Ατομική Διπλωματική Εργασία

ΜΕΛΕΤΗ ΕΥΡΩΣΤΙΑΣ ΣΥΣΤΗΜΑΤΩΝ ΔΙΑΧΕΙΡΙΣΗΣ ΔΕΔΟΜΕΝΩΝ ΣΤΗΝ ΠΑΡΟΥΣΙΑ ΔΙΑΚΟΠΩΝ ΤΡΟΦΟΔΟΣΙΑΣ ΙΣΧΥΟΣ ΣΕ ΚΛΗΣΕΙΣ ΣΥΣΤΗΜΑΤΟΣ

Κωνσταντίνος Παπαευριπίδης

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕΛΕΤΗ ΕΥΡΩΣΤΙΑΣ ΣΥΣΤΗΜΑΤΩΝ ΔΙΑΧΕΙΡΙΣΗΣ ΔΕΔΟΜΕΝΩΝ ΣΤΗΝ ΠΑΡΟΥΣΙΑ ΔΙΑΚΟΠΩΝ ΤΡΟΦΟΔΟΣΙΑΣ ΙΣΧΥΟΣ ΣΕ ΚΛΗΣΕΙΣ ΣΥΣΤΗΜΑΤΟΣ

Κωνσταντίνος Παπαευριπίδης

Επιβλέπων Καθηγητής
κ. Χάρης Βώλος

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2022

Ευχαριστίες

Για την ανάπτυξη της διπλωματικής μου εργασίας ήταν αναγκαία η χρήση ενός testbed για τις διακοπές τροφοδοσίας σε συστήματα βάσεων δεδομένων συνδυάζοντας ένα κύκλωμα διακοπής τροφοδοσίας. Γι' αυτό το λόγο, θέλω να ευχαριστήσω τον Terence Kelly, ο οποίος σχεδίασε το συγκεκριμένο κύκλωμα. Ο Terence Kelly είναι καθηγητής πληροφορικής με διδακτορικό στην επιστήμη πληροφορικής στο πανεπιστήμιο «University of Michigan». Επίσης, ο Terence Kelly σχεδίασε το λογισμικό το οποίο προκαλεί διακοπές τροφοδοσίας μέσω του Raspberry Pi, κάνοντας χρήση των GPIO pins (General Purpose Input Output).

Εκτός από το κύκλωμα, και όπως μόλις έχω αναφέρει, χρειάστηκα ένα υπολογιστή Raspberry Pi από το οποίο υλοποίησα όλα τα scripts και έτρεξα όλα τα πειράματά μου. Το Raspberry Pi το προμηθεύτηκα μαζί με ένα monitor από το Πανεπιστήμιο Κύπρου.
Συνεισφορά στην συγκεκριμένη διπλωματική εργασία είχε και η Ιωάννα Γεωργίου, απόφοιτος του τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου, η οποία είχε άμεση απασχόληση με το συγκεκριμένο θέμα όταν υλοποιούσε την δική της διπλωματική εργασία. Συγκεκριμένα, η Γεωργίου είχε συμβάλει σε πολύ μεγάλο βαθμό στην υλοποίηση των φόρτων εργασίας που γίνονται τα πειράματα, καθώς επίσης και για την επαλήθευση των πειραμάτων.

Τέλος, μεγάλη συνεισφορά στην διπλωματική μου εργασία είχε και ο κ. Βώλος Χάρης, ο επιβλέπων καθηγητής μου. Με καθοδηγούσε και μου παρείχε στήριξη καθ' όλη τη διάρκεια της διπλωματικής, τόσο ως προς την ανάπτυξη και υλοποίηση του συστήματος, όσο και την αντιμετώπιση διαφόρων προβλημάτων που εμφανίστηκαν. Με προμήθευσε με το κατάλληλο υλικό και βοηθήματα έτσι ώστε να πετύχουμε το καλύτερο δυνατό αποτέλεσμα. Πιστεύω πως η συνεργασία μας ήταν άψογη, κάτι το οποίο φαίνεται και από τα αποτελέσματα της διπλωματικής μου εργασίας.

Περίληψη

Τα συστήματα διαχείρισης και αποθήκευσης δεδομένων αποτελούν σημαντικό κομμάτι της ζωής μας. Με την τεράστια εξέλιξη στον τομέα της τεχνολογίας ο όγκος δεδομένων που παράγεται αυξάνεται, και μαζί του αυξάνονται και οι ανάγκες/απαιτήσεις των συστημάτων αυτών. Συγκεκριμένα, θέλουν την εξελιγμένη προστασία ACID (ατομικότητα, συνέπεια, απομόνωση και ανθεκτικότητα) που παρέχουν οι σύγχρονες βάσεις δεδομένων. Ωστόσο, οι ιδιότητες ACID δεν είναι τόσο εύκολο να τις προσφέρει η βάση δεδομένων, ιδίως όταν πρέπει να επιτευχθεί υψηλή απόδοση. Γι' αυτό το λόγο, η συγκεκριμένη μελέτη θα προσπαθήσει να διερευνήσει κατά πόσο η βάση δεδομένων SQLite πληρεί τις ιδιότητες ACID, κάτω από διακοπές τροφοδοσίας σε κλήσεις συστήματος. Έχει γίνει ανάπτυξη δύο φόρτων εργασίας σε γλώσσα Python. Κάθε φόρτος εργασίας έχει λογική αυτοελέγχου που επιτρέπει στις ιδιότητες ACID (κυρίως την ατομικότητα και την ανθεκτικότητα) να ελέγχονται σε μια εικόνα βάσης δεδομένων μετά το σφάλμα. Κάθε φόρτος εργασίας έχει διαφορετικές πολυπλοκότητες και ελέγχει διαφορετικές πτυχές. Επίσης, έχει γίνει ανάπτυξη μιας βιβλιοθήκης για να μπορέσουμε να εισάξουμε διακοπές τροφοδοσίας σε κλήσεις συστήματος (σε καθορισμένα σημεία) και αυτό έχει επιτευχθεί με τη βοήθεια της γνωστής τεχνικής function interposition. Μετά από πολλά πειράματα, έχουμε καταφέρει να εντοπίσουμε απώλεια δεδομένων από διακοπή τροφοδοσίας σε συγκεκριμένο σημείο του commit πρωτοκόλλου. Παρόλο της μικρής πιθανότητας της διακοπής τροφοδοσίας στο συγκεκριμένο σημείο, δεν παύει από το να είναι ένα σφάλμα το οποίο θα πρέπει να αγνοήσουμε και το οποίο καθιστά την βάση δεδομένων SQLite μη ανθεκτική.

Περιεχόμενα

Ευχαριστίες .....

Περίληψη .....

Περιεχόμενα .....

Κεφάλαιο 1 **Εισαγωγή** 1
1.1 Παρακίνηση 1
1.2 Περιγραφή προβλήματος 3
1.3 Συνεισφορά 5
1.4 Περίγραμμα Εργασίας 7

Κεφάλαιο 2 **Υπόβαθρο και Σχετική Εργασία** 9
2.1 Υπόβαθρο 9
2.2 Σχετική Εργασία 10

Κεφάλαιο 3 **Περιγραφή testbed** 16
3.1 Εισαγωγή 16
3.2 Κύκλωμα διακοπής τροφοδοσίας 18
3.3 Testbed 19
3.4 Υλοποίηση των εκτελέσιμων script 22
3.4.1 Επεξήγηση του rab.csh 22
3.4.2 Επεξήγηση του ptf_check.csh 23
3.4.3 Επεξήγηση του ptf_run.csh 24
3.5 Το αρχείο sqlite.py 24
3.6 Έλεγχος δεδομένων 25
3.6.1 Εισαγωγή 25
3.6.2 Επεξήγηση του checkthedata.py 26
3.6.3 Υλοποίηση βοηθητικών αρχείων 29
3.6.3.1 Το αρχείο progress_report 29
3.6.3.2 Το αρχείο check.txt 30

Κεφάλαιο 4 **Υλοποίηση Βιβλιοθήκης** 32
4.1 Εισαγωγή 32
4.2 Η σημασία του Function Interposition 34
4.2.1 Ορισμός 34
4.2.2 Απαιτήσεις για επιτυχημένο function interposition 35
4.2.3 Μέθοδος 1 – ctypes Python library 37
4.2.4 Μέθοδος 2 – LD_PRELOAD 38
4.3 Υλοποίηση συνάρτησεων 40
4.3.1 Εισαγωγή 40
4.3.2 Η συνάρτηση unlink() 40
4.3.3 Η συνάρτηση fsync() 42
4.3.4 Η συνάρτηση fdatasync() 43
4.4 Χτίσιμο και σύνδεση της βιβλιοθήκης 44
4.4.1 Χτίσιμο (build) της βιβλιοθήκης 44
4.4.2 Επαλήθευση 45

Κεφάλαιο 5 **Υλοποίηση φόρτων εργασίας και σχετικών αρχείων** 47
5.1 Εισαγωγή 47
5.2 Φόρτος Εργασίας 1 48
5.3 Φόρτος Εργασίας 2 51
5.4 Η χρησιμότητα του flush() και του fsync_disk() 53
5.5 Αρχεία καταγραφής κλήσεων 54
5.6 Προβλήματα υλοποίησης 54

Κεφάλαιο 6 **Μεθόδοι Διακοπής Τροφοδοσίας** 57
6.1 Εισαγωγή 57
6.2 Επεξήγηση killpower.sh 58
6.3 Μέθοδοι προηγούμενης μελέτης 59
6.4 Μέθοδος 1 - Τυχαίες διακοπές 61
6.5 Μέθοδος 2 - Διακοπές σε συγκεκριμένες κλήσεις συστήματος 62

Κεφάλαιο 7 **Πειράματα** 66
7.1 Εισαγωγή 66

7.2 Journal Modes και Journal Files 67
7.3 PRAGMA Synchronous statements 70
7.4 Checkpointing 72
7.5 Πειράματα στους Φόρτους Εργασίας 74
7.5.1 SQLite Version 2013 (έκδοση 3.7.16.2) 74
7.5.2 SQLite Version 2021 (έκδοση 3.35.4) 76
7.6 Παρατηρήσεις/Σχόλια από τα πειράματα στους φόρτους εργασίας 78
7.7 Ισχυρισμοί SQLite 79
7.8 Πειράματα αλλαγής του checkpoint threshold 81
7.9 Πειράματα αφαίρεσης των subqueries στους φόρτους 82

Κεφάλαιο 8 **Συμπεράσματα** 87
8.1 Συμπεράσματα 87
8.2 Εισηγήσεις 88

Βιβλιογραφία 90

Κεφάλαιο 1

Εισαγωγή

1.1 Παρακίνηση 1
1.2 Περιγραφή προβλήματος 3
1.3 Συνεισφορά 5
1.4 Περίγραμμα Εργασίας 7

1.1 Παρακίνηση

Ένα από τα σημαντικότερα καθήκοντα των συστημάτων αποθήκευσης και διαχείρισης δεδομένων είναι η προστασία της ακεραιότητας των δεδομένων. Σκεφτείτε μεγάλους οργανισμούς που διαθέτουν στην κατοχή τους τεράστιο όγκο από δεδομένα ή κυβερνήσεις με απόρρητες πληροφορίες που δεν πρέπει για κανένα λόγο να διαρρεύσουν, ή και να χαθούν. Όπως καταλαβαίνετε είναι πολύ σημαντική η διαχείριση αυτών των δεδομένων. Οι ξαφνικές αποτυχίες αποτελούν την κύρια απειλή για την ακεραιότητα των δεδομένων σε συστήματα αποθήκευσης και ταυτόχρονα σε εφαρμογές.

Η κατάρρευση λόγω σφαλμάτων σε εφαρμογές και στο λειτουργικό σύστημα είναι αρκετά σημαντικές ανησυχίες. Όμως, η χειρότερη ανησυχία σε μια εφαρμογή ή ένα σύστημα είναι η ύπαρξη μιας διακοπής της τροφοδοσίας. Με άλλα λόγια αν υπάρξει μια διακοπή τροφοδοσίας την στιγμή που ενημερώνονται τα δεδομένα της εφαρμογής στη βάση δεδομένων, τα δεδομένα θα μπορούσαν να καταστραφούν έμμεσα ή άμεσα. Έμμεσα εννοούμε ότι υπάρχει πιθανότητα να καταστραφούν τα μεταδεδομένα των δεδομένων, προκαλώντας επίσης απώλειες στην εφαρμογή μας. Για παράδειγμα, μπορεί να χαθεί η ημερομηνία δημιουργίας ενός πίνακα ή το όνομα του πίνακα.

Τα συστήματα αποθήκευσης και διαχείρισης δεδομένων έχουν αναπτύξει διάφορους μηχανισμούς οι οποίοι υπόσχονται να επαναφέρουν τα δεδομένα σε μια συνεπή κατάσταση που ορίζεται από την εφαρμογή μετά από μια αιφνίδια διακοπή τροφοδοσίας, επιτρέποντας την ανάκτηση των χαμένων δεδομένων.

Δυστυχώς, το ιστορικό όσον αφορά τις διακοπές τροφοδοσίας πάνω σε μηχανισμούς αποτυχίας και ατομικής ενημέρωσης (failure-atomic update mechanisms) αφήνει αρκετά ερωτηματικά στο κατά πόσο οι συγκεκριμένοι μηχανισμοί είναι αξιόπιστοι. Προηγούμενες μελέτες που έχουν γίνει για τον έλεγχο μηχανισμών αποτυχίας και ατομικής ενημέρωσης, όπως του Mai Zheng [1] έχουν δείξει ότι μπορεί να παραβιαστούν τέτοιοι μηχανισμοί. Συγκεκριμένα, ο Mai Zheng έχει δείξει ότι η ύπαρξη σφαλμάτων σε συστήματα βάσεων δεδομένων μπορεί να έχει σαν αποτέλεσμα την παραβίαση της ατομικότητας και την απώλεια δεδομένων. Άρα με αυτή την ένδειξη καταλαβαίνουμε πως εάν κάποια βάση δεδομένων έχει σφάλματα τότε δεν είναι αξιόπιστη. Παραδείγματα μηχανισμών αποτυχίας και ατομικής ενημέρωσης περιλαμβάνουν οι ACID συναλλαγές σε συστήματα σχεσιακών βάσεων δεδομένων (RDBMS - Relational DataBase Management Systems) και ημερολογίων αλλαγών (journaling) σε συστήματα αρχείων και συστήματα αποθήκευσης κλειδιού-τιμής (key-value stores).

Ένα απλό παράδειγμα για την αντίληψη της σημαντικότητας του πιο πάνω προβλήματος είναι το παράδειγμα της τραπεζικής συναλλαγής. Δηλαδή, φανταστείτε μια εφαρμογή που μεταφέρει χρήματα από ένα λογαριασμό σε ένα άλλο. Τι θα γίνει εάν την στιγμή που μεταφέρονται τα χρήματα υπάρξει μια διακοπή τροφοδοσίας στο σύστημα; Μια διακοπή τροφοδοσίας θα μπορούσε να παραβιάσει την ακεραιότητα και την ανθεκτικότητα των δεδομένων, όπου στην συγκεκριμένη περίπτωση είναι τα χρήματα. Δηλαδή, υπάρχει περίπτωση η συναλλαγή να φαίνεται ότι εκτελέστηκε σωστά, ενώ στην πραγματικότητα η εκτέλεση της να ήταν λανθασμένη. Γι’ αυτό το λόγο είναι σημαντικός ο τρόπος ο οποίος αντιμετωπίζονται οι διακοπές τροφοδοσίας από τους μηχανισμούς αποτυχίας και ατομικής ενημέρωσης, έτσι ώστε να παρέχουν αξιοπιστία και ανάκτηση των δεδομένων σε περίπτωση οποιοδήποτε σφάλματος.

1.2 Περιγραφή προβλήματος

Με βάση την πιο πάνω εισαγωγή είναι φανερό ότι η σημασία της ύπαρξης αξιόπιστων μηχανισμών αποτυχίας και ατομικής ενημέρωσης από τα συστήματα αποθήκευσης και διαχείρισης των δεδομένων είναι μεγάλη. Όλο και περισσότεροι οργανισμοί μεταφέρουν τα δεδομένα τους διαδικτυακά συστήματα και αποφεύγουν τον παραδοσιακό τρόπο αποθήκευσης, δηλαδή σε χαρτί. Γι' αυτό το λόγο, κρίναμε αναγκαίο να γίνει μια περαιτέρω μελέτη όσον αφορά το πρόβλημα της διακοπής τροφοδοσίας σε συστήματα, έτσι ώστε να αξιολογηθεί η ευρωστίας τους.

Στόχος της συγκεκριμένης διπλωματικής εργασίας είναι η αξιολόγηση της ευρωστίας σε συστήματα αποθήκευσης και διαχείρισης δεδομένων έναντι διακοπών τροφοδοσίας σε συγκεκριμένα σημεία τα οποία πιστεύουμε ότι μπορούν να παραβιάσουν τις ιδιότητες ACID. Συγκεκριμένα, θα γίνει αξιολόγηση του συστήματος αποθήκευσης SQLite σε δύο εκδόσεις. Την έκδοση 3.7.16.2, δηλαδή την έκδοση του 2013, καθώς επίσης και την τελευταία έκδοση που μας παρέχει η SQLite, την 3.35.4, δηλαδή την έκδοση του 2021. Η SQLite είναι ένα σύστημα διαχείρισης δεδομένων και θεωρείται ένα embedded database system, δηλαδή μια ενσωματωμένη βάση δεδομένων. Με άλλα λόγια, η δομή της είναι ενσωματωμένη μέσα στην ίδια την εφαρμογή, αφού δεν είναι ένα αυτόνομο πρόγραμμα. Αντιθέτως, είναι μια βιβλιοθήκη γραμμένη στην γλώσσα προγραμματισμού C και χρησιμοποιείται ως προσωρινό σύνολο δεδομένων για επεξεργασία με ορισμένα δεδομένα μέσα σε μια εφαρμογή.

Οι βάσεις δεδομένων χρησιμοποιούνται όταν θέλουμε να πετύχουμε υψηλό επίπεδο αξιοπιστίας. Συγκεκριμένα, θέλουν την εξελιγμένη προστασία ACID (ατομικότητα, συνέπεια, απομόνωση και ανθεκτικότητα) που παρέχουν οι σύγχρονες βάσεις δεδομένων. Το ACID (Ατομικότητα, Συνέπεια, Απομόνωση, Ανθεκτικότητα) αναφέρεται σε ένα τυποποιημένο σύνολο ιδιοτήτων που εγγυώνται ότι οι συναλλαγές μιας βάσης δεδομένων υποβάλλονται σε επεξεργασία αξιόπιστα. Το ACID ασχολείται ιδιαίτερα με τον τρόπο με τον οποίο μια βάση δεδομένων ανακάμπτει από οποιαδήποτε αποτυχία που μπορεί να προκύψει κατά την επεξεργασία μιας συναλλαγής.

Ωστόσο, οι ιδιότητες ACID δεν είναι τόσο εύκολο να τις προσφέρει η βάση δεδομένων, ιδίως όταν πρέπει να επιτευχθεί υψηλή απόδοση. Ο έλεγχος για τις ιδιότητες ACID σε περίπτωση αστοχίας είναι αξιοσημείωτα δύσκολος, καθώς ένα σενάριο αποτυχίας ενδέχεται να μην μπορεί να αναπαραχθεί εύκολα. Ένα σύστημα διαχείρισης βάσεων δεδομένων (DBMS) συμβατό με τις ιδιότητες ACID διασφαλίζει ότι τα δεδομένα στη βάση δεδομένων παραμένουν ακριβή και συνεπή παρά τις τυχόν τέτοιες αστοχίες.

Γενικότερα, ένα DBMS εγγυάται ότι μπορεί να πετύχει serializability (σειριοποίηση). Συγκεκριμένα, αναφέρεται στο ότι ένα χρονοδιάγραμμα θεωρείται serialized, εάν είναι ισοδύναμο με ένα άλλο serialized χρονοδιάγραμμα. Επίσης, ο όρος serialization χαρακτηρίζεται από το concurrency και το recoverability. Ένα concurrent χρονοδιάγραμμα διασφαλίζει ότι η εκτέλεση θα γίνει σειριακά, δηλαδή η μια διεργασία θα εκτελεστεί μετά την άλλη. Ενώ, το recoverability αναφέρεται σε dirty reads τα οποία μπορεί να εκτελεστούν σε συναλλαγές που δεν έχουν γίνει commit ακόμα. Δηλαδή, ένα DMBS μπορεί να χαρακτηριστεί recoverable εάν μπορεί να διαχειριστεί σωστά uncommitted συναλλαγές μετά από την ύπαρξη κάποιου σφάλματος (πχ με rollback). Η συγκεκριμένη διπλωματική εργασία θα επικεντρωθεί στο recoverable κομμάτι εφόσον οι διακοπές τροφοδοσίας μπορεί να το επηρεάσουν.

Για την αξιολόγηση του recoverability του συστήματος αποθήκευσης SQLite επιλέξαμε να ελέγξουμε δύο ιδιότητες ACID, την ατομικότητα (atomicity) και την ανθεκτικότητα (durability). Ατομικότητα σημαίνει ότι ένα DBMS εγγυάται ότι είτε όλη η συναλλαγή θα πετύχει (θα γίνει commit), είτε δεν θα γίνει καθόλου. Δηλαδή, δεν μπορεί σε μια συναλλαγή να γίνουν commit τα μισά δεδομένα. Εάν ένα μέρος της συναλλαγής αποτύχει, ολόκληρη η συναλλαγή αποτυγχάνει. Με την ατομικότητα, είναι είτε “όλα ή τίποτα”. Ανθεκτικότητα σημαίνει ότι, μόλις δεσμευτεί μια συναλλαγή, θα παραμείνει στο σύστημα ακόμα και αν υπάρξει κατάρρευση του συστήματος αμέσως μετά τη συναλλαγή. Οποιεσδήποτε αλλαγές από τη συναλλαγή πρέπει να αποθηκεύονται μόνιμα. Εάν το σύστημα ενημερώνει το χρήστη ότι η συναλλαγή έχει επιτύχει, η συναλλαγή πρέπει, στην πραγματικότητα, να έχει επιτύχει. Δηλαδή, ας ξαναπάρουμε το παράδειγμα με την τραπεζική συναλλαγή. Όταν γίνει η μεταφορά των χρημάτων από τον ένα λογαριασμό στον άλλο, τόσο ο παραλήπτης, όσο και ο αποστολέας θέλουν να ξέρουν ότι η μεταφορά έγινε και στην πραγματικότητα, όχι μόνο με βάση την ειδοποίηση επιβεβαίωσης που θα

αποστείλει η εφαρμογή. Αλλιώς, η συγκεκριμένη εφαρμογή δεν θα πληρούσε τις ιδιότητες ACID και θα ήταν αναξιόπιστη.

1.3 Συνεισφορά

Με τη συγκεκριμένη διπλωματική εργασία θέλουμε να μελετήσουμε σε βάθος της αξιοπιστία του συστήματος αποθήκευσης και βάσης δεδομένων, SQLite. Η σημασία των αποτελεσμάτων που θα συλλέξουμε από τα πειράματα μπορούν να μας βοηθήσουν να κατανοήσουμε καλύτερα την λειτουργία των βάσεων δεδομένων. Ταυτόχρονα, θα συνεισφέρουν και στο ευρύτερο πεδίο της επιστήμης του συγκεκριμένου θέματος, τόσο σε θέματα αρχιτεκτονικής των συστημάτων αποθήκευσης δεδομένων, όσο και σε θέματα υλοποίησης μελλοντικών συστημάτων. Δηλαδή, τα προβλήματα που εντοπίσαμε στην παρούσα διπλωματική εργασία μπορούν να αποτελέσουν σημείο αναφοράς για τους προγραμματιστές που ασχολούνται με τη συγκεκριμένη επιστήμη, αλλά και μελλοντικούς προγραμματιστές. Επίσης, τα αποτελέσματα θα βοηθήσουν τους υπάρχων προγραμματιστές συστημάτων και εφαρμογών που χρησιμοποιούν τη SQLite ως γλώσσα αποθήκευσης των δεδομένων τους, να κατανοήσουν καλύτερα το πρωτόκολλο και τις λειτουργίες της. Αυτό είναι πολύ σημαντικό, γιατί έτσι μπορούν να υλοποιήσουν περαιτέρω τεχνικές μέσα στις εφαρμογές τους, έτσι ώστε να μπορούν να αντιμετωπίσουν τα προβλήματα ευρωστίας που παρουσιάζει η SQLite. Παράλληλα, θα μπορούν να προσφέρουν καλύτερη ανθεκτικότητα και ατομικότητα στις εφαρμογές τους, ειδικά σε περιπτώσεις διακοπής τροφοδοσίας. Μην ξεχνάτε ότι αυτό είναι και από τα σημαντικότερα καθήκοντα των συστημάτων αποθήκευσης και διαχείρισης δεδομένων, δηλαδή η προστασία της ακεραιότητας των δεδομένων.

Τα πειράματα που έχουν γίνει στην συγκεκριμένη εργασία βασίζονται σε 2 φόρτους εργασίας. Επιπλέον, έχουν μελετηθεί 2 εκδόσεις της SQLite, μια παλαιότερη έκδοση (version 2013) και η τελευταία έκδοση (version 2021). Με αυτό το τρόπο θα μπορούμε να έχουμε μια πιο σφαιρική εικόνα για τα προβλήματα που υπάρχουν και να δούμε κατά πόσο πιο ανθεκτική είναι η καινούρια έκδοση σε σχέση με την παλιά. Η υλοποίηση των δύο αυτών φόρτων εργασίας στοχεύει ειδικά τη συλλογή αποτελεσμάτων που αφορούν τα πιο πάνω ζητήματα σε συστήματα αποθήκευσης και διαχείρισης δεδομένων. Δηλαδή σε θέματα ατομικότητας και ανθεκτικότητας των βάσεων δεδομένων. Πιστεύω πως η

συγκεκριμένη διπλωματική εργασία θα βοηθήσει τους προγραμματιστές να συνειδητοποιήσουν τη σημασία της ατομικότητας και της ανθεκτικότητας σε βάσεις δεδομένων, αλλά και να τους παροτρύνει στο να λαμβάνουν καλύτερα μέτρα προστασίας στα δεδομένα των συστημάτων τους, ειδικά όταν παρουσιαστεί κάποια διακοπή τροφοδοσίας.

Επίσης, θα χρησιμοποιήσουμε την γνωστή τεχνική function interposition, δηλαδή την αντικατάστασης κλήσεων σε συναρτήσεις σε δυναμικές βιβλιοθήκες με κλήσεις σε wrapper classes οι οποίες είναι υλοποιημένες από το χρήστη. Για να το πετύχουμε αυτό θα χρησιμοποιήσουμε το LD_PRELOAD, μια χρήσιμη τεχνική που χρησιμοποιείται για να επηρεάσει τη σύνδεση κοινόχρηστων βιβλιοθηκών και την ανάλυση συναρτήσεων κατά το χρόνο εκτέλεσης. Με άλλα λόγια το LD_PRELOAD χρησιμοποιείται για να αντικαταστήσει συναρτήσεις βιβλιοθηκών με δικές μας συναρτήσεις. Η τεχνική function interposition θα εφαρμοστεί σε συνδυασμό με μια βιβλιοθήκη που θα υλοποιήσουμε η οποία θα περιέχει τις συναρτήσεις που θέλουμε να μελετήσουμε.

Αξίζει να σημειωθεί ότι έχει ήδη γίνει μια μελέτη που έχει άμεση σχέση με την παρούσα διπλωματική εργασία, αλλά οι διακοπές τροφοδοσίας γίνονταν σε τυχαίες χρονικές στιγμές. Ως αποτέλεσμα, μετά από 6144 τυχαίες διακοπές τροφοδοσίας στον πρώτο φόρτο εργασίας δεν παρουσιάστηκε κάποια απώλεια δεδομένων. Επίσης, παρόμοια συμπεράσματα παρατηρήθηκαν και μετά από 4900 τυχαίες διακοπές τροφοδοσίας στον δεύτερο φόρτο εργασίας. Στην παρούσα διπλωματική εργασία θα επιχειρήσουμε να εισάξουμε διακοπές τροφοδοσίας σε κλήσεις συστήματος, σε συγκεκριμένα σημεία τα οποία πιστεύουμε ότι μπορεί να προκαλέσουν απώλεια δεδομένων και παράλληλα παραβίαση στις ιδιότητες ACID. Για την εισαγωγή διακοπών τροφοδοσίας θα πρέπει να αναπτυχθεί μια βιβλιοθήκη και να γίνει ενσωμάτωση της μέσα στους φόρτους εργασίας (με τη βοήθεια της τεχνικής function interposition). Αναλυτική επεξήγηση για την ανάπτυξη της βιβλιοθήκης περιγράφεται στο *Κεφάλαιο 4*.

Το κύριο αποτέλεσμα της συγκεκριμένης μελέτης είναι ότι καταφέραμε να εντοπίσουμε απώλεια δεδομένων, κάτι το οποίο δεν κατάφερε η μελέτη της Ιωάννας Γεωργίου με τη μεθοδολογία των τυχαίων διακοπών τροφοδοσίας. Συγκεκριμένα, με τη μεθοδολογία των διακοπών τροφοδοσίας σε κλήσεις συστήματος, καταφέραμε να δείξουμε ότι η βάση

δεδομένων δεν είναι όσο ανθεκτική (durable), όσο ισχυρίζεται. Το σφάλμα εντοπίστηκε με διακοπή τροφοδοσίας σε συγκεκριμένο σημείο, το οποίο περιγράφεται λεπτομερώς στο *Κεφάλαιο 7.9*.

1.4 Περίγραμμα Εργασίας

Η παρούσα διπλωματική εργασία αποτελείται από 8 κεφάλαια. Αυτά είναι η «Εισαγωγή», το «Υπόβαθρο και Σχετική εργασία», η «Περιγραφή testbed», η «Υλοποίηση βιβλιοθήκης», η «Υλοποίηση φόρτων εργασίας και σχετικών αρχείων», το «Μέθοδοι διακοπής τροφοδοσίας», το «Πειράματα» και «Συμπεράσματα».

Σε αυτό το κεφάλαιο αναφέρθηκαν οι λόγοι που ώθησαν στην διεξαγωγή έρευνας στο συγκεκριμένο πεδίο και τα κύρια προβλήματα που υπάρχουν γύρω από το συγκεκριμένο θέμα. Επίσης, γίνεται αναφορά της συνεισφοράς της συγκεκριμένης μελέτης στο ευρύτερο επιστημονικό πεδίο.

Στο δεύτερο κεφάλαιο, περιγράφονται διάφορες σχετικές εργασίες που έχουν γίνει στο πεδίο ενασχόλησης της εργασίας αυτής, καθώς επίσης και κάποιες βασικές έννοιες που χρησιμοποιούνται. Εξηγείται η βάση δεδομένων που θα μελετήσουμε και ο τρόπος χρήσης της. Επίσης, περιγράφεται το WAL μια λειτουργία της βάσης δεδομένων SQLite.

Στο τρίτο κεφάλαιο, γίνεται περιγραφή του testbed που έχει χρησιμοποιηθεί για να γίνει εφικτή η συγκεκριμένη μελέτη. Συγκεκριμένα, περιγράφονται το κύκλωμα διακοπής τροφοδοσίας, ο υπολογιστής που έχει χρησιμοποιηθεί, διάφορα βοηθητικά αρχεία και scripts, αλλά και ο τρόπος που γίνεται ο έλεγχος των δεδομένων. Για την ανάπτυξη του testbed έχει συμβάλει σε μεγάλο βαθμό ο καθηγητής-ερευνητής Terence Kelly, καθώς επίσης και η Ιωάννα Γεωργίου η οποία έκανε μια προηγούμενη μελέτη σε παρόμοιο θέμα.

Στο τέταρτο κεφάλαιο, περιγράφεται η βιβλιοθήκη που έχει υλοποιηθεί για να είναι εφικτή η πραγματοποίηση των πειραμάτων στο βάση δεδομένων SQLite. Συγκεκριμένα, περιγράφονται οι συναρτήσεις που έχουν υλοποιηθεί μέσα στη βιβλιοθήκη και πώς γίνεται το φόρτωμα της μέσα στους φόρτους εργασίας. Δηλαδή, με την τεχνική function interposition, η οποία βασίστηκε σε μεγάλο βαθμό η συγκεκριμένη διπλωματική εργασία.

Επίσης, παρουσιάζεται ο τρόπος με τον οποίο έγινε το χτίσιμο της βιβλιοθήκης και πως επιβεβαιώνουμε ότι όλα λειτουργούν σωστά.

Στο πέμπτο κεφάλαιο, περιγράφεται η υλοποίηση των φόρτων εργασίας που έχουν χρησιμοποιηθεί για να την πραγματοποίηση των πειραμάτων και την εξαγωγή αποτελεσμάτων. Επίσης, παρουσιάζονται κάποια βοηθητικά αρχεία καταγραφής κλήσεων και συναρτήσεις που έχουν χρησιμοποιηθεί μέσα στους φόρτους εργασίας.

Το έκτο κεφάλαιο, επικεντρώνεται στην περιγραφή των μεθόδων που έχουν υλοποιηθεί για την εισαγωγή διακοπών τροφοδοσίας μέσα στους φόρτους εργασίας. Συγκεκριμένα, περιγράφονται οι μέθοδοι που έχουν αναπτυχθεί από την Ιωάννα Γεωργίου, αλλά και μέθοδοι που έχουν αναπτυχθεί στην παρούσα διπλωματική εργασίας και πώς διαφέρουν μεταξύ τους.

Το έβδομο κεφάλαιο, παρουσιάζει τα πειράματα που έχουν γίνει στην παρούσα διπλωματική εργασία. Εξηγούνται κάποιες βασικές έννοιες και λειτουργίες που μας παρέχει η βάση δεδομένων SQLite και έχουν χρησιμοποιηθεί για την εκτέλεση διαφόρων πειραμάτων. Επίσης, στο συγκεκριμένο κεφάλαιο αναφέρονται κάποιοι ισχυρισμοί που βρίσκονται μέσα στο documentation της SQLite και θα γίνει έλεγχος κατά πόσο αληθεύουν.

Και τέλος, το όγδοο κεφάλαιο αναφέρει τα συμπεράσματα που έχουν εξαχθεί από τα πειράματα στην συγκεκριμένη μελέτη, αλλά και συμπεράσματα στο σύνολο. Επίσης, αναφέρονται κάποιες εισηγήσεις που θα μπορούσαν να γίνουν σε μελλοντικές μελέτες που έχουν σχέση με το συγκεκριμένο πεδίο και θα βοηθούσαν τους ερευνητές να είναι πιο αποτελεσματικοί με τα πειράματά τους.

Κεφάλαιο 2

Υπόβαθρο και Σχετική Εργασία

2.1 Υπόβαθρο 9
2.2 Σχετική Εργασία 10

2.1 Υπόβαθρο

Για την συγκεκριμένη διπλωματική εργασίας έχει χρησιμοποιηθεί η βάση δεδομένων SQLite, όπως προαναφέρθηκε. Αρχικά, θα πρέπει να γίνει το build της βιβλιοθήκης SQLite. Ο λόγος που κάνουμε build την SQLite, είναι επειδή το συγκεκριμένο σύστημα διαχείρισης δεδομένων είναι βιβλιοθήκη που μπορεί κάποιος να συμπεριλάβει στην εφαρμογή του, και όχι ξεχωριστό σύστημα όπως τα άλλα συστήματα διαχείρισης και αποθήκευσης δεδομένων. Η πιο κάτω εξήγηση αφορά τόσο την παλαιότερη έκδοση (2013), όσο και την τελευταία (2021).

Έστω ότι θέλουμε να κάνουμε build την έκδοση του 2013. Θα πρέπει να ανοίξουμε ένα terminal και να εκτελέσουμε τις πιο κάτω εντολές, δεδομένου ότι έχουμε ήδη κατεβάσει την βιβλιοθήκη της SQLite.

```
1 cd /home/pi/workspace/project/pinap/third-party/sqlite-src-3071602/build
2 /home/pi/workspace/project/pinap/third-party/sqlite-src-3071602/configure
3 make
```

Σχήμα 2.1: Πώς γίνεται το build της βιβλιοθήκης SQLite

Όπως βλέπετε πιο πάνω, με την γραμμή 1 θέλουμε να μπούμε στο φάκελο που μας παρέχει η SQLite για το χτίσιμο της βιβλιοθήκης. Στη συνέχεια, αφού βρισκόμαστε στην

κατάλληλη τοποθεσία, θα εκτελέσουμε το αρχείο configure που είναι ήδη υλοποιημένο από την SQLite. Τέλος, θα εκτελέσουμε την γραμμή 3, και περιμένουμε μέχρι να εμφανιστεί στο terminal το μήνυμα ότι η βιβλιοθήκη έχει κτιστεί με επιτυχία.
Ο λόγος που είναι σημαντικό να καταλάβουμε πώς επιτυγχάνεται το χτίσιμο της βιβλιοθήκης SQLite, είναι επειδή όταν τα εκτελέσουμε τα πειράματα μας, όπως θα εξηγηθεί στο *Κεφάλαιο 7*, θα χρειαστεί να αλλάξουμε μια παράμετρο από τον πηγαίο κώδικα της βιβλιοθήκης.

Επίσης, για τη συγκεκριμένη διπλωματική εργασία έχει χρησιμοποιηθεί η έννοια WAL, η οποία μας παρέχει η βιβλιοθήκη SQLite και περιγράφει αναλυτικά στο documentation της [8]. Το journal mode WAL (Write Ahead Log), είναι ένα αρχείο καταγραφής στο οποίο προστίθεται οποιαδήποτε εγγραφή πραγματοποιείται στη βάση . Σε περίπτωση αποτυχίας, όταν το σύστημα επανέλθει, όλες οι εγγραφές του WAL επαναλαμβάνονται, και η βάση επανέρχεται στην αρχική κατάσταση. Οι εγγραφές στο WAL πραγματοποιούνται ατομικά, δηλαδή είτε δεν πραγματοποιούνται καθόλου είτε ολοκληρώνονται πλήρως και τα δεδομένα αποθηκεύονται μόνιμα και με ασφάλεια στον δίσκο, χωρίς να υπάρχει κίνδυνος παραφθοράς

Στο Write-Ahead Logging εξαναγκάζεται η εγγραφή ιστορικού για μια ενημέρωση, πριν πάει στο δίσκο η αντίστοιχη σελίδα δεδομένων, έτσι εγγυάται η ατομικότητα. Επίσης, πρέπει να γράφονται όλες οι εγγραφές ιστορικού μιας συναλλαγής πριν την ολοκλήρωσή της και με αυτό εγγυάται η Μονιμότητα. Η βασική αρχή του Write-Ahead Log είναι ότι προτού γράψει οτιδήποτε στην βάση δεδομένων πρέπει να καταχωρηθεί η αντίστοιχη εγγραφή στο log.

2.2 Σχετική Εργασία

Μελετήσαμε διάφορα άρθρα για την ανάλυση της αξιοπιστίας λογισμικών αποθήκευσης. Ένα άρθρο που βασίστηκε η εργασία είναι του Mai Zheng [1]. Το άρθρο δείχνει ότι, σχεδόν κάθε σχεσιακή βάση δεν υλοποιεί σωστά τις ατομικές συναλλαγές. Όλες χάνουν δεδομένα λόγω αποτυχιών που ισχυρίζονται ότι ανέχονται. Αξιολογεί συστήματα βάσεων δεδομένων. Σε αντίθεση με αυτή την διπλωματική εργασία, η εργασία του Zheng χρησιμοποιεί μοντελοποίηση αποτυχιών/βλαβών(faults) στο λειτουργικό επίπεδο και όχι

πραγματικών αποτυχιών/βλαβών (faults) στο επίπεδο εφαρμογών. Οπότε δεν μπορεί να αξιολογήσει την ευρωστία σε πραγματικό υλικό.

Η εργασία αυτή μοντελοποιεί την διακοπή τροφοδοσίας ως πιθανή απώλεια οποιωνδήποτε λειτουργιών οι οποίες δεν είναι δεσμευμένες, διότι αν γινόταν μοντελοποίηση και των δεσμευμένων λειτουργιών τότε αυτό θα σήμαινε πως θα πρέπει να αφήσει και όλες τις λειτουργίες οι οποίες εξαρτώνται ρητά από αυτή. Αν και το σύστημά μπορεί να προκαλέσει τα πιο περίπλοκα σφάλματα άλλων μοντέλων σφαλμάτων , τα οποία θεωρούν πιο περίπλοκη καταστροφή δεδομένων και ασυνεπή συμπεριφορά, εστιάζει στην πιο απλή περίπτωση.

Ένα άλλο άρθρο που μελετήθηκε για την ανάλυση αξιοπιστίας λογισμικού είναι των Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, και Vijay Chidambaram [11], το οποίο αναφέρεται σε μία νέα προσέγγιση για τη δοκιμή συνέπειας συντριβής συστήματος αρχείων (file-system crash consistency): οριοθετημένη δοκιμή συντριβής μαύρου κουτιού (bounded black-box crash testing). Δοκιμάζει το σύστημα αρχείων με τον τρόπο του μαύρου κουτιού χρησιμοποιώντας φόρτους εργασίας των λειτουργιών του συστήματος αρχείων.

Δεδομένου ότι ο χώρος των πιθανών φόρτων εργασίας είναι άπειρος, περιορίζει αυτόν τον χώρο με βάση παραμέτρους όπως ο αριθμός των λειτουργιών του συστήματος αρχείων ή ποιες λειτουργίες πρέπει να συμπεριληφθούν και παράγει εξαντλητικά φόρτους εργασίας μέσα σε αυτόν τον οριοθετημένο χώρο. Κάθε φόρτο εργασίας δοκιμάζεται στο σύστημα αρχείων προορισμού με προσομοίωση σφαλμάτων απώλειας (simulating power-loss crashes) ισχύος κατά την εκτέλεση του φόρτου εργασίας και ελέγχει εάν το σύστημα αρχείων ανακάμπτει σε σωστή κατάσταση μετά από κάθε συντριβή.

Η οριοθετημένη δοκιμή συντριβής μαύρου κουτιού (bounded black-box crash testing) βασίζεται σε πληροφορίες που προέκυψαν από τη μελέτη τους για σφάλματα συνέπειας συντριβής που αναφέρθηκαν στα συστήματα αρχείων Linux. Παρατηρήσαν ότι τα περισσότερα σφάλματα που αναφέρθηκαν μπορούν να αναπαραχθούν χρησιμοποιώντας μικρό φόρτο εργασίας τριών ή λιγότερων λειτουργιών του συστήματος αρχείων σε ένα

σύστημα αρχείων, και ότι όλα τα αναφερόμενα σφάλματα οφείλονται σε συντριβές μετά από fsync() και σχετικές κλήσεις συστήματος. Κατασκεύασαν δύο εργαλεία, CRASHMONKEY και ACE, για να αποδείξουν την αποτελεσματικότητα αυτής της προσέγγισης.

Bug #	File System	Consequence	# of ops	Bug present since
1	btrfs	Rename atomicity broken (file disappears)	3	2014
2	btrfs	Rename atomicity broken (file in both locations)	3	2018
3	btrfs	Directory not persisted by fsync*	3	2014
4	btrfs	Rename not persisted by fsync	3	2014
5	btrfs	Hard links not persisted by fsync	2	2014
6	btrfs	Directory entry missing after fsync on directory	2	2014
7	btrfs	Fsync on file does not persist all its paths	1	2014
8	btrfs	Allocated blocks lost after fsync*	1	2014
9	F2FS	File recovers to incorrect size*	1	2015
10	F2FS	Persisted file disappears*	2	2016

Σχήμα 2.2: Σφάλματα από τα εργαλεία CRASHMONKEY και ACE

Ο πιο πάνω πίνακας του σχήματος 2.2 δείχνει τα νέα σφάλματα που βρέθηκαν με την βοήθεια του CRASHMONKEY και ACE. Τα σφάλματα έχουν σοβαρές συνέπειες, που κυμαίνονται από την απώλεια κατανεμημένων μπλοκ σε ολόκληρα αρχεία και καταλόγους που εξαφανίζονται. Τα σφάλματα υπάρχουν εδώ και αρκετά χρόνια στον πυρήνα (kernel), δείχνοντας έτσι την ανάγκη για συστηματικές δοκιμές. Ακόμη και οι φόρτοι εργασίας με λειτουργία ενός συστήματος αρχείων έχουν οδηγήσει σε σφάλματα. Οι προγραμματιστές έχουν υποβάλει μια ενημερωμένη έκδοση κώδικα για σφάλματα που έχουν επισημανθεί με *.

Ακόμη άλλο άρθρο που χρησιμοποιήθηκε για αυτή την εργασία είναι των Mai Zheng, Joseph Tucek, Feng Qin and Mark Lillibridge από το Ohio State University [12] που δείχνει ότι από πριν από μερικά χρόνια, πολλοί δίσκοι SSD έχαναν δεδομένα όταν υπήρχε διακοπή τροφοδοσίας. Απογοητεύσεις όπως αυτές σε ώριμα εμπορικά προϊόντα που έχουν αποσταλεί σε μεγάλο αριθμό πελατών που πληρώνουν εδώ και χρόνια θέτουν υπό αμφισβήτηση όλο το υλικό και το λογισμικό. Δεν μπορούμε να εμπιστευτούμε το λογισμικό ή το υλικό κάτω από την εφαρμογή μας.

Ο προσεκτικός σχεδιασμός, η προσεκτική εφαρμογή και η επαλήθευση όπου είναι δυνατόν είναι σημαντικοί κρίκοι στην αλυσίδα. Σε αντίθεση με την εργασία αυτή, τα πιο πάνω άρθρα [11] [12] επικεντρώνονται είτε στην ακεραιότητα και ορθή λειτουργία των συστημάτων βάσεων δεδομένων είτε των δίσκων SSD , ενώ η εργασία επικεντρώνεται στην ακεραιότητα όλης της στοίβας συστήματος, συμπεριλαμβανομένου βάσης δεδομένων, λειτουργικού συστήματος, του αρχείου συστημάτων και του δίσκου.

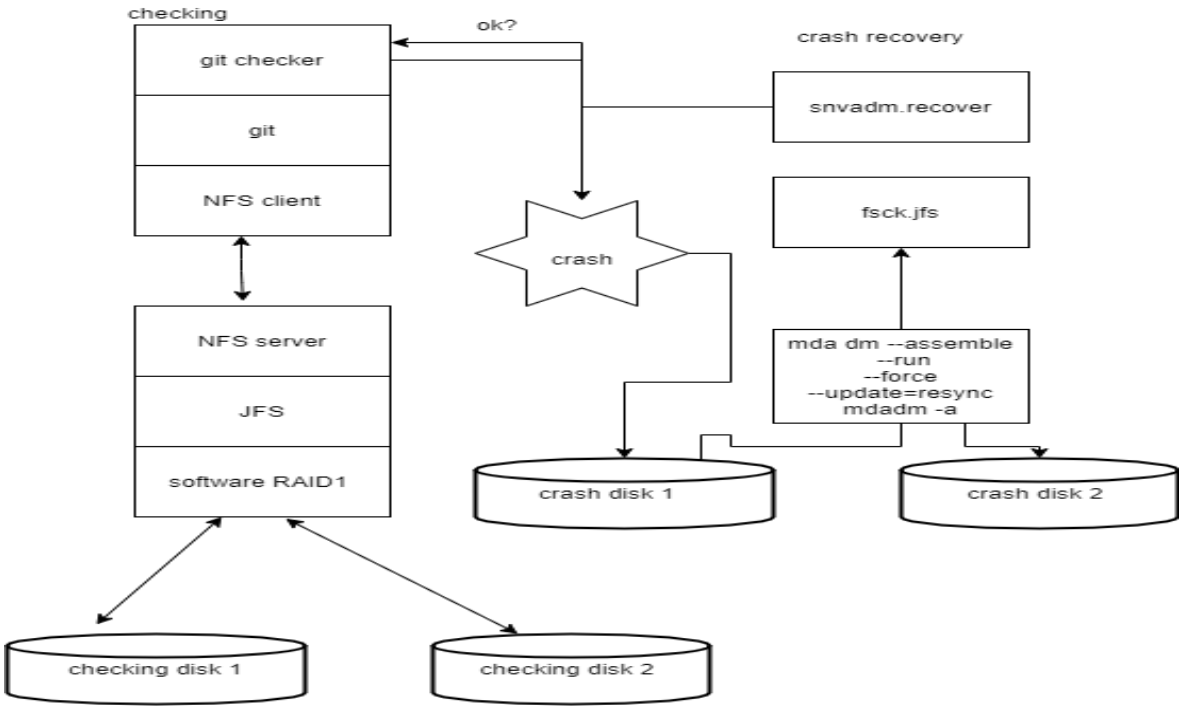
Επίσης, έγινε μελέτη του άρθρου των Junfeng Yang, Can Sar, και Dawson Engler [3] το οποίο δημοσιεύτηκε το 2006. Σύμφωνα με το συγκεκριμένο άρθρο τα σφάλματα του συστήματος αποθήκευσης είναι από τα πιο καταστροφικά πιθανά σφάλματα. Μπορούν να καταστρέψουν τα επίπονα δεδομένα (persistent data), με πολύ κακές συνέπειες εάν το σύστημα έχει μόνο ένα αντίγραφο. Δυστυχώς, ο κώδικας αποθήκευσης (storage code) είναι ταυτόχρονα τόσο δύσκολος στη λογική όσο και στη δοκιμή. Εάν το σύστημα καταρρεύσει σε οποιοδήποτε σημείο του προγράμματος, ανεξάρτητα από τα ποια δεδομένα προλαβαίνουν να αποθηκευτούν στον δίσκο, τότε πρέπει πάντα να ανακάμψει (rollback) σωστά σε μια έγκυρη κατάσταση.

Το πιο πάνω άρθρο παρουσιάζει το eXplode, ένα σύστημα που καθιστά εύκολο τον λεπτομερή έλεγχο των πραγματικών συστημάτων για τέτοια σφάλματα ανάκτησης σφαλμάτων. Δίνει στους πελάτες(clients) ένα καθαρό πλαίσιο για να χτίσουν και να συνδέσουν μαζί τα ισχυρά συστήματα. Το eXplode διευκολύνει τους ελεγκτές (checkers) να βρουν σφάλματα στον κώδικα ανάκτησης σφαλμάτων: καθώς τρέχουν σε ένα ζωντανό σύστημα (live system), λένε στο eXplode πότε πρέπει να δημιουργήσει τις εικόνες δίσκου που θα μπορούσαν να προκύψουν εάν το σύστημα αποτύχει στο τρέχον σημείο εκτέλεσης (execution point), το οποίο στη συνέχεια ελέγχουν για σφάλματα. Το eXplode σχεδιάστηκε έτσι ώστε οι πελάτες να μπορούν να ελέγξουν σύνθετες στοίβες αποθήκευσης που έχουν κατασκευαστεί από πολλά διαφορετικά υποσυστήματα. Επίσης, το eXplode μπορεί να ελέγξει διεξοδικά μεγάλες ποσότητες κώδικα συστήματος αποθήκευσης με λίγη προσπάθεια.

Ένα άλλο άρθρο που μελετήθηκε για την ανάλυση αξιοπιστίας του λογισμικού αποθήκευσης είναι του Donald R. Slutz [10] , το οποίο αναφέρεται σε στοχαστικές

δοκιμές SQL , οι οποίες δεν μπορούν να καλύψουν επαρκώς τον τομέα εισόδου SQL (input domain) . Το RAGS, ένα σύστημα, χτίστηκε για να δημιουργεί στοχαστικά έγκυρες δηλώσεις SQL ένα εκατομμύριο φορές πιο γρήγορα από έναν άνθρωπο και ακολούθως να τις εκτελεί. Αυτό που κάνει το RAGS, δηλαδή η παραγωγή στοχαστικών έγκυρων δηλώσεων SQL είναι ένα αρκετά δύσκολο ζήτημα.

Κάνοντας κάποια πειράματα έχουν εξάγει το συμπέρασμα ότι τα RAGS μπορούν να δημιουργήσουν σταθερά σφάλματα στα απελευθερωμένα προϊόντα SQL (released SQL products). Για να κάνουν όμως αποτελεσματικά τα RAGS, ήταν απαραίτητο να αυτοματοποιηθούν άλλα βήματα της διαδικασίας πειραμάτων, κυρίως θετικές συγκρίσεις αποτελεσμάτων (positive result comparisons) και απλοποίηση δηλώσεων για περιπτώσεις σφαλμάτων (statement simplification for error cases).



Σχήμα 2.3: Το σύστημα RAGS

Η ιδέα ότι είναι ότι σε αυτό το σύστημα όπως φαίνεται στο σχήμα 2.3 έχουν ένα πρόγραμμα που γίνεται η διαχείριση του πηγαίου κώδικα (git). Υπάρχει το NFS το οποίο είναι ένας διακομιστής για εξυπηρέτηση αρχείων, δηλαδή υπάρχει ένας εξυπηρετητής (server) κάπου, όπου εκεί είναι αποθηκευμένα τα αρχεία και γίνετε η προσπέλαση μέσω του δικτύου. Σε κάποια στιγμή το σύστημα δημιουργεί ένα σφάλμα (crash), κάνει ένα

στιγμιότυπο (snapshot), δηλαδή αντιγράφει το checking disk και τον θεωρεί σαν ένα crash disk και προσπαθεί να κάνει ανάκτηση (recovery) από αυτό το snapshot αυτού του δίσκου για να δει τι γίνεται.

Επιπλέον, έχουν ληφθεί υπόψη και τα αποτελέσματα του SIGMOD Programming Contest 2014 [13]. Συγκεκριμένα, στα πλαίσια του πιο πάνω διαγωνισμού, η διαγωνιστική ομάδα του τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου υπό την καθοδήγηση του Καθ. Δημήτρη Ζεϊναλιπούρ έχει αναπτύξει μια πλατφόρμα για την εξομοίωση βλαβών μέσω του τερματισμού διεργασίας (process kill).

Τέλος, για την παρούσα διπλωματική εργασίας έγινε μελέτη μιας προηγούμενης μελέτης που σχετίζεται σε μεγάλο βαθμό με το συγκεκριμένο θέμα. Η μελέτη έχει γίνει από την Ιωάννα Γεωργίου, απόφοιτο του Τμήματος Πληροφορικής στο Πανεπιστήμιο Κύπρου και είχε ως σκοπό την μελέτη ευρωστίας συστημάτων διαχείρισης δεδομένων έναντι διακοπών τροφοδοσίας σε τυχαίες χρονικές στιγμές. Η συγκεκριμένη μελέτη δεν κατάφερε να προκαλέσει κάποιο σφάλμα ή απώλεια δεδομένων από τυχαίες διακοπές τροφοδοσίας. Παρ΄ όλα αυτά, μέρος του κώδικα της Ιωάννας Γεωργίου έχει χρησιμοποιηθεί στην συγκεκριμένη διπλωματική εργασία για την ανάπτυξη, νέων πιο αποτελεσματικών μεθόδων για διακοπές τροφοδοσίας.

Κεφάλαιο 3

Περιγραφή testbed

3.1 Εισαγωγή 16
3.2 Κύκλωμα διακοπής τροφοδοσίας 18
3.3 Testbed 19
3.4 Υλοποίηση των εκτελέσιμων script 22
3.4.1 Επεξήγηση του rab.csh 22
3.4.2 Επεξήγηση του ptf_check.csh 23
3.4.3 Επεξήγηση του ptf_run.csh 24
3.5 Το αρχείο sqlite.py 24
3.6 Έλεγχος δεδομένων 25
3.6.1 Εισαγωγή 25
3.6.2 Επεξήγηση του checkthedata.py 26
3.6.3 Υλοποίηση βοηθητικών αρχείων 29
3.6.3.1 Το αρχείο progress_report 29
3.6.3.2 Το αρχείο check.txt 30

3.1 Εισαγωγή

Όπως προαναφέρθηκε, για την υλοποίηση της συγκεκριμένης διπλωματικής εργασίας ήταν αναγκαία η χρήση ενός testbed το οποίο θα χρησίμευε στις διακοπές τροφοδοσίας στο σύστημα της βάσης δεδομένων, SQLite. Το συγκεκριμένο κύκλωμα το σχεδίασε ο Terence Kelly, ανεξάρτητος ερευνητής με διδακτορικό από το πανεπιστήμιο «University of Michigan». Επίσης, ο Terence Kelly σχεδίασε το λογισμικό το οποίο προκαλεί διακοπές τροφοδοσίας μέσω του Raspberry Pi, κάνοντας χρήση των GPIO pins (General Purpose Input Output). Η σχεδίαση του testbed για το κύκλωμα διακοπής ισχύος που

περιγράφεται στο συγκεκριμένο κεφάλαιο είναι μια συνοπτική περιγραφή της δημοσίευσης που έχει κάνει ο Terence Kelly στο άρθρο «Is Persistent Memory Persistent? », το οποίο δημοσιεύτηκε τον Απρίλιο του 2020 [2].

Όπως αναφέρει και στο άρθρο του ο Terence Kelly, η πιο σημαντική προϋπόθεση για μια δοκιμή διακοπής ρεύματος είναι ότι το λογισμικό που εκτελείται στον κεντρικό υπολογιστή πρέπει να είναι σε θέση να διακόψει απότομα την ισχύ του κεντρικού υπολογιστή σε χρονικές στιγμές της επιλογής του (λογισμικού). Στη συνέχεια, η ισχύς πρέπει να αποκατασταθεί με κάποιο τρόπο στον κεντρικό υπολογιστή, ο οποίος πρέπει να ανταποκριθεί με επανεκκίνηση και εκκίνηση του επόμενου κύκλου δοκιμών. Ο κεντρικός υπολογιστής θα πρέπει να είναι ανθεκτικός, ικανός να αντέχει πολλές χιλιάδες διακοπές ρεύματος και θα πρέπει να είναι σε θέση να εκτελεί κύκλους απενεργοποίησης / ενεργοποίησης γρήγορα. Επίσης, με το συγκεκριμένο μοντέλο διακοπής τροφοδοσίας μπορούμε να αναπαραστήσουμε πραγματικές διακοπές τροφοδοσίας οι οποίες θα επηρέαζαν και δεδομένα που είναι γραμμένα σε buffers στο λειτουργικό σύστημα ή το δίσκο. Αυτό είναι κάτι που δεν μπορεί να επιτευχθεί με την εντολή kill process, η οποία δεν θα προκαλέσει corruption σε δεδομένα που υπάρχουν στην ενδιάμεση μνήμη σελίδων του λειτουργικού συστήματος. Και γι’ αυτό το συγκεκριμένο μοντέλο είναι ιδανικό για τη μελέτη μας.

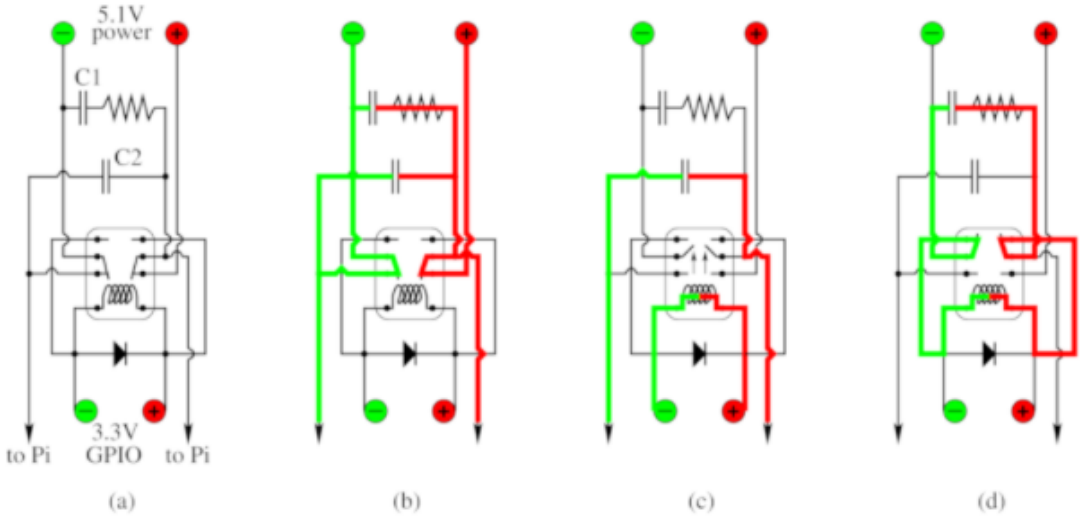
Η επιλογή του υπολογιστή Raspberry Pi για την συγκεκριμένη διπλωματική εργασία είναι η κατάλληλη τόσο για την υλοποίηση των φόρτων εργασίας, όσο και για το σύστημα διακοπής τροφοδοσίας. Οι υπολογιστές Raspberry Pi είναι μικροί και αρκετά φθηνοί για να είναι αναλώσιμοι και να αντλούν πολύ λίγη ενέργεια. Το Pi τρέχει το λειτουργικό σύστημα Linux και σχεδόν όλο το λογισμικό Linux. Επίσης, εκκινείται γρήγορα και αυτόματα όταν ενεργοποιείται. Οι ακίδες GPIO pins (General Purpose Input Output) επιτρέπουν στο λογισμικό να ελέγχει εύκολα τα εξωτερικά κυκλώματα. Το testbed που έχει υλοποιήσει ο Terence Kelly χρησιμοποιεί το Raspberry Pi 3 Model B+.

Το υπόλοιπο αυτής της ενότητας περιγράφει τον κεντρικό υπολογιστή(Raspberry Pi), τα βοηθητικά κυκλώματα και το λογισμικό που μαζί επιτυγχάνουν αυτούς τους στόχους.

3.2 Κύκλωμα διακοπής τροφοδοσίας

Το testbed που έχει υλοποιήσει ο Terence Kelly χρησιμοποιεί ένα μινιμαλιστικό κύκλωμα διακοπής τροφοδοσίας. Και εκτός από τη φτηνότερη υλοποίηση του κυκλώματος, υποστηρίζει και πιο επίπονες δοκιμές επιτρέποντας έτσι στη μηχανή υποδοχής να ελέγχει απευθείας τις διακοπές ρεύματος. Η καρδιά του κυκλώματος διακοπής ισχύος είναι ένα μικροσκοπικό ηλεκτρομηχανικό ρελέ (electromechanical relay). Η τροφοδοσία από την παροχή ρεύματος περνάει από τις κανονικά κλειστές επαφές του ρελέ έως ότου το κύκλωμα διακοπής κόψει την ισχύ αλλάζοντας τους πόλους του ρελέ στις κανονικά ανοιχτές επαφές.

Χρησιμοποιούμε ένα ρελέ (relay) επειδή αποσυνδέει φυσικά την τροφοδοσία, ακριβώς όπως θα γινόταν αν τραβούσαμε το καλώδιο τροφοδοσίας με το χέρι μας. Η υπολειπόμενη ενέργεια στο τροφοδοτικό δεν μπορεί να βοηθήσει τον κεντρικό υπολογιστή μέχρι το ρελέ μεταξύ τους να ανοίξει τα καλώδια. Τα κυκλώματα παροχής ηλεκτρικού ρεύματος μπορεί να περιέχουν μια εκπληκτική ποσότητα υπολειμματικής ενέργειας, αρκετή για να επιτρέψει ακόμη και στους υπολογιστές κλάσης διακομιστή να κλείσουν, αναφέρει ο Terence Kelly στο άρθρο του.



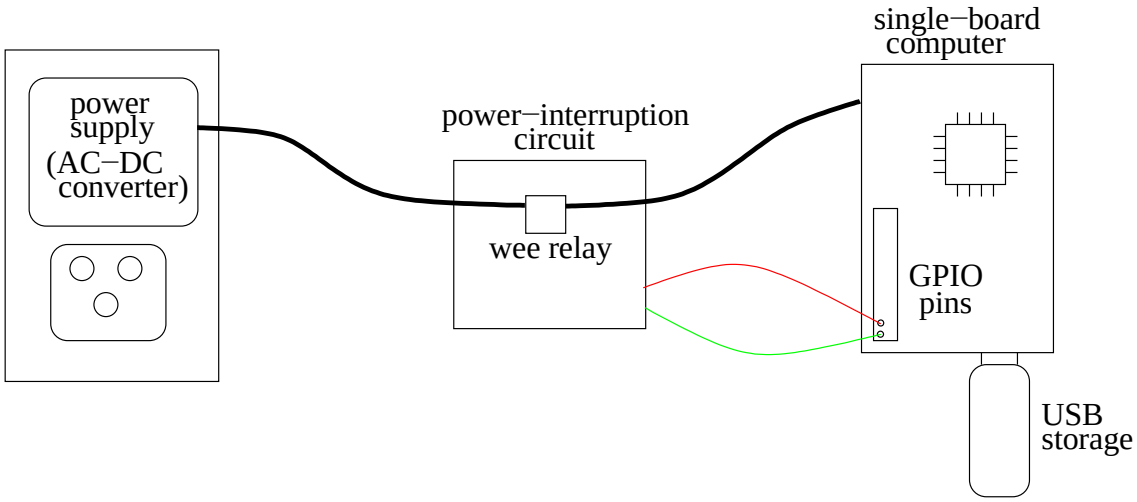
Σχήμα 3.1: Κύκλωμα διακοπής τροφοδοσίας (PiNap)
Πηγή https://queue.acm.org/detail.cfm?id=3400902

Το πιο πάνω σχήμα 3.1 δείχνει το κύκλωμα PiNap, στα διάφορα στάδια λειτουργίας του (με χρονολογική σειρά) που διακόπτει την ισχύ στο Raspberry Pi για αρκετά δευτερόλεπτα. Πιο κάτω περιγράφεται αναλυτικά η τεχνολογία πίσω από το κύκλωμα διακοπής τροφοδοσίας.

Όπως φαίνεται στο σχήμα στην εικόνα (α) το κύκλωμα περιλαμβάνει δύο πυκνωτές, C1 και C2. Ο ένας βοηθά στην αλλαγή του ρελέ και ο άλλος αναλαμβάνει το ρόλο της ενεργοποίησης του πηνίου του ρελέ, ενώ το Raspberry Pi είναι απενεργοποιημένο. Στην εικόνα (β) απεικονίζεται η κανονική λειτουργία του κυκλώματος, δηλαδή η εξωτερική πηγή 5,1 V που παρέχει ισχύ στο Raspberry Pi μέσω των κλειστών επαφών του ρελέ. Επίσης, φορτίζει και τους δύο πυκνωτές. Η εικόνα (γ) δείχνει την παροδική κατάσταση, καθώς το λογισμικό στον κεντρικό υπολογιστή ενεργοποιεί το πηνίο του ρελέ μέσω μιας ακίδας GPIO. Η εξωτερική παροχή ρεύματος αποσυνδέεται αμέσως από τον κεντρικό υπολογιστή, αλλά η ενέργεια από τον πυκνωτή C2 επιτρέπει στην ακίδα GPIO να ωθήσει τους πόλους του ρελέ μέχρι να κλείσουν οι κανονικά ανοικτές επαφές. Στη συνέχεια, όπως φαίνεται στην εικόνα (δ), ο πυκνωτής C1 εκφορτίζεται μέσω της σπείρας, διατηρώντας το Raspberry Pi απενεργοποιημένο για λίγα δευτερόλεπτα. Όταν η ενέργεια στο C1 εξαντληθεί, οι πόλοι του ρελέ επιστρέφουν στις κανονικά κλειστές επαφές, επαναφέροντας την ισχύ στο Raspberry Pi, το οποίο εκκινείται ξανά για τον επόμενο κύκλο δοκιμής.

3.3 Testbed

Μέχρι τώρα έχουμε εξηγήσει τη λειτουργία του κυκλώματος διακοπής τροφοδοσίας, το οποίο έχει δοκιμαστεί αρκετές φορές από τον Terence Kelly και είναι αρκετά αξιόπιστο στις διακοπές και επανεκκινήσεις για τους επόμενους κύκλους δοκιμής.
Στο συγκεκριμένο υποκεφάλαιο εξηγείται το υπόλοιπο μέρος του testbed, δηλαδή το testbed σαν σύνολο.



Σχήμα 3.2: Testbed

Πηγή: Από τη διπλωματική εργασία της Ιωάννας Γεωργίου

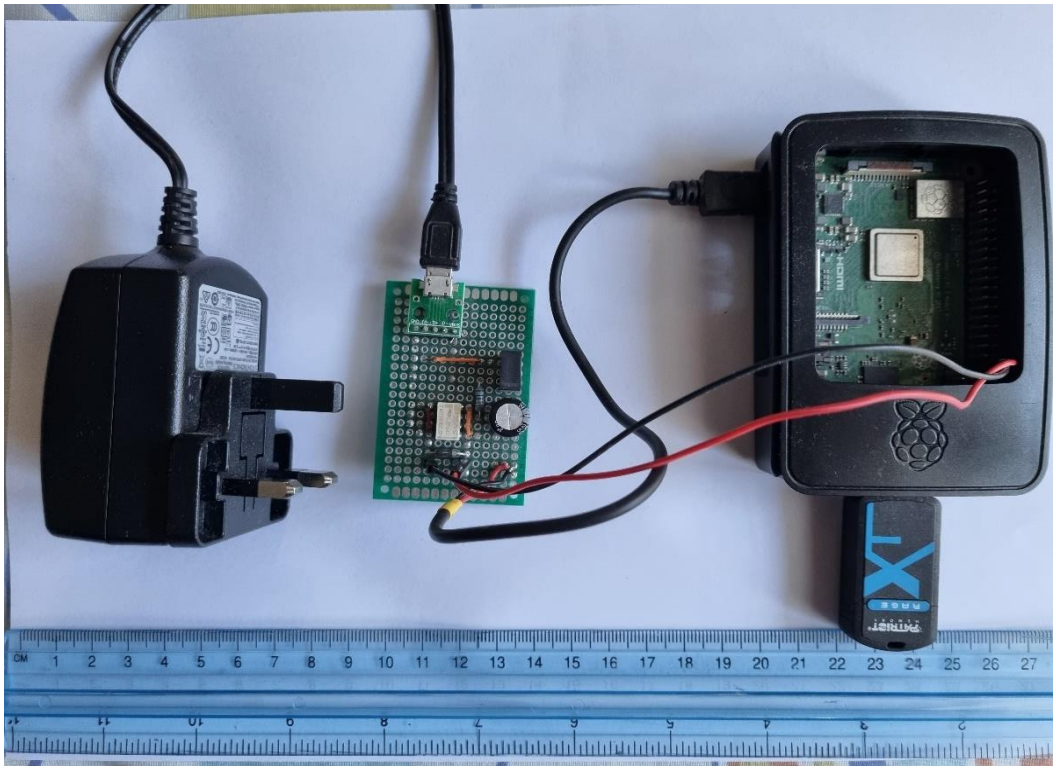
Το σχήμα 3.2 παρουσιάζει το testbed που έχει αναπτύξει ο Terence Kelly. Όπως φαίνεται στο σχήμα, το αριστερό κομμάτι παρουσιάζει την πηγή τροφοδοσίας ρεύματος για την λειτουργία του λογισμικού μας στον υπολογιστή Raspberry Pi.

Όπως βλέπετε, η πηγή τροφοδοσίας δεν είναι άμεσα συνδεδεμένη με το Raspberry Pi, αλλά περνά από ένα ενδιάμεσο κύκλωμα, το κύκλωμα διακοπής τροφοδοσίας PiNap που εξηγήθηκε λεπτομερώς στο προηγούμενο υποκεφάλαιο. Για τη σύνδεση το PiNap μαζί με τη πηγή παροχής ηλεκτρικού ρεύματος, Terence Kelly έκοψε το καλώδιο τροφοδοσίας και τα συγκόλλησε με καλώδια χαλκού 22 AWG στα αντίστοιχα καλώδια της παροχής. Με αυτό τον τρόπο, οποιοσδήποτε χρήστης (host) θα μπορεί να διακόψει την παροχή τροφοδοσίας, στέλνοντας ένα σήμα μέσω των ακίδων GPIO του κυκλώματος, που βρίσκονται στο κέντρο.

Με παρόμοιο τρόπο έγινε και η σύνδεση του κυκλώματος διακοπής τροφοδοσίας με το δεξιό κομμάτι του σχήματος, δηλαδή το Raspberry Pi. Όπως έχετε προσέξει, στο δεξιό μέρος του σχήματος, εκτός από το Raspberry Pi, υπάρχει και ένα USB storage. Τα επίμονα δεδομένα εφαρμογής (persistent application data) βρίσκονται πάνω στην συγκεκριμένη συσκευή αποθήκευση που συνδέεται με τον κεντρικό υπολογιστή μας, το Raspberry Pi. Ο λόγος που έχουμε επιλέξει να αποθηκεύουμε τα δεδομένα σε μια εξωτερική συσκευή αποθήκευσης, αντί στην κύρια μη-πτητική μνήμη αποθήκευσης δεδομένων του Raspberry Pi, είναι για να έχουμε καλύτερο έλεγχο των δεδομένων και

για την αποφυγή τυχών προβλημάτων που μπορούν να προκληθούν (πχ corruption), λόγω των πολλαπλών διακοπών τροφοδοσίας. Συγκεκριμένα, με την εγγραφή των αρχείων στο USB, αποτρέπουμε να προκληθεί κάποια βλάβη στο OS file system του Raspberry Pi, αφού οι κλήσεις της συνάρτησης write() δεν θα το επηρεάζουν. Με αυτό τον τρόπο μειώνεται η πιθανότητα να επηρεαστούν τα δεδομένα από οποιαδήποτε βλάβη και επομένως τα αποτελέσματα των πειραμάτων θα είναι πιο αξιόπιστα.

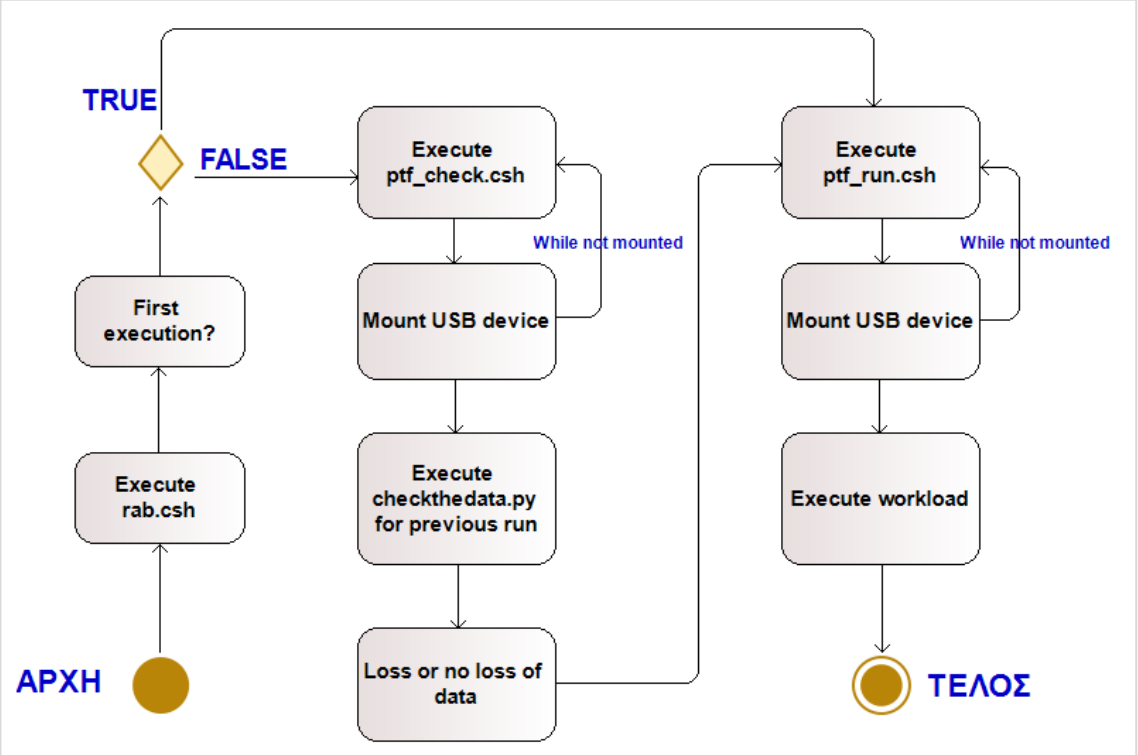
Στο σχήμα 3.3 που παρουσιάζεται πιο κάτω, μπορούμε να δούμε το testbed στην πραγματική του μορφή και το κύκλωμα διακοπής τροφοδοσίας πάνω σε ένα breadboard. Με το συγκεκριμένο σχήμα έχουμε μια καλύτερη αντίληψη για το μέγεθος του testbed. Για σύγκριση, υπάρχει ρίγα που δείχνει το μήκος εκατοστά σε σχέση με όλο το κύκλωμα. Επίσης, μπορείτε να δείτε και το μέγεθος ενός κέρματος της Αμερικής σε σχέση με το κύκλωμα. Όπως έχουμε αναφέρει, το συγκεκριμένο testbed έχει δοκιμαστεί αρκετές φορές χωρίς να παρουσιαστεί κάποιο πρόβλημα. Επομένως μπορούμε να θεωρήσουμε την κατασκευή αυτή είναι αξιόπιστη για τα πειράματα και τα αποτελέσματα μας. Αξίζει να σημειωθεί ότι τα εξαρτήματα του πιο κάτω σχήματος δεν ξεπερνούν τα 100 δολάρια στο σύνολο.



Σχήμα 3.3: Το testbed στην πραγματικότητα

3.4 Υλοποίηση των εκτελέσιμων script

Μέχρι τώρα έχουμε περιγράψει το testbed όσον αφορά το hardware, δηλαδή τα εξαρτήματα και τα υλικά τα οποία χρησιμοποιήθηκαν για να κατασκευαστεί το συγκεκριμένο testbed. Για να πραγματοποιηθούν τα πειράματα μας θα πρέπει να υλοποιηθούν και κάποια scripts τα οποία θα μπορούν να χρησιμοποιούν τα δεδομένα που υπάρχουν πάνω στην εξωτερική συσκευή αποθήκευσης, αλλά και να αξιοποιούν τις ακίδες GPIO. Για την επίτευξη του πιο πάνω υλοποιήθηκαν τρία κύρια scripts. Στο σχήμα 3.4 φαίνεται πώς τα τρία κύρια scripts συνδέονται μεταξύ τους, καθώς επίσης και η όλη διαδικασία ελέγχου και ροής.



Σχήμα 3.4: Διάγραμμα εκτέλεσης των script και της διαδικασίας

3.4.1 Επεξήγηση του rab.csh

Το συγκεκριμένο script είναι το κύριο εκτελέσιμο αρχείο των πειραμάτων μας. Δηλαδή, κάθε φορά που θα τρέξουμε ένα πείραμα θα πρέπει να καλέσουμε το rab.csh. Η κύρια δουλειά του rab.csh είναι να καλέσει τα άλλα δύο scripts διαδοχικά. Πρώτα το pft_check.csh και ακολούθως το pft_run.csh. Προαιρετικά, μπορούμε να

χρησιμοποιήσουμε και τα print μηνύματα που έχει προσθέσει ο Terence Kelly, τα οποία ανακατευθύνονται σε ένα αρχείο, το bootfile, έτσι ώστε να μπορούμε να κρατούμε πληροφορίες για την ημερομηνία και ώρα εκτέλεσης του script.

3.4.2 Επεξήγηση του ptf_check.csh

Το συγκεκριμένο script καλείται από το rab.csh και είναι υπεύθυνο για την εκτέλεση των φόρτων εργασίας. Επίσης, σε αυτό το σημείο γίνεται και το function interposition, το οποίο γίνεται με τη βοήθεια του LD_PRELOAD, όπως θα εξηγηθεί στη συνέχεια. Πριν όπως από την εκτέλεση του φόρτου εργασίας, θα πρέπει να συνδεθεί (mount) η εξωτερική συσκευή αποθήκευσης σε μια συγκεκριμένη τοποθεσία που θα ορίζουμε εμείς. Το πιο κάτω κομμάτι κώδικα δείχνει ακριβώς πως επιτυγχάνεται αυτό.

```
1 set d   = `pwd`
2 if ( '/home/pi/PMTest' != $d ) then
3     exit
4 endif
5 set mpt = $d/MntPt
6 set tnf = $mpt/testnum
7 sudo umount            /dev/sda
8 sudo umount            $mpt
9 sudo mount /dev/sda /home/pi/PMTest/MntPt -o uid=pi,gid=pi
```

Σχήμα 3.5: Πώς γίνεται το mount του USB

Το σχήμα 3.5 παρουσιάζει με ενδεικτικό τρόπο πως επιτυγχάνεται το mount. Όπως βλέπετε στην γραμμή 1, ορίζουμε την τοποθεσία που θέλουμε να γίνει το mount και βεβαιωνόμαστε ότι είναι η σωστή τοποθεσία. Στη συνέχεια, στην γραμμή 7, θα πρέπει να κάνουμε unmount την προηγούμενη σύνδεση του USB για την αποφυγή οποιονδήποτε σφαλμάτων που παρουσιάστηκαν στην προηγούμενη σύνδεση. Και στην γραμμή 9 γίνεται το mount του /dev/sda στην τοποθεσία /home/pi/PMTest/MntPt.

3.4.3 Επεξήγηση του ptf_run.csh

Το συγκεκριμένο script καλείται στην αρχή του rab.csh και είναι υπεύθυνο για τον έλεγχο των δεδομένων μετά από μια διακοπή τροφοδοσίας. Δηλαδή, το ptf_check.csh όταν θα εκτελεστεί, θα ελέγξει τα δεδομένα του φόρτου εργασίας που εκτελέστηκε στο προηγούμενο κάλεσμα του rab.csh. Αυτή είναι η πιο λογική μεθοδολογία για τον έλεγχο των δεδομένων, ειδικά όταν θέλουμε να τρέξουμε πολλαπλές δοκιμές για ένα συγκεκριμένο πείραμα ή για μια αυτοματοποιημένη διαδικασία η οποία θα τρέχει πολλές μέρες.

Η υλοποίηση του ptf_check.csh είναι πολύ παρόμοια με του ptf_run.csh, η οποία εξηγείται πιο πάνω. Δηλαδή, πρέπει και εδώ να γίνει unmount και mount η εξωτερική συσκευή αποθήκευσης των δεδομένων. Ακολούθως, θα καλεστεί το script της python που είναι υπεύθυνο για τον έλεγχο των δεδομένων με την εντολή "python3 /home/pi/workspace/project/pinap/workload2/checkthedata.py". Αξίζει να σημειωθεί ότι μέσα στα τρία script που μόλις εξηγήσαμε γίνονται διάφορα sync σε διάφορα σημεία για να συγχρονίσουμε την μέχρι τώρα διαδικασία με το δίσκο. Πιο αναλυτική εξήγηση για το πως γίνεται ο έλεγχος των δεδομένων στο αρχείο checkthedata,py βρίσκεται στο *Κεφάλαιο 3.6*.

3.5 Το αρχείο sqlite.py

Για την εκτέλεση των πιο πάνω φόρτων εργασίας ήταν απαραίτητη και η χρήση κάποιων βοηθητικών αρχείων έτσι ώστε να μπορούν να εξυπηρετηθούν κάποιες λειτουργίες που ήταν αναγκαίες για την χρήση της βιβλιοθήκης της SQLite, καθώς επίσης και για την εξαγωγή αποτελεσμάτων.

Το συγκεκριμένο αρχείο είναι μια wrapper class της βιβλιοθήκης SQLite και γίνεται import μέσα στους φόρτους εργασίας. Οι συναρτήσεις της συγκεκριμένης κλάσης χρησιμοποιούν τις συναρτήσεις της SQLite. Όμως, επειδή η SQLite είναι γραμμένη σε C γλώσσα, γίνεται χρήση της standard βιβλιοθήκης που μας παρέχει η python, η ctypes, έτσι ώστε να μπορούμε να εκτελούμε C κώδικα από την python. Για παράδειγμα, μέσα στο αρχείο sqlite.py υπάρχει η συνάρτηση SetSqliteLibrary(), όπως αναφέρθηκε και

προηγουμένως. Η συγκεκριμένη συνάρτηση, όπως φαίνεται και στο σχήμα 3.6, έχει ως σκοπό να ενσωματώσει την βιβλιοθήκη της SQLite μέσα στους φόρτους εργασίας μας.

```
1 def SetSqliteLibrary(path):
2   global libsql
3   libsql = ctypes.CDLL(path)
```

Σχήμα 3.6: Συνάρτηση για δήλωση έκδοσης βιβλιοθήκης SQLite

Όπως βλέπετε στο σχήμα 3.6 η συνάρτηση SetSqliteLibrary() δηλώνεται μέσα στο αρχείο sqlite.py με την γραμμή 1. Με τις εντολές 2 και 3 αναθέτουμε σε μια καθολική μεταβλητή που θα μπορεί να χρησιμοποιείται από όλες τις συναρτήσεις, τη βιβλιοθήκη της SQLite που βρίσκεται στην τοποθεσία path. Αυτό όπως αναφέραμε και πριν έγινε με τη βοήθεια της συνάρτησης CDLL που μας παρέχει η βιβλιοθήκη ctypes. Με την πιο πάνω συνάρτηση επιτυγχάνουμε να έχουμε πρόσβαση σε όλες τις συναρτήσεις της SQLite. Εκτός από τη SetSqliteLibrary() έχουν υλοποιηθεί κι άλλες wrapper συναρτήσεις που χρησιμοποιούμε μέσα στους φόρτους εργασίας μας, όπως: connect(), execute() και prepare().

3.6 Έλεγχος δεδομένων

3.6.1 Εισαγωγή

Ένα πολύ βασικό κομμάτι της διπλωματικής εργασίας εμπεριέχεται στο συγκεκριμένο κεφάλαιο, δηλαδή τον τρόπο με τον οποίο γίνεται ο έλεγχος των δεδομένων. Ο έλεγχος των δεδομένων θα πρέπει να γίνεται μετά από κάθε διακοπή τροφοδοσίας έτσι ώστε να μπορέσουμε να συλλέξουμε αποτελέσματα για τα δεδομένα μας. Για να καταφέρουμε να ελέγξουμε απώλειες δεδομένων ή την ύπαρξη σφαλμάτων ήταν αναγκαία η ανάπτυξη μιας μεθοδολογίας. Η συγκεκριμένη μεθοδολογία θα πρέπει με εύκολο τρόπο να μπορεί να μας ενημερώνει εάν έχει παρατηρηθεί κάποια απώλεια χωρίς να χρειάζεται να γίνεται χειροκίνητος έλεγχος κάθε φορά. Για την υλοποίηση της μεθοδολογίας και συγκεκριμένα του script που είναι γραμμένο στη γλώσσα προγραμματισμού python έχει συμβάλει η Ιωάννα Γεωργίου.

Η σημασία της ορθής υλοποίησης της πιο πάνω μεθοδολογίας είναι πολύ μεγάλη, λόγω του ότι όλα τα πειράματα και τα αποτελέσματα βασίζονται στον έλεγχο των δεδομένων μετά από την εκτέλεση του συγκεκριμένου script. Δηλαδή όλα τα αποτελέσματα και συμπεράσματα που έχουμε συλλέξει έχουν γίνει με βάση τη συγκεκριμένη μεθοδολογία.

Γι' αυτό το λόγο ήταν αναγκαία η επαλήθευση της ορθότητας της συγκεκριμένης μεθοδολογίας. Δηλαδή, έχουμε δοκιμάσει να τρέξουμε τους φόρτους εργασίας χωρίς κάποια διακοπή τροφοδοσίας. Επομένως, ο έλεγχος των δεδομένων θα πρέπει να δείχνει ότι δεν παρουσιάστηκε κάποια απώλεια δεδομένων. Κάτι το οποίο έγινε με επιτυχία. Επίσης, έχουμε δοκιμάσει να εισάξουμε εσκεμμένα μια διακοπή τροφοδοσίας σε σημείο πριν από τη εκτέλεση του commit πρωτοκόλλου, όπου ξέρουμε ότι δεν θα δημιουργηθούν οι πίνακες kvpairs και forloop. Επομένως, η μη ύπαρξη πινάκων θα ισοδυναμούσε στην απώλεια δεδομένων, αφού δεν θα γίνουν insert τα δεδομένα εξ αρχής. Ο συγκεκριμένος έλεγχος έγινε με επιτυχία και παρουσιάστηκε απώλεια δεδομένων. Με αυτό τον τρόπο μπορούμε να επαληθεύσουμε την ορθότητα της μεθοδολογίας που αναπτύχτηκε για τον έλεγχο των δεδομένων η οποία εξηγείται πιο κάτω. Ακόμη, στο συγκεκριμένο κεφάλαιο θα εξηγηθούν και τα βοηθητικά αρχεία που υλοποιήθηκαν για να πετύχουμε τον έλεγχο των δεδομένων.

3.6.2 Επεξήγηση του checkthedata.py

Η μεθοδολογία η οποία αναφέρεται πιο πάνω στην πιο πάνω εισαγωγή έχει αναπτυχθεί μέσα στο αρχείο checkthedata.py. Με το συγκεκριμένο αρχείο μπορούμε να ελέγχουμε κατά πόσο έχει παρουσιαστεί απώλεια δεδομένων μετά από μια διακοπή τροφοδοσίας. Το αρχείο checkthedata.py καλείται αυτόματα από το script rab.csh και ελέγχει τα δεδομένα πριν από την εκτέλεση της επόμενης δοκιμής. Έτσι, μπορούμε να έχουμε πιο αποδοτικό και γρήγορο έλεγχο των δεδομένων. Επίσης, με το

Η ιδέα πίσω από την υλοποίηση του συγκεκριμένου αρχείου βασίζεται στο ότι θέλουμε να ελέγξουμε τα δεδομένα που έχουν εισαχθεί στους πίνακες κατά της διάρκεια εκτέλεσης των φόρτων εργασίας. Άρα, εάν ξανατρέξουμε τα queries με τα transactions που εκτελέστηκαν προηγουμένως, και εισάξουμε τα δεδομένα τους σε δύο ίδιους, αλλά διαφορετικούς πίνακες, τότε υπό κανονικές συνθήκες θα πρέπει να έχουμε τα ίδια

δεδομένα καταχωρημένα. Με τη φράση «υπό κανονικές συνθήκες» εννοούμε την ομαλή εκτέλεση του φόρτου εργασίας, χωρίς την εισαγωγή κάποιας διακοπής τροφοδοσίας. Αντιθέτως, εάν γίνει κάποια διακοπή, και μετά από τον έλεγχο των δεδομένων παρατηρηθεί ότι οι πίνακες δεν έχουν τα ίδια δεδομένα καταχωρημένα τότε θα μπορούμε να συμπεράνουμε ότι έχει παρουσιαστεί κάποια απώλεια δεδομένων. Κάτι το οποίο αποτελεί ένα από τους κύριους στόχους της συγκεκριμένης διπλωματικής εργασίας.

Επομένως, μέσα στο αρχείο checkthedata.py θα χρησιμοποιήσουμε τα ίδια queries τα οποία έχουν εκτελεστεί προηγουμένως, στους φόρτους εργασίας. Συγκεκριμένα, αναφέρομαι στο init_query το οποίο χρησιμοποιείται για την αρχικοποίηση των πινάκων και το query workload_query το οποίο έχει μετονομαστεί σε correct_query εφόσον δεν θα υπάρξει κάποια διακοπή σε αυτό και αποτελεί την κανονική λειτουργία του φόρτου εργασίας 1. Παράλληλα, έχουν χρησιμοποιηθεί και τα αντίστοιχα queries του φόρτου εργασίας 2 στο αρχείο checkthedata.py που βρίσκεται μέσα στο αντίστοιχο folder. Επίσης, με την εκτέλεση του init_query αναγκάζεται και η βάση δεδομένων να ανακάμψει (recover) από την ξαφνική διακοπή τροφοδοσίας που είχε προηγηθεί. Επομένως, όταν θα εκτελεστεί ξανά η συναλλαγή με το correct_query, διασφαλίζουμε ότι τα δεδομένα δεν θα τύχουν αλλοίωση, λόγω μιας πιθανής unrecovered βάσης.

Επιπλέον, εκτός από τα πιο πάνω queries έχει χρησιμοποιηθεί και το query check_if_all_is_correct. Το συγκεκριμένο query δεν βρίσκεται μέσα στους φόρτους εργασίας, αλλά αποκλειστικά στο αρχείου του ελέγχου των δεδομένων. Σκοπός του check_if_all_is_correct είναι να συγκρίνει τον πίνακα που έχει δημιουργηθεί στο μέσα στο query του φόρτου εργασίας μαζί με τον πίνακα που έχει δημιουργηθεί μέσα στο checkthedata.py. Το σχήμα 3.7 πιο κάτω δείχνει αναλυτικά την υλοποίηση του συγκεκριμένου query.

```
1 check_if_all_is_correct = """
2 SELECT * FROM kvpairs
3 EXCEPT
4 SELECT * FROM correct
5 UNION
6 SELECT * FROM correct
7 EXCEPT
8 SELECT * FROM kvpairs
9 """
```

Σχήμα 3.7: Query για έλεγχο δεδομένων

Το σχήμα 3.7 δείχνει το query που θα εκτελεστεί για να γίνει η σύγκριση. Ο πίνακας kvpairs αναφέρεται στον πίνακα που περιέχει τα δεδομένα που έχουν εισαχθεί κατά τη διάρκεια εκτέλεσης του φόρτου εργασίας και ο πίνακας correct αναφέρεται στον πίνακα που περιέχει τα δεδομένα που έχουν εισαχθεί κατά τη διάρκεια εκτέλεσης του αρχείου checkthedata.py για να γίνει ο έλεγχος.

Στην γραμμή 2-4 επιλέγουμε όλα τα δεδομένα από τον πίνακα kvpairs εκτός από όλα τα δεδομένα του πίνακα correct. Υπό κανονικές συνθήκες, δηλαδή χωρίς διακοπή τροφοδοσίας το αποτέλεσμα της συγκεκριμένης επιλογής πρέπει να είναι κενό. Αυτό, επειδή οι δύο πίνακες θα περιέχουν ακριβώς τα ίδια δεδομένα και η επιλογή του ενός θα αναιρέσει την επιλογή του άλλου. Σε περίπτωση που υπάρξει απώλεια δεδομένων στον πίνακα kvpairs, τότε η επιλογή όλων των δεδομένων του πίνακα correct θα αναιρέσει τα κοινά δεδομένα με τον πίνακα kvpairs (τα δεδομένα που δεν έχουν χαθεί) και έτσι το αποτέλεσμα από την εξίσωση θα είναι τα δεδομένα που έχουν χαθεί. Με παρόμοιο τρόπο, η συγκεκριμένη εξίσωση εφαρμόζεται και στις γραμμές 6-8 με τη διαφορά ότι οι πίνακες είναι αντίστροφοι έτσι ώστε να καλύψουμε όλες τις πιθανές τιμές των δεδομένων που ίσως να έχουν χαθεί. Στην γραμμή 5, χρησιμοποιείται η εντολή UNION της γλώσσας SQLite για να ενοποιήσουμε το αποτέλεσμα της πρώτης εξίσωσης με το αποτέλεσμα της δεύτερης εξίσωσης.

Με αυτό τον τρόπο μπορούμε να ελέγξουμε εάν έχουν χαθεί οποιαδήποτε δεδομένα. Δηλαδή, εάν το αποτέλεσμα της εκτέλεσης όλου του query είναι κενό, τότε η λειτουργία του φόρτου εργασίας πραγματοποιήθηκε κανονικά. Διαφορετικά εάν το αποτέλεσμα δεν

είναι κενό και επιστρέψει δεδομένα, αυτό σημαίνει ότι η διακοπή τροφοδοσίας έχει προκαλέσει σφάλμα στο commit πρωτόκολλο.

Για να καθορίσουμε όμως την απώλεια δεδομένων θα πρέπει να γνωρίζουμε σε ποια διεργασία ακριβώς έγινε η διακοπή τροφοδοσία, καθώς επίσης και μια ένδειξη η οποία θα μας δείχνει εάν παρουσιάστηκε απώλεια δεδομένων ή όχι. Γι' αυτό το λόγο, ήταν αναγκαία η υλοποίηση κάποιων βοηθητικών αρχείων τα οποία εξηγούνται στην συνέχεια.

3.6.3 Υλοποίηση βοηθητικών αρχείων

3.6.3.1 Το αρχείο progress_report

Το συγκεκριμένο αρχείο χρησιμοποιείται για να γίνεται καταγραφή των ολοκληρωμένων συναλλαγών που έχουν γίνει μέχρι μια χρονική στιγμή. Για παράδειγμα, εάν μέχρι τώρα έχουν εκτελεστεί 3 συναλλαγές μέσα στο for loop του φόρτου εργασίας, τότε το progress_report θα έχει καταγραμμένη την ακολουθία '0123'. Το 0 δεν θεωρείται σαν μια συναλλαγή, αλλά χρησιμοποιείται για να μην είναι κενό το αρχείο και να προκληθεί σφάλμα στη καταγραφή συναλλαγών.

Ο λόγος που χρειάζεται να γίνεται η καταγραφή των ολοκληρωμένων συναλλαγών μέσα στο progress_report είναι για να μπορούμε να ελέγχουμε για κάθε ολοκληρωμένη συναλλαγή εάν παρατηρήθηκε κάποια απώλεια δεδομένων. Δηλαδή, το query check_if_all_is_correct που έχει εξηγηθεί στο προηγούμενο υποκεφάλαιο, θα εκτελεστεί σε κάθε ολοκληρωμένη συναλλαγή για να ελέγξει όλες τα πιθανά σημεία στα οποία μπορεί να έχει προκληθεί διακοπή τροφοδοσίας και όχι μόνο στην τελευταία συναλλαγή (στην οποία έγινε η διακοπή τροφοδοσίας).

Μέσα στο αρχείο checkthedata.py θα ανοιχτεί το αρχείο progress_report και θα διαβαστεί το τελευταίο ψηφίο της ακολουθίας που περιέχει μέσα και θα αποθηκευτεί σε μια μεταβλητή last_char. Αν πάρουμε το παράδειγμα που αναφέρθηκε πιο πάνω, τότε ο τελευταίος χαρακτήρας της ακολουθίας είναι το 3 και αυτό σημαίνει ότι έχουν ολοκληρωθεί 3 συναλλαγές πριν την διακοπή τροφοδοσίας. Άρα, το πιο πιθανό η διακοπή τροφοδοσίας να έγινε κατά τη διάρκεια εκτέλεσης της 4ης συναλλαγής μέσα στο φόρτο

εργασίας. Γι' αυτό το λόγο θα πρέπει να ελέγξουμε και τα δεδομένα της συναλλαγής, άρα θα πρέπει να γίνει last_char = last_char+1.

Στη συνέχεια, μέσα στον κώδικα του checkthedata.py θα εκτελεστεί το for loop για όλες τις συναλλαγές που έχουν ολοκληρωθεί και της τελευταίας συναλλαγής. Ο έλεγχος των δεδομένων θα γίνει σε κάθε συναλλαγή και γι' αυτό ήταν αναγκαίο να γνωρίζουμε τον αριθμό των ολοκληρωμένων συναλλαγών από το progress_report.

3.6.3.2 Το αρχείο check.txt

Το συγκεκριμένο αρχείο χρησιμοποιείται επίσης μέσα στο αρχείο checkthedata.py. Μέσα στο check.txt θα καταγράφεται το αποτέλεσμα των πειραμάτων, δηλαδή παρουσιάστηκε οποιαδήποτε απώλεια δεδομένων μετά από τη διακοπή τροφοδοσίας ή όχι. Εάν η τιμή που περιέχεται μέσα στο αρχείο check.txt είναι 1, τότε σημαίνει ότι η εκτέλεση του φόρτου εργασίας λειτούργησε κανονικά και δεν παρουσιάστηκε κάποιο σφάλμα. Διαφορετικά, εάν η τιμή είναι 0, τότε υπήρξε κάποια απώλεια δεδομένων. Με αυτό τον τρόπο μπορούμε εύκολα και αποδοτικά να ενημερωθούμε εάν έχει παρατηρηθεί κάποια απώλεια δεδομένων χωρίς να χρειάζεται να γίνεται χειροκίνητος έλεγχος κάθε φορά.

Συγκεκριμένα το αρχείο check.txt χρησιμοποιείται μέσα στο for loop του checktehdata.py, μετά την εκτέλεση του query check_if_all_is_correct. Το σχήμα 3.8 αποτελεί μέρος εκτέλεσης του for loop, στο οποίο φαίνεται η χρησιμότητα του check.txt.

```
1 tuples = sqlite.execute(db, check_if_all_is_correct.encode())
2 if ((n==(last_char-1)) and (not tuples)):
3       print('1', file=open('/home/pi/workspace/project/pinap/workload1_sqlite/check.txt', 'w'))
4       sys.exit(1)
5   elif ((n==(last_char-1)) and (tuples)):
6       print('0', file=open('/home/pi/workspace/project/pinap/workload1_sqlite/check.txt', 'w'))
7       sys.exit(0)
```

Σχήμα 3.8: Εγγραφή απώλειας δεδομένων ή όχι στο αρχείο check.txt

Στην γραμμή 1 βλέπουμε την εκτέλεση του query check_if_all_is_correct. Όπως αναφέραμε και πιο πάνω, αν το αποτέλεσμα είναι 0 τότε δεν παρουσιάστηκε κάποια απώλεια, διαφορετικά παρουσιάστηκε κάποιο σφάλμα. Στην γραμμή 2 γίνεται έλεγχος

εάν βρισκόμαστε στον έλεγχο της τελευταίας συναλλαγής και αν το αποτέλεσμα του check_if_all_is_correct είναι κενό. Εάν είναι true, τότε θα γραφτεί στο αρχείο check.txt η τιμή 1 και θα τερματίσει το πρόγραμμα. Αντίθετα, στην γραμμή 5 εάν το αποτέλεσμα του check_if_all_is_correct είναι κενό, δηλαδή υπάρχουν tuples, τότε θα γραφτεί στο αρχείο check.txt η τιμή 0 δείχνοντας μας έτσι την ύπαρξη απώλειας δεδομένων.

Κεφάλαιο 4

Υλοποίηση Βιβλιοθήκης

4.1 Εισαγωγή 32
4.2 Η σημασία του Function Interposition 34
4.2.1 Ορισμός 34
4.2.2 Απαιτήσεις για επιτυχημένο function interposition 35
4.2.3 Μέθοδος 1 – ctypes Python library 37
4.2.4 Μέθοδος 2 – LD_PRELOAD 38
4.3 Υλοποίηση συνάρτησεων 40
4.3.1 Εισαγωγή 40
4.3.2 Η συνάρτηση unlink() 40
4.3.3 Η συνάρτηση fsync() 42
4.3.4 Η συνάρτηση fdatasync() 43
4.4 Χτίσιμο και σύνδεση της βιβλιοθήκης 44
4.4.1 Χτίσιμο (build) της βιβλιοθήκης 44
4.4.2 Επαλήθευση 45

4.1 Εισαγωγή

Μέχρι τώρα έχουν γίνει διάφορες μελέτες που στοχεύουν σε παρόμοια ζητήματα με την συγκεκριμένη διπλωματική εργασία, δηλαδή την μελέτη ευρωστίας σε συστήματα αποθήκευσης και διαχείρισης δεδομένων. Η μελέτη που έχει κάνει η Ιωάννα Γεωργίου, απόφοιτος του Τμήματος Πληροφορικής στο Πανεπιστήμιο Κύπρου, είχε ως σκοπό τη μελέτη της ευρωστίας της βάσης δεδομένων SQLite σε τυχαίες διακοπές τροφοδοσίας. Συγκεκριμένα, χρησιμοποίησε τη συνάρτηση sqlite3OsDelete(), η οποία βρίσκεται μέσα στο αρχείο pager.c του πηγαίου κώδικα της SQLite. Η συγκεκριμένη συνάρτηση σε

κάποια φάση της υλοποίησης της ήταν γνωστό ότι καλούσε τη συνάρτηση unlink(), από την standard library της γλώσσας προγραμματισμού C. Για την τυχαία διακοπή τροφοδοσίας, τοποθέτησε μέσα στο αρχείο rab.csh μια τυχαία καθυστέρηση η οποία παραγόταν από μια τυχαία ψευδό-ακολουθία. Με αυτό το τρόπο μπορούσε να διακόψει την τροφοδοσία σε ένα τυχαίο χρονικό διάστημα το οποίο ενεργοποιούσε το κάλεσμα της συνάρτησης sqlite3OsDelete(). Με βάση τις πιο πάνω τυχαίες διακοπές τροφοδοσίας δεν παρατηρήθηκε κάποια απώλεια δεδομένων από τους φόρτους εργασίας. Αυτό το αποτέλεσμα πιθανόν να οφείλεται στο ότι υπάρχουν άπειρες χρονικές στιγμές που μπορεί να γίνει διακοπή τροφοδοσίας και είναι δύσκολο να πετύχουμε απώλεια δεδομένων στο συγκεκριμένο σημείο που πιστεύουμε ότι μπορεί να προκαλέσει σφάλμα.

Με βάση τα πιο πάνω αποτελέσματα είναι ξεκάθαρο ότι οι διακοπές τροφοδοσίας δεν πρέπει να γίνονται σε τυχαίες διακοπές γιατί έτσι είναι δύσκολο να εντοπίσουμε πιθανά σφάλματα της συγκεκριμένης βάσης δεδομένων. Θα ήταν πολύ πιο αποδοτικό οι διακοπές τροφοδοσίας να μπορούν να γίνουν σε συγκεκριμένα σημεία, τα οποία ξέρουμε ότι έχουν σχέση με το commit πρωτόκολλο της SQLite και πιθανόν να κάνουν trigger διάφορα σφάλματα που θέλουμε να μελετήσουμε.

Σε γενικές γραμμές, η σημασία των ευρετικών μεθόδων στην εξαγωγή αποτελεσμάτων για το συγκεκριμένο θέμα είναι πολύ σημαντική. Κάθε μεθοδολογία μπορεί να επιφέρει διαφορετικά αποτελέσματα. Για παράδειγμα με την μέθοδο των τυχαίων διακοπών τροφοδοσίας μπορούσαμε να διακόψουμε την τροφοδοσία σε σημεία σε ολόκληρη την εκτέλεση του φόρτου εργασίας. Απ’ ότι φαίνεται όμως, η συγκεκριμένη μέθοδος δεν κατάφερε να εστιάσει σε σημεία που πιστεύουμε ότι μπορεί να προκαλέσουν κάποιο σφάλμα. Μια άλλη μεθοδολογία θα μπορούσε να στοχεύσει σε πιο αποτελεσματική εισαγωγή διακοπών τροφοδοσίας, σε σημεία που είναι επιρρεπή σε σφάλματα.

Γι’ αυτό το λόγο είναι αναγκαία η δημιουργία μιας βιβλιοθήκης η οποία θα υλοποιεί συναρτήσεις που έχουν άμεση σχέση με τις συναρτήσεις που χρησιμοποιά το πρωτόκολλο της SQLite. Με αυτό το τρόπο θα μπορούμε να εισάγουμε τις διακοπές τροφοδοσίας στο ακριβές σημείο που επιθυμούμε, κάνοντας έτσι πιο αποτελεσματική την έρευνα για την εντόπιση διαφόρων σφαλμάτων και γενικότερα τη μελέτη ευρωστίας της βάσης δεδομένων SQLite. Βέβαια, για να μπορεί να χρησιμοποιηθεί η βιβλιοθήκη

που θα υλοποιήσουμε θα πρέπει με κάποιο τρόπο να την αντικαταστήσουμε με την βιβλιοθήκη της γλώσσας C, εφόσον το πρωτόκολλο της SQLite χρησιμοποιά συναρτήσεις της C.

Ο συγκεκριμένος τρόπος που επιτυγχάνουμε αυτή την αντικατάσταση περιγράφεται στο επόμενο υποκεφάλαιο. Επίσης, στο συγκεκριμένο κεφάλαιο παρουσιάζονται οι τρόποι με τους οποίους μπορεί να γίνει η αντικατάσταση μιας βιβλιοθήκης με την standard library της C, καθώς επίσης και οι συναρτήσεις που υλοποίησα και πώς λειτουργούν όλα μαζί.

4.2 Η σημασία του Function Interposition

4.2.1 Ορισμός

Με τον όρο function interposition εννοούμε την αντικατάστασης κλήσεων σε συναρτήσεις σε δυναμικές βιβλιοθήκες με κλήσεις σε wrapper classes οι οποίες είναι υλοποιημένες από το χρήστη. Το function interposition μπορεί να χρησιμοποιηθεί για διάφορους λόγους. Για παράδειγμα όταν θέλουμε να μετρήσουμε πόσες φορές έγινε κλήση μιας συνάρτησης και για να φυλάξουμε χρήσιμες πληροφορίες και παραμέτρους που έχουν περαστεί σε συναρτήσεις από κλήσεις χρηστών. Επίσης, μπορεί να χρησιμοποιηθεί για εντοπισμό διαφόρων memory leaks και για την προσθήκη επιπλέων μέτρων προστασίας στις συναρτήσεις μας, λόγω της δυνατότητας καταγραφής της ακολουθίας των κλήσεων συναρτήσεων μιας διεργασίας.

Με βάση το άρθρο που δημοσίευσε ο Adrian Kummerländer τον Φεβρουάριο του 2016[4] καταλαβαίνουμε πιο αναλυτικά την έννοια και τη σημασία του function interposition. Οι περισσότερες εφαρμογές και υπηρεσίες που χρησιμοποιούμε σήμερα εξαρτώνται από μια σειρά από ξεχωριστές βιβλιοθήκες που παρέχουν χρήσιμες αφαιρετικότητες στις οποίες βασίζεται η πραγματική εφαρμογή. Καθώς οι περισσότερες εφαρμογές μοιράζονται ένα κοινό σύνολο εξαρτήσεων βιβλιοθήκης, θα ήταν αρκετά σπάταλο να συσκευάζετε κάθε εφαρμογή με το δικό της ξεχωριστό αντίγραφο μιας δεδομένης βιβλιοθήκης. Αυτός είναι ο λόγος για τον οποίο οι περισσότερες βιβλιοθήκες συνδέονται συνήθως δυναμικά, πράγμα που σημαίνει ότι π.χ. μια κλήση σε τυχαία

συνάρτηση της βιβλιοθήκης δεν ορίζεται κατά τη διάρκεια του compile αντικαθιστώντας την με τον πηγαίο κώδικα της βιβλιοθήκης, αλλά ορίζεται ανάλογα και αντίστοιχα κατά το χρόνο εκτέλεσης. Όπως αναφέρει ο Adrian Kummerländer, το function interposition είναι μια χρήσιμη τεχνική σε διάφορα πρακτικά σενάρια, όπως η παροχή προσαρμοσμένων διαμεσολαβητών μνήμης (memory allocators) για αντικατάσταση των τυπικών λειτουργιών μιας βιβλιοθήκης, καθώς επίσης και η παρακολούθηση διαφόρων κλήσεων μιας συνάρτησης μιας εφαρμογής ως πρόσθετο βοήθημα για εντοπισμό σφαλμάτων.

4.2.2 Απαιτήσεις για επιτυχημένο function interposition

Όπως αναφέραμε και πιο πάνω, το function interposition μας βοηθά να αντικαθιστούμε κλήσεις συναρτήσεων με δυναμικές βιβλιοθήκες που είναι υλοποιημένες από το χρήστη, δηλαδή κάνουν την ίδια λειτουργία με την αρχική συνάρτηση χωρίς να επηρεάζουν και να αλλάζουν την λειτουργία της. Για να πετύχει όμως ένα function interposition θα χρειαστεί να κατανοήσουμε πρώτα τις έννοιες RTLD_NEXT, dlsym(), καθώς επίσης και τους function pointers. Οι συγκεκριμένες έννοιες/εντολές έχουν χρησιμοποιηθεί στην υλοποίηση της βιβλιοθήκης μου και αποτελούν σημαντικές απαιτήσεις για να λειτουργήσει σωστά το function interposition.

Ξεκινώντας, με το RTLD_NEXT, είναι μια σήμανση (flag) η οποία υποδεικνύει σε μια συνάρτηση να επιστρέψει τη διεύθυνση του επόμενου συμβόλου του οποίο το όνομα είναι το ίδιο με το όνομα το οποίο θα ζητήσει ο χρήστης. Με άλλα λόγια το RTLD_NEXT επιτρέπει στον χρήστη να παρέχει μια δική του συνάρτηση σε μια άλλη βιβλιοθήκη, πάνω από την αρχική συνάρτηση. Μια διευκρίνηση είναι ότι πρέπει η συνάρτηση του χρήστη να έχει ακριβώς το ίδιο όνομα με την αρχική συνάρτηση, έτσι ώστε να καταλάβει το RTLD_NEXT ποια διεύθυνση πρέπει να επιστρέψει.

Ακολούθως, η συνάρτηση dlsym() χρησιμοποιά το RTLD_NEXT για να επιστρέψει τη διεύθυνση του επόμενου συμβόλου το οποίο είναι φορτωμένο στη μνήμη. Δηλαδή, το συντακτικό του dlsym() με το RTLD_NEXT για την δημιουργία μιας δικής μας συνάρτησης, πχ της malloc() συνάρτησης, πάνω από την αρχική συνάρτηση malloc() θα έπρεπε να γραφτεί με τον εξής τρόπο: **dlsym(RTLD_NEXT, "malloc").**

Όπως αναφέρει και ο Vishal Chovatiya στο άρθρο του με τίτλο «How to hack C/C++ application using RTLD_NEXT with an easy example» [5] που δημοσίευσε το Σεπτέμβριο του 2016, η συγκεκριμένη μέθοδος για αντικατάσταση συναρτήσεων με wrapped συναρτήσεις μπορεί να θεωρηθεί και ως ευπάθεια. Ο λόγος είναι επειδή μπορεί ένας χάκερ να αποθηκεύσει σε δομές δεδομένων ή πίνακες, πληροφορίες όπως κρυπτογραφημένα κλειδιά και άλλα δεδομένα που χρησιμοποιούνται για την αυθεντικοποίηση ενός χρήστη. Βέβαια, υπάρχουν τρόποι για να αποτρέψουμε αυτή την ενέργεια τους οποίους εξηγά ο Vishal Chovatiya στο άρθρο του.

Στη συνέχεια θα εξηγήσουμε τους function pointers. Ένας function pointer, είναι ένας δείκτης που οδηγεί σε μια συνάρτηση. Σε αντίθεση με την αρχικοποίηση μιας μεταβλητής με τιμές δεδομένων, ένας function pointer οδηγεί σε εκτελέσιμο κώδικα μέσα στη μνήμη. Ο λόγος που χρησιμοποιούμε function pointer στην συγκεκριμένη περίπτωση, δηλαδή στο function interposition, είναι επειδή θέλουμε η συνάρτηση dlsym() να μας επιστρέψει την αρχική συνάρτηση της wrapper συνάρτησης που δημιουργήσαμε. Δηλαδή, με την wrapped συνάρτηση μας, δημιουργούμε ένα δείκτη (function pointer) στην αρχική συνάρτηση για να εκτελεστεί ο κώδικας της αρχικής συνάρτησης, όπως ακριβώς περιγράφεται και στον ορισμό του function pointer. Στο παράδειγμα που αναφέραμε πιο πάνω με τη συνάρτηση malloc() για να πετύχουμε function interposition, ένας function pointer θα έπρεπε να χρησιμοποιηθεί με τον εξής τρόπο (σχήμα 3.8):

```
1 void (*fptr)(void) = NULL;
2 fptr = (void (*)(void))dlsym(RTLD_NEXT, "malloc");
```

Σχήμα 4.1: Αρχικοποίηση του function pointer

Η πρώτη γραμμή χρησιμοποιείται για την αρχικοποίηση του function pointer. Σημαντική διευκρίνηση είναι ότι ο τύπος του function pointer πρέπει να είναι ο ίδιος με της αρχικής συνάρτησης που θέλουμε να κάνουμε interpose. Έτσι, δεν θα επηρεάσουμε τη λειτουργία της αρχικής συνάρτησης και θα επιστρέφουμε στον function pointer ότι ακριβώς επιστρέφει (κάνει return) η αρχική συνάρτηση malloc(). Η δεύτερη γραμμή συνδυάζει τις έννοιες που έχουμε αναφέρει μέχρι τώρα. Δηλαδή, θέτουμε τον function pointer (fptr) να είναι ίσος με τη διεύθυνση που θα μας επιστρέψει η συνάρτηση dlsym(), της διεύθυνσης

του επόμενου συμβόλου (RTLD_NEXT) του οποίο το όνομα είναι ίσο με την αρχική συνάρτηση, δηλαδή της συνάρτησης malloc().

Αξίζει να σημειωθεί ότι οι πιο πάνω εντολές για το function interposition βρίσκονται μέσα στη wrapped συνάρτηση που πρέπει να δημιουργήσει ο χρήστης. Και εφόσον έχουμε εξηγήσει τις απαιτήσεις για ένα πετυχημένο function interposition, το επόμενο βήμα είναι να δηλώσουμε στον μεταγλωττιστή ότι θέλουμε να φορτώσει την βιβλιοθήκη μας στη θέση της standard βιβλιοθήκης που μας παρέχει γλώσσα προγραμματισμού C. Υπάρχουν δύο μέθοδοι με τους οποίους μπορούμε να πούμε στον μεταγλωττιστή το πιο πάνω ζήτημα οι οποίοι θα εξηγηθούν πιο κάτω.
Σημείωση: οι δύο μέθοδοι που περιγράφονται πιο κάτω έχουν δεδομένο ότι η βιβλιοθήκη είναι κατασκευασμένη (build) από πριν. Ο τρόπος με τον οποίο γίνεται το χτίσιμο της βιβλιοθήκης περιγράφεται στο επόμενο υποκεφάλαιο.

4.2.3 Μέθοδος 1 – ctypes Python library

Η γλώσσα προγραμματισμού Python διευκολύνει τη συγγραφή των φόρτων εργαίας μας, λόγω των διαφόρων βιβλιοθηκών που μας παρέχει. Όπως για παράδειγμα τη βιβλιοθήκη ctypes. Η συγκεκριμένη βιβλιοθήκη παρέχει τύπους δεδομένων συμβατούς με τη γλώσσα C και επιτρέπει κλήσεις συναρτήσεων σε DLLs ή κοινόχρηστες (shared) βιβλιοθήκες. Συγκεκριμένα, με τη συνάρτηση CDLL που παρέχει η βιβλιοθήκη ctypes, μπορούμε να φορτώσουμε μια εξωτερική βιβλιοθήκη και να χρησιμοποιήσουμε τις συναρτήσεις της μέσα στον κώδικα. Και αφού οι φόρτοι εργασίας που έχουμε υλοποιήσει είναι γραμμένοι σε Python, τότε θα δοκιμάσουμε να χρησιμοποιήσουμε τη συγκεκριμένη μεθοδολογία.

Έστω ότι έχουμε δημιουργήσει τη βιβλιοθήκη mylibrary.so, τότε ο τρόπος με τον οποίο θα φορτώσουμε τη βιβλιοθήκη μέσα στον python κώδικα μας είναι όπως φαίνεται στο σχήμα 4.2.

```
1 import ctypes
2 global handler
3     handler = ctypes.CDLL("path/mylibrary.so")
```

Σχήμα 4.2: Φόρτωμα βιβλιοθήκης στο φόρτο εργασίας

Η πρώτη γραμμή χρησιμοποιείται για να εισάξουμε την βιβλιοθήκη ctypes στον κώδικα μας. Η δεύτερη γραμμή είναι μια global μεταβλητή η οποία θα χρησιμοποιείται από τον υπόλοιπο κώδικα ως ένας handler, από τον οποίο θα μπορούμε να καλούμε τις συναρτήσεις της βιβλιοθήκης μας. Η τρίτη γραμμή καλεί την συνάρτηση CDLL από τη βιβλιοθήκη ctypes, λέγοντας της έτσι ότι θέλουμε να φορτώσουμε τη βιβλιοθήκης μας (mylibrary.so) μέσα στον κώδικα. Με αυτό τον τρόπο μπορούμε να έχουμε πρόσβαση στις συναρτήσεις της βιβλιοθήκης μας.

Παρ' όλα αυτά η συγκεκριμένη μέθοδος, δυστυχώς, δεν μπορεί να εφαρμοστεί στην περίπτωση μας. Ο λόγος είναι επειδή εμείς θέλουμε να χρησιμοποιούμε τις συναρτήσεις που χρησιμοποιά η βάση δεδομένων SQLite στο commit πρωτόκολλο της. Οι συναρτήσεις αυτές καλούνται την χρονική στιγμή που πραγματοποιείται ένα transaction σε ένα query της SQLite. Δηλαδή, δεν μπορούμε μέσω της Python να καλέσουμε και να έχουμε πρόσβαση στις συγκεκριμένες συναρτήσεις που χρησιμοποιούνται τη συγκεκριμένη χρονική στιγμή σε μια συναλλαγή. Εμάς μας ενδιαφέρουν μόνο οι συναρτήσεις που χρησιμοποιούνται κατά τη διάρκεια εκτέλεσης του πρωτοκόλλου, και όχι οποιεσδήποτε κλήσεις. Άλλωστε, θέλουμε να μελετήσουμε την ευρωστία της SQLite κατά τη διάρκεια μιας συναλλαγής, επομένως αν η διακοπή τροφοδοσίας γινόταν σε μια τυχαία κλήση μιας συνάρτησης (πριν ή μετά την χρήση του commit πρωτοκόλλου), δεν θα είχαμε απώλεια δεδομένων και τα αποτελέσματα μας θα ήταν ασαφή.

4.2.4 Μέθοδος 2 – LD_PRELOAD

Εφόσον η πρώτη μέθοδος δεν αρμόζει στην περίπτωση μας, δηλαδή δεν μπορούμε να χρησιμοποιήσουμε τον handler της συνάρτησης ctypes.CDLL() για κάλεσμα μιας συνάρτησης την ώρα εκτέλεσης του query, θα πρέπει να χρησιμοποιήσουμε μια άλλη μεθοδολογία.

Πρέπει με κάποιο τρόπο να ορίσουμε κατά τη εκτέλεση του προγράμματος ότι θέλουμε να φορτώσουμε την βιβλιοθήκη μας πριν από οποιαδήποτε άλλη βιβλιοθήκη. Με αυτό τον τρόπο θα γίνεται χρήση της πρώτης συνάρτησης που βρίσκει ο μεταγλωττιστής (της

wrapper συνάρτησης), αντί της αρχικής συνάρτησης. Για να το πετύχουμε αυτό θα χρησιμοποιήσουμε το LD_PRELOAD.

Το LD_PRELOAD είναι μια χρήσιμη τεχνική που χρησιμοποιείται για να επηρεάσει τη σύνδεση κοινόχρηστων βιβλιοθηκών και την ανάλυση συναρτήσεων κατά το χρόνο εκτέλεσης. Με άλλα λόγια το LD_PRELOAD χρησιμοποιείται για να αντικαταστήσει συναρτήσεις βιβλιοθηκών με δικές μας συναρτήσεις, όπως ακριβώς θέλουμε να πετύχουμε στην περίπτωση μας. Τα προγράμματα συστήματος Linux ld.so και ld-linux.so (dynamic linker/loader) χρησιμοποιούν το LD_PRELOAD για τη φόρτωση καθορισμένων κοινόχρηστων βιβλιοθηκών. Συγκεκριμένα, πριν από οποιαδήποτε άλλη βιβλιοθήκη, ο δυναμικός φορτωτής θα φορτώσει πρώτα κοινόχρηστες βιβλιοθήκες που βρίσκονται μέσα στο LD_PRELOAD.

Έστω ότι έχουμε δημιουργήσει τη βιβλιοθήκη mylibrary.so, και το πρόγραμμα στο οποίο θέλουμε να χρησιμοποιήσουμε τη συγκεκριμένη βιβλιοθήκη ονομάζεται myprogram.c. Επίσης, το πρόγραμμα myprogram.c βρίσκεται στην τοποθεσία "/home/pi/examples". Ο τρόπος με τον οποίο θα χρησιμοποιήσουμε το LD_PRELOAD παρουσιάζεται στο σχήμα 4.3.

```
1 gcc -o myprogram myprogram.c
2 LD_PRELOAD = /home/pi/examples/mylibrary.so ./myprogram
3
4 ή
5
6 LD_PRELOAD = `pwd`/mylibrary.so ./myprogram
```

Σχήμα 4.3: Χρήση του LD_PRELOAD

Η πρώτη γραμμή χρησιμοποιείται για να δημιουργήσουμε ένα object του προγράμματος μας. Η δεύτερη γραμμή χρησιμοποιά το LD_PRELOAD για να πει στο μεταγλωττιστή ότι θα φορτώσει πρώτα την βιβλιοθήκη mylibrary.so που βρίσκεται στην τοποθεσία που ορίσαμε και στην ίδια γραμμή εκτελεί και το πρόγραμμα myprogram.o. Εναλλακτικά, θα μπορούσαμε να χρησιμοποιήσουμε την γραμμή 6, η οποία είναι η ίδια με την γραμμή 2, μόνο που η τοποθεσία της βιβλιοθήκης ορίζεται με το `pwd` το οποίο συμβολίζει την συγκεκριμένο path που βρισκόμαστε τώρα.

Η συγκεκριμένη μεθοδολογία είναι η κατάλληλη για την διεξαγωγή των πειραμάτων μας επειδή έχουμε καταφέρει να ορίσουμε ως προεπιλεγμένες συναρτήσεις τις συναρτήσεις της βιβλιοθήκης μας. Περισσότερες λεπτομέρειες για την επαλήθευση την επιτυχημένης χρήσης του LD_PRELOAD αναφέρονται στο τελευταίο υποκεφάλαιο του συγκεκριμένου κεφαλαίου.

4.3 Υλοποίηση συνάρτησεων

4.3.1 Εισαγωγή

Οι συναρτήσεις που θα υλοποιηθούν έχουν άμεση σχέση με τις συναρτήσεις που χρησιμοποιά το πρωτόκολλο της SQLite. Συγκεκριμένα, θα υλοποιήσουμε τις unlink(), fsync() και fdatasync(). Οι τρεις αυτές συναρτήσεις θα είναι υλοποιημένες ως wrapper (περιτύλιγμα) συναρτήσεις μέσα στην βιβλιοθήκη μας, έτσι ώστε να έχουμε έλεγχο/πρόσβαση μέσα στο πρωτόκολλο commit. Με αυτό το τρόπο θα μπορούμε να εισάγουμε τις διακοπές τροφοδοσίας στο ακριβές σημείο που επιθυμούμε, κάνοντας έτσι πιο αποτελεσματική την έρευνα για την εντόπιση διαφόρων σφαλμάτων και γενικότερα τη μελέτη ευρωστίας της βάσης δεδομένων SQLite. Επίσης, στις υλοποιήσεις των συγκεκριμένων συναρτήσεων επιλέξαμε να χρησιμοποιήσουμε κλήσεις σε βιβλιοθήκες για την εγγραφή των δεδομένων στο δίσκο, αντί την χρήση της σήμασης O_DIRECT. Για παράδειγμα, γίνεται χρήση της συνάρτησης fopen(), αντί της open(). Με την fopen() έχουμε τη δυνατότητα να αποθηκεύουμε τα δεδομένα προσωρινά σε I/O buffer, κάτι το οποίο καθιστά την συγκεκριμένη μέθοδο πιο γρήγορη και αποτελεσματική. Επίσης, για την αποφυγή σύγχρονων εγγραφών στο σύστημα αποθήκευσης το οποίο μπορεί να επηρεάσει τις εγγραφές της βάσης δεδομένων Επιπλέον, με κλήσεις σε βιβλιοθήκες μπορούμε να χρησιμοποιούμε αντικείμενα όπως το FILE *, το οποίο μας παρέχει με χρήσιμες stdio συναρτήσεις, όπως την fscanf().

4.3.2 Η συνάρτηση unlink()

Η συνάρτηση unlink που μας παρέχει η γλώσσα προγραμματισμού C, είναι από τις πιο βασικές και χρήσιμες συναρτήσεις της βιβλιοθήκης unistd.h. Με την συγκεκριμένη

συνάρτηση μπορεί κάποιος να διαγράψει ένα αρχείο. Κατ' ακρίβεια, με την συνάρτηση unlink διαγράφεται το όνομα του αρχείου και όχι το αρχείο. Εξαιρείται η περίπτωση που το μόνο όνομα του αρχείου είναι το συγκεκριμένο όνομα που επιχειρούμε να κάνουμε unlink. Μια επιτυχημένη κλήση της unlink θα επιστρέψει 0, ενώ αν υπάρξει κάποιο error θα επιστρέψει -1. Υπάρχουν και άλλα error name conditions τα οποία περιγράφονται αναλυτικά μέσα στο manual που μας δίνει η C.

Γι' αυτό το λόγο έχω υλοποιήσει περιτύλιγμα (wrapper) για τη συνάρτηση unlink() η οποία επεξηγείται με λεπτομέρεια πιο κάτω. Η υλοποίηση της unlink() είναι βασισμένη στην αρχική unlin(), όπως μας την παραθέτει το documentation [6] της C.
Σημείωση: στον πιο κάτω κώδικα περιγράφεται μέρος της unlink και όχι όλη η συνάρτηση. Το σχήμα 4.4 δείχνει μέρος της unlink που είναι αρκετό για να πετύχουμε το function interposition.

```
1 int unlink(const char *path){
2     static int (*fptr)(const char *) = NULL;
3     if (fptr == NULL){
4         fptr = (int (*)(const char *))dlsym(RTLD_NEXT, "unlink");
5         if (fptr == NULL){
6             printf("dlsym: %s\n", dlerror());
7             return -1;}}
8     printf("\nMy Unlink is called\n");
9     return (*fptr)(path); // Calling original unlink
```

Σχήμα 4.4: Μέρος υλοποίησης unlink()

Όπως βλέπετε, στην γραμμή 1 ορίζουμε τη συνάρτηση unlink η οποία επιστρέφει ένα integer, όπως εξηγήσαμε πιο πάνω, δηλαδή 0 για επιτυχία και -1 για αποτυχία. Επίσης, η unlink παίρνει ως παράμετρο ένα σταθερό char pointer στην τοποθεσία του αρχείου που θέλουμε να κάνουμε unlink. Αξίζει να σημειωθεί ότι το signature της συνάρτησης πρέπει να είναι το ίδιο με την κανονική unlink(). Η δεύτερη γραμμή ορίζει ένα function pointer που θα μας οδηγήσει στη συνέχεια στην αρχική συνάρτηση της unlink. Στην γραμμή 4 ορίζουμε ποια συνάρτηση θέλουμε, στην περίπτωση μας την αρχική unlink. Όπως βλέπετε αυτό επιτυγχάνεται με την βοήθεια του dlsym() και του RTLD_NEXT, τα οποία εξηγήσαμε σε προηγούμενα υποκεφάλαια. Η γραμμή 8 χρησιμοποιήθηκε απλά για σκοπούς επαλήθευσης της λειτουργίας της wrapper unlink και της απόδειξης ότι το

function interposition λειτουργεί όπως ακριβώς πρέπει. Και τέλος, η γραμμή 9 χρησιμοποιά τον function pointer που ορίσαμε πιο πάνω για να χρησιμοποιήσουμε τη λειτουργία της αρχικής unlink, η οποία θα επιστρέψει 0 ή -1.

4.3.3 Η συνάρτηση fsync()

Η συνάρτηση fsync() που μας παρέχει η C μέσω της βιβλιοθήκης unistd.h., είναι επίσης πολύ χρήσιμη για την μελέτη μας. Με την συνάρτηση fsync συγχρονίζονται όλες οι αλλαγές σε ένα αρχείο. Συγκεκριμένα, θα ζητήσει να μεταφερθούν όλα τα δεδομένα του open file descriptor στη συσκευή αποθήκευσης που σχετίζεται με το αρχείο που περιγράφεται από τον file descriptor. Η φύση της μεταφοράς καθορίζεται από την υλοποίηση. Η συνάρτηση fsync () δεν θα επιστρέψει έως ότου το σύστημα ολοκληρώσει αυτήν την ενέργεια ή μέχρι να εντοπιστεί ένα σφάλμα.

Γι' αυτό το λόγο, εάν επιχειρήσουμε να διακόψουμε την τροφοδοσία κατά τη διάρκεια της εκτέλεσης μιας fsync, μέσα στο πρωτόκολλο, ίσως να εντοπίσουμε κάποια απώλεια δεδομένων. Η υλοποίηση στο σχήμα 4.5 της συνάρτησης fsync() αποτελεί μέρος της κώδικα και όχι ολόκληρη τη συνάρτηση.

```
1 int fsync(int fd){
2     static int (*fptr)(const char *) = NULL;
3     if (fptr == NULL){
4         fptr = (int (*)(int))dlsym(RTLD_NEXT, "fsync");
5         if (fptr == NULL){
6             printf("dlsym: %s\n", dlerror());
7             return -1;}}
8     printf("\nMy Fsync is called\n");
9  return (*fptr)(fd); // Calling original fsync
```

Σχήμα 4.5: Μέρος υλοποίησης fsync()

Όπως βλέπετε, στην γραμμή 1 ορίζουμε τη συνάρτηση fsync η οποία επιστρέφει ένα integer, 0 για επιτυχία και -1 για αποτυχία. Επίσης, η fsync παίρνει ως παράμετρο ένα ακέραιο αριθμό για τον file descriptor. Η υπόλοιπη υλοποίηση της fsync είναι παρόμοια με την unlink, μόνο που στην γραμμή 4 ορίσαμε ότι στην συνάρτηση dlsym() θέλουμε το

RTLD_NEXT να είναι ίσο με fsync. Επίσης, στην τελευταία γραμμή επιστρέφουμε τον function pointer που δείχνει στην αρχική fsync, όπως είναι ορισμένη από τη βιβλιοθήκη unistd.h.

4.3.4 Η συνάρτηση fdatasync()

Η συνάρτηση fdatasync() που μας παρέχει η C μέσω της βιβλιοθήκης unistd.h., είναι επίσης πολύ χρήσιμη για την μελέτη μας και αρκετά παρόμοια με την fsync. Η διαφορά της με την fsync είναι ότι δεν ξεπλένει τα τροποποιημένα μεταδεδομένα, εκτός εάν αυτά τα μεταδεδομένα χρειάζονται για να επιτραπεί ο σωστός χειρισμός μιας επακόλουθης ανάκτησης δεδομένων. Ο στόχος της συνάρτησης fdatasync είναι να μειώσει τη δραστηριότητα του δίσκου για εφαρμογές που δεν απαιτούν τον συγχρονισμό όλων των μεταδεδομένων με το δίσκο (πχ timestamps).

Η υλοποίηση της fdatasync είναι ακριβώς η ίδια με την υλοποίηση της fsync, όπως φαίνεται στον κώδικα πιο πάνω. Οι μόνες διαφορές είναι στο σημείο που ζητάμε από τη συνάρτηση dlsym() να μας επιστρέψει την fdatadync με τη βοήθεια του RTLD_NEXT. Επίσης, η τελευταία γραμμή θέλουμε να μας επιστρέψει τον function pointer που δείχνει στην αρχική fdatasync, όπως είναι ορισμένη από τη βιβλιοθήκη unistd.h.

Ίσως η υλοποίηση της fdatasync εκτός από την υλοποίηση fsync να σας δημιουργούν το εξής ερώτημα. Για ποιο λόγο έγινε υλοποίηση και των δύο συναρτήσεων από τη στιγμή που η λειτουργία τους είναι η ίδια και ο σκοπός και των δύο είναι ο συγχρονισμός των δεδομένων. Η απάντηση στο συγκεκριμένο ερώτημα είναι η εξής. Μετά από τα διάφορα πειράματα που εκτελέσαμε παρατηρήσαμε ότι η τελευταία έκδοση της SQLite, του 2021, δεν χρησιμοποιά τη συνάρτηση fsync, αλλά την fdatasync. Επομένως, για την εισαγωγή διακοπών τροφοδοσίας στα συγκεκριμένα σημεία που γίνεται ο συγχρονισμός των δεδομένων είναι απαραίτητη η υλοποίηση της fdatasync. Ενώ, για την πιο παλιά έκδοση, του 2013, χρησιμοποιήσαμε την fsync επειδή δεν παρατηρήσαμε κάποια κλήση της fdatasync.

4.4 Χτίσιμο και σύνδεση της βιβλιοθήκης

4.4.1 Χτίσιμο (build) της βιβλιοθήκης

Μέχρι τώρα έχουμε εξηγήσει το πώς έχουμε υλοποιήσει τις συναρτήσεις τις βιβλιοθήκης μας και την σημασία της κάθε μιας. Έχουμε αναφέρει τον τρόπο που πετύχαμε το function interposition, δηλαδή την αντικατάσταση των αρχικών συναρτήσεων με τις wrapped συναρτήσεις τις βιβλιοθήκης μας. Όμως, αυτό δεν είναι αρκετό για να μπορέσουμε να εκτελέσουμε τα πειράματα για μελέτη της ευρωστίας της βάσης δεδομένων SQLite.

Αυτό που πρέπει να γίνει τώρα είναι να χτίσουμε (build) την βιβλιοθήκη μας για να μπορούμε να την συνδέσουμε με τους φόρτους εργασίας, όπως θα εξηγηθεί στο επόμενο υποκεφάλαιο. Η βιβλιοθήκη θα πρέπει να γίνεται build κάθε φορά που γίνεται μια αλλαγή στον κώδικα ή προσθέτουμε νέο κώδικα/συνάρτηση. Οι εντολές στο σχήμα 4.6 πιο κάτω δείχνουν τον τρόπο με τον οποίο επιτυγχάνουμε το build. Επίσης, οι πιο κάτω εντολές έχουν χρησιμοποιηθεί με την βοήθεια ενός άρθρο από τη σελίδα Journal Dev [7].

```
1 cd /home/pi/workspace/project/pinap/workload1
2 gcc -o mylibrary.so -shared -fPIC mylibrary.c -D_GNU_SOURCE -ldl
```

Σχήμα 4.6: Build της βιβλιοθήκης mylibrary.c

Με παρόμοιο τρόπο, η εντολή 1 χρησιμοποιείται για να μπούμε στην τοποθεσία όπου βρίσκεται ο κώδικας .c της βιβλιοθήκης μας. Ακολούθως, θα δημιουργήσουμε ένα αντικείμενο της βιβλιοθήκης τύπου .so για να μπορούμε να το συμπεριλαμβάνουμε μέσα στους φόρτους εργασίας. Η σήμανση -fPIC χρησιμοποιείται από το μεταγλωττιστή για τη δημιουργία ενός αντικειμένου σε ανεξάρτητο κώδικα. Είναι απαραίτητο για τη δημιουργία shared αντικειμένων. Επίσης, η σήμανση D_GNU_SOURCE ορίζει στο μεταγλωττιστή ότι θα χρησιμοποιήσει το GNU standard για μεταγλώττιση. Αξίζει να σημειωθεί ότι ο πιο πάνω τρόπος είναι ο επίσημος τρόπος για να χτίσουμε την βιβλιοθήκη μας. Όμως, για λόγους ευκολίας, το χτίσιμο της βιβλιοθήκης το έχω τοποθετήσει μέσα στο script rab.csh. Δηλαδή, ανοίγω ένα νέο terminal, μέσα στο terminal που είναι ήδη ανοιχτό για το rab.csh. Αυτό γίνεται με τη βοήθεια παρενθέσεων () , μέσα στις οποίες τοποθετούνται οι δύο εντολές. Με αυτό τον τρόπο, το build της βιβλιοθήκης γίνεται

αυτόματα σε κάθε εκτέλεση του rab.csh, χωρίς να χρειάζεται να το κάνω χειροκίνητα κάθε φορά που κάνω μια αλλαγή στη βιβλιοθήκη μου.

4.4.2 Επαλήθευση

Με τον πιο πάνω τρόπο, καταφέραμε επιτυχώς να δημιουργήσουμε αντικείμενο της βιβλιοθήκης μας, το οποίο θα μπορεί να χρησιμοποιείται από τους φόρτους εργασίας. Όπως, εξηγήθηκε σε προηγούμενο υποκεφάλαιο, η βιβλιοθήκη mylibrary.so ενσωματώνεται μέσα στους φόρτους εργασίας με την τεχνική function interposition που έγινε με τη βοήθεια του LD_PRELOAD.

Το μόνο που απομένει τώρα, είναι να επαληθεύσουμε ότι η βιβλιοθήκη μας έχει ενσωματωθεί σωστά μέσα στους φόρτους εργασίας και όλα λειτουργούν όπως ακριβώς θα περιμέναμε. Δηλαδή, να χρησιμοποιούνται οι συναρτήσεις μας αντί οι συναρτήσεις που μας παρέχει η γλώσσα προγραμματισμού C. Εάν όλα λειτουργούν σωστά, τότε θα μπορούμε να προχωρήσουμε στο επόμενο βήμα που είναι η υλοποίηση των μεθόδων διακοπής τροφοδοσίας.

Για την επαλήθευση αποφάσισα να ακολουθήσω μια πολύ απλή μεθοδολογία, αλλά ταυτόχρονα αρκετά αποτελεσματική. Συγκεκριμένα, έχω εισάγει μηνύματα τα οποία θα τυπωθούν στο terminal με το κάλεσμα των συναρτήσεων μου. Με αυτό τον τρόπο επιβεβαιώνουμε ότι όντως έχει καλεστεί η δική μου συνάρτηση, αφού οι συναρτήσεις της γλώσσας C δεν έχουν κάποια μηνύματα μέσα στον κώδικα τους. Για παράδειγμα, όσον αφορά την συνάρτηση unlink που υλοποίησα έχω τοποθετήσει μέσα στον κώδικα την εντολή printf("\nMy Unlink is called").

Σχήμα 4.7: Κομμάτι από το terminal μετά το execution

Στο σχήμα 4.7 πιο πάνω, παρουσιάζεται το terminal μετά από μια εκτέλεση του rab.csh. Όπως βλέπετε, υπάρχει το μήνυμα “My Unlink is called”, το οποία μας επαληθεύει ότι έχει χρησιμοποιηθεί η wrapper συνάρτηση unlink, αντί της κανονικής unlink. Επίσης, τυπώνω το path το οποίο χρησιμοποιά ως παράμετρο η unlink, καθώς επίσης και τα δεδομένα μετά από την ολοκλήρωση μιας συναλλαγής. Με το τύπωμα των δεδομένων επιβεβαιώνουμε ότι wrapper unlink είναι υλοποιημένη σωστά και κάνει ότι ακριβώς κάνει η αρχική unlink χωρίς να επηρεάζει την διαδικασία αποδέσμευσης του αρχείου. Με παρόμοιο τρόπο, έχω προσθέσει μηνύματα μέσα στον κώδικα και των άλλων δύο συναρτήσεων και επιβεβαίωσα ότι χρησιμοποιούνται από τους φόρτους εργασίας και λειτουργούν όπως ακριβώς οι αντίστοιχες αρχικές συναρτήσεις.

Επιπλέον, θα κάνουμε επαλήθευση ότι η unlink() μας καλεί όντως την κανονική unlink(). Αυτό έχει γίνει με τη βοήθεια κάποιου dummy αρχείου το οποίο θα επιχειρήσουμε να διαγράψουμε μέσω της wrapper unlink(). Μετά από δοκιμή που έχει γίνει, παρατηρήσαμε ότι σωστά γίνεται διαγραφεί αρχείου επιβεβαιώνοντας έτσι ότι η wrapper unlink() είναι υλοποιημένη ορθά.

Κεφάλαιο 5

Υλοποίηση φόρτων εργασίας και σχετικών αρχείων

5.1 Εισαγωγή 47
5.2 Φόρτος Εργασίας 1 48
5.3 Φόρτος Εργασίας 2 51
5.4 Η χρησιμότητα του flush() και του fsync_disk() 53
5.5 Αρχεία καταγραφής κλήσεων 54
5.6 Προβλήματα υλοποίησης 54

5.1 Εισαγωγή

Για την παρούσα διπλωματική εργασία αναπτύξαμε δύο φόρτους εργασίας με διαφορετικές πολυπλοκότητες, για να ελέγξουμε εάν μια βάση δεδομένων παρέχει δύο από τις τέσσερις ιδιότητες ACID. Συγκεκριμένα, την ατομικότητα (atomicity) και την ανεκτικότητα (durability). Όπως αναφέραμε και σε προηγούμενο κεφάλαιο, ατομικότητα σημαίνει ότι μια συναλλαγή θα εκτελεστεί και θα γίνει commit ολόκληρη και δεν θα χαθούν δεδομένα. Διαφορετικά, αν υπάρξει κάποιο σφάλμα θα πρέπει να κάνει rollback και η βάση δεδομένων SQLite πρέπει να επιστρέψει στην τελευταία κατάσταση που βρισκόταν πριν το σφάλμα. Δηλαδή, αν ένα μέρος της συναλλαγής αποτύχει, ολόκληρη η συναλλαγή αποτυγχάνει. Από την άλλη, ανθεκτικότητα σημαίνει ότι, μόλις δεσμευτεί μια συναλλαγή, θα παραμείνει στο σύστημα ακόμα και αν υπάρξει κατάρρευση του συστήματος αμέσως μετά τη συναλλαγή.

Κάθε φόρτος εργασίας τονίζει μία ή περισσότερες πτυχές των βάσεων δεδομένων, συμπεριλαμβανομένων των μεγάλων συναλλαγών και της συνέπειας πολλαπλών συναλλαγών. Κάθε φόρτος εργασίας έχει ιδιότητες αυτοελέγχου (π.χ. γνωστές τιμές και

παραγγελίες για εγγραφές) που τις χρησιμοποιεί για τον έλεγχο των ιδιοτήτων ACID. Πιο αναλυτική περιγραφή για την υλοποίηση των δύο φόρτων εργασίας υπάρχει στη συνέχεια. Επίσης, αξίζει να αναφερθεί ότι για την υλοποίηση των φόρτων εργασίας έχει συμβάλει σε μεγάλο βαθμό η Ιωάννα Γεωργίου, από μια προηγούμενη μελέτη που έχει γίνει για την ευρωστία σε βάσεις δεδομένων μετά από διακοπές τροφοδοσίας.

5.2 Φόρτος Εργασίας 1

Ο φόρτος εργασίας 1 εκτελεί μια συναλλαγή μεγέθους txn, το οποίο θα πρέπει να ορίσουμε εμείς. Επίσης, η εκτέλεση του (φόρτου εργασίας) χρησιμοποιά μόνο ένα νήμα. Χρησιμοποιούμε αυτόν τον φόρτο εργασίας για να δούμε αν οι μεγάλες συναλλαγές (π.χ. μεγαλύτερες από ένα μπλοκ) προκαλούν σφάλματα. Με τις δύο χρονικές σημάνσεις πριν από τη δέσμευση και μετά τη δέσμευση καταγράφουμε τα χρονικά όρια της δέσμευσης. Ο ελεγκτής για αυτόν τον φόρτο εργασίας είναι απλός: εάν το σφάλμα ήταν μετά τη δέσμευση, βεβαιωνόμαστε ότι υπάρχουν όλες οι σειρές μεγέθους txn. Διαφορετικά, βεβαιωνόμαστε ότι δεν υπάρχει καμία από τις σειρές. Εάν υπάρχουν μόνο μερικές σειρές, αυτό αποτελεί παραβίαση ατομικότητας. Για ένα φόρτο εργασίας με ένα νήμα, η απομόνωση δεν προκαλεί ανησυχία. Για έλεγχο συνέπειας, επικυρώνουμε ότι μια σάρωση εύρους δίνει το ίδιο αποτέλεσμα με το ρητό αίτημα κάθε κλειδιού ξεχωριστά σε ερωτήματα σημείων. Επιπλέον, κάθε ανακτήσιμο ζεύγος κλειδιού-τιμής πρέπει να ταιριάζει. Δηλαδή, το κλειδί N πρέπει πάντα να σχετίζεται με την τιμή N.

Εάν δούμε πιο αναλυτικά το φόρτο εργασίας 1, βλέπουμε ότι αρχικά κάνουμε εισαγωγή της βιβλιοθήκης της βάσης δεδομένων SQLite. Η εισαγωγή γίνεται με την πιο κάτω γραμμή μέσα στον κώδικα.

sqlite.SetSqliteLibrary(os.path.join(base_path, "third-party/sqlite-src-3071602 / build/.libs/libsqlite3.so"))

Αξίζει να σημειωθεί ότι η συνάρτηση SetSqliteLibrary() που χρησιμοποιείται πιο πάνω είναι υλοποιημένη σε ένα ξεχωριστό αρχείο python που υλοποιήσαμε μαζί με τον επιβλέπων καθηγητή μου. Περισσότερες λεπτομέρειες για το αρχείο sqlite.py αναφέρονται πιο κάτω.

Ακολούθως, μέσα στον φόρτο εργασίας 1 υλοποιήσαμε δύο κύρια queries. Το πρώτο ονομάζεται init_query και έχει ως σκοπό να δημιουργήσει τους απαιτητούς πίνακες τους οποίους θα χρησιμοποιά στην συνέχεια το άλλο query. Συγκεκριμένα, θα διαγραφτούν πρώτα οι πίνακες της προηγούμενης εκτέλεσης του φόρτου εργασίας 1 και μετά θα ξαναδημιουργηθούν οι καινούριοι. Σημαντικό είναι να αναφέρουμε ότι πριν την εκτέλεση του init_query θα πρέπει να αρχικοποιήσουμε μια σύνδεση (connection) με τη βάση δεδομένων SQLite. Ακολουθεί η υλοποίηση του δεύτερου query (σχήμα 5.1), του workload_query, η οποία αποτελεί και τη συναλλαγή στην οποία θέλουμε να εισάξουμε τις διακοπές τροφοδοσίας.

```
workload_query = """
begin transaction;
INSERT INTO kvpairs
select "k-%05d-" || cast (n as text ),"v-"||cast (n as text )
from numbers join forloop on (numbers.n >= forloop.startrange and numbers.n < endrange);
create table if not exists  beforetimestamp(beforecommit);
insert into beforetimestamp (beforecommit)
values (time('now'));
select * from beforetimestamp;
end transaction;
--DROP TABLE IF EXISTS aftertimestamp;
create table  if not exists aftertimestamp(aftercommit time);
insert into aftertimestamp (aftercommit)
values (time('now'));
select * from aftertimestamp;
"""
```

Σχήμα 5.1: Workload query 1

Το πιο πάνω σχήμα 5.1 παρουσιάζει το workload_query μέσα στο φόρτο εργασίας 1. Αρχικά δηλώνει ότι θα δημιουργήσουμε μία συναλλαγή (begin transaction , end transaction). Μία συναλλαγή είναι μία ομάδα από SQL εντολές που εκτελούνται μαζί ως ένα ενιαίο τμήμα των εργασιών που πρέπει να υλοποιηθούν. Αν όλες αυτές οι εντολές εκτελεστούν με επιτυχία, τότε η πράξη έχει γίνει commit, δηλαδή οι αλλαγές που γίνονται στα δεδομένα είναι οριστικές. Εάν οποιαδήποτε από τις εντολές μέσα στην συναλλαγή αποτύχει, τότε ολόκληρη η συναλλαγή ακυρώνεται, και στην SQL η συναλλαγή θα πρέπει να αναιρεθεί όλη.

Ακολούθως με την INSERT INTO στην γραμμή 3, εισάγουμε στον πίνακα KVPAIRS δεδομένα στις 2 στήλες. Με την εντολή join, την οποία χρησιμοποιούμε σε αυτό τον κώδικα, μπορούμε να επιλέξουμε δεδομένα που προέρχονται από δύο ή περισσότερους πίνακες. Θα πρέπει οι πίνακες αυτοί να σχετίζονται μεταξύ τους μέσω κάποιας συσχέτισης των πεδίων τους.

Στη συνέχεια, με την γραμμή 4, δηλαδή «select "k-%05d-" || cast (n as text),"v-"||cast (n as text) from numbers join forloop on (numbers.n >= forloop.startrange and numbers.n < endrange);» , θα εισαχθεί στην 1η στήλη του πίνακα KVPAIRS το γράμμα k μετά μία παύλα (κ-) και ακολούθως σε 5 θέσεις θα τυπωθεί κάποιο νούμερο το οποίο είναι μεγαλύτερο ή ίσο από το startrange του πίνακα forloop και μικρότερο από το endrange του πίνακα forloop και αυτό το νούμερο υπάρχει στον πίνακα numbers (ο πίνακας numbers περιέχει τους αριθμούς από το 1 μέχρι το 100000). Το ίδιο ακριβώς θα γίνει και με την 2η στήλη που αντί “κ-“ θα έχει “v-“ . Άρα εάν το startrange=0 και το endrange=100000 τότε η πρώτη εγγραφή του πίνακα kvpairs θα είναι k-00000,v-00000 και η τελευταία θα είναι k-99999,v-99999 .

Ακολούθως, στην γραμμή 6 δημιουργούμε τον πίνακα beforetimestamp εάν δεν είναι ήδη δημιουργημένος . Αυτός ο πίνακας θα περιέχει μία στήλη η οποία θα ονομάζεται beforecommit. Με την γραμμές 7 και 8 θα τοποθετήσουμε σε αυτή την στήλη την ώρα που εκτελέστηκε η συγκεκριμένη εντολή (time(now)). Τέλος, στην γραμμή 9 τερματίζουμε την συναλλαγή. Αφού τερματίσουμε την συναλλαγή με την γραμμή 10, θα δημιουργήσουμε με την γραμμή 12 τον πίνακα aftertimestamp εάν αυτός δεν υπάρχει. Ο πίνακας αυτός περιέχει την στήλη aftercommit και σε εκείνη την στήλη θα τοποθετήσουμε με τις εντολές 13 και 14 την ώρα που εκτελείται η συγκεκριμένη εντολή (η 12) . Άρα αυτή η ώρα θα απεικονίζει την ώρα ακριβώς μετά το τέλος της συναλλαγής.

Όσον αφορά την εκτέλεση του workload_query, αποφάσισα να το εκτελέσω 4 φορές σε κάθε εκτέλεση του φόρτου εργασίας 1 για καλύτερη σύγκριση αποτελεσμάτων. Έτσι θα μπορούμε να διακόπτουμε την τροφοδοσία σε περισσότερα σημεία και σε σημεία πριν η μετά τα workload_query του συγκεκριμένου πειράματος. Επίσης, θα μπορούμε να βλέπουμε πόσες από τις 4 εκτελέσεις έχουν γίνει commit κανονικά και πόσες όχι. Ο αριθμός των εκτελέσεων αποφασίστηκε να είναι τέσσερα, όμως μπορεί ανά πάσα στιγμή

να αλλάξει, αφού είναι αποθηκευμένος σε μια μεταβλητή (n). Οι τέσσερις εκτελέσεις πραγματοποιούνται μέσα σε ένα for loop και κάθε φορά, μετά την ολοκλήρωση του κάθε workload_query, γράφεται στο αρχείο progress_report, ο αριθμός της συναλλαγής που ολοκληρώθηκε. Αυτό είναι πολύ σημαντικό γιατί έτσι μπορούμε να ελέγξουμε την ανθεκτικότητα της SQLite, μετά από μια διακοπή τροφοδοσίας.

5.3 Φόρτος Εργασίας 2

Ο φόρτος εργασίας 2 βασίζεται στο παράδειγμα που αναφέραμε στην εισαγωγή της διπλωματικής εργασίας, το παράδειγμα με την τραπεζική συναλλαγή. Δηλαδή, φανταστείτε μια εφαρμογή που μεταφέρει χρήματα από ένα λογαριασμό σε ένα άλλο. Τι θα γίνει εάν την στιγμή που μεταφέρονται τα χρήματα υπάρξει μια διακοπή τροφοδοσίας στο σύστημα;

Αυτός ο φόρτος εργασίας δοκιμάζει τη συνεκτικότητα πολλαπλών σειρών με ένα νήμα προσομοιώνοντας ταυτόχρονες, μη επικαλυπτόμενες τραπεζικές συναλλαγές. Εκτελεί διαδοχικές συναλλαγές, καθεμία από τις οποίες μεταφέρει χρήματα μεταξύ διαφορετικού συνόλου λογαριασμών μεγέθους txn. Κάθε λογαριασμός ξεκινά με κάποια χρήματα και το ήμισυ των χρημάτων σε κάθε ζυγό λογαριασμό μετακινείται στον επόμενο λογαριασμό με υψηλότερη αρίθμηση (δηλαδή, τα μισά από τα χρήματα του kt-2i μεταφέρονται στο kt- (2i + 1) όπου t είναι το αναγνωριστικό συναλλαγής).

Όπως και με τον φόρτο εργασίας 1, ελέγχουμε ότι κάθε συναλλαγή είναι όλα-ή-τίποτα, ότι οι συναλλαγές που έχουν πραγματοποιηθεί πριν από το σφάλμα υπάρχουν. Επιπλέον, δεδομένου ότι καμία από τις συναλλαγές δεν αλλάζει το συνολικό χρηματικό ποσό, ο ελεγκτής δοκιμάζει κάθε ζεύγος σειρών με κλειδιά της φόρμας kt-2i, kt-(2i+1) για να δει αν τα ποσά τους ανέρχονται στην ίδια τιμή όπως στην αρχική κατάσταση.

Ακολουθεί η υλοποίηση σε SQL του φόρτου εργασίας 2. Για αυτή την υλοποίηση χρησιμοποιήθηκε και ο πίνακας forloop ο οποίος δημιουργεί αριθμούς μέσα σε συγκεκριμένο εύρος.

```
workload_query = """
begin transaction;
INSERT INTO tmp1
select  "k-%02d-" || cast (n as text) || "+%02d-" ,"k-%01d-"||  cast (n as text) || "+%02d - "
|| "k-%01d-" || cast (n/2 as text) || "+%02d"
from numbers join forloop on (numbers.n >= forloop.startrange and numbers.n <forloop.endrange);
INSERT INTO tmp2
select  "k-%02d-" || cast (n as text) || "+%02d-" ,"k-%01d-"||  cast (n as text) || "+%02d - "
|| "k-%01d-" || cast (n/2 as text) || "+%02d"
from numbers join forloop on (numbers.n >= forloop.startrange and numbers.n <forloop.endrange);
DROP TABLE IF EXISTS beforetimestamp;
create table if not exists  beforetimestamp(beforecommit);
insert into beforetimestamp (beforecommit)
values (time('now'));
select * from beforetimestamp;
end transaction;
DROP TABLE IF EXISTS aftertimestamp;
create table  if not exists aftertimestamp(aftercommit time);
insert into aftertimestamp (aftercommit)
values (time('now'));
select * from aftertimestamp;
"""
```

Σχήμα 5.2: Workload query 1

Στον πιο πάνω κώδικα έχουμε δημιουργήσει μία συναλλαγή (transaction). Μέσα σε αυτή την συναλλαγή δημιουργούμε 2 πίνακες, τον πίνακα tmp1 (γραμμή 2) και τον πίνακα tmp2 (γραμμή 4). Η γραμμή cast (n as text), θα μετατρέψει τον αριθμό n από ακέραιο σε συμβολοσειρά, ούτως έτσι να μπορεί να ενωθεί με την ολοκληρωμένη συμβολοσειρά. Και πάλι με την εντολή JOIN ενώνουμε 2 πίνακες, τον πίνακα numbers και forloop με την εξής σχέση: Θέλουμε όλους τους αριθμούς από την στήλη n του πίνακα numbers όπου να είναι μεγαλύτεροι ή ίσοι από το start range και μικρότεροι από το end range του πίνακα forloop. Άρα δηλαδή με αυτό τον τρόπο ορίζουμε το κάτω και πάνω όριο το εύρος τον αριθμών που θα θέλαμε.

Με την γραμμή 3 τυπώνουμε το “κ-“ και δίπλα σε 2 θέσεις τον αριθμό που θα παίρνουμε σαν παράμετρο όταν καλείτε αυτή η συνάρτηση (δηλαδή όλος ο πιο πάνω κώδικας είναι ένα function, και όταν καλείτε από την python, τότε η πρώτη παράμετρος που θα δοθεί θα αντικαταστήσει την παράμετρο d), μετά θα αποθηκευτεί ένα “-” ακολούθως το νούμερο n, έτσι όπως έχει εξηγηθεί πιο πάνω. Και τέλος για το περιεχόμενο αυτής της στήλης έχουμε "+%02d-", όπου σημαίνει πως δίπλα θα υπάρξει το «+» και σε 2 θέσεις θα τυπωθεί η δεύτερη παράμετρος που δόθηκε για το κάλεσμα αυτής της συνάρτησης. Για την 2η στήλη του πίνακα tmp1 κάνει ακριβώς το ίδιο πράγμα με προηγουμένως. Άρα

αν υποθέσουμε ότι αυτό που έχουμε σχηματίσει προηγουμένως είναι η μεταβλητή tmp1 τότε σχηματίζουμε ξανά το tmp1. Ακολούθως, ενώνουμε με ένα – το tmp1/2, δηλαδή αντί για cast(n as text) θα έχουμε το cast(n/2 as text).

Η γραμμή 5 κάνει ακριβώς ότι κάνει και η γραμμή 3 όμως παίρνει σαν παράμετρο από το κύριο πρόγραμμα της python (δηλαδή αυτό που θα αντικαταστήσει την μεταβλητή d) το 2*i+1. Το i είναι μία μεταβλητή του κύριου μας προγράμματος που δείχνει σε ποια εγγραφή της συγκεκριμένης συναλλαγής είμαστε .

Ακολούθως, στην γραμμή 6 εάν ο πίνακας beforetimestamp δεν υπάρχει τότε τον δημιουργεί, εάν υπάρχει όμως δεν κάνει τίποτα. Βάζει στην στήλη beforecommit την ακριβή ώρα που εκτελείται η συγκεκριμένη εντολή. Ακολούθως τερματίζουμε την συναλλαγή (END TRANSACTION). Άρα έχουμε την χρονική στιγμή πριν την ολοκλήρωση της συγκεκριμένης συναλλαγής με την στήλη beforecommit του πίνακα beforetimestamp. Ακολούθως, αφού έχουμε τελειώσει με την συναλλαγή δημιουργούμε τον πίνακα aftertimestamp εάν δεν υπάρχει και βάζουμε στην στήλη aftercommit την ακριβή ώρα που εκτελείτε αυτή η εντολή. Άρα τώρα έχουμε και την ακριβή χρονική στιγμή μετά την ολοκλήρωση της συναλλαγής.

5.4 Η χρησιμότητα του flush() και του fsync_disk()

Το flush() και το fsync_disk() είναι δύο συναρτήσεις οι οποίες χρησιμοποιούνται μέσα στους δύο φόρτους εργασίας και μας βοηθούν κυρίως στο συγχρονισμό των δεδομένων για να έχουμε αξιοπιστία στα αποτελέσματά μας. Με τη συνάρτηση flush() εξασφαλίζουμε ότι τα δεδομένα θα γραφτούν μέσα στο αρχείο. Αυτό γίνεται επειδή εάν γίνει διακοπή τροφοδοσίας δεν θα ξέρουμε σίγουρα εάν τα δεδομένα γράφτηκαν ή όχι στο αρχείο. Επομένως με το flush() λέμε στον buffer του αρχείου να γράψει ότι έχει στο buffer αναγκαστικά.

Επίσης εχτός από το flush(), γίνεται χρήση και της συνάρτησης fsync_disk() έτσι ώστε να είμαστε σίγουροι ότι η τελευταία έκδοση του αρχείου είναι γραμμένη στο δίσκο, επειδή όπως το αρχείο μας έχει buffer, έτσι και το λειτουργικό σύστημα έχει δικό του buffer. Ο λόγος που χρησιμοποιούμε τις δύο αυτές συναρτήσεις είναι για να

εξασφαλίσουμε ότι τα πειράματα μας δεν επηρεάζονται από καθυστερήσεις στο γράψιμο των δεδομένων πάνω στο δίσκο. Έτσι, τα αποτελέσματα μας θα είναι εγγυημένα αυτά που θα συλλέξουμε και θα μπορούμε να εξακριβώσουμε αν όντως έχει εντοπιστεί κάποιο σφάλμα ή όχι από μια διακοπή τροφοδοσίας.

5.5 Αρχεία καταγραφής κλήσεων

Τα αρχεία στα οποία αναφέρομαι είναι τα αρχεία που έχω εισάξει μέσα στην βιβλιοθήκη που έχω δημιουργήσει. Με τα συγκεκριμένα αρχεία, μπορούμε να καταγράφουμε με ακρίβεια τον αριθμό των κλήσεων μιας συγκεκριμένης συνάρτησης που έχω υλοποιήσει (πχ unlink). Ο λόγος που χρειάζεται να καταγράφονται η κλήσεις της unlink(), αλλά και των άλλων συναρτήσεων, είναι επειδή με αυτό τον τρόπο θα μπορούμε να προσθέσουμε μια διακοπή τροφοδοσίας σε ένα συγκεκριμένο σημείο, όπως θα εξηγηθεί και στο *Κεφάλαιο 5*. Για παράδειγμα, θέλουμε να μελετήσουμε την συμπεριφορά της SQLite όταν γίνει διακοπή τροφοδοσίας μετά το 4^{ο} unlink στο commit πρωτόκολλο της συναλλαγής. Αυτό θα γίνει με τη βοήθεια του αρχείου unlink.txt που θα χρησιμοποιηθεί μέσα στην συνάρτηση unlink(). Δηλαδή, όταν η συνάρτηση unlink() διαβάζει από το unlink.txt ότι έχουμε φτάσει στο 4^{ο} commit, θα ειδοποιήσει την εντολή που προκαλεί διακοπή τροφοδοσίας. Αντίστοιχα, σε πειράματα που αφορούν τη μελέτη των συναρτήσεων fsync() και fdatasync() θα χρησιμοποιηθούν τα αρχεία fsync.txt και fdatasync.txt.

5.6 Προβλήματα υλοποίησης

Σε γενικές γραμμές μπορούμε να πούμε ότι δεν αντιμετωπίσαμε ιδιαίτερα προβλήματα στη συγκεκριμένη διπλωματική εργασία. Τόσο όσον αφορά την κατασκευή/υλοποίηση των φόρτων εργασίας και σχετικών αρχείων, όσο και τεχνικά προβλήματα.

Χρειάστηκε να αντιμετωπίσουμε ένα πρόβλημα που είχε σχέση με το mount, όπως φαίνεται στο σχήμα 4 πιο κάτω.

Σχήμα 4: Κομμάτι από το τεχνικό πρόβλημα στο terminal

Απ' ότι φαίνεται το πιο πάνω πρόβλημα είχε να κάνει με το ότι κάτι δεν πήγαινε σωστά στην διαδικασία του mount στη συσκευή εξωτερικής αποθήκευσης (USB Drive). Ενώ το mounting point φαινόταν να ήταν σωστό και mounted, για κάποιο λόγο δεν μας άφηνε να δημιουργήσουμε οποιοδήποτε αρχείο μέσα στο mounting path, δηλαδή το /home/pi/PMTest/MntPt. Όπως βλέπετε και στο σχήμα 4, στην τελευταία γραμμή τυπώνεται το "Permission denied".

Τελικά για να λύσουμε το πρόβλημα χρησιμοποιήσαμε την εντολή sudo mount /dev/sda /home/pi/PMTest/MntPt -o uid=pi,gid=pi στην οποία καθορίζουμε χειροκίνητα το ownership στο pi για να έχουμε πρόσβαση. Τη συγκεκριμένη εντολή την προσθέσαμε στο ptf_check.csh και στο ptf_run.csh όπου γίνεται mount.

Επίσης, ένα κατασκευαστικό πρόβλημα που εμφανίστηκε, έχει να κάνει με την υλοποίηση των φόρτων εργασίας. Αν και αυτό το πρόβλημα δεν μας απότρεψε από την εκτέλεση των πειραμάτων και τη συλλογή αποτελεσμάτων, σίγουρα θα μας διευκόλυνε αν είχε προσεγγιστεί με διαφορετικό τρόπο.

Συγκεκριμένα, αναφέρομαι στην γλώσσα που επιλέχτηκε για να γραφτούν οι φόρτοι εργασίας, την Python. Παρ' όλο που η Python είναι μια γλώσσα υψηλού επιπέδου και μας διευκολύνει με τις βιβλιοθήκες που μας παρέχει (πχ ctypes), στην περίπτωση μας θα ήταν καλύτερο να χρησιμοποιηθεί η C. Ο λόγος είναι επειδή η βιβλιοθήκη της βάσης δεδομένων SQLite είναι γραμμένη σε C και θα μπορούσαμε να εισάγουμε πιο εύκολα κώδικα αν ήταν γραμμένος στην ίδια γλώσσα. Αντίθετα, χρειάζεται να χρησιμοποιούμε

την βιβλιοθήκη ctypes της Python για να έχουμε πρόσβαση στις συναρτήσεις της βιβλιοθήκης της SQLite.

Επίσης, η υλοποίηση των φόρτων εργασίας σε γλώσσα Python, μας περιόριζε στη χρήση του εργαλείου back trace. Συγκεκριμένα, με τη χρήση της συνάρτησης backtrace() που μας παρέχει η C, μπορούμε να δούμε αναλυτικά ποιες συναρτήσεις έχουν καλεστεί με σειρά κατά της διάρκεια εκτέλεσης ενός εκτελέσιμου αρχείου. Με αυτό τον τρόπο θα μπορούσαμε να είχαμε μια καλύτερη οπτική για τη σειρά που εκτελούνται οι συναρτήσεις μέσα στο φόρτο εργασίας, αλλά και ποιες συναρτήσεις χρησιμοποιούνται από το commit πρωτόκολλο της SQLite. Έχω δοκιμάσει να ενσωματώσω, παρόμοια εργαλεία στον κώδικα μου, αλλά λόγω της πολυπλοκότητας που υπάρχει στα διάφορα script, δεν κατέστη δυνατή η χρήση κάποιου εργαλείου.

Ο λόγος που δεν υλοποίησα τους φόρτους εργασίας στην γλώσσα C, είναι επειδή ήταν ήδη υλοποιημένοι σε μεγάλο βαθμό από την Ιωάννα Γεωργίου όταν έκανε τη δικιά της μελέτη που έχει άμεση σχέση με τη συγκεκριμένη διπλωματική εργασία. Επομένως, λόγω του περιορισμένου χρόνου αποφασίστηκε να κρατήσουμε την αρχική υλοποίηση των φόρτων εργασίας και να χτίσουμε πάνω σε αυτά. Βέβαια, αν είχαμε εντοπίσει από την αρχή το συγκεκριμένο πρόβλημα υλοποίησης, θα γράφαμε τους φόρτους εργασίας σε C. Η συγκεκριμένη παρατήρηση μπορεί να αποτελέσει σημείο αναφοράς για μελλοντικές μελέτες που έχουν να κάνουν με τον εντοπισμό σφαλμάτων σε συστήματα αποθήκευσης και διαχείρισης δεδομένων.

Κεφάλαιο 6

Μέθοδοι διακοπής τροφοδοσίας

6.1 Εισαγωγή 57
6.2 Επεξήγηση killpower.sh 58
6.3 Μέθοδοι προηγούμενης μελέτης 59
6.4 Μέθοδος 1 - Τυχαίες διακοπές..... 61
6.5 Μέθοδος 2 - Διακοπές σε συγκεκριμένες κλήσεις συστήματος 62

6.1 Εισαγωγή

Στην παρούσα διπλωματική εργασία θέλουμε να ελέγξουμε τη συμπεριφορά της βάσης δεδομένων SQLite, μετά από διακοπές τροφοδοσίας. Για να μπορέσουμε να πετύχουμε μια διακοπή τροφοδοσίας ήταν αναγκαία η χρήση ενός testbed. Το συγκεκριμένο κύκλωμα το σχεδίασε ο Terence Kelly το οποίο αξιοποιά τις ακίδες GPIO (General Purpose Input Output) που επιτρέπουν στο λογισμικό να ελέγχει εύκολα ένα εξωτερικό κύκλωμα. Το κύκλωμα διακοπής τροφοδοσίας περιγράφεται αναλυτικά στο *Κεφάλαιο 2,* στο οποίο μπορείτε να δείτε και πως φαίνεται στην πραγματικότητα, μέσα από την εικόνα στο σχήμα 2.2.

Σκοπός του συγκεκριμένου κεφαλαίου είναι η επεξήγηση των μεθόδων που αναπτύχθηκαν για διακοπές τροφοδοσίας, κάνοντας χρήση του πιο πάνω κυκλώματος διακοπής τροφοδοσίας. Για να έχουμε μια πιο ολοκληρωμένη εικόνα/συμπέρασμα γύρω από τη συμπεριφορά της SQLite, μετά από διακοπές τροφοδοσίας, τόσο στο αν παρατηρείται απώλεια δεδομένων, όσο και αν παραβιάζονται οι ιδιότητες ACID (Ατομικότητα, Συνέπεια, Απομόνωση, Ανθεκτικότητα), χρειάστηκε να αναπτυχθούν διάφορες μεθοδολογίες.

Συγκεκριμένα, θα εξηγηθεί η μεθοδολογία της Ιωάννας Γεωργίου, στην δική της μελέτη για διακοπές τροφοδοσίας, καθώς επίσης και οι νέες μεθοδολογίες που χρειάστηκαν να υλοποιηθούν για περαιτέρω μελέτη και εξαγωγή αποτελεσμάτων. Επίσης, θα εξηγηθεί το script το οποίο χρησιμοποιείται για να γίνει η διακοπή τροφοδοσίας και πως αξιοποιά τις ακίδες GPIO.

6.2 Επεξήγηση killpower.sh

Το script killpower.sh είναι υπεύθυνο για την διακοπή τροφοδοσίας αξιοποιώντας της ακίδες GPIO. Οι ακίδες GPIO pins (General Purpose Input Output) επιτρέπουν στο λογισμικό να ελέγχει εύκολα τα εξωτερικά κυκλώματα. Το συγκεκριμένο script το έχει υλοποιήσει ο Terence Kelly, μαζί με το υπόλοιπο testbed για διακοπές τροφοδοσίας σε συστήματα διαχείρισης και αποθήκευσης δεδομένων. Το σχήμα 6.1 πιο κάτω δείχνει πώς ακριβώς επιτυγχάνεται η διακοπή παροχής ρεύματος.

```
1 if [ -f "/var/tmp/kp" ]; then
2  gpio -g write $opin 1
3  gpio -g write $opin 0
4 fi
```

Σχήμα 6.1: Script διακοπής τροφοδοσίας

Το σχήμα 6.1 είναι μέρος κώδικα του αρχείου killpower.sh και είναι γραμμένος σε τύπο shell code στο Raspberry Pi. Με το συγκεκριμένο if statement ελέγχουμε κατά πόσο υπάρχει δημιουργημένο το αρχείο kp που βρίσκεται στην τοποθεσία "/var/tmp". Το συγκεκριμένο αρχείο δημιουργείται κατά την εκτέλεση του φόρτου εργασίας, μετά την εκτέλεση του init_query, έτσι ώστε να μην γίνει διακοπή τροφοδοσίας σε συναρτήσεις που αφορούν το συγκεκριμένο query. Ο λόγος που χρησιμοποιείται και το init_query είναι για να αρχικοποιηθούν οι πίνακες μέσα στη βάση δεδομένων για να μπορέσουμε στη συνέχεια να εισάξουμε τα δεδομένα μέσα σε αυτούς τους πίνακες. Οι συναρτήσεις που καλούνται στο init_query δεν μας αφορούν και δεν θέλουμε να διακόψουμε τη τροφοδοσίας σε εκείνα τα σημεία, καθώς δεν θα μας αποφέρουν κάποιο όφελος στα

αποτελέσματα μας. Όπως βλέπετε με τις εντολές 2 και 3 αξιοποιούμε τις ακίδες GPIO για να διακόψουμε την τροφοδοσία, όταν το if statement είναι true.

6.3 Μέθοδοι προηγούμενης μελέτης

Η Ιωάννα Γεωργίου έχει αναπτύξει δύο μεθοδολογίες. Η πρώτη μέθοδος πραγματοποιεί τυχαίες διακοπές τροφοδοσίας με βάση τυχαίους αριθμούς που θα δημιουργηθούν. Ο κώδικας πιο κάτω (σχήμα 6.2) δείχνει με λεπτομέρεια την υλοποίηση της μεθόδου.

```
1  python /home/pi/workspace/project/pinap/workload2/workload.py &
2  sleep `seq -f "%f" 0.8345 0.0125 2.2 | shuf | head -1`
3  set opin = 21
4  gpio -g mode $opin output
5  while (1)
6    gpio -g write $opin 1
7    sleep 5
8    gpio -g write $opin 0
9    sleep 55
10 end
```

Σχήμα 6.2: Μέθοδος τυχαίων διακοπών

Στην 1η γραμμή του πιο πάνω κώδικα βλέπουμε πως καλείται το πρόγραμμα workload το οποίο είναι γραμμένο στην python και μέσα περιέχει τον κώδικα σε SQL για τον φόρτο εργασίας 1. Στην 2η γραμμή βλέπουμε ότι θα γίνει sleep για τυχαίο χρονικό διάστημα. Αυτό θα γίνει αρχικά με την εντολή seq η οποία θα δημιουργήσει αριθμούς από το 0.8345 μέχρι το 2.2 (αυτοί οι αριθμοί επιλέχθηκαν μετά από αρκετές δοκιμές της Ιωάννας Γεωργίου, γιατί στόχος ήταν η διακοπή τροφοδοσίας να γίνει κατά την διάρκεια της συναλλαγής, όχι πριν αρχή της συναλλαγής, καθώς ούτε και μετά τον τερματισμό της συναλλαγής) και θα αυξάνει κατά 0.0125 κάθε φορά, για κάθε αριθμό που δημιουργεί. Ακολούθως, αυτοί οι αριθμοί θα ανακατευτούν με την εντολή shuf, δηλαδή θα τυχαιοποιηθούν, και θα επιλεχθεί ο πρώτος αριθμός από αυτούς (εντολή head-1) για να γίνει το sleep για αυτό το χρονικό διάστημα. Από την 3η γραμμή μέχρι την 9η βλέπουμε πως θα γίνει η διακοπή τροφοδοσίας με την βοήθεια των GPIOs.

Η συγκεκριμένη μεθοδολογία, παρόλο που προσπάθησε να εστιάσει στην χρονική στιγμή που γινόταν η συναλλαγή και το commit πρωτόκολλο, δεν κατάφερε να εντοπίσει κάποιο πρόβλημα. Αυτό πιθανόν να οφείλεται στο ότι υπάρχουν άπειρες χρονικές στιγμές που μπορεί να γίνει διακοπή τροφοδοσίας και είναι δύσκολο να πετύχουμε απώλεια δεδομένων στο συγκεκριμένο σημείο που πιστεύουμε ότι μπορεί να προκληθεί σφάλμα. Γι' αυτό το λόγο πρέπει να γίνεται διακοπή σε συγκεκριμένο σημείο, με βάση το πρωτόκολλο.

Η δεύτερη μεθοδολογία που ανέπτυξε η Ιωάννα Γεωργίου είχε σκοπό να επιλύσει το πρόβλημα της πρώτης μεθοδολογίας, δηλαδή το ότι υπάρχουν άπειρες χρονικές στιγμές για να γίνει διακοπή τροφοδοσίας. Γι' αυτό το λόγο, αποφάσισε να τροποποιήσει τον πηγαίο κώδικα της βιβλιοθήκης, και να εισάξει μια διακοπή τροφοδοσίας πάνω σε μια μέθοδο που υπήρχε ήδη υλοποιημένη από την SQLite. Συγκεκριμένα, αναφέρομαι στη συνάρτηση sqlite3OsDelete(), η οποία βρίσκεται μέσα στο αρχείο pager.c του πηγαίου κώδικα της SQLite. Η συγκεκριμένη συνάρτηση σε κάποια φάση της υλοποίησης της ήταν γνωστό ότι καλούσε την συνάρτηση unlink(), από την standard library της γλώσσας προγραμματισμού C. Επομένως, η Γεωργίου τοποθέτησε την διακοπή τροφοδοσίας στο τέλος της συνάρτησης sqlite3OsDelete() για να επηρεάσει το πρωτόκολλο commit της SQLite.

Η ιδέα ήταν ότι, όταν θα εκτελεστεί η συναλλαγή στους φόρτους εργασίας και θα γίνει trigger το commit πρωτόκολλο, θα καλεστεί και η συνάρτηση unlink() που χρησιμοποιείται από το πρωτόκολλο. Ταυτόχρονα, θα καλεστεί και η συνάρτηση sqlite3OsDelete() και θα διακόψει την τροφοδοσία στο συγκεκριμένο σημείο. Αν και η συγκεκριμένη μεθοδολογία έχει στοχεύσει σε πιο πιθανά σημεία ύπαρξης σφάλματος, το πρόβλημα είναι ότι η συνάρτηση unlink() δεν καλείται μόνο από το commit πρωτόκολλο, αλλά καλείται και σε άλλα σημεία του φόρτου εργασίας. Για παράδειγμα, όταν θα γίνει drop ένας πίνακας της βάσης δεδομένων. Επομένως, δεν ξέρουμε σε πιο ακριβώς unlink() γίνεται trigger η διακοπή τροφοδοσίας και άρα δεν έχουν αξιόπιστα αποτελέσματα για το αν έχουν χαθεί δεδομένα ή όχι.

Γι' αυτό το λόγο, ήταν αναγκαία η υλοποίηση διαφορετικών, πιο εξειδικευμένων μεθοδολογιών για να μπορούμε να διακόψουμε την τροφοδοσία ακριβώς στο σημείο που

θέλουμε. Κάτι το οποίο έγινε στη συγκεκριμένη διπλωματική εργασία. Έχω αναπτύξει δύο μεθοδολογίες για διακοπή τροφοδοσίας, οι οποίες βασίζονται στην υλοποίηση της βιβλιοθήκης, όπως εξηγήθηκε στο *Κεφάλαιο 3*. Δηλαδή, με τη χρήση μιας δικιάς μας βιβλιοθήκης έχουμε υλοποιήσει συναρτήσεις που έχουν άμεση σχέση με τις συναρτήσεις που χρησιμοποιά το πρωτόκολλο της SQLite. Με αυτό το τρόπο θα μπορούμε να εισάγουμε τις διακοπές τροφοδοσίας στο ακριβές σημείο που επιθυμούμε, κάνοντας έτσι πιο αποτελεσματική την έρευνα για την εντόπιση διαφόρων σφαλμάτων και γενικότερα τη μελέτη ευρωστίας της βάσης δεδομένων SQLite.

6.4 Μέθοδος 1 - Τυχαίες διακοπές σε κλήσεις συστήματος

Η συγκεκριμένη μεθοδολογία που αναπτύχθηκε στην παρούσα διπλωματική εργασία μπορεί αρχικά να φαίνεται ίδια με την μέθοδο της Ιωάννας Γεωργίου στην προηγούμενη μελέτη. Όμως, αν δούμε λίγο πιο αναλυτικά, υπάρχει μεγάλη διαφορά στην υλοποίηση. Όπως αναφέρθηκε και πιο πάνω, έγινε υλοποίηση μιας βιβλιοθήκης με wrapper συναρτήσεις των συναρτήσεων που χρησιμοποιά το commit πρωτόκολλο. Οι τυχαίες διακοπές που γίνονται εδώ, βρίσκονται μέσα στις συναρτήσεις της βιβλιοθήκης που υλοποίησα. Αυτό σημαίνει ότι αν για παράδειγμα γίνει κλήση της συνάρτησης unlink(), θα γίνει διακοπή σε μια unlink() συνάρτηση με βάση μια πιθανότητα. Επομένως, παρόλο που είναι τυχαία η διακοπή, είναι μια τυχαία unlink που θα επιλεχτεί και όχι μια τυχαία χρονική στιγμή κατά τη διάρκεια εκτέλεσης του φόρτου εργασίας. Με αυτό τον τρόπο μπορούμε να επικεντρωθούμε σε όλες τις κλήσεις της unlink. Ο κώδικας πιο κάτω (σχήμα 6.3) δείχνει πιο αναλυτικά την υλοποίηση της μεθοδολογίας.

```
1 srand(time(0));
2 int p = rand()%100 + 1;
3     if (p>50){
4         return (*fptr)(path); // Calling original unlink
5     }
6 system("/home/pi/workspace/project/pinap/workload1_sqlite/killpower.sh");
```

Σχήμα 6.3: Τυχαίες διακοπές στην βιβλιοθήκη

Αρχικά, να αναφέρουμε ότι το πιο πάνω απόσπασμα από κώδικα βρίσκεται μέσα στην συνάρτηση unlink μέσα στην βιβλιοθήκη που υλοποίησα. Με την γραμμή 1

αρχικοποιούμε ένα random number generator (seed) για να έχουμε διαφορετικούς αριθμούς κάθε φορά. Γι' αυτό βάζουμε την χρονική στιγμή της αρχικοποίησης ως seed. Με την γραμμή 2 αναθέτουμε σε μια μεταβλητή τον τυχαίο αριθμό που θα παραχθεί από το 1 μέχρι το 100. Ακολούθως, στην γραμμή 3 γίνεται έλεγχος του αριθμού και αν είναι μεγαλύτερος του 50, τότε δεν θα γίνει διακοπή, αλλά θα επιστρέψουμε τον function pointer που δείχνει στην αρχική συνάρτηση unlink. Αλλιώς, θα γίνει διακοπή τροφοδοσίας στην γραμμή 6 με τη βοήθεια του killpower.sh, όπως εξηγήθηκε προηγουμένως. Δηλαδή, έχει 50% πιθανότητα να γίνει διακοπή τροφοδοσίας όταν καλεστεί μια συνάρτηση unlink. Τα πειράματα και τα αποτελέσματα που συλλέξαμε από τη συγκεκριμένη μεθοδολογία περιγράφονται στο *Κεφάλαιο 7* και *Κεφάλαιο 8* αντίστοιχα.

6.5 Μέθοδος 2 - Διακοπές σε συγκεκριμένες κλήσεις συστήματος

Όπως και στην προηγούμενη μεθοδολογία, μπορεί να φαίνεται η ίδια με αυτή της Ιωάννας Γεωργίου, όπως περιγράφεται στη μελέτη της, αλλά είναι διαφορετική υλοποίηση. Η διαφορά είναι ότι το συγκεκριμένο σημείο που γίνεται η διακοπή τροφοδοσίας εδώ, αναφέρεται σε συγκεκριμένο κάλεσμα μιας συνάρτησης (πχ unlink). Δηλαδή, η μεθοδολογία είναι αρκετά έξυπνη έτσι ώστε να μπορεί να διακόψει την τροφοδοσία στο κάλεσμα που θα της ορίσουμε εμείς. Για παράδειγμα αν θέλω να διακόψω την τροφοδοσία στο 3^{ο} unlink, μπορώ να το ορίσω μέσα στη μεθοδολογία.

Αυτό είναι πολύ σημαντικό για την εκτέλεση των πειραμάτων μας, επειδή μπορούμε να εισάξουμε διακοπή τροφοδοσίας σε σημεία που δεν μπορούσαμε προηγουμένως και σε σημεία που υποψιαζόμαστε ότι μπορεί να προκαλέσουν απώλειες δεδομένων ή σφάλματα. Το μειονέκτημα της μεθοδολογίας στην προηγούμενη μελέτη ήταν ότι μπορούσε να εισάξει διακοπή τροφοδοσίας μόνο μετά από ένα συγκεκριμένο unlink, μετά από μια συγκεκριμένη χρονική στιγμή που ήταν η ίδια κάθε φορά. Ενώ, με αυτή τη μέθοδο, μπορούμε να διακόψουμε την τροφοδοσία πριν ή και μετά οποιασδήποτε κλήσης μιας συνάρτησης από τη βιβλιοθήκη μας.

Πιο κάτω φαίνεται αναλυτικά η υλοποίηση της μεθοδολογίας για διακοπή τροφοδοσίας σε συγκεκριμένο σημείο. Όπως και πριν, το πιο κάτω κομμάτι κώδικα (σχήμα 6.4)

βρίσκεται μέσα σε κάθε συνάρτηση στη βιβλιοθήκη μας, έτσι ώστε να έχουμε πλήρη έλεγχο των διακοπών τροφοδοσίας.

```
1  char c[100];
2  FILE *fp2;
3  if ((fp2 = fopen("/home/pi/workspace/project/pinap/workload1/fsync.txt",
   "r")) == NULL) {
4      printf("Error! File cannot be opened.");
5      exit(1);
6      }
7      // reads text until newline is encountered
8  fscanf(fp2, "%[^\n]", c);
9  fclose(fp2);
10 fp2 = fopen("/home/pi/workspace/project/pinap/workload1/fsync.txt", "a+");
11 fprintf(fp2, "%d", (int)c[strlen(c)-1]-'0'+1);
12 printf("\nMy Fsync is called %s%d\n", c,(int)c[strlen(c)-1]-'0'+1);
13 if((int)c[strlen(c)-1]-'0'+1 != 1){
14     fclose(fp2);
15     return (*fptr)(fd); // Calling original fsync
16     }
17 system("/home/pi/workspace/project/pinap/workload1/killpowerfsync.sh");
18 fclose(fp2);
19 return (*fptr)(fd); // Calling original fsync
```

Σχήμα 6.4: Μέθοδος διακοπής τροφοδοσίας σε συγκεκριμένο σημείο

Οι εντολές 1 και 2 χρησιμοποιούνται για να αρχικοποιήσουμε κάποιες βοηθητικές μεταβλητές, έτσι ώστε να πετύχουμε διακοπή τροφοδοσίας. Συγκεκριμένα, αρχικοποιούμε ένα πίνακα τύπου char μέσα στον οποίο θα αποθηκεύονται όλοι οι χαρακτήρες του αρχείου fsync.txt (βοηθητικό αρχείο κλήσεων, βλ. *Κεφάλαιο 5.5*). Επίσης, αρχικοποιούμε ένα file pointer για να μπορούμε να διαβάσουμε το βοηθητικό αρχείο fsync.txt. Οι εντολές 3 – 6 δείχνουν πως γίνεται το διάβασμα του βοηθητικού αρχείου fsync.txt και πως διαχειρίζονται την περίπτωση όπου δεν βρεθεί το αρχείο. Ακολούθως, αφού μπορούμε να έχουμε πρόσβαση στο αρχείο, θα διαβάσουμε όλο το περιεχόμενο της πρώτης γραμμής του αρχείου, όπως φαίνεται στην γραμμή 7. Στην γραμμή 9, θα ξανά-ανοίξουμε το βοηθητικό αρχείο, αυτή τη φορά σε μορφή ‘α+’, δηλαδή δεν θα γίνουν overide τα δεδομένα που έχει ήδη, αλλά θα προστεθούν (append) στο ήδη υπάρχων αρχείο.

Σκεφτείτε ότι αυτός ο κώδικας θα τρέχει πολλαπλές φορές κατά τη διάρκεια εκτέλεσης ενός φόρτου εργασίας. Η ιδέα είναι ότι θέλουμε να μετρούμε πόσες φορές καλέστηκε η συνάρτηση που ελέγχουμε, η fsync στην συγκεκριμένη περίπτωση. Κάθε φορά θα γίνεται append ο αριθμός της εκτέλεσης της συνάρτησης. Δηλαδή, αν η συνάρτηση unlink καλεστεί 5 φορές κατά της διάρκεια εκτέλεσης του φόρτου εργασίας, αυτό σημαίνει ότι το βοηθητικό αρχείο fsync.txt θα περιέχει την ακολουθία χαρακτήρων '012345'. Το μηδέν δεν υπολογίζεται, αλλά χρειάζεται να υπάρχει για τον λόγο που θα εξηγηθεί πιο κάτω.

Συνεχίζοντας στην γραμμή 10, βλέπουμε πως γίνεται το append του επόμενου χαρακτήρα μέσα στο βοηθητικό αρχείο. Επίσης, θα γίνει +1 αφού μέχρι τώρα η συνάρτηση fsync έχει καλεστεί X φορές. Άρα, τώρα διανύουμε την X+1 fsync. Με αυτή την τεχνική μπορούμε να καταγράφουμε πόσες φορές έχει καλεστεί η συνάρτηση που ελέγχουμε, έτσι ώστε να μπορούμε να διακόψουμε την τροφοδοσία μετά από μια συγκεκριμένη κλήση. Στην γραμμή 11, τυπώνουμε στο terminal ότι ακριβώς έχει καταγραφεί στην γραμμή 10 για να ξέρουμε σε πραγματικό χρόνο σε ποια κλήση της συνάρτησης βρισκόμαστε. Με αυτό τον τρόπο μπορούμε να δούμε καλύτερα το τι γίνεται κατά τη διάρκεια εκτέλεσης του φόρτου εργασίας και να μπορούμε να δούμε ποιες και πόσες συναρτήσεις καλούνται κάθε φορά.

Η γραμμή 12 είναι από τις πιο σημαντικές εντολές της μεθόδου. Με της συγκεκριμένη εντολή ορίζουμε πότε θα γίνει η διακοπή τροφοδοσίας. Δηλαδή, γίνεται έλεγχος του τελευταίου χαρακτήρα μέσα στον πίνακα c μαζί με τον αριθμό που θα ορίζουμε εμείς. Με άλλα λόγια, γίνεται σύγκριση του αριθμού της μέχρι τώρα κλήσης της συνάρτησης + 1 μαζί με ένα αριθμό και αν δεν είναι ίσα τότε δεν θα γίνει διακοπή τροφοδοσίας, αλλιώς θα γίνει. Για καλύτερη κατανόηση της συγκεκριμένης μεθόδου μπορούμε να εξηγήσουμε τον πιο πάνω έλεγχο με ένα παράδειγμα.

Έστω ότι θέλουμε να ελέγξουμε τη συνάρτηση fsync και να κάνουμε διακοπή τροφοδοσίας ακριβώς πριν το 5^{ο} κάλεσμα της. Η πιο πάνω μέθοδος θα τρέξει την πρώτη φορά και το αρχείο fsync.txt θα περιέχει τους χαρακτήρες '01' (μόλις εκτελεστεί η γραμμή 10). Συνεχίζοντας, θα καλεστεί σε μια άλλη χρονική στιγμή η συνάρτηση fsync ξανά. Το αρχείο τώρα θα περιέχει τους χαρακτήρες '012'. Αυτό θα συνεχιστεί μέχρι και

την 4η κλήση της συνάρτησης fsync, όπου το αρχείο θα περιέχει τους χαρακτήρες ‘01234’. Όταν θα γίνει η κλήση της fsync για 5η φορά, θα διαβαστεί το αρχείο fsync.txt στην γραμμή 7 και θα γεμίζει ο πίνακας c με τους χαρακτήρες 0, 1, 2, 3 και 4 (επειδή ακόμη δεν έχει γραφτεί η 5η κλήση της fsync). Επομένως, όταν θα γίνει ο έλεγχος στην γραμμή 12, θα πρέπει να ελέγξουμε το τελευταίο στοιχείο του πίνακα c (το 4) και να προσθέσουμε +1 για να γίνει 5, εφόσον βρισκόμαστε στην 5η κλήση της συνάρτησης fsync. Θα γίνει ο έλεγχος 5 !=5 που είναι false, άρα δεν θα μπει μέσα στο if statement και θα γίνει διακοπή τροφοδοσίας. Σημείωση: η διακοπή γίνεται μεταξύ 4ης και 5ης κλήσης της fsync, αφού η αρχική fsync θα κληθεί κανονικά μετά από την διακοπή τροφοδοσίας στην γραμμή 18.

Η γραμμή 16 δείχνει το σημείο που γίνεται η διακοπή τροφοδοσίας, δηλαδή θα εκτελεστεί το killpowerfsync.sh script. Η γραμμή 18 θα καλέσει τον function pointer που δείχνει στην αρχική unlink, εναλλακτικά ο function pointer θα καλεστεί στην γραμμή 14, στις περιπτώσεις που δεν θα γίνει διακοπή τροφοδοσίας. Κανονικά, η γραμμή 17 και 18 δεν πρέπει να εκτελεστούν ποτέ, απλά λόγο συντακτικού error στη μεταγλώττιση έπρεπε να προστεθούν.

Η πιο πάνω μεθοδολογία είναι πολύ σημαντική για την συγκεκριμένη διπλωματική εργασία. Με αυτή την υλοποίηση μπορούμε να πετύχουμε διακοπές τροφοδοσίας σε συγκεκριμένα σημεία, μεταξύ συναρτήσεων ή πριν και μετά από μια κλήση. Η υλοποίηση της μεθοδολογίας επιτρέπει την εκτέλεση πολλών διαφορετικών πειραμάτων χωρίς να χρειάζεται ιδιαίτερες αλλαγές σε κάθε διαφορετικό πείραμα. Το μόνο που χρειάζεται να αλλάξει κάθε φορά είναι ο αριθμός στην γραμμή 12, που συμβολίζει τον αριθμό της κλήσης της συνάρτησης που θέλουμε να ελέγξουμε/διακόψουμε την τροφοδοσίας (δηλαδή τη διακοπή τροφοδοσίας που θα γίνει πριν από την κλήση της συνάρτησης που θα ορίσουμε).

Κεφάλαιο 7

Πειράματα

7.1 Εισαγωγή 66
7.2 Journal Modes και Journal Files 67
7.3 PRAGMA Synchronous statements 70
7.4 Checkpointing 72
7.5 Πειράματα στους Φόρτους Εργασίας 74
7.5.1 SQLite Version 2013 (έκδοση 3.7.16.2) 74
7.5.2 SQLite Version 2021 (έκδοση 3.35.4) 76
7.6 Παρατηρήσεις/Σχόλια από τα πειράματα στους φόρτους εργασίας 78
7.7 Ισχυρισμοί SQLite 79
7.8 Πειράματα αλλαγής του checkpoint threshold 81
7.9 Πειράματα αφαίρεσης των subqueries στους φόρτους 82

7.1 Εισαγωγή

Μέχρι τώρα έχουν εξηγηθεί όλα τα μέρη της διπλωματικής εργασίας τα οποία έπρεπε να υλοποιηθούν. Δηλαδή, εξηγήσαμε την υλοποίηση του testbed (η οποία έγινε από τον Terence Kelly) μαζί με τα βοηθητικά scripts για να τρέχουμε τους φόρτους εργασίας (rab.csh, ptf_run.csh). Εξηγήσαμε την υλοποίηση της βιβλιοθήκης η οποία μας βοηθά στο να καλούμε τις δικές μας συναρτήσεις αντί των συναρτήσεων που μας παρέχει η standard library της γλώσσας προγραμματισμού C. Αυτό έγινε με τη βοήθεια του function interposition και της εντολής LD_PRELOAD. Επίσης, εξηγήσαμε την υλοποίηση των φόρτων εργασίας, καθώς και των βοηθητικών αρχείων που ήταν αναγκαία για να συλλέξουμε τα αποτελέσματα μας. Συγκεκριμένα, αναπτύχθηκαν 2 φόρτοι εργασίας. Ο πρώτος ελέγχει αν οι μεγάλες συναλλαγές προκαλούν σφάλματα και ο δεύτερος ελέγχει

την συνεκτικότητα πολλαπλών σειρών με ένα νήμα προσομοιώνοντας ταυτόχρονες μη επικαλυπτόμενες τραπεζικές συναλλαγές.

Επιπλέον, εξηγήθηκε η υλοποίηση των μεθόδων διακοπής τροφοδοσίας. Τόσο των μεθόδων που έχουν υλοποιηθεί από την μελέτη της Ιωάννας Γεωργίου, όσο και των μεθόδων που αναπτύχθηκαν στην συγκεκριμένη εργασία. Ακόμη, έχει εξηγηθεί ο τρόπος με τον οποίο γίνεται ο έλεγχος των δεδομένων, μετά από μια διακοπή τροφοδοσίας, καθώς επίσης και των βοηθητικών αρχείων που συμβάλλουν στην εξαγωγή των αποτελεσμάτων.

Σκοπός του συγκεκριμένου κεφαλαίου είναι η περιγραφή των πειραμάτων που έχουμε εκτελέσει μαζί με τον επιβλέπον καθηγητή μου κ. Χάρη Βώλο. Στόχος είναι η αξιολόγηση της ευρωστίας σε συστήματα αποθήκευσης και διαχείρισης δεδομένων έναντι διακοπών τροφοδοσίας σε συγκεκριμένα σημεία τα οποία πιστεύουμε ότι μπορούν να παραβιάσουν τις ιδιότητες ACID. Για την αξιολόγηση του συστήματος αποθήκευσης SQLite επιλέξαμε να ελέγξουμε δύο ιδιότητες ACID, την ατομικότητα (atomicity) και την ανθεκτικότητα (durability). Επομένως, τα πειράματα που θα εκτελέσουμε θα πρέπει να στοχεύουν στην εύρεση οποιονδήποτε σφαλμάτων που μπορεί να προκληθούν από την παραβίαση της ατομικότητας και της ανθεκτικότητας.

Για να το πετύχουμε αυτό θα χρησιμοποιήσουμε κάποιες ιδιότητες και χαρακτηριστικά που μας παρέχει η γλώσσα SQLite τα οποία μπορούν να αλλάξουν τον τρόπο με τον οποίο συμπεριφέρεται και διαχειρίζεται τα δεδομένα η βάση δεδομένων. Συγκεκριμένα, αναφέρομαι στα διάφορα Journal Modes που μπορεί να οριστεί η SQLite για να εκτελεστεί ένα query. Επίσης, αναφέρομαι στα Pragma Statements τα οποία βοηθούν στην αύξηση της αποδοτικότητας και της ασφάλειας στη διακίνηση των δεδομένων και στην τεχνική Checkpointing η οποία αλλάξει την ροή αποθήκευσης (και συγχρονισμού) των δεδομένων. Οι συγκεκριμένες ιδιότητες/μέθοδοι εξηγούνται αναλυτικά στα επόμενα υποκεφάλαια.

7.2 Journal Modes και Journal Files

Τα journal files χρησιμοποιούνται από το SQLite για την επαναφορά συναλλαγών λόγω κάποιου rollback, ή επειδή έχει παρουσιαστεί κάποιο σφάλμα το οποίο δεν είναι

ανακτήσιμο. Κανονικά όμως, απαιτείται ένα journal file για την επεξεργασία οποιασδήποτε εντολής SQL που προκαλεί τροποποίηση σε αρχείο βάσης δεδομένων. Η ενεργή διάρκεια ζωής ενός journal file εκτείνεται από την πρώτη τροποποίηση της βάσης δεδομένων έως ότου γίνει το commit.

Το journal_mode είναι η επιλογή του πως θα ελέγχονται και θα συμπεριφέρονται τα journal files. Εάν έχουμε επιλεγμένο το journal_mode=DELETE,το οποίο είναι και το προεπιλεγμένο , τότε αυτό σημαίνει πως με την ολοκλήρωση μιας συναλλαγής η βάση δεδομένων θα εξαναγκάσει (force) το journal_file να κλείσει και αμέσως μετά να το διαγράψει. Μια ολοκληρωμένη συναλλαγή εγγυάται ότι όλες οι εγγραφές βρίσκονται στο δίσκο. Σε περίπτωση που υπάρξει κάποιο system fail πριν προλάβει η βάση να διαγράψει το αρχείο και να ορίσει τη συναλλαγή ως committed, τότε η συναλλαγή θα θεωρηθεί ως αποτυχημένη. Ως συνέχεια, θα γίνει rollback η (failed) συναλλαγή με την εκτέλεση της επόμενης συναλλαγής.

Η SQLite μας παρέχει 5 διαφορετικά journal modes τα οποία μπορούμε να χρησιμοποιήσουμε κατά τη διάρκεια μιας συναλλαγής. Σημαντική σημείωση είναι ότι η SQLite θυμάται την τελευταία αλλαγή στο journal mode/ Επομένως, όλα τα επόμενα queries θα εκτελούνται με το τελευταίο mode που έχουμε ορίσει και για να το αλλάξουμε θα πρέπει να γράψουμε την εντολή “PRAGMA journal_mode = mode;”, όπου mode είναι το είδος του journal mode. Τα 5 modes είναι:

1. DELETE:
   Όπως αναφέραμε είναι το προεπιλεγμένο mode, δηλαδή όταν τελειώσει η συναλλαγή θα διαγραφτεί το journal file.

2. TRUNCATE
   Το συγκεκριμένο mode περικόπτει (μικραίνει) το μέγεθος του journal file σε μηδέν bytes. Αυτή η μέθοδος χρησιμοποιείται σε πολλά λειτουργικά συστήματα, επειδή είναι πιο γρήγορη από το DELETE.

3. PERSIST

Το συγκεκριμένο mode μηδενίζει τα πρώτα 28 bytes του journal file. Με αυτό τον τρόπο υποδεικνύουμε ότι το header του journal file είναι ακυρωμένο και άρα ταυτόχρονα όλο το journal file είναι ακυρωμένο. Ένα ακυρωμένο journal file δεν μπορεί να γίνει rolled back. Αυτή η μέθοδος επίσης χρησιμοποιείται λόγω του ότι είναι πιο γρήγορη από το DELETE.

4. MEMORY

   Με αυτό το mode το journal file παραμένει στη μνήμη, αντί στο δίσκο. Ο journal file descriptor απλά κλείνει, καταστρέφοντας έτσι το journal ου βρίσκεται στη μνήμη.

5. WAL

   Το Write Ahead Log (WAL) παρέχει περισσότερη συνοχή, καθώς οι readers δεν εμποδίζουν τους writers και ένας writer δεν εμποδίζει τους readers. Η ανάγνωση και η γραφή μπορούν να προχωρήσουν ταυτόχρονα.

Στην παρούσα διπλωματική εργασία έχουμε αποφασίσει να εκτελέσουμε πειράματα σε 2 διαφορετικά modes. Σε DELETE και WAL. Έχουμε επιλέξει το DELETE επειδή είναι το προεπιλεγμένο journal mode και θέλουμε να δούμε πως συμπεριφέρεται η βάση δεδομένων SQLite χωρίς οποιεσδήποτε αλλαγές και τροποποιήσεις στις λειτουργίες της. Επίσης, έχουμε επιλέξει το WAL mode λόγω των πλεονεκτημάτων που παρέχει το συγκεκριμένο mode. Δηλαδή, είναι σημαντικά ταχύτερο στα περισσότερα σενάρια. Η ανάγνωση και η γραφή, όπως αναφέραμε και πιο πάνω, μπορούν να προχωρήσουν ταυτόχρονα. Οι λειτουργίες εισόδου/εξόδου στο δίσκο τείνουν να είναι πιο διαδοχικές χρησιμοποιώντας το WAL. Ακόμη, χρησιμοποιεί πολύ λιγότερες λειτουργίες fsync() και, ως εκ τούτου, είναι λιγότερο ευάλωτη σε προβλήματα σε συστήματα όπου η κλήση της συνάρτησης fsync() παρουσιάσει κάποιο σφάλμα (διακοπή τροφοδοσίας).

Η προεπιλεγμένη μέθοδος στην οποία πραγματοποιείται ένα rollback λειτουργεί γράφοντας ένα αντίγραφο του αρχικού αμετάβλητου περιεχομένου της βάσης σε ένα ξεχωριστό αρχείο καταγραφής rollback και, στη συνέχεια, γράφοντας αλλαγές απευθείας στο αρχείο βάσης δεδομένων. Σε περίπτωση διακοπής λειτουργίας ή κάποιου rollback, το αρχικό περιεχόμενο που περιέχεται στις εγγραφές του αρχείου rollback αναπαράγεται

στο αρχείο βάσης δεδομένων για να επανέλθει το αρχείο της βάσης στην αρχική του κατάσταση. Το commit παρουσιάζεται όταν διαγραφτεί το rollback journal.

Η επιλογή της μεθόδου WAL ως journal mode αντιστρέφει το πιο πάνω. Το αρχικό περιεχόμενο διατηρείται στο αρχείο βάσης δεδομένων και οι αλλαγές γίνονται append σε ένα ξεχωριστό αρχείο WAL. Ένα commit παρουσιάζεται όταν μια ειδική εγγραφή που υποδεικνύει μια δέσμευση προσαρτάται στο WAL αρχείο. Με αυτό τον τρόπο, ένα commit μπορεί να συμβεί χωρίς ποτέ να γράψει στην αρχική βάση δεδομένων, η οποία επιτρέπει στους readers να συνεχίσουν να λειτουργούν από την αρχική αναλλοίωτη βάση δεδομένων, ενώ οι αλλαγές δεσμεύονται ταυτόχρονα στο WAL. Πολλαπλές συναλλαγές μπορούν να προσαρτηθούν στο τέλος ενός μόνο αρχείου WAL.

7.3 PRAGMA Synchronous statements

Στην παρούσα διπλωματική εργασία έχουμε πειραματιστεί και με τα PRAGMA Synchronous statements [9] τα οποία παρέχουν διάφορους τρόπους συγχρονισμού και διαχείρισης των journal files. Επίσης, όπως αναφέρει η SQLite μπορούν να μας υποστηρίξουν σε περίπτωση που υπάρξει κάποια διακοπή τροφοδοσίας. Και εφόσον αυτό είναι και το κύριο θέμα που μας αφορά στην συγκεκριμένη μελέτη θα ήταν καλό να τρέξουμε διάφορα πειράματα με τα διαφορετικά synchronous statements. Η SQLite μας παρέχει 4 διαφορετικά PRAGMA Synchronous statements:

1. FULL

   Όταν το synchronous statement έχει οριστεί σε FULL, ο μηχανισμός βάσης δεδομένων SQLite θα χρησιμοποιήσει τη μέθοδο xSync του VFS (SQLite OS interface) για να διασφαλίσει ότι όλο το περιεχόμενο εγγράφεται με ασφάλεια στην επιφάνεια του δίσκου πριν να προχωρήσει. Αυτό εξασφαλίζει ότι η ύπαρξη κάποιου σφάλματος στο λειτουργικό σύστημα ή μια διακοπή τροφοδοσίας δεν θα καταστρέψει τη βάση δεδομένων. Ο πλήρης συγχρονισμός είναι πολύ ασφαλής, αλλά είναι επίσης και πιο αργός. Γι’ αυτό το λόγο το FULL χρησιμοποιείται κυρίως όταν το journal mode δεν είναι ρυθμισμένο σε WAL.

2. NORMAL

Όταν το synchronous statement έχει οριστεί σε NORMAL, ο μηχανισμός βάσης δεδομένων SQLite θα εξακολουθεί να συγχρονίζεται στις πιο κρίσιμες στιγμές, αλλά λιγότερο συχνά από ό, τι σε λειτουργία FULL. Υπάρχει μια πολύ μικρή (αν και μη μηδενική) πιθανότητα ότι μια διακοπή τροφοδοσίας τη λάθος στιγμή θα μπορούσε να καταστρέψει τη βάση δεδομένων στο journal_mode = DELETE, αλλά αυτό ίσχυε σε παλαιότερες εκδόσεις της SQLite. Ο συνδυασμός του journal_mode = WAL μαζί με το synchronous_statement = NORMAL είναι θεωρητικά πιο ασφαλής από την ύπαρξη μιας καταστροφής. Το ίδιο ισχύει και σε λειτουργία DELETE. Το NORMAL είναι μια καλή επιλογή για τις περισσότερες εφαρμογές που εκτελούνται σε λειτουργία WAL.

3. EXTRA

   Όταν το synchronous statement έχει οριστεί σε EXTRA τότε η λειτουργία του είναι πολύ παρόμοια με το FULL. Η διαφορά είναι ότι η διεύθυνση που περιέχει τις εγγραφές με το rollback συγχρονίζεται μετά την αποσύνδεση της εγγραφής για την ολοκλήρωση μιας συναλλαγής (σε journal_mode = DELETE). Το EXTRA παρέχει πρόσθετη ανθεκτικότητα σε περίπτωση που υπάρξει μια διακοπή τροφοδοσίας ακριβώς μετά το commit.

4. OFF

   Όταν το synchronous statement έχει οριστεί σε OFF, η SQLite συνεχίζει χωρίς να συγχρονίσει τα δεδομένα που έχει παραδώσει στο λειτουργικό σύστημα. Εάν υπάρξει οποιαδήποτε διακοπή τροφοδοσίας ή κάποιο άλλο σφάλμα πριν από την εγγραφή των δεδομένων στο δίσκο, τα δεδομένα θα είναι ασφαλή αλλά η βάση δεδομένων ενδέχεται να καταστραφεί. Παρ’ όλα αυτά το synchronous statement OFF είναι πολύ γρήγορο.

Με βάση τις πιο πάνω περιγραφές, όπως μας τις παραθέτει η SQLite στο documentation της για τα διάφορα PRAGMA synchronous statements καταλαβαίνουμε ότι η χρήση τους μπορεί να επηρεάσει τη λειτουργία των πειραμάτων μας και άρα και των αποτελεσμάτων μας. Γι’ αυτό το λόγο θα προσπαθήσουμε να επικεντρωθούμε στις περιπτώσεις που ισχυρίζεται η βάση δεδομένων SQLite ότι δεν θα παρατηρηθεί κάποια απώλεια δεδομένων.

7.4 Checkpointing

Εκτός από την αλλαγή του journal mode και τον ορισμό διαφόρων Synchronous statements, έχουμε πειραματιστεί και με το Checkpointing. Η μετακίνηση των συναλλαγών από το αρχείο WAL πίσω στη βάση δεδομένων ονομάζεται Checkpointing. Αν και αυτό είναι μια αυτοματοποιημένη διαδικασία, είναι κάτι που θα ήταν καλό να έχουμε υπόψη μας, εφόσον επηρεάζει τη μεταφορά δεδομένων. Δηλαδή, όλες οι αλλαγές που έχουμε κάνει και έχουν γραφτεί στο αρχείο WAL, τελικά, θα πρέπει να μεταφερθούν πίσω στην αρχική βάση δεδομένων. Ένας άλλος τρόπος για να σκεφτείτε τη διαφορά μεταξύ του rollback και του write-ahead-log είναι ότι στην προσέγγιση rollback-journal, υπάρχουν δύο πρωτόγονες λειτουργίες, το reading και το writing, ενώ με ένα αρχείο καταγραφής εγγραφών υπάρχουν τρεις πρωτόγονες λειτουργίες: reading, writing και checkpointing.

Από προεπιλογή, η βάση δεδομένων SQLite κάνει αυτόματα checkpointing όταν το αρχείο WAL φτάσει σε μέγεθος threshold 1000 σελίδων. (Η επιλογή SQLITE_DEFAULT_WAL_AUTOCHECKPOINT κατά τη μεταγλώττιση μπορεί να χρησιμοποιηθεί για τον καθορισμό διαφορετικής προεπιλογής threshold). Οι εφαρμογές που χρησιμοποιούν το WAL ως προεπιλεγμένο journal mode δεν χρειάζεται να κάνουν τίποτα για να συμβεί το checkpointing. Αλλά αν το θέλουν, οι εφαρμογές μπορούν να προσαρμόσουν το όριο του αυτόματου σημείου checkpointing. Εναλλακτικά, μπορούν να το απενεργοποιήσουν και να το εκτελέσουν κατά τη διάρκεια αδρανών στιγμών ή σε ξεχωριστό νήμα ή διεργασία.

Αξίζει να σημειωθεί ότι υπάρχουν 4 διαφορετικά modes στα οποία μπορεί να οριστεί το checkpointing όταν βρίσκεται σε WAL mode. Τα 4 modes που αναφέρονται πιο κάτω μπορούν να καλεστούν με τη χρήση του “schema.wal_checkpoint(mode)” μέσα στο query που θα εκτελέσουμε.

1. PASSIVE

   Με το passive, ελέγχονται όσο το δυνατόν περισσότερα frames χωρίς να περιμένει να ολοκληρωθεί το reading και writing στη βάση δεδομένων. Γίνεται συγχρονισμός του αρχείου της βάσης εάν όλα τα frames στο αρχείο καταγραφής

είναι συνδεδεμένα. Αυτή η λειτουργία είναι η ίδια με την κλήση του interface sqlite3_wal_checkpoint() στη C.

2. FULL

   Με τη λειτουργία FULL γίνεται block ο call-back handler μέχρι να μην υπάρχει κάποιος writer στη βάση. Ακολούθως, κάνει checkpointing όλα τα frames στο log file και τα συγχρονίζει με το αρχείο της βάσης δεδομένων. Επίσης, το FULL αποκλείει ταυτόχρονους writers καθώς τρέχει.

3. RESTART

   Παρόμοια λειτουργία με το FULL, με την προσθήκη ότι μετά το checkpointing του log file γίνεται block (το log file) μέχρι όλοι οι readers να τελειώσουν. Αυτό εξασφαλίζει ότι ο επόμενος που θα γράψει στο αρχείο βάσης δεδομένων θα επανεκκινήσει το αρχείο καταγραφής από την αρχή.

4. TRUNCATE

   Παρόμοια λειτουργία με το RESTART, με την προσθήκη ότι το WAL file θα αποκοπεί σε μηδέν bytes όταν ολοκληρωθεί επιτυχώς η συναλλαγή.

Εκτός από την πιο πάνω αλλαγή στον τρόπο λειτουργίας του checkpointing, μπορούμε να αλλάξουμε και το μέγεθος του threshold, δηλαδή του αριθμού σελίδας που είναι προγραμματισμένη η SQLite να κάνει checkpoint το WAL αρχείο. Όπως αναφέρθηκε και πιο πάνω, το προεπιλεγμένο μέγεθος σελίδας που κάνει checkpoint η SQLite είναι 100 σελίδες. Υπάρχουν 2 τρόποι με τους οποίους μπορούμε να αλλάξουμε το threshold.

1ος τρόπος

Χρησιμοποιώντας το “PRAGMA wal_autocheckpoint=N;” μέσα στο query που θα εκτελέσουμε, όπου N συμβολίζει τον αριθμό σελίδων που θέλουμε να γίνει το checkpointing. Το συγκεκριμένο PRAGMA είναι μια wrapper γύρω από το sqlite3_wal_autocheckpoint() interface στη γλώσσα C. Όλα τα αυτόματα checkpoints είναι σε λειτουργία PASSIVE. Επιπλέον, αν ο αριθμός N οριστεί σε μηδέν ή μια αρνητική τιμή, τότε το wal_autocheckpoint απενεργοποιείται.

2ος τρόπος

Μπορούμε να αλλάξουμε την τιμή του προεπιλεγμένου threshold απευθείας από τον πηγαίο κώδικα που μας παρέχει η βιβλιοθήκη της SQLite. Συγκεκριμένα, όπως φαίνεται στο σχήμα 7, η εντολή που ορίζει το threshold βρίσκεται στην γραμμή 13456 μέσα στο αρχείο sqlite3.c και έχει την τιμή 1000. Μπορούμε να αλλάξουμε την τιμή αυτή αλλά θα πρέπει να ξανακάνουμε build την βιβλιοθήκη της SQLite. Γενικότερα, μετά από κάθε αλλαγή θα πρέπει να γίνεται build η βιβλιοθήκη.

```
13453  ** checkpointing the database in WAL mode.
13454  */
13455: #ifndef SQLITE_DEFAULT_WAL_AUTOCHECKPOINT
13456: # define SQLITE_DEFAULT_WAL_AUTOCHECKPOINT  1000
13457  #endif
```

Σχήμα 7.1: σημείο του κώδικας που ορίζει το checkpointing threshold

7.5 Πειράματα στους Φόρτους Εργασίας

7.5.1 SQLite Version 2013 (έκδοση 3.7.16.2)

Όπως αναφέραμε και πιο πάνω έχουμε πραγματοποιήσει πειράματα και σε παλαιότερο version της SQLite. Ο λόγος είναι για να μελετήσουμε εάν τα αποτελέσματα διαφέρουν σε σχέση με την τελευταία έκδοση, αλλά και για να δούμε αν έχει βελτιωθεί η βάση δεδομένων SQLite.

Ξεκινώντας στο φόρτο εργασίας 1, με το journal_mode = DELETE, έχουμε κάνει διακοπές τροφοδοσίας σε όλες τις κλήσεις των συναρτήσεων που έχουμε υλοποιήσει στη βιβλιοθήκη μας και καλούνται κατά τη διάρκεια εκτέλεσης της συναλλαγής. Συγκεκριμένα, διαπιστώσαμε ότι κατά τη διάρκεια εκτέλεσης μιας συναλλαγής σε DELETE mode, γίνεται κλήση με σειρά: τέσσερα fsync(), ένα unlink(), ένα fsync() και τέλος ένα unlink(). Έχουμε κάνει διακοπές τροφοδοσίας πριν, ενδιάμεσα και μετά την κλήση των συγκεκριμένων συναρτήσεων. Με βάση τα αποτελέσματα που έχουμε συλλέξει με τη βοήθεια του script checkthedata.py δεν παρατηρήθηκε κάποια απώλεια δεδομένων που να αφορά το durability και atomicity της SQLite. Αξίζει να σημειωθεί ότι παρατηρήθηκε απώλεια δεδομένων, αλλά αυτό αφορά το γεγονός ότι η διακοπή

τροφοδοσίας έγινε κατά της διάρκεια της συναλλαγής σε σημείο όπου τα δεδομένα δεν είχαν ακόμη γίνει commit. Επομένως, η βάση δεδομένων SQLite, σωστά έκανε rollback στην προηγούμενη κατάσταση που βρισκόταν πριν από τη διακοπή. Επίσης, έχουν γίνει διακοπές τροφοδοσίας και πριν από την κλήση του πρώτου fsync(), χωρίς όμως να παρατηρηθεί κάποια απώλεια.

Συνεχίζοντας στο φόρτο εργασίας 1, αλλάξαμε το mode σε journal_mode = WAL. Έχουμε κάνει διακοπές τροφοδοσίας στις κλήσεις που αφορούν το commit πρωτόκολλο κατά της διάρκεια εκτέλεσης μιας συναλλαγής. Δηλαδή, πριν, κατά και μετά τις κλήσεις των unlink() και fsync() του πρωτοκόλλου. Με βάση τα αποτελέσματα από τον έλεγχο του checkthedata.py δεν παρατηρήσαμε κάποια απώλεια δεδομένων που να αφορά το durability και atomicity της SQLite. Οι διακοπές τροφοδοσίας πριν από το unlink() προκάλεσαν τη βάση δεδομένων σε rollback. Το ίδιο αποτέλεσμα παρατηρήθηκε και από τη διακοπή τροφοδοσίας μετά το unlink() αλλά πριν από το fsync(). Αντίθετα, η διακοπή τροφοδοσίας μετά το fsync() δεν προκάλεσε κάποιο rollback, ούτε κάποια απώλεια δεδομένων. Αυτό ήταν αναμενόμενο επειδή η βάση δεδομένων με το fsync() κατάφερε να εξασφαλίσει την επιτυχή καταγραφή των δεδομένων στο δίσκο, επομένως μια διακοπή τροφοδοσίας μετά τη συγκεκριμένη χρονική στιγμή δεν θα επηρεάσει τα δεδομένα.

Ακολούθως, εκτελέσαμε πειράματα στο δεύτερο φόρτος εργασίας. Ο δεύτερος φόρτος εργασίας ελέγχει την συνεκτικότητα πολλαπλών σειρών με ένα νήμα προσομοιώνοντας ταυτόχρονες μη επικαλυπτόμενες τραπεζικές συναλλαγές. Αρχικά, δοκιμάσαμε το journal_mode = DELETE. Παρατηρήσαμε ότι κατά τη διάρκεια εκτέλεσης μιας συναλλαγής πραγματοποιούνται τέσσερα fsync(), ένα unlink() και τέλος ένα fsync(). Διακοπές τροφοδοσίας που έγιναν πριν από το πρώτο fsync() δεν προκάλεσαν κάποια απώλεια δεδομένων. Αυτό ήταν αναμενόμενο, γιατί οι συγκεκριμένες χρονικές στιγμές δεν αφορούν τη συναλλαγή (δεν έχει ξεκινήσει η μεταφορά δεδομένων ακόμη). Παρ' όλα αυτά, διακοπές τροφοδοσίας που έγιναν στις μεταξύ του πρώτου fsync() μέχρι και το χρονικό σημείο που τελειώνει η κλήση του πρώτου unlink() παρατηρήθηκε rollback της βάσης δεδομένων. Η SQLite σωστά έκανε rollback, λόγω του ότι οι διακοπές τροφοδοσίας έγιναν κατά της διάρκεια της συναλλαγής σε σημεία όπου τα δεδομένα δεν είχαν γίνει ακόμη commit. Άρα δεν παρατηρείται κάποια παράξενη συμπεριφορά ή απώλεια δεδομένων από τις συγκεκριμένες διακοπές. Το ίδιο αποτέλεσμα είχαμε και όταν

έγινε διακοπή τροφοδοσίας μετά το τελευταίο fsync() στη συναλλαγή. Ο λόγος είναι ο ίδιος με προηγούμενα πειράματα στο φόρτο εργασίας 1. Δηλαδή, η βάση δεδομένων με το τελευταίο fsync() συγχρονίζει τα δεδομένα στο δίσκο και δεν επηρεάζονται από οποιαδήποτε διακοπή τροφοδοσίας.

Στη συνέχεια, στο φόρτο εργασίας 2 αλλάξαμε το mode σε journal_mode = WAL. Παρατηρήσαμε ότι κατά τη διάρκεια εκτέλεσης μιας συναλλαγής πραγματοποιούνται ένα unlink(), τέσσερα fsync(), ένα unlink() και τέλος ένα fsync(). Τα αποτελέσματα ήταν παρόμοια με τα πειράματα του φόρτου εργασίας 1 σε WAL. Δηλαδή, οι διακοπές τροφοδοσίας πριν από το unlink() προκάλεσαν τη βάση δεδομένων σε rollback. Το ίδιο αποτέλεσμα παρατηρήθηκε και από τη διακοπή τροφοδοσίας μετά το unlink() αλλά πριν από το fsync(). Αντίθετα, η διακοπή τροφοδοσίας μετά το fsync() δεν προκάλεσε κάποιο rollback, ούτε κάποια απώλεια δεδομένων.

7.5.2 SQLite Version 2021 (έκδοση 3.35.4)

Όσον αφορά τις διακοπές τροφοδοσίας στον 1ο φόρτο εργασίας, για την τελευταία έκδοση της βάσης δεδομένων SQLite, θα αναμένουμε να υπάρξει κάποια βελτίωση/αλλαγή στον τρόπο που λειτουργεί το πρωτόκολλο έτσι ώστε να μειωθεί η πιθανότητα ύπαρξης κάποιου σφάλματος μετά από διακοπή τροφοδοσίας.

Έχουμε δοκιμάσει το journal_mode = DELETE και παρατηρήσαμε ότι κατά τη διάρκεια εκτέλεσης μιας συναλλαγής πραγματοποιούνται με σειρά: τέσσερα fdatasync(), 1 unlink(), τέσσερα fdatasync() και 1 unlink(). Όπως αναφέρθηκε και σε προηγούμενα κεφάλαια, η συνάρτηση fsync() δεν χρησιμοποιείται στην συγκεκριμένη έκδοση της SQLite, κάτι το οποίο μας έκανε να συμπεράνουμε ότι κάποια άλλη συνάρτηση χρησιμοποιείται στη θέση της, δηλαδή η fdatasync() κάτι το οποίο επιβεβαιώνεται από τα πειράματά μας. Έχουμε κάνει διακοπές τροφοδοσίας πριν, κατά και μετά την κλήση όλων των συναρτήσεων που χρησιμοποιούνται σε μια συναλλαγή με τη συγκεκριμένη έκδοση. Παρατηρήσαμε ότι οποιαδήποτε διακοπή στα 4 πρώτα fdatasync(), μέχρι και το σημείο ακριβώς μετά την εκτέλεση του πρώτου unlink(), η βάση δεδομένων πραγματοποίησε rollback. Όπως και με την προηγούμενη έκδοση, η βάση δεδομένων SQLite σωστά κάνει rollback, λόγω του ότι η διακοπή τροφοδοσίας έγινε κατά της

διάρκεια της συναλλαγής σε σημείο όπου τα δεδομένα δεν είχαν γίνει ακόμη commit. Όταν διακόψαμε την τροφοδοσία, μετά το 5ο fdatasync() δεν παρατηρήθηκε κάποιο rollback, ούτε κάποια απώλεια δεδομένων. Το ίδιο ισχύει και για όλες τις επόμενες διακοπές τροφοδοσίας από το συγκεκριμένο σημείο και μετά, στο φόρτο εργασίας 1, στην τελευταία έκδοση. Αυτό το αποτέλεσμα μας έκανε να συνειδητοποιήσουμε ότι μόνο τα πρώτα 4 fdatasync() και το πρώτο unlink() αφορούν την διαχείριση των δεδομένων και συγκεκριμένα την εκτέλεση του commit πρωτοκόλλου. Γι' αυτό το λόγω ίσως θα πρέπει να πρέπει να διαμορφωθεί ο τρόπος που γίνεται η συναλλαγή. Περισσότερες λεπτομέρειες στο *υποκεφάλαιο 7.9*.

Στη συνέχεια κάναμε πειράματα με journal_mode = WAL, για την τελευταία έκδοση της SQLite στον φόρτο εργασίας 1. Παρατηρήσαμε ότι κατά τη διάρκεια εκτέλεσης μιας συναλλαγής πραγματοποιούνται 2 – 3 fdatasync() και μετά ένα unlink(). Οποιεσδήποτε διακοπές τροφοδοσίας πριν από την εκτέλεση του πρώτου fdatasync() προκαλούσαν την βάση δεδομένων να πραγματοποιήσει rollback. Αντίθετα, διακοπές τροφοδοσίας σε οποιοδήποτε σημείο μετά, δεν προκαλεί κάποιο rollback ή απώλεια δεδομένων.

Ακολούθως δοκιμάσαμε το φόρτο εργασίας 2 στην τελευταία έκδοση της SQLite, έτσι ώστε να μελετήσουμε την ύπαρξη οποιονδήποτε διαφορών/βελτιώσεων σε σχέση με την προηγούμενη έκδοση, αλλά και για τον εντοπισμό κάποιου σφάλματος. Όπως και με το φόρτο εργασίας 1, η τελευταία έκδοση της SQLite δεν κάνει χρήση της συνάρτησης fsync(), αλλά της συνάρτησης fdatasync().

Ξεκινώντας με το journal_mode = DELETE, παρατηρήσαμε ότι κατά τη διάρκεια εκτέλεσης μιας συναλλαγής πραγματοποιούνται τέσσερα fdatasync(), ένα unlink(), τέσσερα fdatasync(), ένα unlink(), τέσσερα fdatasync() και τέλος τρία unlink(). Βέβαια, όχι όλες οι κλήσεις των πιο πάνω συναρτήσεων αφορούν το commit πρωτόκολλο, κάτι το οποίο διαπιστώσαμε μετά την εκτέλεση των πειραμάτων μας. Συγκεκριμένα, διακοπές τροφοδοσίας πριν το πρώτο fdatasync() δεν προκαλούν κάποια απώλεια δεδομένων. Μεταξύ του πρώτου fdatasync(), μέχρι και το πέμπτο fdatasync() (δηλαδή μετά το πρώτο unlink), παρατηρήσαμε ότι η βάση δεδομένων κάνει rollback αφού οι διακοπές επηρεάζουν χρονικές στιγμές στις οποίες γίνεται επεξεργασία/μεταφορά δεδομένων. Επομένως, σωστά η SQLite κάνει rollback. Όσον αφορά τα πειράματα με τις διακοπές

τροφοδοσίας από το πέμπτο fdatasync() και μετά, δεν παρατηρήθηκε κάποια απώλεια δεδομένων, ούτε κάποιο άλλο σφάλμα. Άρα, όπως αναφέρθηκε και πριν, υπάρχει πιθανότητα κάποιες συναρτήσεις να μην αφορούν το commit πρωτόκολλο.

Συνεχίζοντας με την τελευταία έκδοση της SQLite για τον δεύτερο φόρτο εργασίας, αλλάξαμε το mode σε journal_mode = WAL. Παρατηρήσαμε ότι κατά τη διάρκεια εκτέλεσης μιας συναλλαγής πραγματοποιούνται 3 – 4 fdatasync() και μετά δύο unlink(). Με βάση το αρχείο checkthedata.py δεν παρατηρήθηκε κάποια απώλεια δεδομένων, παρά μόνο όταν η διακοπή τροφοδοσίας γινόταν πριν από το πρώτο fdatasync(). Αυτό μας παραξένεψε λίγο, διότι αναμέναμε ο ότι ο τρόπος με τον οποίο δουλεύει το WAL θα παρουσιαζόταν κάποια απώλεια δεδομένων. Γι' αυτό είναι αναγκαία η εκτέλεση περαιτέρω πειραμάτων, όπως θα εξηγηθεί σε επόμενα υποκεφάλαια.

7.6 Παρατηρήσεις/Σχόλια από τα πειράματα στους φόρτους εργασίας

Μετά από τα πολλαπλά πειράματα που εκτελέσαμε στους δύο φόρτους εργασίας, τόσο στην έκδοση 2013, όσο και στην τελευταία έκδοση, παρατηρήσαμε κάποιες διαφορές που αξίζει να αναφερθούν.

Πρώτη παρατήρηση είναι ότι μειώθηκαν σε μεγάλο βαθμό οι κλήσεις συναρτήσεων συγχρονισμού fsync στην έκδοση 2021, σε σχέση με την έκδοση 2013. Αυτό μειώνει ταυτόχρονα και την πιθανότητα να παρουσιαστεί κάποια απώλεια δεδομένων, αφού τα κρίσιμα σημεία είναι λιγότερα. Επίσης, μειώθηκαν γενικότερα οι κλήσεις συναρτήσεων, δηλαδή η τελευταία έκδοση έχει γίνει πιο γρήγορη και πιο αποδοτική σε σχέση με την έκδοση 2013.

Δεύτερη παρατήρηση είναι ότι δεν παρουσιάστηκε κάποια απώλεια δεδομένων, τόσο σε DELETE mode όσο και σε WAL mode. Η βάση δεδομένων SQLite πραγματοποιούσε rollback, όταν οι διακοπές τροφοδοσίας γίνονταν σε σημεία όπου μεταφέρονταν/ επεξεργάζονταν δεδομένα. Αυτό είναι κάτι που μας προβληματίζει, τουλάχιστον σε WAL mode, λόγω του τρόπου με τον οποίο αποθηκεύει τα δεδομένα, Γι' αυτό και θα γίνουν περαιτέρω πειράματα, όπως αναφέρεται σε επόμενα υποκεφάλαια.

Τρίτη παρατήρηση είναι ότι η έκδοση 2013 χρησιμοποιά την συνάρτηση fsync() για τον συγχρονισμό των δεδομένων, ενώ η έκδοση 2021 χρησιμοποιά την συνάρτηση fdatasync(). Την συγκεκριμένη παρατήρηση την ανακαλύψαμε μετά την εκτέλεση των πειραμάτων, όπου δεν παρατηρήσαμε κάποια κλήση της fsync(). Αυτό μας έκανε να αναρωτηθούμε αν γίνεται κλήση κάποιας άλλης συνάρτησης, δηλαδή της fdatasync().

7.7 Ισχυρισμοί SQLite

Εκτός από πειράματα με διαφορετικό journal mode, αποφασίσαμε να ελέγξουμε κάποιους ισχυρισμούς που παραθέτει η βάση δεδομένων SQLite στον οδηγό χρήσης της (documentation). Οι πιο κάτω ισχυρισμοί αφορούν τα διάφορα PRAGMA synchronous statements που έχουν εξηγηθεί πιο πάνω και αφορούν την αλλαγή της συμπεριφοράς της SQLite. Συγκεκριμένα, πόσο πιο durable μπορεί να γίνει η βάση δεδομένων μετά από μια διακοπή τροφοδοσίας από την αλλαγή του synchronous statement σε συνδυασμό με διαφορετικά journal modes. Τα πειράματα που έχουμε πραγματοποιήσει έχουν εκτελεστεί μόνο στην τελευταία έκδοση της SQLite, λόγω του ότι το documentation που μας δίνει η SQLite αναφέρεται στην τελευταία έκδοση. Επίσης, αποφασίσαμε να ελέγξουμε μόνο τους συνδυασμούς journal mode και synchronous statements που ισχυρίζεται η SQLite ότι είναι πιο durable, καθώς αυτοί οι συνδυασμοί θα αποφέρουν τα αποτελέσματα τα οποία ισχυρίζονται.

1ος Ισχυρισμός

"WAL mode is safe from corruption with synchronous=NORMAL. WAL mode is always consistent with synchronous=NORMAL, but WAL mode does lose durability. A transaction committed in WAL mode with synchronous=NORMAL might roll back following a power loss or system crash."

Με βάση τον πιο πάνω ισχυρισμό θα ορίσουμε journal_mode = WAL και synchronous statement = NORMAL και θα ελέγξουμε αν μια διακοπή τροφοδοσίας θα προκαλέσει τη βάση σε rollback ή σε loss of durability.

Ο πιο πάνω ισχυρισμός είναι σωστός. Μετά από διακοπή τροφοδοσίας κατά τη διάρκεια εκτέλεσης των φόρτων εργασίας παρατηρήσαμε ότι οι συναλλαγές δεν εκτελέστηκαν με

επιτυχία. Αντιθέτως, πραγματοποιήθηκε rollback και η SQLite επανήλθε στην τελευταία της κατάσταση πριν από το συγκεκριμένο transaction. Δηλαδή, στην μη δημιουργία του πίνακα kvpairs. Άρα, θα πρέπει τα προγράμματα που χρησιμοποιούν τη βάση δεδομένων SQLite να έχουν ειδικές διεργασίες που εκτελούνται μετά από διακοπές τροφοδοσίας έτσι ώστε να αναγνωρίζουν ότι ένα transaction έκανε rollback για να το ξαναεκτελούν. Κατά τη γνώμη μου, αυτό δεν είναι πολύ πρακτικό και είναι δύσκολο στην υλοποίηση.

2ος Ισχυρισμός

" *With synchronous=FULL in WAL mode, an additional sync operation of the WAL file happens after each transaction commit. The extra WAL sync following each transaction help ensure that transactions are durable across a power loss."*

Με βάση τον πιο πάνω ισχυρισμό θα ορίσουμε journal_mode = WAL και synchronous statement = FULL και θα ελέγξουμε αν μια διακοπή τροφοδοσίας δεν θα επηρεάσει τα δεδομένα και θα παραμείνει durable, λόγω του επιπλέον fsync.

Ο πιο πάνω ισχυρισμός είναι σωστός. Μετά από διακοπή τροφοδοσίας κατά τη διάρκεια εκτέλεσης των φόρτων εργασίας παρατηρήσαμε ότι, πράγματι, γίνεται ένα επιπλέον fdatasync μετά από κάθε συναλλαγή. Παρ' όλα αυτά, υπάρχει μια μικρή πιθανότητα να γίνει διακοπή τροφοδοσίας πριν το συγκεκριμένο fdatasync και να μην προλάβουν να γραφτούν τα δεδομένα στο δίσκο. Μετά από τον έλεγχο της συγκεκριμένης περίπτωσης παρατηρήσαμε ότι η SQLite πραγματοποιεί rollback και επανέρχεται στην τελευταία της κατάσταση πριν από το συγκεκριμένο transaction.

3ος Ισχυρισμός

" EXTRA synchronous is like FULL with the addition that the directory containing a rollback journal is synced after that journal is unlinked to commit a transaction in DELETE mode. EXTRA provides additional durability if the commit is followed closely by a power loss."

Με βάση τον πιο πάνω ισχυρισμό θα ορίσουμε journal_mode = DELETE και synchronous statement = EXTRA και θα ελέγξουμε αν μια διακοπή τροφοδοσίας δεν θα επηρεάσει τα δεδομένα και θα παραμείνει durable.

Ο πιο πάνω ισχυρισμός είναι σωστός αν και δεν παρατηρήσαμε “extra” προστασία. Μετά από διακοπή τροφοδοσίας κατά τη διάρκεια εκτέλεσης των φόρτων εργασίας παρατηρήσαμε ότι, και εδώ γίνεται ένα επιπλέον fdatasync μετά από κάθε συναλλαγή. Αυτό είναι σωστό επειδή το EXTRA είναι παρόμοιο με το FULL. Παρ’ όλα αυτά, το “additional durability” που ισχυρίζεται η SQLite δεν ισχύει. Μετά από την εκτέλεση των πειραμάτων παρατηρήσαμε ότι η SQLite πραγματοποιεί rollback και επανέρχεται στην τελευταία της κατάσταση πριν από το συγκεκριμένο transaction. Επομένως, δεν υπάρχει κάποιο επιπλέον durability σε σχέση με το synchronous statement = FULL, εφόσον τα σημεία στα οποία παρατηρήθηκε rollback ήταν τα ίδια.

4ος Ισχυρισμός

“ *With synchronous OFF (0) [...]. If the application running SQLite crashes, the data will be safe, but the database might become corrupted if the operating system crashes or the computer loses power[...]”*

Με βάση τον πιο πάνω ισχυρισμό θα ορίσουμε journal_mode = DELETE και synchronous statement = OFF και θα ελέγξουμε αν μια διακοπή τροφοδοσίας δεν θα επηρεάσει τα δεδομένα και η βάση θα γίνει corrupted.

Ο πιο πάνω ισχυρισμός είναι σωστός. Μετά από διακοπή τροφοδοσίας κατά τη διάρκεια εκτέλεσης των φόρτων εργασίας παρατηρήσαμε ότι οι συναλλαγές δεν εκτελέστηκαν με επιτυχία. Αντιθέτως, πραγματοποιήθηκε rollback και η SQLite επανήλθε στην τελευταία της κατάσταση πριν από το συγκεκριμένο transaction. Δηλαδή, στην μη δημιουργία του πίνακα kvpairs, όπως ακριβώς έγινε και με το PRAGMA synchronous NORMAL. Άρα, θα πρέπει τα προγράμματα που χρησιμοποιούν τη βάση δεδομένων SQLite να έχουν ειδικές διεργασίες που εκτελούνται μετά από διακοπές τροφοδοσίας έτσι ώστε να αναγνωρίζουν ότι ένα transaction έκανε rollback για να το ξαναεκτελούν.

7.8 Πειράματα αλλαγής του checkpoint threshold

Όπως αναφέρθηκε σε προηγούμενο υποκεφάλαιο, η μετακίνηση των συναλλαγών από το αρχείο WAL πίσω στη βάση δεδομένων ονομάζεται Checkpointing. Είναι μια

αυτοματοποιημένη διαδικασία που αναλαμβάνει όλες οι αλλαγές που έχουμε κάνει και έχουν γραφτεί στο αρχείο WAL. Από προεπιλογή, η βάση δεδομένων SQLite κάνει αυτόματα checkpointing όταν το αρχείο WAL φτάσει σε μέγεθος threshold 1000 σελίδων.

Σκοπός των πειραμάτων που ακολουθούν είναι να γίνει αλλαγή στο μέγεθος του αρχείου WAL το οποίο είναι υπεύθυνο για τη μετακίνηση των συναλλαγών πίσω στο δίσκο. Συγκεκριμένα, θα μειώσουμε το μέγεθος του αρχείου WAL σε μέγεθος 1 σελίδας, από το προεπιλεγμένο μέγεθος που είναι 1000 σελίδες. Με αυτή την αλλαγή ευελπιστούμε ότι λόγω του μικρού μεγέθους, η βάση δεδομένων θα αναγκάζεται να γράφει σε πιο μικρά χρονικά διαστήματα τα δεδομένα στο δίσκο, μειώνοντας έτσι την πιθανότητα ύπαρξης κάποιας απώλειας δεδομένων ή κάποιου rollback.

Ο τρόπος με τον οποίο μπορούμε να αλλάξουμε το μέγεθος σελίδας, είναι πολύ απλά με την εισαγωγή της εντολής “PRAGMA wal_autocheckpoint=1;” μέσα στο πρώτο query του φόρτου εργασίας τον οποίο ελέγχουμε. Η αλλαγή σελίδας μπορεί να γίνει και μέσω του πηγαίου κώδικα της βάσης δεδομένων SQLite, αλλά θα πρέπει να γίνει ξανά build η βιβλιοθήκη, επομένως χρησιμοποιήσαμε την πρώτη μέθοδο, η οποία είναι πιο πρακτική.

Μετά από διακοπές τροφοδοσίας σε συναρτήσεις των φόρτων εργασίας με wal_autocheckpoint=1 και journal_mode=WAL, παρατηρήσαμε ότι δεν υπήρξε κάποια απώλεια δεδομένων. Συγκεκριμένα, στα πρώτα 2 πρώτα fdatasync η βάση δεδομένων SQLite πραγματοποίηση rollback, κάτι το οποίο είναι χειρότερο σε σχέση με το wal_autocheckpoint=1000, στο οποίο rollback γινόταν μόνο στο πρώτο fdatasync(). Επίσης, παρατηρήσαμε ότι το wal_autocheckpoint=1 κάνει αισθητά πιο αργή την εκτέλεση των φόρτων εργασίας, λόγω των πολλών κλήσεων συναρτήσεων συγχρονισμού. Δοκιμάσαμε να αλλάξουμε το wal_autocheckpoint σε ένα ενδιάμεσο αριθμό πχ 300, αλλά τα αποτελέσματα ήταν παρόμοια. Επομένως, η συγκεκριμένη τακτική δεν συστήνεται να χρησιμοποιείται σε εφαρμογές και συστήματα.

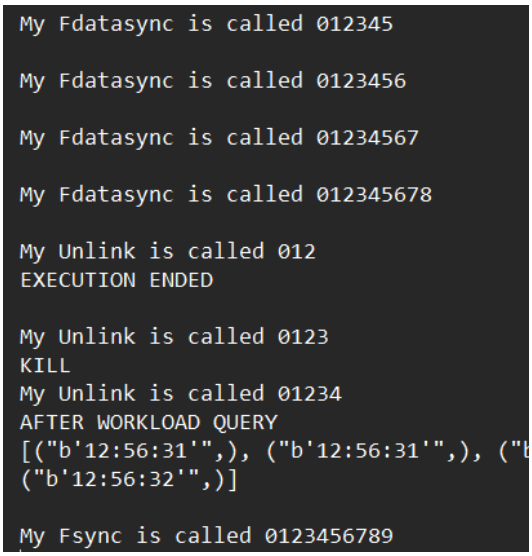
7.9 Πειράματα αφαίρεσης των subqueries στους φόρτους

Μέχρι στιγμής, δεν έχουμε καταφέρει να αποδείξουμε κάποια απώλεια δεδομένων που να έχει προκληθεί από κάποια διακοπή τροφοδοσίας. Εάν δούμε ξανά το πώς είναι υλοποιημένη η συναλλαγή του φόρτου εργασίας 1, θα δούμε ότι στην πραγματικότητα είναι 6 εντολές που εκτελούνται σε μια συναλλαγή. Οι τελευταίες 3 εντολές αφορούν την εισαγωγή κάποιων timestamps μέσα στον πίνακα aftertimestamp. Αυτές οι εντολές δεν αφορούν την εισαγωγή των δεδομένων μέσα στον πίνακα kvpairs ο οποίος είναι ο πίνακας στον οποίο γίνεται η συναλλαγή. Με βάση αυτό, καταλαβαίνουμε ότι οι επιπλέον εντολές ίσως να επηρεάζουν την λειτουργία του πρωτοκόλλου και άρα τα αποτελέσματά μας.

Επίσης, όπως αναφέρει στο άρθρο του ο Mai Zheng με τίτλο «Torturing Databases for Fun and Profit» [1], ο τρόπος με τον οποίο συγχρονίζονται τα δεδομένα μιας συναλλαγής είναι με το ξεκίνημα της επόμενης συναλλαγής. Δηλαδή, όταν το commit πρωτόκολλο τελειώσει, το journal file θα διαγραφτεί με τη βοήθεια της εντολής unlink. Ακολούθως, με το ξεκίνημα της επόμενης εντολής ή συναλλαγής, η βάση δεδομένων SQLite θα κάνει ένα fsync έτσι ώστε να συγχρονιστούν τα δεδομένα της προηγούμενης συναλλαγής.

Με βάση τις πιο πάνω πληροφορίες, οι 3 επιπλέον εντολές που χρησιμοποιούνται μέσα στο φόρτο εργασίας μας, οι οποίες είναι για σκοπούς καταγραφής χρονικών στιγμών εισαγωγής δεδομένων μέσα στον πίνακα kvpairs, ίσως να ευθύνονται για τον συγχρονισμό των δεδομένων.

Στόχος του συγκεκριμένου πειράματος είναι να απομονώσουμε το σημείο όπου γίνονται commit τα δεδομένα για να μην δώσουμε περιθώριο στην SQLite να κάνει fsync μετά το commit (βάση πρωτοκόλλου). Γι’ αυτό, θα αφαιρέσουμε τις γραμμές στις οποίες εκτελούνται οι 3 επιπλέον εντολές μέσα στο φόρτο εργασίας μας. Μετά από μια γρήγορη επιβεβαίωση της αφαίρεσης των επιπλέων εντολών παρατηρήσαμε μείωση στα fsyncs.



Σχήμα 7.2: Screenshot εκτέλεσης χωρίς τις επιπλέον εντολές

Όπως φαίνεται και στο σχήμα 7.2 με το τέλος του commit πρωτοκόλλου (στο σημείο "EXECUTION ENDED"), παρατηρούμε ότι δεν εκτελούνται οποιεσδήποτε κλήσεις συναρτήσεων συγχρονισμού. Το μόνο fsync που παρατηρείται είναι η τελευταία γραμμή του σχήματος όπου θα γραφτεί στο progress_report ότι ολοκληρώθηκε το transaction.

Επομένως, αν γίνει διακοπή τροφοδοσίας σε κάποιο σημείο μετά την ολοκλήρωση του commit πρωτοκόλλου, αλλά πριν την έναρξη της επόμενης συναλλαγής (η οποία δεν φαίνεται στο σχήμα), τι ακριβώς θα γίνει; Επίσης, το fsync που γράφει στο progress_report, συγχρονίζει και τα δεδομένα της βάσης? Δηλαδή θα γίνουν synced και τα δεδομένα της βάσης ή μόνο το write στο αρχείο progress_report?

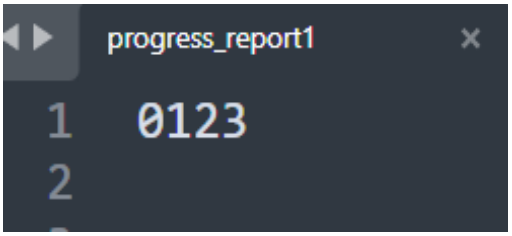
Τα πιο πάνω ερωτήματα θα απαντηθούν μετά τις αλλαγές στους φόρτους εργασίας όπως εξηγήθηκαν πιο πάνω. Αν δεν συγχρονιστούν τα δεδομένα στο δίσκο, αλλά γραφτεί ότι έχει ολοκληρωθεί το transaction μέσα στο αρχείο progress_report, τότε αποκαλύπτουμε ένα σημαντικό bug που καθιστά την βάση δεδομένων SQLite non durable.

```
for n in range(0, 4):
  q=create_numbers %(txsize)
  tuples = sqlite.execute(db, q.encode())
  q = workload_query % (n)
  tuples = sqlite.execute(db, q.encode())
  print(tuples)
  file=open('/var/tmp/kp', 'w')
  # report completion of last query
  progress_report_file.write('%s' % n)
  progress_report_file.flush()
  os.fsync(progress_report_file.fileno()os.system(
"/home/pi/workspace/project/pinap/workload1_sqlite/killpower.sh");
```

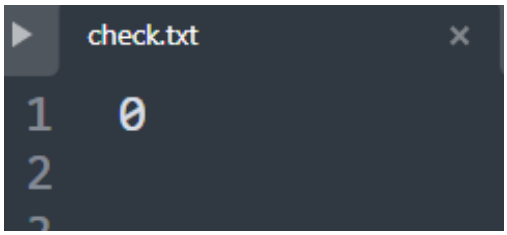
Διακοπή μετά το fsync του progress_report

Σχήμα 7.3: Σημείο τοποθέτησης διακοπής τροφοδοσίας

Όπως φαίνεται και στο σχήμα 7.3 θα τοποθετήσουμε την εντολή που προκαλεί διακοπή τροφοδοσίας ακριβώς μετά το σημείο όπου γίνεται εγγραφή της ολοκλήρωσης μιας συναλλαγής στο αρχείο progress_report. Η διακοπή τροφοδοσίας γίνεται πριν να ξεκινήσει η επόμενη συναλλαγή, άρα η βάση δεν προλαβαίνει να συγχρονίσει τα δεδομένα στο δίσκο, αφού δεν πραγματοποιείται κάποια άλλη εντολή (λόγω αφαίρεσης των επιπλέον εντολών). Όπως αναφέρθηκε και πριν, το fsync στην προτελευταία γραμμή του σχήματος, δεν αφορά τη βάση δεδομένων SQLite και αν όντως δεν επηρεάζει τα δεδομένα της βάσης, τότε ίσως να αποκαλυφθεί κάποιο σφάλμα απώλειας δεδομένων.



Σχήμα 7.4.1



Σχήμα 7.4.2

Στο πείραμα θα διακόψουμε την τροφοδοσία μετά την ολοκλήρωση της 3ης συναλλαγής, την οποία ορίσαμε με ένα απλό if statement μέσα στο for loop. Μετά την εκτέλεση του συγκεκριμένου πειράματος με την αφαίρεση των επιπλέων εντολών παρατηρήσαμε τα εξής αποτελέσματα, όπως φαίνονται στα σχήματα 7.4.1 και 7.4.2. Το αρχείο progress_report όπως φαίνεται στο σχήμα 7.4.1 δείχνει ότι έχουν ολοκληρωθεί με

επιτυχία οι 3 πρώτες συναλλαγές. Όμως, το αρχείο check.txt όπως φαίνεται στο σχήμα 7.4.2 δείχνει ότι έχουμε απώλεια δεδομένων, η οποία συμβολίζεται με τον αριθμό μηδέν.

Με βάση τα πιο πάνω αποτελέσματα συμπεραίνουμε ότι κάτι δεν είναι φυσιολογικό. Από την μια έχουμε ότι η συναλλαγές ολοκληρώθηκαν με επιτυχία, ενώ από την άλλη έχουμε ότι υπήρξε κάποια απώλεια δεδομένων. Τελικά, συνάρτηση fsync() που γράφει στο αρχείο progress_report δεν επηρεάζει τα δεδομένα της βάσης SQLite. Άρα, το commit πρωτόκολλο βασίζεται στην συνάρτηση fsync() που θα εκτελεστεί από την επόμενη συναλλαγή. Και εφόσον η διακοπή τροφοδοσίας πραγματοποιήθηκε πριν την έναρξη της επόμενης συναλλαγής, τα δεδομένα δεν συγχρονίστηκαν στο δίσκο.

Το συγκεκριμένο πείραμα έχει καταφέρει να αποδείξει ότι βάση δεδομένων SQLite δεν είναι durable στις διακοπές τροφοδοσίας. Παρόλο της μικρής πιθανότητας της διακοπής τροφοδοσίας στο συγκεκριμένο σημείο, δεν παύει από το να είναι ένα σφάλμα το οποίο θα πρέπει να αγνοήσουμε.

Κεφάλαιο 8

Συμπεράσματα

8.1 Συμπεράσματα 87
8.2 Εισηγήσεις 88

8.1 Συμπεράσματα

Στην παρούσα διπλωματική εργασία έγινε μελέτη της ευρωστίας της βάσης δεδομένων SQLite έναντι διακοπών τροφοδοσίας σε συγκεκριμένα σημεία. Έγινε χρήση μιας βιβλιοθήκης που υλοποιήσαμε έτσι ώστε να δοκιμάσουμε διαφορετικές μεθοδολογίες σε σχέση με προηγούμενες μελέτες. Έχουν δοκιμαστεί 2 διαφορετικές μεθοδολογίες για τη συγκεκριμένη μελέτη: Τυχαίες διακοπές τροφοδοσίας μέσα στη βιβλιοθήκη και διακοπές σε συγκεκριμένες κλήσεις συναρτήσεων της βιβλιοθήκης.

Από την πρώτη μεθοδολογία δεν καταφέραμε να εντοπίσουμε κάποιο σφάλμα ή απώλεια δεδομένων, τόσο σε προηγούμενη έκδοση της SQLite, όσο και στην τελευταία έκδοση. Το ίδιο αποτέλεσμα είχαμε και στην αλλαγή του journal mode από DELETE σε WAL.

Όμως, από τη χρήση της δεύτερης μεθοδολογίας καταφέραμε να εντοπίσουμε απώλεια δεδομένων από διακοπή σε συγκεκριμένο σημείο. Συγκεκριμένα, η διακοπή τροφοδοσίας έγινε μετά την ολοκλήρωση της συναλλαγής, αλλά πριν την κλήση της συνάρτησης συγχρονισμού (fsync), βάση το πρωτόκολλο στο journal_mode=DELETE. Αυτό που παρατηρήσαμε είναι ότι η βάση δεδομένων SQLite, συγχρονίζει τα δεδομένα στο δίσκο με την κλήση της συνάρτησης fsync, της επόμενης συναλλαγής. Επομένως, αφού διακόψαμε την τροφοδοσία ακριβώς πριν από το συγκεκριμένο fsync, η βάση δεν πρόλαβε να συγχρονίσει τα δεδομένα στο δίσκο (βλ. πείραμα στο *κεφάλαιο 7.8*). Τα

αποτελέσματα απώλειας δεδομένων αφορούν τόσο την τελευταία έκδοση, όσο και την παλαιότερη.

Άρα έχουμε δείξει ότι ακόμα και μια φαινομενικά καλά δοκιμασμένη βάση δεδομένων, όπως η SQLite μπορεί να χάσει δεδομένα . Αυτό είναι μία αφύπνιση για οποιονδήποτε δημιουργό λογισμικού συστημάτων αποθήκευσης που χρησιμοποιά την SQLite, για να δώσει περισσότερη έμφαση στην ορθότητα, διότι αυτή είναι ένα πολύ σημαντικό συστατικό για την εμπιστοσύνη των χρηστών και για την επιτυχία του λογισμικού. Οποιοσδήποτε χρήστης , ειδικά κάποιος χρήστης της τελευταίας έκδοσης της SQLite , θα περίμενε μία πιο ασφαλή συμπεριφορά στις προκαθορισμένες ρυθμίσεις της. Κανείς δεν θα ήθελε να χάσει λεφτά από τον λογαριασμό του χωρίς να πάνε στον προορισμό τους, λόγω μιας διακοπής τροφοδοσίας. Και επίσης, κανείς δεν θα ήθελε να χρησιμοποιεί κάποια βάση δεδομένων γνωρίζοντας ότι υπάρχει πιθανότητα απώλειας δεδομένων.

8.2 Εισηγήσεις

Έχουμε αποδείξει ότι η βάση δεδομένων SQLite είναι ευάλωτη σε διακοπές τροφοδοσίας. Αυτό όμως ισχύει για τις προεπιλεγμένες ρυθμίσεις της SQLite που είναι το journal_mode=DELETE. Τα πειράματα που έχουμε κάνει σε WAL mode δεν έχουν παρουσιάσει κάποιο σφάλμα ή απώλεια δεδομένων (μέχρι στιγμής). Αυτό που θα πρότεινα σε όσους χρησιμοποιούν την βάση δεδομένων SQLite στις εφαρμογές τους είναι να ορίσουν το journal mode σε WAL. Με αυτό τον τρόπο θα μειώσουν την πιθανότητα ύπαρξης κάποιου σφάλματος λόγω διακοπής τροφοδοσίας.

Επίσης καλό θα ήταν να γίνει υλοποίηση των φόρτων εργασίας και σε άλλα συστήματα βάσεων δεδομένων για να δούμε και εκεί πως θα ανταποκριθεί η βάση δεδομένων σε σημεία διακοπής τροφοδοσίας που έχουμε εντοπίσει απώλεια δεδομένων Άρα, εκεί θα δούμε εάν αυτό το σφάλμα είναι ένα σφάλμα που αφορά μόνο την βάση δεδομένων SQLite , ή εάν είναι κάτι πιο γενικό που αφορά τις περισσότερες βάσεις δεδομένων , ή και όλες.

Ακόμη, σε μια μελλοντική εργασία θα μπορούσαν οι φόρτοι εργασίας να υλοποιηθούν σε γλώσσα προγραμματισμού C, αντί σε Python όπως είναι υλοποιημένοι εδώ. Με αυτό

τον τρόπο θα γίνει πιο εύκολος ο εντοπισμός κάποιου σφάλματος λόγω της υποστήριξης εργαλείων όπως το GCC Coverage και μεθόδων όπως το backtrace().

Τέλος, θα ήταν καλό να χρησιμοποιηθεί και ένα άλλο μοντέλο για διακοπές τροφοδοσίας, διαφορετικό από αυτό που μας έχει δώσει ο Terence Kelly. Για παράδειγμα, θα μπορούσε να αναπτυχθεί ένα μοντέλο το οποίο θα χρησιμοποιά την εντολή kill process και να συγκρίνουμε τα αποτελέσματα με αυτά που έχουμε αποδείξει (βλ. *Κεφάλαιο 7.9*) στην συγκεκριμένη διπλωματική εργασία. Επίσης, να παρουσιαστούν πιθανά πλεονεκτήματα ή μειονεκτήματα του μοντέλου kill process.

Βιβλιογραφία

[1] Mai Zheng, “Torturing Databases for Fun and Profit” , 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’14)

[2] Terence Kelly. 2020. Is persistent memory persistent? Commun. ACM 63, 9 (September 2020), 48–54.

[3] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: a lightweight, general system for finding serious storage system errors. In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06). USENIX Association, USA, 131–146.

[4] Kummerländer, Adrian, 2016. Notes on function interposition in C++ @ /home/adrian. [online] Blog.kummerlaender.eu. Available at: <https://blog.kummerlaender.eu/article/notes_on_function_interposition_in_cpp/

[5] Chovatiya, Vishal, 2016. Hack C/C++ application using RTLD_NEXT with an easy example. [online] Available at: <http://www.vishalchovatiya.com/hack-c-cpp- application-using-rtld-next-with-an-easy-example/>

[6] https://man7.org/linux/man-pages/man2/unlink.2.html

[7] https://www.journaldev.com/31907/calling-c-functions-from-python

[8] https://www.sqlite.org/wal.html

[9] https://www.sqlite.org/pragma.html#pragma_synchronous

[10] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 618–622.

[11] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding crash-consistency bugs with bounded black-box crash testing. In Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI'18). USENIX Association, USA, 33–50.

[12] Mai Zheng, Joseph Tucek, Feng Qin and Mark Lillibridge, The Ohio State University - HP Labs «Zheng et al. Understanding the Robustness of SSDs under Power Fault, FAST 2013»

[13] www.cs.ucy.ac.cy/~dzeina/talks/research-led-teaching-15-12-2015.txt