLE;** -- Amount to be transfered by A SET @amountA = 200; SET @balanceA = 0; -- Init value SELECT balance INTO @balanceA FROM Accounts WHERE acctID = 101; SET @balanceA = @balanceA - @amountA; SELECT @balanceA;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL SERIALIZABLE;** -- Amount to be transfered by B SET @amountB = 500; SET @balanceB = 0; -- Init value SELECT balance INTO @balanceB FROM Accounts WHERE acctID = 101; SET @balanceB = @balanceB - @amountB;
3	UPDATE Accounts SET balance = @balanceA WHERE acctID = 101;	
4		-- continue without waiting for A! UPDATE Accounts SET balance = @balanceB WHERE acctID = 101;
5	SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**	
6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**

Ερωτήσεις

1. Ποιό είναι το συμπέρασμα στο οποίο καταλήγετε;
2. Τι γίνεται στην περίπτωση όπου το επίπεδο απομόνωσης των δύο συναλλαγών γίνεται REPEATABLE READ, αντί του SERIALIZABLE;

Σημείωση: Στην MySQL/InnoDB το επίπεδο απομόνωσης REPEATABLE READ υλοποιείται μέσω ελέγχου ταυτοχρονισμού με πολλαπλές εκδόσεις (MVCC) κατά την ανάγνωση, ενώ το επίπεδο SERIALIZABLE υλοποιείται με κλείδωμα πολλαπλού βαθμού ευαισθησίας (MGL).

ΑΣΚΗΣΗ 2.2β

Επανάληψη της άσκησης 2.2α, αυτή τη φορά με ενέργειες "ευαίσθητης ενημέρωσης" (sensitive updates) στο σενάριο εκτέλεσης, χωρίς να γίνεται χρήση τοπικών μεταβλητών.

Επαναφέρεται το αρχικό στιγμιότυπο του πίνακα Accounts:

```
-------------------------------------
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-------------------------------------
```

Πίνακας 2.5β Ανταγωνισμός τύπου SELECT **–** UPDATE με ενέργειες "ευαίσθητης ενημέρωσης"

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL SERIALIZABLE;** SELECT balance FROM Accounts WHERE acctID = 101;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL SERIALIZABLE;** SELECT balance FROM Accounts WHERE acctID = 101;
3	UPDATE Accounts SET balance = balance – 200 WHERE acctID = 101;	
4		UPDATE Accounts SET balance = balance – 500 WHERE acctID = 101;
5	SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**	
6		SELECT acctID, balance FROM Accounts WHERE acctID = 101; **COMMIT;**

Ερωτήσεις

1. Ποιό είναι το συμπέρασμα στο οποίο καταλήγετε;

ΑΣΚΗΣΗ 2.3 Σύγκρουση UPDATE **–** UPDATE σε δύο πόρους, με διαφορετική σειρά

Επαναφέρεται το αρχικό στιγμιότυπο του πίνακα Accounts:

```
-------------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-------------------------------------
```

Πίνακας 2.6 Σενάριο UPDATE-UPDATE

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** UPDATE Accounts SET balance = balance - 200 WHERE acctID = 202;
3	UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;	
4		UPDATE Accounts SET balance = balance + 200 WHERE acctID = 101;
5	**COMMIT;**	
6		**COMMIT;**

Ερωτήσεις

1. Ποιό είναι το συμπέρασμα στο οποίο καταλήγετε;

Σημείωση: Το επίπεδο απομόνωσης στο οποίο εκτελούνται οι συναλλαγές δεν επηρεάζει το παραπάνω σενάριο, παρόλα αυτά συνιστά καλή πρακτική ο καθορισμός του επιπέδου απομόνωσης στο ξεκίνημα της κάθε συναλλαγής! Υπάρχουν περιπτώσεις επεξεργασίας η οποία γίνεται στο παρασκήνιο, όπως ο έλεγχος της ακεραιότητας των δεδομένων επί των τιμών ξένων κλειδιών και εναυσμάτων που σχετίζονται με ενέργειες ανάγνωσης. Ο σχεδιασμός των εναυσμάτων συνιστά αρμοδιότητα των διαχειριστών των βάσεων δεδομένων και δεν εξετάζεται στο πλαίσιο του παρόντος μαθήματος.

ΑΣΚΗΣΗ 2.4 Πρόχειρη ανάγνωση

Συνεχίζοντας με την εξέταση των προβλημάτων στον ταυτοχρονισμό των συναλλαγών, επιδιώκεται τώρα η αναπαραγωγή του προβλήματος της πρόχειρης ανάγνωσης. Η συναλλαγή Α εκτελείται με επίπεδο απομόνωσης REPEATABLE READ (προεπιλογή στη MySQL), ενώ η συναλλαγή Β εκτελείται με επίπεδο απομόνωσης READ UNCOMMITED:

Επαναφέρεται το αρχικό στιγμιότυπο του πίνακα Accounts:

```
-------------------------------------
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-------------------------------------
```

Πίνακας 2.7 Το πρόβλημα της πρόχειρης ανάγνωσης

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL REPEATABLE READ;** UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101; UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ** **UNCOMMITTED;** SELECT * FROM Accounts; **COMMIT;**
3	**ROLLBACK;** SELECT * FROM Accounts; **COMMIT;**	

Ερωτήσεις

1. Σε ποιό συμπέρασμα καταλήγετε; Πόσο αξιόπιστη μπορεί να είναι η συναλλαγή της συνεδρίας Β;
2. Πως θα μπορούσαμε να αντιμετωπίσουμε αυτό το πρόβλημα;

ΑΣΚΗΣΗ 2.5 Μη-επαναλήψιμη ανάγνωση

Ακολουθεί το πρόβλημα της μη-επαναλήψιμης ανάγνωσης:

Επαναφέρεται το αρχικό στιγμιότυπο του πίνακα Accounts:

```
-----------------------------------
SET AUTOCOMMIT=0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Πίνακας 2.8 Το πρόβλημα της μη-επαναλήψιμης ανάγνωσης

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** SELECT * FROM Accounts WHERE balance > 500;	
2		SET AUTOCOMMIT = 0; UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101; UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202; SELECT * FROM Accounts; **COMMIT;**
3	-- Repeating the same query SELECT * FROM Accounts WHERE balance > 500; **COMMIT;**	

Ερωτήσεις

1. Στο βήμα 3, διαβάζει η συναλλαγή της συνεδρίας (session) Α το ίδιο αποτέλεσμα με εκείνο που είχε διαβάσει στο βήμα 1;
2. Διαφοροποιείται η κατάσταση όταν η συναλλαγή της συνεδρίας Α εκτελείται με επίπεδο απομόνωσης REPEATABLE READ?

ΑΣΚΗΣΗ 2.6

Στο σημείο αυτό επιχειρείται η αναπαραγωγή του **προβλήματος του φαντάσματος κατά την εισαγωγή δεδομένων** που αναφέρεται στη σχετική βιβλιογραφία:

Επαναφέρεται το αρχικό στιγμιότυπο του πίνακα Accounts:

```
-----------------------------------
SET AUTOCOMMIT = 0;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
SELECT * FROM Accounts;
COMMIT;
-----------------------------------
```

Πίνακας 2.9 Το πρόβλημα του φαντάσματος κατά την εισαγωγή δεδομένων

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL REPEATABLE READ;** START TRANSACTION READ ONLY;	
2		SET AUTOCOMMIT = 0; INSERT INTO Accounts (acctID, balance) VALUES (301,3000); **COMMIT;**
3	SELECT * FROM Accounts WHERE balance > 1000;	
		INSERT INTO Accounts (acctID, balance) VALUES (302,3000); **COMMIT;**
4	-- Can we see accounts 301 and 302? SELECT * FROM Accounts WHERE balance > 1000; **COMMIT;**	

Ερωτήσεις

1. Υπάρχει σημείο στην πορεία της εκτέλεσής της όπου συναλλαγή της συνεδρίας (session) Β περιμένει να ολοκληρώσει η συναλλαγή στη συνεδρία Α;
2. Η εγγραφή 301 ή/και η εγγραφή 302 που εισάγονται στην πορεία της εκτέλεσης των Α και Β, είναι ορατές στο περιβάλλον της συναλλαγής Α;
3. Αλλάζει το αποτέλεσμα που ανακτάται στο βήμα 4 όταν αντιμεταθούν οι ενέργειες των βημάτων 2 και 3;
4. (προχωρημένου επιπέδου ερώτηση) Ποιά η διαφορά στη διαχείριση των εγγραφών-φαντασμάτων όταν η συναλλαγή της συνεδρίας Α εκτελείται σε επίπεδο απομόνωσης SERIALIZABLE, αντί του REPEATABLE READ;

Το περιβάλλον MySQL/InnoDB υλοποιεί το επίπεδο απομόνωσης REPEATABLE READ με Multi-Versioning. Υπάρχει μία χρονική στιγμή (timestamp) κατά την οποία ενεργοποιείται το στιγμιότυπο της βάσης (snapshot). Είναι, άραγε, η στιγμή εκτέλεσης της εντολής START TRANSACTION, ή η στιγμή της εκτέλεσης της πρώτης εντολής SQL μέσα στη συναλλαγή; Υπενθυμίζεται ότι ακόμη και όταν έχει ήδη οριστεί το επίπεδο απομόνωσης, μπορεί κάποιος να χρησιμοποιήσει την εντολή

START TRANSACTION για να ορίσει επιπλέον ιδιότητες της συναλλαγής, όπως την ιδιότητα READ ONLY στο συγκεκριμένο παράδειγμα.

ΑΣΚΗΣΗ 2.7

Μελέτη του στιγμιότυπου βάσης (snapshot) με ένα διαφορετικό είδος φαντασμάτων. Αυτήν τη φορά ενεργοποιείται και τρίτο παράθυρο/συνεδρία, αυτό του “Πελάτη C” (Client C). Το παράθυρο θα χρησιμοποιηθεί στην πορεία της παρούσας άσκησης.

Στην αρχή δημιουργείται ο πίνακας/περιβάλλον της επεξεργασίας που θα ακολουθήσει:

```
mysql TestDB
SET AUTOCOMMIT = 0;
DROP TABLE T;
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(40), i SMALLINT);
INSERT INTO T (id, s, i) VALUES (1, 'first', 1);
INSERT INTO T (id, s, i) VALUES (2, 'second', 2);
INSERT INTO T (id, s, i) VALUES (3, 'third', 1);
INSERT INTO T (id, s, i) VALUES (4, 'forth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);
COMMIT;
```

Πίνακας 2.10 Το πρόβλημα του φαντάσματος κατά την εισαγωγή και την ενημέρωση, ενημέρωση γραμμής η οποία έχει διαγραφεί (γραμμή-οφθαλμαπάτη / ghost row)

	Session A	**Session B**
1	SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL REPEATABLE READ;** SELECT * FROM T WHERE i = 1;	
2		SET AUTOCOMMIT = 0; **SET TRANSACTION** **ISOLATION LEVEL READ COMMITTED;** UPDATE T SET s = **'Update by B'** WHERE id = 1; INSERT INTO T (id, s, i) VALUES (6, **'Insert Phantom'**, 1); UPDATE T SET s = **'Update Phantom'**, i = 1 WHERE id = 2; DELETE FROM T WHERE id = 5; SELECT * FROM T;
3	-- Repeat the query and do updates SELECT * FROM T WHERE i = 1; INSERT INTO T (id, s, i) VALUES (7, **'inserted by A'**, 1); UPDATE T SET s = **'update by A inside the snapshot'** WHERE id = 3; UPDATE T SET s = **'update by A outside the snapshot'** WHERE id = 4;	

	UPDATE T SET s = **'update by A after update by B'** WHERE id = 1;	
3.5		**-- Client C:** **-- what's the current content?** SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; SELECT * FROM T;
4		**-- Client B continues** **-- without waiting for A** **COMMIT;** SELECT * FROM T;
5	SELECT * FROM T WHERE i = 1; UPDATE T SET s = **'updated after delete?'** WHERE id = 5; SELECT * FROM T WHERE i = 1; **COMMIT;**	
6	SELECT * FROM T; **COMMIT;**	
7		**-- 7. Client C does the final** **-- select** SELECT * FROM T; COMMIT;

Ερωτήσεις

1. Γίνονται ορατές στο περιβάλλον της συναλλαγής Α οι γραμμές που εισάγονται και οι γραμμές που ενημερώνονται από τη συναλλαγή Β;
2. Τι συμβαίνει όταν η συναλλαγή Α επιχειρεί να ενημερώσει τη γραμμή με id=1 που έχει ήδη ενημερώσει η συναλλαγή Β;
3. Τι συμβαίνει όταν η συναλλαγή Α επιχειρεί να ενημερώσει τη γραμμή με id=5 που έχει ήδη διαγράψει η συναλλαγή Β;

Σημείωση: Στο περιβάλλον MySQL/InnoDB, όταν συμβαίνει συναλλαγή να εκτελείται σε επίπεδο απομόνωσης REPEATABLE READ, με το που εκτελείται μία εντολή SQL SELECT, δημιουργείται ένα συνεπές στιγμιότυπο του περιεχομένου της βάσης. Αν στη συνέχεια η συναλλαγή επεξεργάζεται τροποποιώντας γραμμές στον πίνακα ή στους πίνακες που συναποτελούν το στιγμιότυπο της βάσης, το τελευταίο παύει να είναι συνεπές όσον αφορά στα δεδομένα που αυτό περιέχει.

Αποτέλεσμα 2.1 Παράδειγμα εκτέλεσης των της άσκησης 2.7:

```
mysql> -- 5. Client A continues
mysql> SELECT * FROM T WHERE i = 1;
+----+-------------------------------+------+
| id | s                             | i    |
+----+-------------------------------+------+
|  1 | update by A after update by B |    1 |
|  3 | update by A inside snapshot   |    1 |
|  5 | to be or not to be            |    1 |
|  7 | inserted by A                 |    1 |
+----+-------------------------------+------+
4 rows in set (0.00 sec)

mysql> UPDATE T SET s = 'updated after delete?' WHERE id = 5;
Query OK, 0 rows affected (0.00 sec)
Rows matched: 0  Changed: 0  Warnings: 0
mysql> SELECT * FROM T WHERE i = 1;
+----+-------------------------------+------+
| id | s                             | i    |
+----+-------------------------------+------+
|  1 | update by A after update by B |    1 |
|  3 | update by A inside snapshot   |    1 |
|  5 | to be or not to be            |    1 |
|  7 | inserted by A                 |    1 |
+----+-------------------------------+------+
4 rows in set (0.00 sec)
mysql> COMMIT;
Query OK, 0 rows affected (0.03 sec)

mysql> -- 6. Client A continues with a new transaction
mysql> SELECT * FROM T;
mysql> SELECT * FROM T;
+----+-------------------------------+------+
| id | s                             | i    |
+----+-------------------------------+------+
|  1 | update by A after update by B |    1 |
|  2 | Update Phantom                |    1 |
|  3 | update by A inside snapshot   |    1 |
|  4 | update by A outside snapshot  |    2 |
|  6 | Insert Phantom                |    1 |
|  7 | inserted by A                 |    1 |
+----+-------------------------------+------+
6 rows in set (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.03 sec)
```

Σημείωση: Στο τέλος του Παραρτήματος 1 παρατίθενται, προς σύγκριση, αποτελέσματα εκτέλεσης της άσκησης 2.7 στο περιβάλλον του SQL Server 2012.

Παράρτημα 1 Πρακτική Εξάσκηση με Συναλλαγές σε SQL Server

Καλώς ήρθατε σε μια "Περιοδεία μυστηρίου" στον κόσμο των SQL συναλλαγών χρησιμοποιώντας το αγαπημένο σας προϊόν ΣΔΒΣ, κατά τη διάρκεια της οποίας θα μπορείτε να πειραματιστείτε και να επαληθεύσετε ό,τι σας έχει παρουσιαστεί σε αυτό το σεμινάριο. Τα προϊόντα ΣΔΒΔ συμπεριφέρονται λίγο διαφορετικά αναφορικά με τις υπηρεσίες συναλλαγών, πράγμα το οποίο μπορεί να εκπλήξει εσάς - και τους πελάτες σας - αν δεν έχετε επίγνωση αυτών των διαφορών κατά την ανάπτυξη εφαρμογών. Αν έχετε το χρόνο να πειραματιστείτε με περισσότερα από ένα προϊόντα, θα αποκομίσετε μια διευρυμένη εικόνα των υπηρεσιών συναλλαγών που παρέχονται από τα προϊόντα ΣΔΒΔ.

Σημείωση: Τα σενάρια κώδικα που έχουν γραφτεί για τα DBMS προιόντα, στο πλαίσιο του εργαστηρίου βάσεων δεδομένων DebianDB, είναι διαθέσιμα εδώ:

http://www.dbtechnet.org/download/SQL_Transactions_Appendix1.zip

Σε αυτό το παράρτημα παρουσιάζουμε μια σειρά πειραμάτισμών μας, που εφαρμόζεται σε SQL Server Express 2012. Αφού ο SQL Server τρέχει μόνο σε πλατφόρμες των Windows, και δεν είναι διαθέσιμός στο DebianDB εργαστήριο, θα παρουσιαστούν επίσης τα περισσότερα από τα αποτελέσματα των ασκήσεων, ώστε να μπορείτε να τα συγκρίνετε με τα αποτελέσματα των δικών σας πειραματισμών. Εάν θέλετε να επαληθεύσετε τα αποτελέσματά μας, μπορείτε να κατεβάσετε το SQL Server Express 2012 δωρεάν, από την ιστοσελίδα της Microsoft για την αντίστοιχη έκδοση Windows του τερματικού εργασίας σας (σε Windows 7 ή νεότερη έκδοση).

Στα ακόλουθα πειράματα, χρησιμοποιούμε το πρόγραμμα SQL Server Management Studio (SSMS). Πρώτα θα δημιουργήσουμε μια νέα βάση δεδομένων την " TestDB", όπως φαίνεται παρακάτω, χρησιμοποιώντας τις προεπιλεγμένες τιμές για όλες τις παραμέτρους διαμόρφωσής της, και με την εντολή USE θα αρχίσουμε να τη χρησιμοποιούμε στην παρακάτω SQL συνεδρία μας.

```
-------------------------------------
CREATE DATABASE TestDB;
USE TestDB;
-------------------------------------
```

ΜΕΡΟΣ 1. ΠΕΙΡΑΜΑΤΙΣΜΟΙ ΜΕ ΣΥΝΑΛΛΑΓΕΣ ΕΝΟΣ ΧΡΗΣΤΗ

Ως προεπιλογή, οι συνεδρίες του SQL Server εκτελούνται σε κατάσταση AUTOCOMMIT, και χρησιμοποιώντας πολλές ρητές (explicit) συναλλαγές, μπορούμε να «χτίσουμε» συναλλαγές πολλαπλών εντολών SQL. Ωστόσο, όλο το στιγμιότυπο του διακομιστή μπορεί να ρυθμιστεί έτσι ώστε να χρησιμοποιεί έμμεσες (implicit) συναλλαγές. Επιπλέον, μπορεί να ρυθμιστεί μία μόνο συνεδρία SQL ώστε να χρησιμοποιεί έμμεσες συναλλαγές, με τη χρήση της επόμενης εντολής του SQL Server

```
SET IMPLICIT_TRANSACTIONS ON;
```

η οποία θα είναι σε ισχύ μέχρι το τέλος της συγκεκριμένης συνεδρίας, ενώ παύει να ισχυεί με την χρήση της εντολής:

```
SET IMPLICIT_TRANSACTIONS OFF;
```

Ας αρχίσουμε τον πειραματισμό με τις συναλλαγές ενός χρήστη. Για αρχή, θα δείξουμε και κάποια βασικά αποτελέσματα:

```
-----------------------------------------------------------------
-- Άσκηση 1.1
-----------------------------------------------------------------
--  Autocommit mode
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), si SMALLINT);
Command(s) completed successfully.

INSERT INTO T (id, s) VALUES (1, 'first');
(1 row(s) affected)

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)

ROLLBACK; -- What happens?
Msg 3903, Level 16, State 1, Line 3
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)


BEGIN TRANSACTION;  -- An explicit transaction begins

INSERT INTO T (id, s) VALUES (2, 'second');

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           second                             NULL
(2 row(s) affected)

ROLLBACK;
```

```
SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
(1 row(s) affected)
```

Κατά τη διάρκεια της συναλλαγής η εντολή ROLLBACK εκτελέστηκε, αλλά τώρα έχουμε γυρίσει σε κατάσταση AUTOCOMMIT!

```
------------------------------------------------------------------
-- Άσκηση 1.2
------------------------------------------------------------------
INSERT INTO T (id, s) VALUES (3, 'third');
(1 row(s) affected)

ROLLBACK;
Msg 3903, Level 16, State 1, Line 3
The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION.

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
3           third                           NULL
(2 row(s) affected)

COMMIT;
Msg 3902, Level 16, State 1, Line 2
The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION.



------------------------------------------------------------------
-- Άσκηση 1.3
------------------------------------------------------------------
BEGIN TRANSACTION;

DELETE FROM T WHERE id > 1;
(1 row(s) affected)

COMMIT;

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL

(1 row(s) affected)


------------------------------------------------------------------
```

-- Άσκηση 1.4
-- DDL stands for Data Definition Language. In SQL, the statements like
-- CREATE, ALTER and DROP are called DDL statements.
-- Now let's test use of DDL commands in a transaction!

SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO T (id, s) VALUES (2, 'will this be committed?');
CREATE TABLE T2 (id INT); -- testing use of a DDL command in atransaction!
INSERT INTO T2 (id) VALUES (1);
SELECT * FROM T2;
ROLLBACK;
GO -- GO marks the end of a batch of SQL commands to be sent to the server
(1 row(s) affected)

(1 row(s) affected)
id

1
(1 row(s) affected)

SELECT * FROM T; **-- What has happened to T ?**
id s si
--- --- ---
1 first NULL
(1 row(s) affected)

SELECT * FROM T2; **-- What has happened to T2 ?**
Msg 208, Level 16, State 1, Line 2
Invalid object name 'T2'.

-- Άσκηση 1.5a

DELETE FROM T WHERE id > 1;
COMMIT;

-- Testing if an error would lead to automatic rollback in SQL Server.
-- @@ERROR is the SQLCode indicator in Transact-SQL, and
-- @@ROWCOUNT is the count indicator of the effected rows

INSERT INTO T (id, s) VALUES (2, **'The test starts by this'**);
(1 row(s) affected)

SELECT 1/0 AS dummy; **-- Division by zero should fail!**
dummy

Msg 8134, Level 16, State 1, Line 1
Divide by zero error encountered.

SELECT @@ERROR AS 'sqlcode'
sqlcode

8134
(1 row(s) affected)

UPDATE T SET s = 'foo' WHERE id = 9999; **-- Updating a non-existing row**
(0 row(s) affected)

SELECT @@ROWCOUNT AS 'Updated'

```
Updated
-----------
0
(1 row(s) affected)

DELETE FROM T WHERE id = 7777;   -- Deleting a non-existing row
(0 row(s) affected)

SELECT @@ROWCOUNT AS 'Deleted'
Deleted
-----------
0
(1 row(s) affected)

COMMIT;

SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           The test starts by this        NULL
(2 row(s) affected)

INSERT INTO T (id, s) VALUES (2, 'Hi, I am a duplicate')
INSERT INTO T (id, s) VALUES (3, 'How about inserting too long string value?')
INSERT INTO T (id, s, si) VALUES (4, 'Smallint overflow for 32769?', 32769);
INSERT INTO T (id, s) VALUES (5, 'Is the transaction still active?');
SELECT * FROM T;
COMMIT;
GO

Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'PK__T__3213E83FD0A494FC'. Cannot insert
duplicate key in object 'dbo.T'. The duplicate key value is (2).
The statement has been terminated.
Msg 8152, Level 16, State 14, Line 2
String or binary data would be truncated.
The statement has been terminated.
Msg 220, Level 16, State 1, Line 3
Arithmetic overflow error for data type smallint, value = 32769.
The statement has been terminated.
Msg 8152, Level 16, State 14, Line 4
String or binary data would be truncated.
The statement has been terminated.
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
2           The test starts by this        NULL
(2 row(s) affected)


BEGIN TRANSACTION;
SELECT * FROM T;
DELETE FROM T WHERE id > 1;
COMMIT;



----------------------------------------------------------------------
```

-- Άσκηση 1.5b
-- This is special to SQL Server only!

SET XACT_ABORT ON; -- In this mode an error generates automatic rollback
SET IMPLICIT_TRANSACTIONS ON;

SELECT 1/0 AS dummy; **-- Division by zero**

INSERT INTO T (id, s) VALUES (6, 'insert after arithm. error');
COMMIT;

SELECT @@TRANCOUNT AS 'do we have an transaction?'
GO

dummy

Msg 8134, Level 16, State 1, Line 3
Divide by zero error encountered.

SET XACT_ABORT OFF; -- In this mode an error does not generate automatic rollback

SELECT * FROM T;
id s si
--- --- ---
1 first NULL
2 The test starts by this NULL
(2 row(s) affected)

-- What happened to the transaction?

-- Exercise 1.6 Experimenting with Transaction Logic

SET NOCOUNT ON; -- Skipping the "n row(s) affected" messages
DROP TABLE Accounts;
SET IMPLICIT_TRANSACTIONS ON;

CREATE TABLE Accounts (
acctID INTEGER NOT NULL PRIMARY KEY,
balance INTEGER NOT NULL
CONSTRAINT unloanable_account CHECK (balance >= 0)
);

COMMIT;

INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);

SELECT * FROM Accounts;
acctID balance
---
101 1000
202 2000

COMMIT;

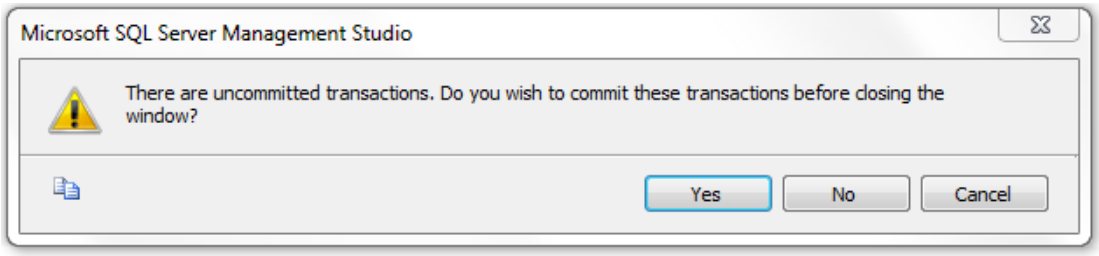
-- **Let's try the bank transfer**
UPDATE Accounts SET balance = balance - 100 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100 WHERE acctID = 202;

SELECT * FROM Accounts;

```
acctID      balance
----------- -----------
101         900
202         2100

ROLLBACK;

-- Let's test if the CHECK constraint actually works:
UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
Msg 547, Level 16, State 0, Line 2
The UPDATE statement conflicted with the CHECK constraint "unloanable_account". The
conflict occurred in database "TestDB", table "dbo.Accounts", column 'balance'.
The statement has been terminated.

UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         4000

ROLLBACK;

-- Transaction logic using the IF structure of Transact-SQL
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

UPDATE Accounts SET balance = balance - 2000 WHERE acctID = 101;
Msg 547, Level 16, State 0, Line 4
The UPDATE statement conflicted with the CHECK constraint "unloanable_account". The
conflict occurred in database "TestDB", table "dbo.Accounts", column 'balance'.
The statement has been terminated.

IF @@error <> 0 OR @@rowcount = 0
  ROLLBACK
ELSE BEGIN
  UPDATE Accounts SET balance = balance + 2000 WHERE acctID = 202;
  IF @@error <> 0 OR @@rowcount = 0
    ROLLBACK
  ELSE
    COMMIT;
  END;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000
COMMIT;

-- How about using a non-existent bank account?
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 777;

SELECT * FROM Accounts ;
acctID      balance
----------- -----------
```

```
101         500
202         2000
ROLLBACK;

-- Fixing the case using the IF structure of Transact-SQL
SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
IF @@error <> 0 OR @@rowcount = 0
  ROLLBACK
ELSE BEGIN
  UPDATE Accounts SET balance = balance + 500 WHERE acctID = 707;
  IF @@error <> 0 OR @@rowcount = 0
    ROLLBACK
  ELSE
    COMMIT;
  END;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000


-----------------------------------------------------------------
-- Άσκηση 1.7   Testing the database recovery
-----------------------------------------------------------------
DELETE FROM T WHERE id > 1;
COMMIT;

BEGIN TRANSACTION;
INSERT INTO T (id, s) VALUES (9, 'What happens if ..');

SELECT * FROM T;
id          s                             si
----------- ----------------------------- ------
1           first                         NULL
9           What happens if ..            NULL
```

Αν δοκιμάσουμε τώρα να κάνουμε **<u>έξοδο από το SQL Server Management Studio</u>**, θα λάβουμε το ακόλουθο μήνυμα:



και για τους σκοπούς του πειράματός μας θα επιλέξουμε το «Όχι».

Με την **επανέναρξη του Management Studio** και κατά τη σύνδεση με την TestDB βάση μας, μπορούμε να μελετήσουμε τι συνέβει στην τελευταία μη επικυρωμένη συναλλαγή μας, απλώς παρατηρώντας τα περιεχόμενα του πίνακα T.

```
SET NOCOUNT ON;
SELECT * FROM T;
id          s                              si
----------- ------------------------------ ------
1           first                          NULL
```

ΜΕΡΟΣ 2 ΠΕΙΡΑΜΑΤΙΣΜΟΙ ΜΕ ΤΑΥΤΟΧΡΟΝΕΣ ΣΥΝΑΛΛΑΓΕΣ

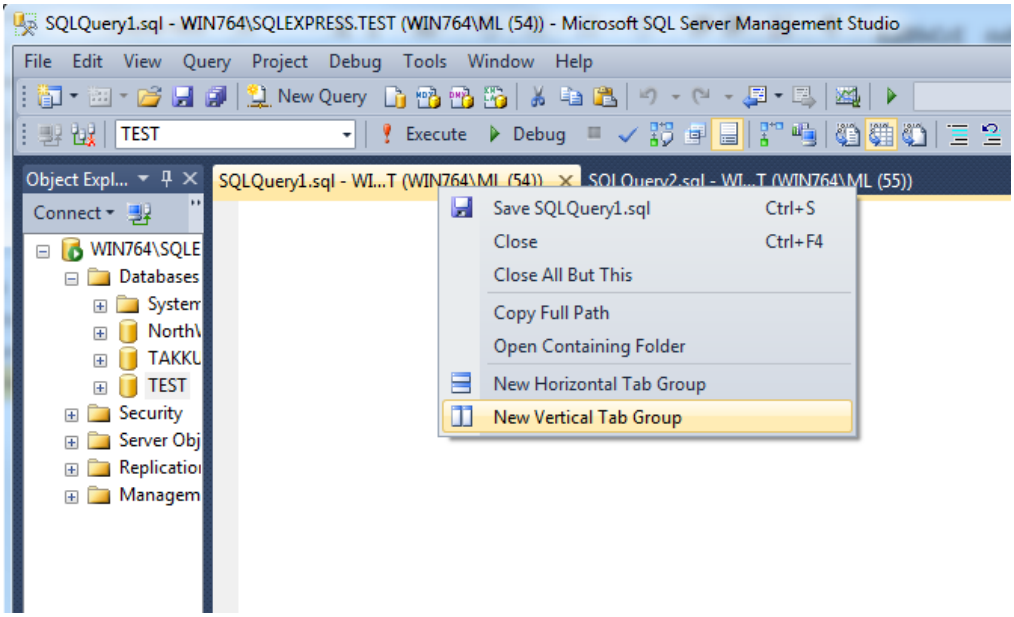
Για τα πειράματα του ταυτοχρονισμού θα ανοίξουμε δύο παράλληλα παράθυρα SQL εντολών, με τις SQL συνεδρίες του "πελάτη Α" και του "πελάτη Β" που θα έχουν πρόσβαση στην ίδια βάση δεδομένων TestDB. Επιλέγουμε το αποτέλεσμα για δύο συνεδρίες να εμφανίζεται σε μορφή κειμένου με τις ακόλουθες επιλογές του μενού

Query > Results To > Results to Text

και επιλέγουμε τη χρήση έμμεσων συναλλαγών και στις δύο συνεδρίες

SET IMPLICIT_TRANSACTIONS ON;

Για την καλύτερη οπτική χρήση του Management Studio, μπορούμε να επιλέξουμε τα παράλληλα παράθυρα SQLQuery να εμφανίζονται κάθετα, το ένα πλάι στο άλλο, πατώντας το δεξί κουμπί του ποντικιού στον τίτλο οποιουδήποτε SQLQuery παραθύρου και διαλέγοντας από το pop-up παράθυρο την επιλογή " New Vertical Tab Group" (βλέπε Εικόνα 1.1).



Εικόνα Α1.1. Άνοιγμα δύο SQLQuery παραθύρων

```
-----------------------------------------------------------------
-- Άσκηση 2.1   "Lost update problem simulation"
-----------------------------------------------------------------
-- 0. To start with fresh contents we enter following commands on a session
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;
```

Το πρόβλημα της χαμένης ενημέρωσης (Lost Update) δημιουργείται αν κάποια εισαγόμενη ή ενημερωμένη γραμμή του πίνακα ενημερωθεί ή διαγραφεί από κάποια δεύτερη ταυτόχρονη συναλλαγή, πριν ολοκληρωθεί η πρώτη συναλλαγή. Αυτό είναι δυνατόν να συμβεί σε λύσεις βασισμένες σε αρχεία τύπου "NoSQL", αλλά όλα τα

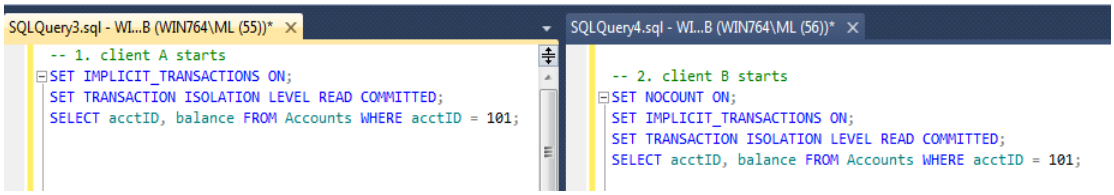
σύγχρονα προϊόντα ΣΔΒΔ θα αποτρέψουν κάτι τέτοιο. Ωστόσο, μετά την επικύρωση της πρώτης συναλλαγής, κάθε άλλη ανταγωνιστική συναλλαγή μπορεί να αντικαταστήσει τις επικυρωμένες γραμμές της πρώτης συναλλαγής.

Στο παρακάτω παράδειγμα, θα προσομοιωθεί το σενάριο της χαμένης ενημέρωσης χρησιμοποιώντας το READ COMMITTED επιπέδο απομόνωσης, το οποίο δεν τηρεί τις S-κλειδαριές. Αρχικά, οι εφαρμογές του πελάτη θα διαβάσουν τις τιμές της στήλης balance, απελευθερώνοντας τις S-κλειδαριές.

-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

```
acctID      balance
----------- -----------
101         1000
```

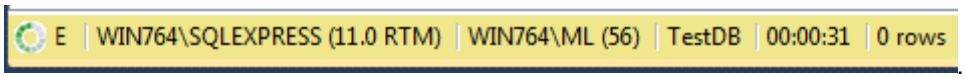
-- 2. Client B starts
SET NOCOUNT ON;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;



Εικόνα Α1.2 Παράθυρα ανταγωνιστικών SQL Συνεδριών

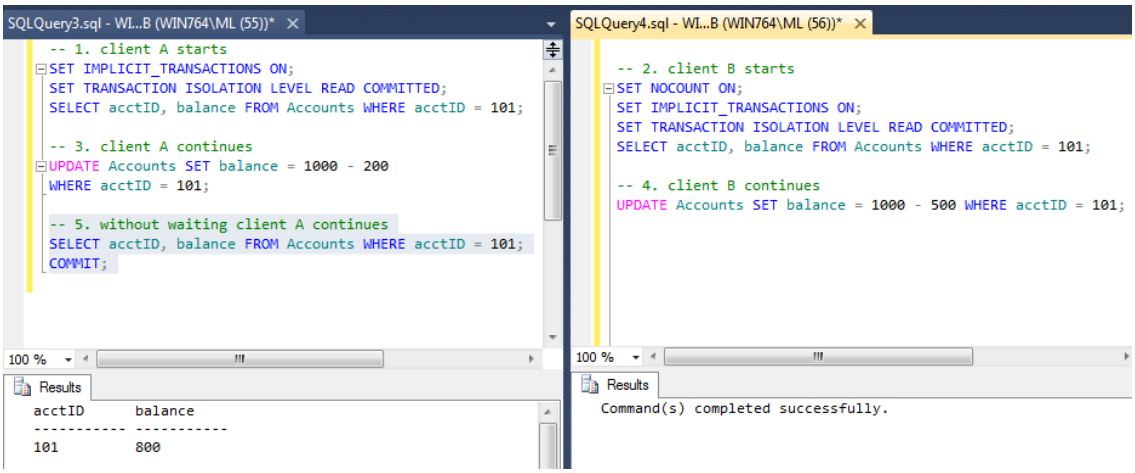
-- 3. Client A continues
UPDATE Accounts SET balance = 1000 - 200 WHERE acctID = 101;

-- 4. Client B continues
UPDATE Accounts SET balance = 1000 - 500 WHERE acctID = 101;

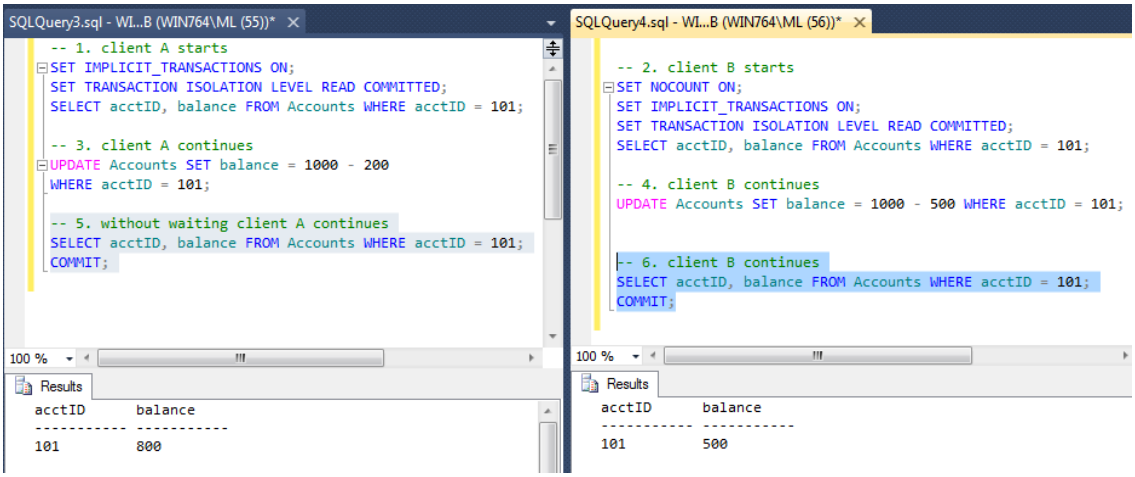


...waiting for A ...

-- 5. Client A continues
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;



-- 6. Client B continues
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
COMMIT;



Έτσι, το τελικό αποτέλεσμα είναι εσφαλμένο!

Σημείωση: Σε αυτό το πείραμα, δεν είχαμε το «πραγματικό» πρόβλημα χαμένης ενημέρωσης, αλλά αφού ο Α επικυρώσει τη συναλλαγή του, ο Β μπορεί να προχωρήσει και να αντικαταστήσει την ενημέρωση του Α. Αυτό το πρόβλημα αξιοπιστίας είναι γνωστό σαν **τυφλή τροποποίηση (Blind Overwriting)** ή **πρόχειρη ενημέρωση (Dirty Write)**. Αυτό το πρόβλημα μπορεί να λυθεί αν οι εντολές ενημέρωσης UPDATE χρησιμοποιούν ***ευαίσθητες ενημερώσεις (sensitive updates)***, όπως

SET balance = balance – 500

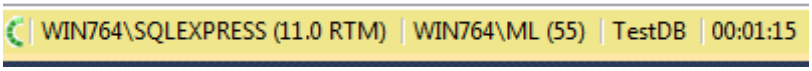
-- **Άσκηση 2.2 "Lost Update Problem" fixed by locks**

-- Competition on a single resource
-- using SELECT .. UPDATE scenarios both client A and B
-- tries to withdraw amounts from the same account.
--
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

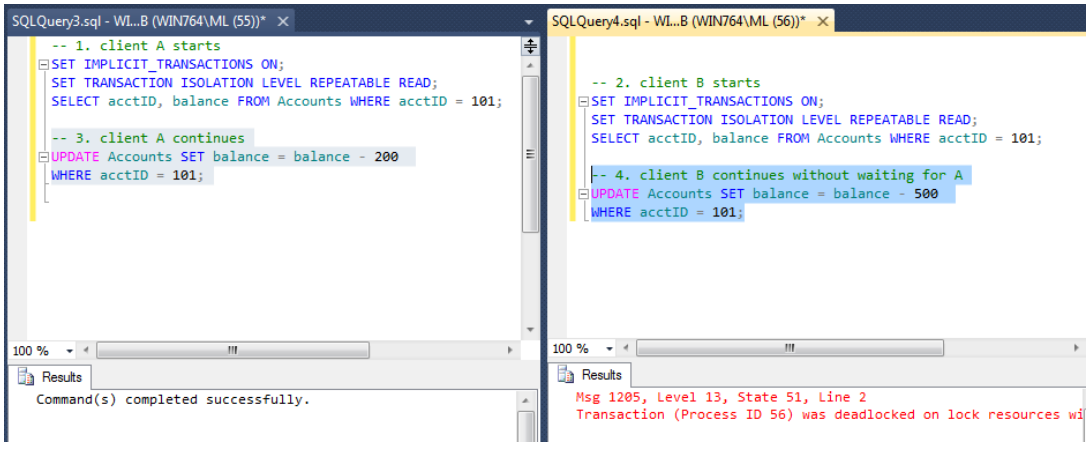
-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT acctID, balance FROM Accounts WHERE acctID = 101;

-- 3. client A continues
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 101;



... waiting for B ...

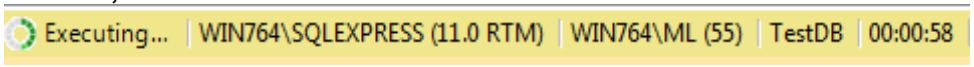
-- 4. Client B continues without waiting for A
UPDATE Accounts SET balance = balance - 500
WHERE acctID = 101;



-- 5. The client which survived will commit
SELECT acctID, balance FROM Accounts WHERE acctID = 101;
acctID   balance
----------- -----------
101   800

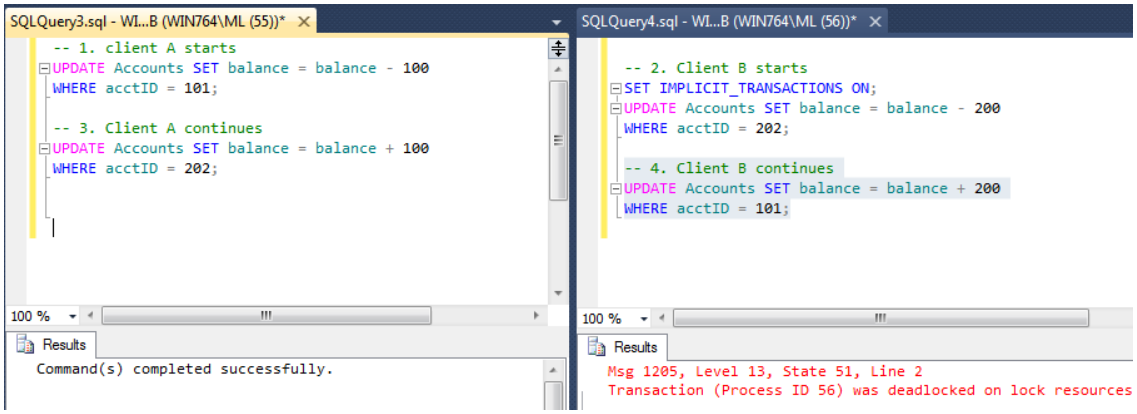
COMMIT;

```
-----------------------------------------------------------------
-- Άσκηση 2.3  Competition on two resources in different order
--             using UPDATE-UPDATE scenarios
-----------------------------------------------------------------
-- Client A transfers 100 euros from account 101 to 202
-- Client B transfers 200 euros from account 202 to 101
--
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;

-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 200
WHERE acctID = 202;

-- 3. Client A continues
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 202;

COMMIT;
```



... waiting for B ...

```
-- 4. Client B continues
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 101;
```




```
-- 5. Client A continues if it can ...
COMMIT;
```

Στις ασκήσεις 2.4 - 2.7, θα πειραματιστούμε με κάποιες **ανωμαλίες ταυτοχρονισμού**, δηλαδή με τους γνωστούς, σύμφωνα με το ISO SQL πρότυπο,

κινδύνους που αφορούν στην αξιοπιστία των δεδομένων. Μπορούμε να τους προσδιορίσουμε; Πώς μπορούμε να διορθώσουμε αυτές τις ανωμαλίες;

```
-----------------------------------------------------------------
-- Άσκηση 2.4   Dirty Read?
-----------------------------------------------------------------
--
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;

UPDATE Accounts SET balance = balance - 100
WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 100
WHERE acctID = 202;


-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         900
202         2100
COMMIT;


-- 3. Client A continues
ROLLBACK;

SELECT * FROM Accounts;
acctID      balance
----------- -----------
101         1000
202         2000

COMMIT;
```

```
-----------------------------------------------------------------
-- Άσκηση 2.5   Non-repeatable Read?
-----------------------------------------------------------------
-- In non-repeatable read anomaly some rows read in the current transaction
-- may not appear in the resultset if the read operation would be repeated
-- before end of the transaction.

-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

--  Listing accounts having balance > 500 euros
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
101         1000
202         2000


-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
UPDATE Accounts SET balance = balance - 500 WHERE acctID = 101;
UPDATE Accounts SET balance = balance + 500 WHERE acctID = 202;
COMMIT;


-- 3. Client A continues
-- Can we still see the same accounts as in step 1?
SELECT * FROM Accounts WHERE balance > 500;
acctID      balance
----------- -----------
202         2500

COMMIT;
```

```
-----------------------------------------------------------------
-- Άσκηση 2.6   Insert Phantom?
-----------------------------------------------------------------
-- Insert phantoms are rows inserted by concurrent transactions and
-- which the current might see before the end of the transaction.
--
-- 0. First restoring the original contents by client A
SET IMPLICIT_TRANSACTIONS ON;
DELETE FROM Accounts;
INSERT INTO Accounts (acctID,balance) VALUES (101,1000);
INSERT INTO Accounts (acctID,balance) VALUES (202,2000);
COMMIT;

-- 1. Client A starts
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- Accounts having balance > 1000 euros
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
101         1000
202         2000


-- 2. Client B starts
SET IMPLICIT_TRANSACTIONS ON;
INSERT INTO Accounts (acctID,balance) VALUES (303,3000);
COMMIT;


-- 3. Client A continues
-- Let's see the results
SELECT * FROM Accounts WHERE balance > 1000;
acctID      balance
----------- -----------
202         2000
303         3000

COMMIT;
```

Ερώτηση

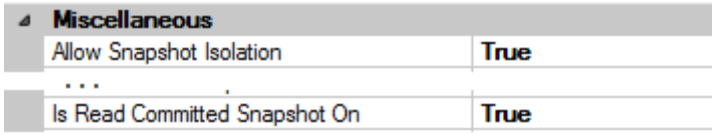
- Πώς μπορούμε να αποφύγουμε τα σφάλματα φαντασμάτων (phantoms);

-- SNAPSHOT STUDIES

-- The database needs to be configured to support SNAPSHOT isolation.
-- For this we create a new database

CREATE DATABASE SnapsDB;

-- to be configured to support snapshots as follows



-- Then both client A and B are switched to use SnapsDB as follows:

USE SnapsDB;

-- Άσκηση 2.7 A Snapshot study with different kinds of Phantoms
---
USE SnapsDB;

-- 0. Setup the test: recreate the table T with five rows
DROP TABLE T;
GO

SET IMPLICIT_TRANSACTIONS ON;
CREATE TABLE T (id INT NOT NULL PRIMARY KEY, s VARCHAR(30), i SMALLINT);
INSERT INTO T (id, s, i) VALUES (1, 'first', 1);
INSERT INTO T (id, s, i) VALUES (2, 'second', 2);
INSERT INTO T (id, s, i) VALUES (3, 'third', 1);
INSERT INTO T (id, s, i) VALUES (4, 'forth', 2);
INSERT INTO T (id, s, i) VALUES (5, 'to be or not to be', 1);
COMMIT;

-- 1. Client A starts
USE SnapsDB;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL SNAPSHOT ;

SELECT * FROM T WHERE i = 1;
id s i
---
1 first 1
3 third 1
5 to be or not to be 1

-- 2. Client B starts
USE SnapsDB;
SET IMPLICIT_TRANSACTIONS ON;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

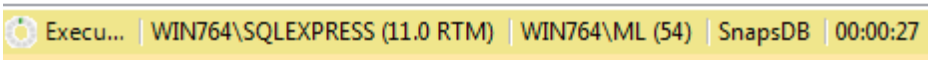
INSERT INTO T (id, s, i) VALUES (6, 'Insert Phantom', 1);
UPDATE T SET s = 'Update Phantom', i = 1 WHERE id = 2;
DELETE FROM T WHERE id = 5;

```
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           first                          1
2           Update Phantom                 1
3           third                          1
4           forth                          2
6           Insert Phantom                 1


-- 3. Client A continues
-- Let's repeat the query and try some updates
SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           first                          1
3           third                          1
5           to be or not to be             1

INSERT INTO T (id, s, i) VALUES (7, 'inserted by A', 1);
UPDATE T SET s = 'update by A inside snapshot' WHERE id = 3;
UPDATE T SET s = 'update by A outside snapshot' WHERE id = 4;
UPDATE T SET s = 'update by A after B' WHERE id = 1;

SELECT * FROM T WHERE i = 1;
id          s                              i
----------- ------------------------------ ------
1           update by A after B            1
3           update by A inside snapshot    1
5           to be or not to be             1
7           inserted by A                  1
```

```
-- 3.5 Client C in a new session starts and executes a query
USE SnapsDB;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           update by A after B            1
2           Update Phantom                 1
3           update by A inside snapshot    1
4           update by A outside snapshot   2
6           Insert Phantom                 1
7           inserted by A                  1
(6 row(s) affected)

-- 4. Client B continues
SELECT * FROM T;
id          s                              i
----------- ------------------------------ ------
1           first                          1
2           Update Phantom                 1
3           third                          1
4           forth                          2
6           Insert Phantom                 1


-- 5. Client A continues
SELECT * FROM T WHERE i = 1;
```

```
id          s                             i
----------- ----------------------------- ------
1           update by A after B           1
3           update by A inside snapshot   1
5           to be or not to be            1
7           inserted by A                 1
```

```
UPDATE T SET s = 'update after delete?' WHERE id = 5;
```



... waiting for B ...

```
-- 6. Client B continues without waiting for A
COMMIT;

-- 7. Client A continues
```

Msg 3960, Level 16, State 2, Line 1
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot isolation to access table 'dbo.T' directly or indirectly in database 'SnapsDB' to update, delete, or insert the row that has been modified or deleted by another transaction. Retry the transaction or change the isolation level for the update/delete statement.

```
-- 8. Client B continues
SELECT * FROM T;
id          s                             i
----------- ----------------------------- ------
1           first                         1
2           Update Phantom                1
3           third                         1
4           forth                         2
6           Insert Phantom                1

(5 row(s) affected)
```

Ερώτηση

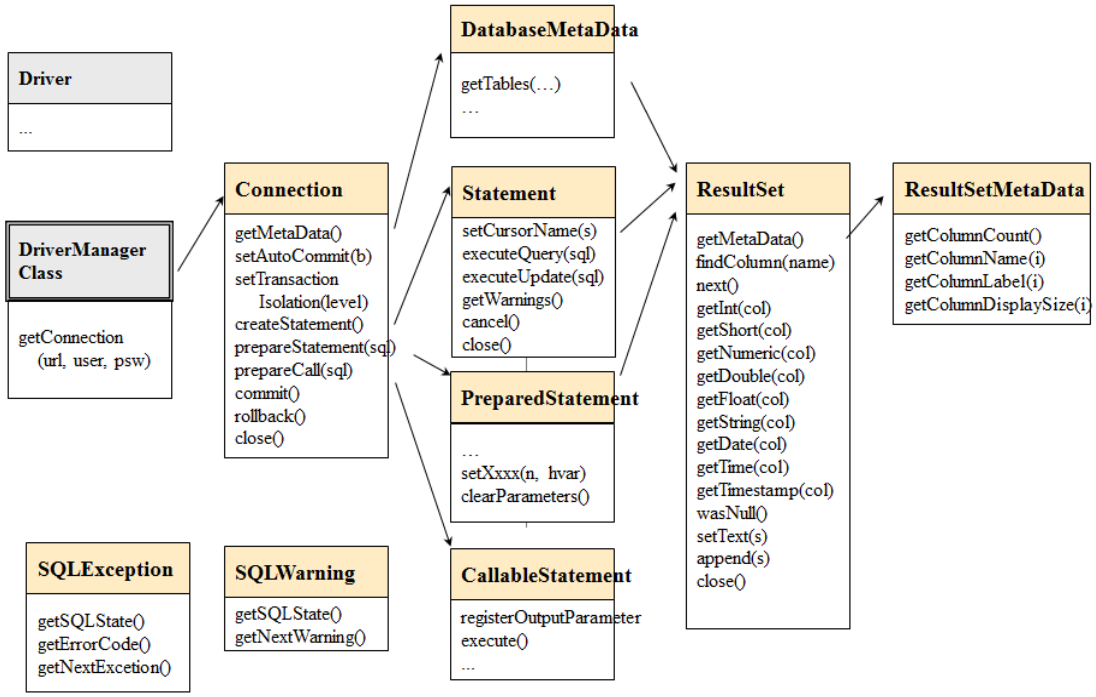
- Εξηγήστε πως εξελίχθηκε το πείραμα. Εξηγήστε τα διαφορετικά αποτελέσματα των βημάτων 3.5 και 4.

--

Για πιο προχωρημένα θέματα σχετικά με τις συναλλαγές στον SQL Server δείτε:
Kalen Delaney (2012), “SQL Server Concurrency, Locking, Blocking and Row Versioning”
ISBN 978-1-906434-90-8

Παράρτημα 2 Συναλλαγές με προγραμματισμό σε Java

Μπορούμε να πειραματιστούμε με τις συναλλαγές στην SQL με πολύ απλό τρόπο, χρησιμοποιώντας τα κατάλληλα εργαλεία που ονομάζονται συντάκτες SQL (SQL editors) που υποστηρίζονται στα σύγχρονα περιβάλλοντα ΣΔΒΔ. Οι συντάκτες αυτοί επιτρέπουν τη σύνταξη και εκτέλεση εντολών SQL σε άμεση επικοινωνία με το ΣΔΒΔ, με διαδραστικό τρόπο. Στην περίπτωση όπου αυτός που επικοινωνεί με το ΣΔΒΔ είναι μία εφαρμογή και όχι ένα άτομο, η επικοινωνία διεκπεραιώνεται από μία προγραμματιστική διεπαφή (Application Programming Interface-API). Στη συνέχεια, παρατίθεται ένα σύντομο παράδειγμα κώδικα που έχει να κάνει με συναλλαγές στην SQL, όπου γίνεται χρήση μιας διεπαφής API. Παρουσιάζουμε παράδειγμα Τραπεζικής Κατάθεσης, που υλοποιείται σε Java, χρησιμοποιώντας την τεχνολογία σύνδεσης με βάσεις δεδομένων σε Java (Java Database Connectivity-JDBC) ως την προγραμματιστική διεπαφή (API) πρόσβασης του πελάτη στα δεδομένα. Υποθέτουμε ότι ο αναγνώστης του παρόντος παραρτήματος είναι ήδη εξοικειωμένος με τον προγραμματισμό σε γλώσσα Java και το JDBC API και η Εικόνα Α2.1 χρησιμεύει μόνο ως μια οπτική επισκόπηση που βοηθά στην κατανόηση των κυρίων αντικειμένων (objects), των μεθόδων (methods) και της αλληλεπίδρασής τους.



Εικόνα Α2.1 Μια απλουστευμένη επισκόπιση της διεπαφής JDBC API

Κώδικας Παραδείγματος Java: Τραπεζική Κατάθεση

Το πρόγραμμα Τραπεζικής Κατάθεσης μεταφέρει ένα ποσό 100 ευρώ από έναν τραπεζικό λογαριασμό (fromAcct) σε κάποιον άλλο τραπεζικό λογαριασμό (toAcct). Το όνομα του προγράμματος οδήγησης, η διεύθυνση URL της βάσης δεδομένων, το όνομα χρήστη, ο κωδικός πρόσβασης χρήστη, οι παράμετροι fromAcct και toAcct διαβάζονται από τις παραμέτρους της γραμμής εντολών του προγράμματος.

Στο πλαίσιο του πειράματος, ο χρήστης θα πρέπει να αρχίσει δύο παραθυρα εντολών, στις Windows πλατφόρμες (ή τερματικά παράθυρα στις πλατφόρμες Linux) και να εκτελέσει το πρόγραμμα συγχρόνως και στα δύο παράθυρα, έτσι ώστε η παράμετρος fromAcct του πρώτου είναι η παράμετρος toAcct του άλλου και αντίστροφα, όπως φαίνεται στα σενάρια παρακάτω.
Μετά την ενημέρωση της παραμέτρου fromAcct το πρόγραμμα περιμένει το πλήκτρο ENTER για να συνεχίσετε, έτσι ώστε να συχρονιστούν οι χρήστες του πειράματος και να μπορούμε να ελέγξουμε τις συγκρούσεις ταυτοχρονισμού. Το πηγαίο πρόγραμμα δείχνει επίσης την ρουτίνα **επαναπροσπάθειας πρόσβασης δεδομένων ReTryer**.

Σενάριο κώδικα A2.1 Ο Java κώδικας για το παράδειγμα Τραπεζική Κατάθεση χρησιμοποιεί JDBC

```
/*  DBTechNet Concurrency Lab 15.5.2008 Martti Laiho

   BankTransfer.java

Save the java program as BankTransfer.java and compile as follows

javac BankTransfer.java

See BankTransferScript.txt for the test scripts applied to SQL Server, Oracle and DB2

Updates:
2.0 2008-05-26 ML preventing rollback by application after SQL Server deadlock
2.1 2012-09-24 ML restructured for presenting the Retry Wrapper block
2.2 2012-11-04 ML exception on non-existing accounts
**************************************************************/
import java.io.*;
import java.sql.*;
public class BankTransfer {
   public static String moreRetries = "N";

   public static void main (String args[]) throws Exception
   {
      System.out.println("BankTransfer version 2.2");

      if (args.length != 6) {
         System.out.println("Usage:" +
         "BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%");
         System.exit(-1);
      }
      java.sql.Connection conn = null;
     // String moreRetries = "N";
      boolean sqlServer = false;
      int counter = 0;
      String URL = args[1];
      String user = args[2];
      String password = args[3];
      String errMsg = "";
      int amount = 100;
      int fromAcct = Integer.parseInt(args[4]);
      int toAcct = Integer.parseInt(args[5]);

      // SQL Server's explicit transactions will require special treatment
      if (URL.substring(5,14).equals("sqlserver")) {
        sqlServer = true;
      }
            // register the JDBC driver and open connection
```

```
    try {
        Class.forName(args[0]);
        conn = java.sql.DriverManager.getConnection(URL,user,password);
    }
    catch (SQLException ex) {
          System.out.println("URL: " + URL);
          System.out.println("** Connection failure: " + ex.getMessage() +
                             "\n SQLSTATE: " + ex.getSQLState() +
                             " SQLcode: " + ex.getErrorCode());
        System.exit(-1);
    }
```

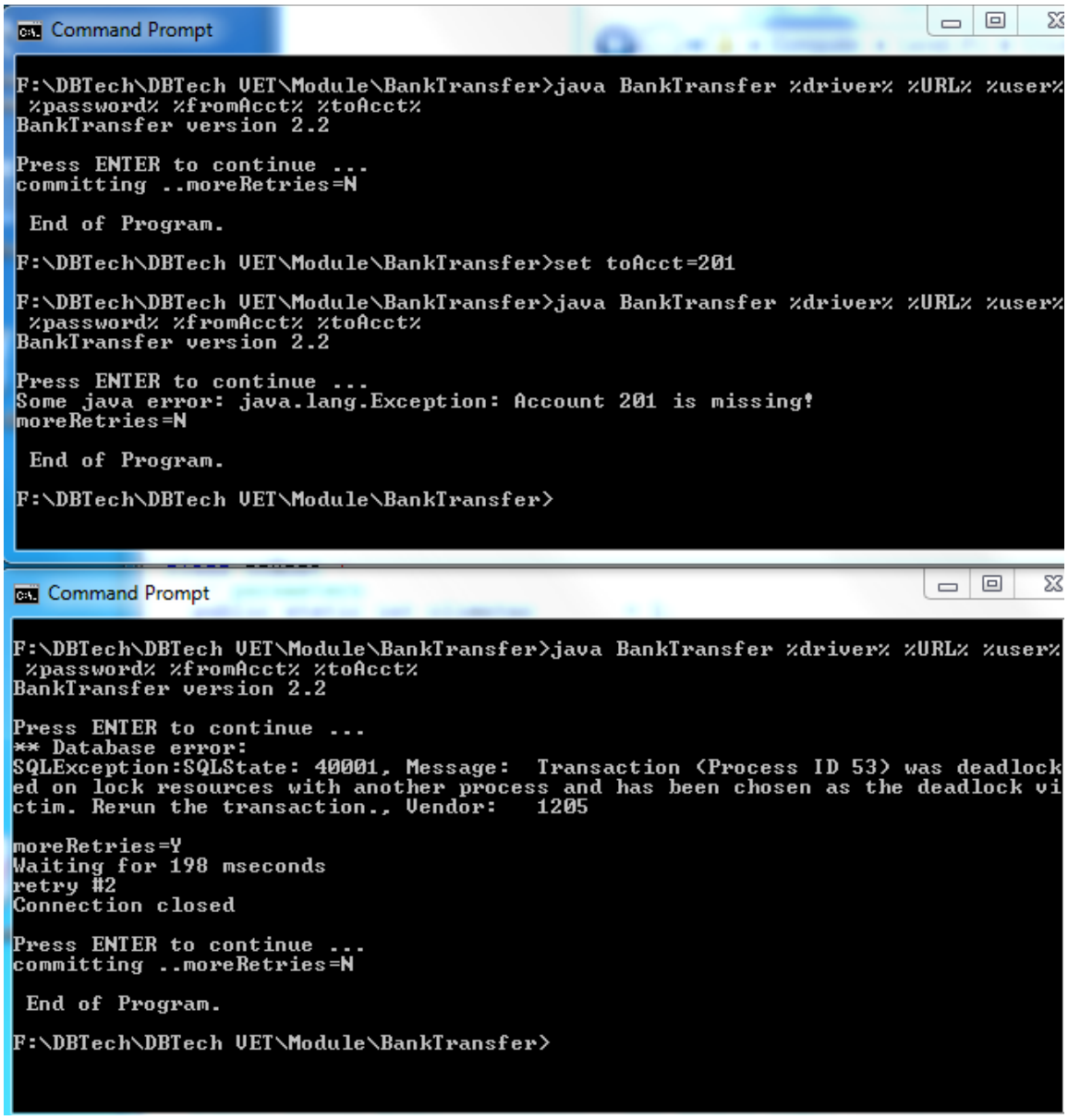
```
    do
    {

    // Retry wrapper block of TransaferTransaction
        if (counter++ > 0) {
            System.out.println("retry #" + counter);
            if (sqlServer) {
              conn.close();
              System.out.println("Connection closed");
              conn = java.sql.DriverManager.getConnection(URL,user,password);
              conn.setAutoCommit(true);
            }
        }
        TransferTransaction (conn,
                             fromAcct, toAcct, amount,
                             sqlServer, errMsg //,moreRetries
                             );
        System.out.println("moreRetries="+moreRetries);
        if (moreRetries.equals("Y")) {
            long pause = (long) (Math.random () * 1000); // max 1 sec.
            System.out.println("Waiting for "+pause+ " mseconds"); // just for testing
            Thread.sleep(pause);
        }
    } while (moreRetries.equals("Y") && counter < 10);  // max 10 retries
    // end of the Retry wrapper block

    conn.close();
    System.out.println("\n End of Program. ");
}

static void TransferTransaction (Connection conn,
        int fromAcct, int toAcct, int amount,
        boolean sqlServer,
        String errMsg //, String moreRetries
        )
        throws Exception {
        String SQLState = "*****";
        try {
            conn.setAutoCommit(false);   // transaction begins
            conn.setTransactionIsolation(
                Connection.TRANSACTION_SERIALIZABLE);
            errMsg = "";
            moreRetries = "N";

            // a parameterized SQL command
            PreparedStatement pstmt1 = conn.prepareStatement(
              "UPDATE Accounts SET balance = balance + ? WHERE acctID = ?");
            // setting the parameter values
            pstmt1.setInt(1, -amount);  // how much money to withdraw
```

```
            pstmt1.setInt(2, fromAcct); // from which account
            int count1 = pstmt1.executeUpdate();
            if (count1 != 1) throw new Exception ("Account "+fromAcct + " is missing!");
            // --- Interactive pause for concurrency testing -----
            // In the following we arrange the transaction to wait
            // until the user presses ENTER key so that another client
            // can proceed with a conflicting transaction.
            // This is just for concurrency testing, so don't apply this
            // user interaction in real applications!!!
              System.out.print("\nPress ENTER to continue ...");
              BufferedReader reader = new BufferedReader(
                      new InputStreamReader(System.in));
              String s = reader.readLine();
            // --- end of waiting -------------------------------------

            pstmt1.setInt(1, amount);  // how much money to add
            pstmt1.setInt(2, toAcct); // to which account
            int count2 = pstmt1.executeUpdate();
            if (count2 != 1) throw new Exception  ("Account "+toAcct + " is missing!");
            System.out.print("committing ..");
            conn.commit();  // end of transaction
            pstmt1.close();
          }
          catch (SQLException ex) {
            try {
              errMsg = "\nSQLException:";
              while (ex != null) {
                  SQLState = ex.getSQLState();
                  // is it a concurrency conflict?
                  if ((SQLState.equals("40001")   // Solid, DB2, SQL Server,...
                    || SQLState.equals("61000")   // Oracle ORA-00060: deadlock detected
                    || SQLState.equals("72000"))) // Oracle ORA-08177: can't serialize access
                      moreRetries = "Y";
                  errMsg = errMsg + "SQLState: "+ SQLState;
                  errMsg = errMsg + ", Message:  "+ ex.getMessage();
                  errMsg = errMsg + ", Vendor:   "+ ex.getErrorCode() + "\n";
                  ex = ex.getNextException();
              }
              // SQL Server does not allow rollback after deadlock !
              if (sqlServer == false) {
                 conn.rollback();  // explicit rollback needed for Oracle
                             // and the extra rollback does not harm DB2
              }
              // println for testing purposes
              System.out.println("** Database error: " + errMsg);
            }
            catch (Exception e) { // In case of possible problems in SQLException handling
              System.out.println("SQLException handling error: " + e);
              conn.rollback(); // Current transaction is rolled back
              ;  // This is reserved for potential exception handling
            }
          } // SQLException
          catch (Exception e) {
            System.out.println("Some java error: " + e);
            conn.rollback(); // Current transaction is rolled back also in this case
            ;  // This is reserved for potential other exception handling
          } // other exceptions
      }
    }
```

Ο παρακάτω κώδικας στο Σενάριο Κώδικα Α2.2 μπορεί να χρησιμοποιηθεί για τη δοκιμή του προγράμματος σε ένα σταθμό εργασίας των Windows σε δύο παράλληλα παράθυρα της γραμμής εντολών (Command Prompt) των Windows. Για τα σενάρια θεωρείται ότι το προϊόν ΣΔΒΔ είναι ο SQL Server Express, η βάση δεδομένων ονομάζεται TestDB, και το πρόγραμμα οδήγησης JDBC αποθηκεύεται στον υποκατάλογο jdbc-drivers του τρέχοντος καταλόγου του προγράμματος.

Σενάριο Κώδικα Α2.2 Πειραματισμός με την Τραπεζική Κατάθεση στα Windows

```
rem Script for the first window:
set CLASSPATH=.;jdbc-drivers\sqljdbc4.jar
set driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
set URL="jdbc:sqlserver://localhost;instanceName=SQLEXPRESS;databaseName=TestDB"
set user="user1"
set password="sql"
set fromAcct=101
set toAcct=202
java BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%


rem Script for the second window:
set CLASSPATH=.;jdbc-drivers\sqljdbc4.jar
set driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
set URL="jdbc:sqlserver://localhost;instanceName=SQLEXPRESS;databaseName=TestDB"
set user="user1"
set password="sql"
set fromAcct=202
set toAcct=101
java BankTransfer %driver% %URL% %user% %password% %fromAcct% %toAcct%
```



Εικόνα Α2.1 Παραδείγματα αποτελεσμάτων του πειράματος Τραπεζική Κατάθεση σε Windows

Ο κώδικας για αλλά προϊόντα ΣΔΒΔ καθώς κα για πλατφόρμες Linux, μπορεί να παραχθεί εύκολα τροποποιώντας τον κώδικα που περιέχεται στο Σενάριο Κώδικα Α2.2

Σενάριο Κώδικα Α2.3 Περιέχει σενάριο για τον πειραματισμό με την Τραπεζική Κατάθεση χρησιμοποιώντας MySQL για την πλατφόρμα DebianDB Linux. Ο κώδικας BankTransfer.java έχει αποθηκευτεί και μεταγλωτιστεί στον κατάλογο Transactions, του χρήστη student και τα προγράμματα οδήγησης JDBC έχουν εγκατασταθεί στον κατάλογο /opt/jdbc-drivers

Σενάριο κώδικα για MySQL σε Linux:

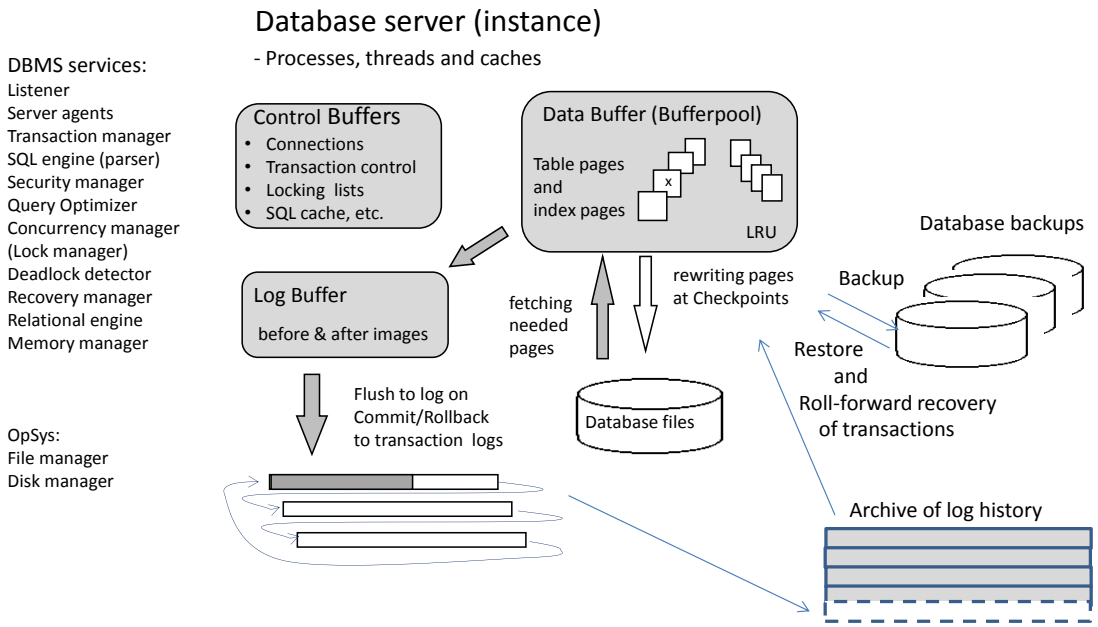
```
# Creating directory /opt/jdbc-drivers for JDBC drivers
cd /opt
mkdir jdbc-drivers
chmod +r+r+r jdbc-drivers
# copying the MySQL jdbc driver to /opt/jdb-drivers
cd /opt/jdbc-drivers
cp $HOME/mysql-connector-java-5.1.23-bin.jar
# allow read access to the driver to everyone
chmod +r+r+r mysql-connector-java-5.1.23-bin.jar


#******* MySQL/InnoDB ***************************************************

# First window:
cd Transactions
export CLASSPATH=/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/testdb
export user=user1
export password=sql
export fromAcct=101
export toAcct=202
java -classpath .:$CLASSPATH BankTransfer $driver $URL $user $password $fromAcct
$toAcct


# Second window:
cd Transactions
export CLASSPATH=/opt/jdbc-drivers/mysql-connector-java-5.1.23-bin.jar
export driver=com.mysql.jdbc.Driver
export URL=jdbc:mysql://localhost/testdb
export user=user1
export password=sql
export fromAcct=202
export toAcct=101
java -classpath .:$CLASSPATH BankTransfer $driver $URL $user $password $fromAcct
$toAcct
#*************************************************************************
```

Παράρτημα 3 Συναλλαγές και Επαναφορά βάσης δεδομένων

Στην Εικόνα Α3.1 παρουσιάζεται σχηματικά η εσωτερική διεκπαιρέωση των συναλλαγών ενός τυπικού διακομιστή βάσης δεδομένων. Οι εκπαιδευτικές διαφάνειες,"Basics of SQL Transactions" είναι διαθέσιμες από:

http://www.dbtechnet.org/papers/BasicsOfSqlTransactions.pdf

στις οποίες παρουσιάζονται, με περισσότερες λεπτομέρειες, οι αρχιτεκτονικές ορισμένων δημοφιλών προϊόντων ΣΔΒΔ, ο τρόπος της διαχείρισης των αρχείων της βάσης δεδομένων και των αρχείων ιστορικού (logs) καταγραφής συναλλαγών (π.χ. κάποια στιγμιότυπα των αρχείων ιστορικού συναλλαγών του MS SQL Server), η διαχείριση της διαθέσιμης ενδιάμεσης μνήμης (bufferpool) (με την εκμετάλλευση της data cache μνήμης, ώστε να επιτυγχάνεται καλύτερη απόδοση του συστήματος, διατηρώντας το I/O του δίσκου στο ελάχιστο), ο τρόπος που οι SQL συναλλαγές εξυπηρετούνται, η αξιοπιστία των ενεργειών COMMIT και η υλοποίηση των ενεργειών ROLLBACK.



Εικόνα Α3.1 Γενική επισκόπηση ενός διακομιστή βάσης δεδομένων

Στη συνέχεια, περιγράφουμε πως το ΣΔΒΔ μπορεί να επαναφέρει τη βάση δεδομένων στην κατάσταση της τελευταίας επικυρωμένης συναλλαγής, σε περίπτωση κάποιας διακοπής ρεύματος ή/και κάποιας ενδεχόμενης κατάρρευσης (crash) του διακομιστή. Σε περίπτωση αποτυχίας του υλικού (hardware), η βάση δεδομένων μπορεί να ανακτηθεί χρησιμοποιώντας το αντίγραφο ασφαλείας της (database backup) και το αποθηκευμένο αρχείο ιστορικού καταγραφής συναλλαγών (log) από το αντίγραφο ασφαλείας της βάσης δεδομένων.
(Δείτε επίσης: http://www.dbtechnet.org/papers/RdbmsRecovery.ppt)

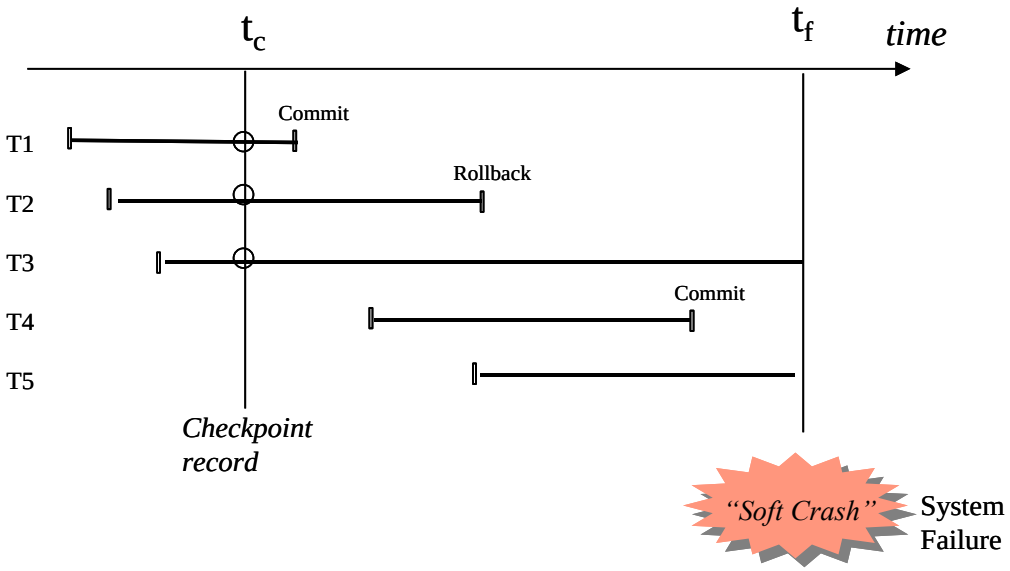
Όταν ξεκινά μια συναλλαγή SQL ο διακομιστής της βάσης δεδομένων θα της δώσει έναν μοναδικό αριθμό ταυτότητας (id) και για κάθε ενέργεια που θα εκτελεστεί κατά

τη διάρκεια αυτής της συναλλαγής, θα κρατήσει τις αντίστοιχες εγγραφές στο αρχείο ιστορικού καταγραφής συναλλαγών. Για κάθε γραμμή εντολής της συναλλαγής, που εκτελείται, το αρχείο ιστορικού περιέχει τον μοναδικό αναγνωριστικό αριθμό ταυτότητας της συναλλαγής καθώς και τα περιεχόμενα της γραμμής αυτής ως «την εικόνα πριν» από την εκτέλεση της γραμμής και τα περιεχόμενα της γραμμής μετά την εκτέλεσή της ως την «εικόνα μετά». Σε περίπτωση των INSERT εντολών, το μέρος της εικόνας «πριν» είναι κενό, και στην περίπτωση των DELETE εντολών, το μέρος της εικόνας «μετά» είναι κενό. Επίσης, για τις εντολές COMMIT και ROLLBACK, οι αντίστοιχες εγγραφές του αρχείου ιστορικού θα καταγραφούν, και στο πλαίσιο αυτό, όλα τα αρχεία καταγραφής του αρχείου ιστορικού της συναλλαγής θα εγγραφούν στον σκληρό δίσκο. Ο έλεγχος μετά από την εντολή COMMIT θα επιστρέψει στον πελάτη μετά την εγγραφή όλου του αρχείου ιστορικού στον δίσκο.

Από καιρό σε καιρό ο διακομιστής βάσης δεδομένων θα προχωρήσει στη δημιουργία ενός σημείου ελέγχου (CHECKPOINT), στο οποίο η εξυπηρέτηση των πελατών θα σταματήσει προσωρινά, όλα τα αρχεία ιστορικού καταγραφής συναλλαγών από τη μνήμη cache θα γραφτούν στο αρχείο ιστορικού καταγραφής συναλλαγών. Όλες οι ενημερωμένες σελίδες δεδομένων (που χαρακτηρίζονται ως «προς εκκαθάριση –βρώμικα- τμήματα»), στη διαθέσιμη ενδιάμεση μνήμη δεδομένων (bufferpool), θα γραφτούν στην θέση που προορίζεται για την εγγραφή των αρχείων της βάσης δεδομένων. Τα «προς εκκαθάριση –βρώμικα- τμήματα» από αυτές τις σελίδες θα εκκαθαριστούν από την γρήγορη μνήμη (data cache) δεδομένων. Στο αρχείο καταγραφής ιστορικού συναλλαγών, θα καταγραφεί ένα σύνολο από checkpoint εγγραφές, συμπεριλαμβανομένης μιας λίστας των μοναδικών αριθμών ταυτότητας (id) των υπό εξέλιξη συναλλαγών. Τελικά, η εξυπηρέτηση των πελάτων θα συνεχιστεί.

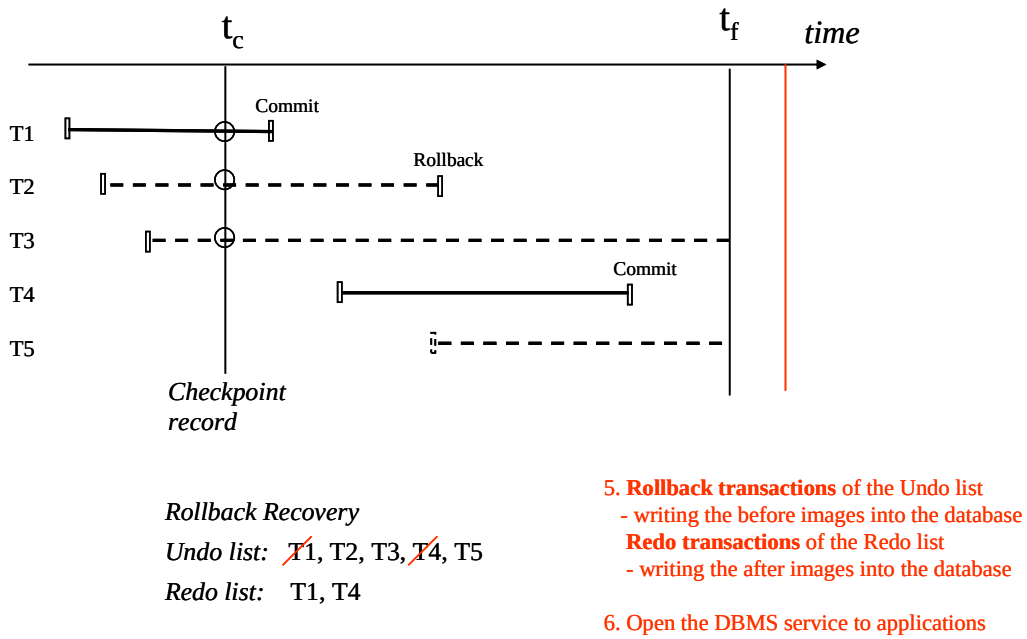
Σημείωση: Σε μια ελεγχόμενη διακοπή λειτουργίας του διακομιστή, δεν θα πρέπει να υπάρχουν ενεργές SQL συνεδρίες στον διακομιστή, έτσι ώστε στην τελευταία ενέργεια καταγραφής του αρχείου ιστορικού συναλλαγών ο διακομιστής να καταχωρήσει την εγγραφή μιας άδειας εγγραφής στο αρχείο ελέγχου (checkpoint record), που θα πιστοποιεί έναν «καθαρό» τερματισμό λειτουργίας του διακομιστή.

Η Εικόνα Α3.2 παρουσιάζει ένα σενάριο του αρχείου ιστορικού καταγραφής συναλλαγών πριν από την κατάρρευση του στιγμιότυπου της βάσης δεδομένων ή πριν από την κατάρρευση του συνόλου του υπολογιστή-διακομιστή, για παράδειγμα, λόγω κάποιας διακοπής ρεύματος. Όλα τα αρχεία στο δίσκο είναι αναγνώσιμα, αλλά τα περιεχόμενα της διαθέσιμης ενδιάμεσης μνήμης (bufferpool) έχουν χαθεί. Η κατάσταση αυτή ονομάζεται **Μαλακή Κατάρρευση Συστήματος (Soft Crash**).



Εικόνα Α3.2 Σενάριο του αρχείου ιστορικού καταγραφής συναλλαγών πριν από μία Μαλακή Κατάρρευση Συστήματος (Soft Crash).

Όταν γίνει επανεκκίνηση του διακομιστή, θα δούμε πρώτα την τελευταία εγγραφή του σημείου ελέγχου στο ιστορικό καταγραφής συναλλαγών (transaction log) . Εάν η εγγραφή στο σημείο ελέγχου είναι κενή δείχνει έναν καθαρό τερματισμό και ο διακομιστής είναι έτοιμος να εξυπηρετήσει τους εισερχόμενους νέους πελάτες. Σε αντίθετη περίπτωση , ο διακομιστής ξεκινά μια διαδικασία αναίρεσης και επαναφοράς ως εξής : Κατ' αρχάς, όλοι οι αριθμοί id των συναλλαγών που καταγράφηκαν στο σημείο ελέγχου (checkpoint) θα αντιγραφούν στη λίστα των συναλλαγών UNDO για να αναιρεθούν, ενώ η λίστα REDO των αριθμών id των συναλλαγών που θα πρέπει να επαναληφθούν στη βάση δεδομένων, θα είναι κενή. Ο διακομιστής τότε θα διαβάσει («σκανάρει») το αρχείο ιστορικού καταγραφής συναλλαγών από το σημείο ελέγχου (checkpoint) μέχρι το τέλος του αρχείου, αποδίδοντας νέους αριθμούς ταυτότητας (id) στις συναλλαγές της λίστας UNDO, ενώ θα μετακινήσει τους αριθμούς ταυτότητας των επικυρωμένων συναλλαγών από τη λίστα UNDO στη λίστα REDO. Στη συνέχεια, ο διακομιστής «προχωρά προς τα πίσω» και καταγράφει στο αρχείο ημερολογίου στη βάση δεδομένων τις «πριν» εικόνες όλων των εγγραφών των συναλλαγών που απαριθμούνται στον κατάλογο UNDO. Στη συνέχεια, «προχωρά προς τα εμπρός» από το σημείο ελέγχου και καταγράφει τις «μετά» εικόνες όλων των εγγραφών των συναλλαγών που απαριθμούνται στη λίστα REDO . Μετά από αυτό, η βάση δεδομένων έχει επανέλθει στο επίπεδο της τελευταίας επικυρωμένης συναλλαγής πριν τη μαλακή κατάρευση του συστήματος (soft crash), και ο διακομιστής μπορεί να αρχίσει να εξυπηρετεί τους πελάτες (βλέπε εικόνα Α3.3).



Εικόνα Α3.3 Αναίρεση και επαναφορά της βάσης δεδομένων

Σημείωση: Εκτός από την απλουστευμένη διαδικασία επαναφοράς που περιγράφεται παραπάνω, τα περισσότερα σύγχρονα προϊόντα ΣΔΒΔ χρησιμοποιούν το πρωτόκολλο ARIES, στο οποίο μεταξύ των σημείων ελέγχου (checkpoint) και όταν δεν υπάρχει «φόρτος» στη βάση δεδομένων, οι προς εκκαθάριση -«βρώμικες»- σελίδες δεδομένων από τη μνήμη γρήγορης προσπέλασης συγχρονίζονται στα αρχεία δεδομένων. Οι συγχρονισμένες σελίδες στον δίσκο χαρακτηρίζονται από έναν αριθμό LSN, που μπορεί στη συνέχεια να παραλειφθεί κατά την αναίρεση και την επαναφορά της βάσης, ενώ όλη η διαδικασία εκτελείται ταχύτερα.