ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βελτιστοποίηση Ροής Σχεσιακών Επερωτήσεων κατά το Χρόνο Εκτέλεσης

Δημήτρης Ι. Φωτόπουλος

Επιβλέποντες: **Αλέξανδρος Δελής**, Καθηγητής ΕΚΠΑ
Μιχάλης Χατζόπουλος, Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2011

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βελτιστοποίηση Ροής Σχεσιακών Επερωτήσεων κατά το Χρόνο Εκτέλεσης

Δημήτρης Ι. Φωτόπουλος
Α.Μ.: M0998

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Αλέξανδρος Δελής,** Καθηγητής ΕΚΠΑ
Μιχάλης Χατζόπουλος, Καθηγητής ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Αλέξανδρος Δελής,** Καθηγητής ΕΚΠΑ
Μιχάλης Χατζόπουλος, Καθηγητής ΕΚΠΑ

ΟΚΤΩΒΡΙΟΣ 2011

ΠΕΡΙΛΗΨΗ

Ένα μεγάλο μέρος των σύγχρονών εταιρικών εφαρμογών λογισμικού και των διαδικτυακών εφαρμογών συμπεριλαμβάνει τη χρήση κάποιας μόνιμης μνήμης, για την αποθήκευση των δεδομένων τους. Τις περισσότερες φορές αυτή η μόνιμη αποθήκευση των δεδομένων αλλά και η προσβασιμότητα σε αυτά εξασφαλίζεται μέσα από μία σχεσιακή βάση δεδομένων. Πολλές σύγχρονες εφαρμογές, πολλαπλών χρηστών, όπως ERPs, CRMs, CMSs, BPMs υποβάλλουν ροές από σχεσιακά ερωτήματα άμεσα σε βάσεις δεδομένων ή σε ενδιάμεσα μεσισμικά. Στη συντριπτική πλειοψηφία των εφαρμογών χρησιμοποιείται κάποια API βιβλιοθήκη που υποστηρίζει κάποιο ευρέως αποδεκτό πρωτόκολλο, όπως τα ODBC και JDBC. Οι επερωτήσεις συνθέτονται με τη μορφή αλφαριθμητικών και αφού συγκεκριμενοποιηθούν οι εκάστοτε πιθανές παράμετροι εισόδου, αποστέλλονται στη βάση δεδομένων.

Οι περισσότερες από αυτές τις εφαρμογές υποφέρουν από μειωμένη διέλευση και υψηλές καθυστερήσεις. Υψηλά φορτία επερωτήσεων δοκιμάζουν τις αντοχές τόσο των μηχανών αποθήκευσης στον πυρήνα των σχεσιακών βάσεων όσο και τους αντίστοιχους διαχειριστές ταυτοχρονισμού. Τα τελευταία χρόνια έχουμε γίνει μάρτυρες μία εντατικής έρευνας τόσο σε σχεσιακές όσο και σε αντικειμενοστραφείς βάσεις δεδομέμων. Για πολλά χρόνια η ερευνητική κοινότητα έχει επικεντρώσει την προσοχή της στην ανάπτυξη εκλεπτυσμένων αποθηκευτικών μηχανών, πιο αποδοτικών επεξεργαστών ερωτημάτων, κλιμακούμενων συστοιχιών από βάσεις δεδομένων, εξυπηρετητών αποθήκευσης κλειδιών-τιμών στην κύρια μνήμη με απώτερο σκοπό να απαλείψουν προβλήματα μειωμένης διέλευσης και να επιτρέψουν περισσότερο ταυτοχρονισμό. Αυτές οι βελτιστοποιήσεις έχουν ως στόχο την αποδοτική ανάκτηση πληροφορίας, τη γρήγορη εγγραφή νέων δεδομένων, την ταχύτατη σύγκλιση στο βέλτιστο πλάνο εκτέλεσης και την βέλτιση εκτέλεση του με βάση τα χαρακτηριστικά του χρησιμοποιούμενου υλικού. Οι κατασκευαστές υλικού έχουν βελτιώσει σημαντικά την ταχύτητα και τη διέλευση των πολύ-πύρηνων επεξεργαστών, της κύριας μνήμης, της ιεραρχικής δευτερεύουσας μνήμης, των σκληρών δίσκων (π.χ. δίσκοι στερεάς κατάστασης), των επίγειων (π.χ. LAN, MAN) και ασύρματων δικτύων (π.χ. Wifi).

Όλα αυτά τα βελτιωμένα τμήματα του υλικού, είναι ευρέως διαθέσιμα και αρκετά φθηνά, ώστε να χρησιμοποιηθούν από επιτραπέζια συστήματα, μέχρι κέντρα υπολογιστών επιτρέποντας τους να διαχειριστούν αποδοτικά και αξιόπιστα ροές σχεσιακών επερωτήσεων που προέρχονται από ένα μεγάλο αριθμό χρηστών. Οι αναμφίβολα σημαντικές πρόοδοι στο υλικό, έχουν αντιμετωπίσει πολλά προβλήματα διέλευσης, αλλά φαίνεται ότι δεν ισχύει το ίδιο για τη παρατηρούμενη καθυστέρηση και ότι δε θα είναι εύκολο να βελτιωθεί η κατάσταση για αυτό στο εγγύς μέλλον. Η διέλευση αυξάνεται συνεχώς με τον ερχομό νέων επίγειων και ασύρματων τεχνολογιών, αλλά δε φαίνεται να συμβαίνει κάτι ανάλογο με την καθυστέρηση (latency) στο δίκτυο. Οι προσπάθειες στη

βελτίωσης της αντιληπτής καθυστέρησης από τον τελικό χρήστη φαίνεται να έχουν μείνει πίσω σε σχέση με τις εξελίξεις στην αποδοτική χρησιμοποίηση του διαθέσιμου εύρους ζώνης.

Για την απαλοιφή των προβλημάτων μειωμένης διέλευσης η ερευνητική κοινότητα στο χώρο των βάσεων δεδομένων έχει επικεντρώσει τις προσπαθειές της σε δύο τεχνικές, στην προσωρινή αποθήκευση και στην προανάκληση των επιστρεφόμενων αποτελεσμάτων των υποβληθέντων σχεσιακών ερωτημάτων. Υψηλά φορτία ροών σχεσιακών ερωτημάτων έχουν γίνει προσπάθειες να αντιμετωπιστούν με αποθήκευση των επιστρεφόμενων αποτελεσμάτων, για κάθε ξεχωριστό στιγμιότυπο ερωτήματος, στην κύρια μνήμη. Βέβαια αυτή η τεχνική δεν είναι και πολύ αποτελεσματική, εκτός και αν υποβάλεται συχνά ακριβώς το ίδιο ερώτημα. Οπότε παρόλο που αυτή η τεχνική μπορεί να αποδειχθεί αποτελεσματική για συγκεκριμένα φορτία δεν είναι γενικά αποδοτική. Η προανάκληση προϋποθέτει ότι το σύστημα μπορεί να προβλέψει μελλοντικά ερωτήματα, τα οποία ενδεχομένως να συσχετίζονται με ερωτήματα που έχουν προηγηθεί. Η πρόβλεψη μελλοντικών ερωτημάτων σε μία σχεσιακή ροή είναι αρκετά περισσότερο περίπλοκη διαδικασία συγκρινόμενη με άλλα εκλεπτυσμένα πληροφοριακά συστήματα, όπως για παράδειγμα ένα κατανεμημένο σύστημα αρχείων. Αυτό οφείλεται στο γεγονός ότι οι σχεσιακές επερωτήσεις χαρακτηρίζονται από παραμέτρους εισόδου (που ενδεχομένως διαφοροποιούνται σε κάθε εκτέλεση) και αποτελέσματα εξόδου (που πιθανότατα διαφοροποιούνται σε κάθε ανάκληση πλειάδας). Οποιαδήποτε συσχέτιση ανάμεσα σε δύο διαφορετικές επερωτήσεις είναι περισσότερο δύσκολο να εντοπιστεί, μια και απαιτεί εκτεταμένες συγκρίσεις μεταξύ όλων των πιθανών παραμέτρων εισόδου και τιμών εξόδου.

Πρόσφατες πρόοδοι στο υλικό και στο λογισμικό των τηλεπικοινωνιακών δικτύων όσον αφορά τη διέλευση δεν έχουν συνοδευτεί από αντίστοιχη πρόοδο στη μείωση της καθυστέρησης που γίνεται αντιλήπτή στον τελικό χρήστη. Σε αυτή τη διπλωματική εργασία επιχειρούμε να αντιμετωπίσουμε αποτελεσματικά το πρόβλημα της καθυστέρησης που γίνεται αντιληπτή από τον τελικό χρήστη. Μελετήσαμε ενα συγκεκριμένο πρότυπο ροών σχεσιακών επερωτήσεων που συναντάται αρκετά συχνά στις περισσότερες εφαρμογές και χαρακτηρίζεται απο έντονο συσχετισμό των ερωτημάτων και σημαντικό βαθμό εμφωλιασμού. Επιβεβαιώσαμε ότι αυτοί οι δύο παράγοντες έχουν ως αποτέλεσμα ένα μεγάλο αριθμό από round-trips, το αποία μπορούν να αποφευχθούν είτε με αλλαγές μέσα στον κώδικα των ίδιων των εφαρμογών, είτε με κάποια δυναμική τροποποίηση και συνδυασμό των επερωτήσεων, στον αέρα, καθώς η ροή στέλνεται για εκτέλεση στη βάση δεδομένων. Παρόλο που η χειροκίνητη αλλαγή του κώδικα μπορεί να είναι πολύ αποδοτική, μια και θα συνδυάζονται 2 ή περισσότερες επερωτήσεις σε μία, δεν ενδείκνυται, γιατί τις περισσότερες φορές ακυρώνει καλές τεχνικές ανάπτυξης λογισμικού. Επιπλεόν, μία εξειδικευμένη βελτιστοποίση μπορεί να μην είναι το ίδιο αποδοτική για μία διαφορετική ρύθμιση του λογισμικού ή για μία διαφορετική κατανομή δεδομένων στη βάση.

Έχουμε αναπτύξει μία βιβλιοθήκη σε Java που επιτρέπει αποδοτικούς συνδυασμούς δύο ή περισσότερων επερωτήσεων σε μία, η οποία υποβάλλεται εναλλακτικά στη βάση δεδομένων. Ουσιαστικά αυτή η βιβλιοθήκη λειτουργεί σαν ένας wrapper γύρω από την εκάστοτε JDBC βιβλιοθήκη του κατασκευαστή της σχεσιακής βάσης δεδομένων. Η βιβλιοθήκη αυτή λειτουργεί σε δύο καταστάσεις. Στην πρώτη φάση, λειτουργεί σε κατάσταση "εκπαίδευσης", δηλαδή παρατηρεί τις αρχικές επερωτήσεις και τις καταγράφει μαζί με επιπρόσθετη μετά-πληροφορία σε ένα n-ary δένδρο, το οποίο ονομάζουμε "context tree". Η πρώτη φάση, κατά προσέγγιση αντιπροσωπεύει το 10% του συνολικού χρόνου της λειτουργίας της επιχειρησιακής εφαρμογής. Μετά το τέλος της πρώτης φάσης, αυτή η βιβλιοθήκη αποφασίζει ποιες αρχικές επερωτήσεις μπορούν και αξίζει (με κριτήριο τη μείωση της καθυστέρησης) να συνδυαστούν μεταξύ τους. Στη δεύτερη φάση που ονομάζεται κατάσταση "κανονικής" λειτουργίας και που συνήθως είναι το 90% του χρόνου, αυτή η βιβλιοθήκη χρησιμοποιεί τις εναλλακτικές, συνδυαστικές επερωτήσεις που παρήγαγε η προηγούμενη φάση για να τις υποβάλλει στη βάση δεδομένων στη θέση των αρχικών απλών επερωτήσεων. Η επανεγγραφή πραγματοποιείται στον αέρα καθώς το σύστημα είναι σε κανονική λειτουργία. Παρόλο, που οι παραγόμενες επερωτήσεις είναι πιο πολύπλοκες, αποκαλύπτουν στον επεξεργαστή επερωτήσεων της σχεσιακής βάσης δεδομένων περισσότερες ευκαιρίες για βελτιστοποίηση. Διαφορετικά αυτές οι βελτιστοποιήσεις θα παρέμεναν κρυμμένες και ανευκμετάλλευτες μέσα στην εφαρμογή. Επιπλέον, έχουμε αναπτύξει ένα μοντέλο για τη κοστολόγηση όλων των προτεινόμενων τεχνικών επανεγγραφής που μας επιτρέπει να τις συγκρίνουμε μεταξύ τους και να λαμβάνουμε υπόψιν συστημικές παραμέτρους, όπως τη μέση καθυστέρηση του δικτύου. Η διαδικασία της κοστολόγησης των διαθέσιμων εναλλακτικών και της επιλογής του βέλτιστου σχήματος για την επανεγγραφή μίας ροής επερωτήσεων, πραγματοποιείται μετά το πέρας της "εκπαιδευτικής" φάσης και πριν την έναρξη της "κανονικής" λειτουργίας. Τέλος, εφαρμόσαμε μία εκτεταμένη σειρά από ελέγχους απόδοσης, ώστε να καταγράψουμε τις βελτιώσεις στο πρόβλημα της δικτυακής καθυστέρησης, που γίνεται αντιληπτή στον τελικό χρήστη. Όλες οι εναλλακτικές στρατηγικές επανεγγραφής αποδείχθηκαν πιο αποδοτικές από το αρχικό σχήμα επερωτήσεων από 2 έως και 4 φορές! Παρόλαυτά, οι προτεινόμενες στρατηγικές δεν είναι σε όλες τις περιπτώσεις πιο αποδοτικές. Διαπιστώσαμε, ότι σε ορισμένα δίκτυα που η μέση καθυστέρηση του δικτύου είναι πολύ μικρή, οι εναλλακτικές επερωτήσεις μπορούν να είναι πιο αργές από τις αρχικές! Αυτή η διαπίστωση μαζί με πλήθος άλλων συμπερασμάτων παρατίθενται και ερμηνεύονται αναλυτικά στο αντίστοιχο κεφάλαιο των πειραματικών ελέγχων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Σχεσιακές Βάσεις Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: σημασιολογική ανάλυση σχεσιακών επερωτήσεων, σημασιολογική βελτιστοποίηση σχεσιακών επερωτήσεων, επανεγγραφή σχεσιακών επερωτήσεων, ροές σχεσιακών επερωτήσεων, επεξεργασία σχεσιακών επερωτήσεων

ABSTRACT

Current multi user applications submit streams of relational queries against a back end database server. For many years the research community has focused its attention in developing sophisticated storage engines, more efficient query processors, scalable clustering systems, in main memory key-value caching servers that would alleviate any throughput bottlenecks and could allow for more concurrency. Query streams initiated by individual users have received some attention in the form of result set caching. However, unless the same query is resubmitted, this remedy has not proved very efficient on minimizing the latency perceived by the end user. In addition, improvements in network latency have lagged developments in bandwidth usage. In this thesis, we attempt to tackle the latency experienced by the end users, which in contrary to the throughput remains a big issue and there does not seem that any networking hardware improvements will alleviate it in the future. We have studied a specific pattern of query streams that is quite often found in most applications and is characterized by a number of query correlations and deep nesting. We verified that these two factors result in excessive numbers of round-trips, which could be avoided with either manual rewriting of the queries or whith some form of runtime rewritings on the fly. Although, the manual rewriting of these applications could result in much more efficient queries, it is not recommended as it is not always clear for which system configuration or application instance we should optimize. Furthermore, good sofware engineering practices promote code modularity and encapsulation, with more simple queries. We have developed a prototype software library, which allows for run time optimization of these query streams. We have implemented a number of alternative query rewritings that are applied during run time and essentially submit at the back end RDBMS a combined query. This combined query although more complex, reveals to the RDBMS query processor more scope for optimization, which otherwise would have remained hidden within the client application code. In addition, we developed an analytic cost model that allows us to compare different alternatives and at the same time take into account any critical system properties like network communication latency. We performed comprehensive benchmarking so as to measure any improvements, on the total latency, seen by the end user. Finally, we present experimental results where all the alternative strategies outperform the orignal queries by 2 to 4 times.

SUBJECT AREA: Relational Database Management Systems

KEYWORDS: semantic analysis of relational queries, semantic optimization of relational queries, relational query rewriting, streams of relational queries, query processing of relational queries

For my mother, Eftihia Sirma

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Alex Delis for his honest support during my graduate studies. He has helped me with his mentorship not only during the thesis period but during the entire period of my presence at the computer science department as well. His guidance was invaluable in a number of semester projects and the thesis itself. I am grateful to him for his support, patience, commitment and dedication which many times where the major factors that motivated me to explore and study in depth challenging problems. His encouragement and enlightening observations, comments during our discussions played a key role for the successful completion of this thesis.

I would also like to thank my beloved mother Eftihia Sirma who has supported me all these years unconditionally with love, patience, persistence. It is more than certain that if it was not for her I would not have achieved many of my goals today. It is the person who has sacrificed numerous personal things so as to support me during my life. She has set a model of behavior for me. She has encouraged me all these years and equipped me with a hard-working personality. Her persistence, commitment and dedication are only some characteristics that has cultivated in my personality. I will always be grateful for her protection, support and love.

In addition, I would like to thank my fiancé Aggeliki Nika for her patience, support and love all this stressful and hard-working period. She has been a source of strength and stability in my life. She endured many late nights, working weekends, working holidays. It is her support and love that encouraged me to keep working hard with commitment and dedication.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ΠΡΟΛΟΓΟΣ

Η παρούσα διπλωματική εκπονήθηκε στο Τμήμα Πληροφορικής και Τηλεπικοινωνιών, του Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών (ΕΚΠΑ). Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον επιβλέποντα Καθηγητή κ. Αλέξανδρο Δελή για την ειλικρινή υποστήριξη του κατά τη διάρκεια των μεταπτυχιακών μου σπουδών στο τμήμα. Κατά τη διάρκεια των σπουδών μου υπήρξε πολύτιμος σύμβουλος, μέντορας και φίλος. Η καθοδήγηση του δεν περιορίστηκε μόνο στη διπλωματική μου εργασία αλλά υπήρχε και ήταν καθοριστική σε όλες τις προγραμματιστικές και μελετητικές εργασίες των μαθημάτων του. Ως σύμβουλος Καθηγητής της 3 κατεύθυνσης, στην "Τεχνολογία Συστημάτων Υπολογιστών", έπαιξε καταλυτικό ρόλο στην επιλογή μαθημάτων που με βοηθήσανε να διευρύνω τόσο τους επαγγελματικούς όσο και τους ερευνητικούς μου ορίζοντες. Θα είμαι πάντα ευγνώμων για την αμέριστη υποστήριξη, την υπομονή, την υπευθυνότητα και την αφοσίωση του. Πολλές φορές αυτά τα στοιχεία ήταν που μου έδωσαν το κίνητρο να ερευνήσω και να μελετήσω σε βάθος δύσκολα προβλήματα. Η ενθάρρυνση, οι παρατηρήσεις και τα σχολιά του κατά τη διάρκεια των συζητήσεών μας έπαιξαν καθοριστικό ρόλο στην ολοκλήρωση αυτής της διπλωματικής εργασίας με επιτυχία.

Επίσης, θα ήθελα να ευχαριστήσω την πολυαγαπημένη μου μητέρα Ευτυχία Σύρμα, φιλόλογο, που με έχει υποστηρίξει όλα αυτά τα χρόνια με πολύ αγάπη, υπομονή και επιμονή. Είναι βέβαιο ότι χωρίς την υποστήριξη και την καθοδήγηση της δεν θα είχα επιτύχει πολλούς από τους στόχους μου σήμερα. Έχει θυσιάσει αναρίθμητα προσωπικά πράγματα για να με υποστηρίξει αποτελεσματικά κατά τη διάρκεια της ζωής μου. Έχει θέσει ένα μοντέλο συμπεριφοράς στο οποίο προσπαθώ να ανταποκριθώ. Η μητέρα μου είναι που με έμαθε να εργάζομαι σκληρά και ήταν εκεί κάθε φορά που τη χρειάστηκα για να με ενθαρρύνει. Η επιμονή, η δέσμευση και η αφοσίωση είναι μόνο μερικές από τις πολύτιμες προσωπικές της αρετές που έχει εμφυσήσει και σε εμένα. Θα της είμαι για πάντα ευγνώμων για την προστασία, την υποστήριξη και την αγάπη της.

Επιπλέον, θα ήθελα να ευχαριστήσω ολόψυχα την αρραβωνιαστικιά μου Αγγελική Νίκα για την υπομονή, την υποστήριξη και την αγάπη που μου πρόσφερε γενναιόδωρα κατά τη διάρκεια αυτής της κοπιαστικής περιόδου των μεταπτυχιακών μου σπουδών. Η Αγγελική αποτελεί πηγή δύναμης και σταθερότητας στη ζωή μου. Υπομονετικά άντεξε υπερωρίες, σαββατοκύριακα και διακοπές γεμάτα με αγχωτική και κοπιαστική δουλειά. Ήταν η υποστήριξη της και η αγάπη της που με ενθάρρυναν να συνεχίσω να δουλεύω σκληρά με υπευθυνότητα και αφοσίωση.

1. INTRODUCTION

A large part of the current enterprise and web applications involve the use of persistent storage. Most of the times this data persistency and accessibility is ensured by a Relational Database Management System (RDBMS). Current database driven web applications and enterprise grade solutions (e.g. ERPs, CRMs, CMSs, BPMs) submit streams of relational queries against a back-end DBMS either directly through some database library API (e.g. ODBC, JDBC) or indirectly through some middleware (e.g. an application server). Most of these applications suffer from both throughput throttling and high latencies. High execution loads stress both the underlying storage engines and any existing concurrency managers as well. The last decades have witnessed an overwhelming research on both relational and object storage systems. For many years researchers have been developing sophisticated algorithms so as to boost the performance of database kernels in various aspects. These optimizations have targeted efficient read, write access and query plan enumeration, evaluation. Hardware manufacturers have significantly improved the speed and throughput of CPUs, hard disks and both terrestrial and wireless networks. Multi-core CPUs, cheap and large main memories, SSD disks, deeper memory hierarchies, larger caches available even in commodity hardware have enabled many business to handle heavy query loads and avoid significant throughput bottlenecks. All these continuously improved hardware components when deployed in data centers have allowed for the efficient execution of query streams, initiated by a large population of users. Although, hardware advancements have alleviated many of these problems, it seems that the latency issue remains and it will not be easy to manage it efficiently. The network throughput is increasing all the time with the advent of new wired and wireless technologies, however this is not the case for network latencies. Improvements in network latency have lagged developments in network bandwidth usage and throughput. In this thesis we attempt to tackle the latency issue which in contrary to the throughput issue remains and there does not seem that any networking hardware improvements will alleviate it effectively.

The database research community has employed techniques based on either caching or prefetching so as to speed up certain types of queries. Caching presumes that the very same queries will be resubmitted in the near future. Although, this solution may work for some specific workloads it is not generally effective on all the observed workloads. Prefetching assumes that the system can predict future requests that may be possibly correlated with previous queries. Predicting future queries in a relational stream is more complicated compared with other software systems, like prefetching disk blocks in a distributed file system. This is because queries are consisted of input parameter bindings and output attributes. Any possible correlation between two different queries is more difficult to be tracked as it requires extensive comparisons between all the possible correlation sources.

Our study is heavily based on the work conducted by Ivan Bowman and Kenneth Salem at [3], [4], [5], [6], [7]. Their research focused on observing various query streams and the source code that produced them. They identified repetitive patterns that occur in a variety of data base driven applications. In this thesis we chose to address one of those patterns, which they called "Nested Request Patterns". We implemented from scratch a prototype, in Java, that loosely resembles their prototype and through comprehensive and exhaustive benchmarks we tried to verify their findings. Our prototype is called Sqlprefetch and it is a wrapper around JDBC that is provided as a library to any Java application that makes use of some RDBMS vendor's JDBC library.

Sqlprefetch is a library that sits between the client application and the RDBMS specific database access API library. It intercepts all the submitted queries and tracks all input and output values. This way it can identify possible correlations and optimize a number of queries during runtime. A key concept of this mechanism is that of the context. It is very important for Sqlprefetch to be able to recognise and distinguish uses of the same query under different situations. Knowing the specific context under which a query is submitted it is of major importance if we are to either alter it or combine it with another query so as to speed up the execution of the entire stream.

Since, Sqlprefetch must understand the submitted queries and distinguish the different contexts that they may occur, the aforementioned researchers have chosen to call this technique "Semantic Prefetching of Relational Query Streams". This technique is mainly based on the combining of individual queries into larger ones, that return clustered results sets to the Sqlprefetch library. Then, these clustered result sets are appropriately traversed so as to return the expected result set values to the corresponding individual requests. A number of alternative techniques are proposed and their corresponding costs are evaluated during a training mode. The developed cost model takes into account various information that spans from selectivity predicates to more dynamic parameters like the experienced network latency of the specific application instance. This way Sqlprefetch takes the current network configuration into account. For example, the optimizations that Sqlprefetch may suggest could differ between a LAN and a WAN. This critical feature consists Sqlprefetch appropriate for a variety of different configurations.

Most database driven applications issue fine grained SQL queries on the back end RDBMS. A simplified time modeling of a database request is given at figure 1 on page 16. A fine grained query will spend most of its time on the communication round-trip rather than on the server itself. The communication latency depends on the deployed network setting, i.e. for a WAN it will be significantly increased compared to a LAN. If a large number of queries is issued against a database then this communication overhead becomes an important performance bottleneck.

We attempt to address these query streams with a number of runtime optimizations that function completely transparently to the application programmer. Additionally, we perform a number of tests that have helped us to identify the advantages and disadvantages of

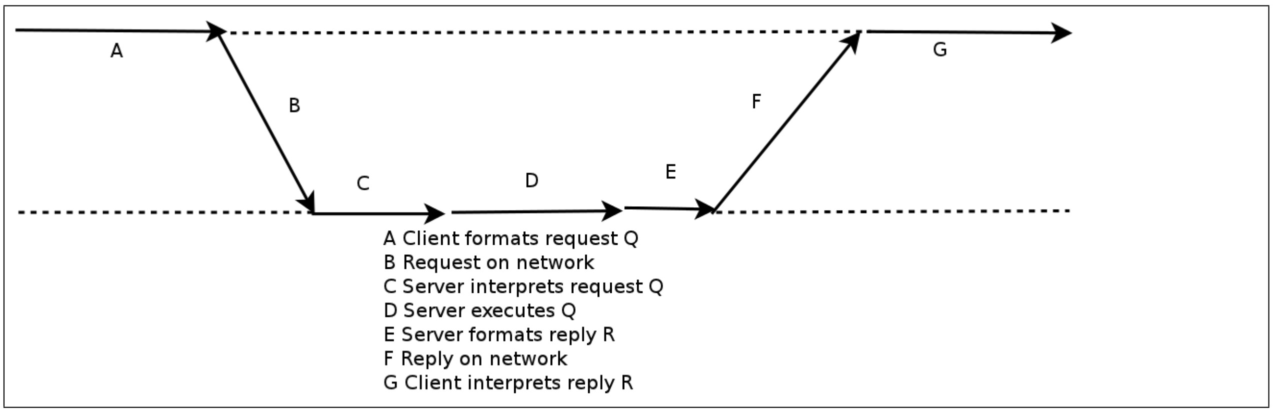

Figure 1: A Sample Communication Trace

the proposed solution. At figure 2 on page 16 a simplified pseudo-code example is given, that exhibits a commonly found characteristic in many database driven applications. As can be seen the code is written in a modular fashion where each method performs a very specific task. Each method issues a fine-grained SQL query that possibly accesses a relatively small number of tuples on a single relation. If each outer tuple results in more that one fetched inner tuples, then a query stream of the form Q_1, Q_2, ..., Q_2 is submitted to the database. Such a query stream pattern involves many unnecessary round-trips that consist the overwhelming majority of the stream's cost. Especially, when the base relations have been cached into main memory.

```
method1(value)
  sql = "select column1, column2
         from table1
         where column1=" + value;
  conn = connect(db);
  cursor = conn.execute(sql);
  while(cursor.next())
    if(column1 <> some_value &&
       global_conf_param <> some_runtime_param)
       method2(cursor.getValue("column2"));
  cursor.close();
  conn.close();

method2(value)
  sql = "select attr1, attr2, attr3
         from table2
         where attr1=" + value;
  conn = connect(db);
  cursor = conn.execute(sql);
  while(cursor.next())
    # Process the returned result set.
  cursor.close();
  conn.close();
```

Figure 2: Nested Query Pattern Example

A query stream pattern of the form Q_1, Q_2, ..., Q_2 could be addressed with an aggressive code refactoring where the two very specific methods could be integrated into one where a combined query is issued, like the one at figure 3 on page 17. If necessary we could keep the initial methods and just add the 3 integrated method.

```
method3(value)
  sql = "select column1, column2
       from table1 left inner join table2
       on table1.column1 = table2.attr1
       where column1=" + value;
  conn = connect(db);
  cursor = conn.execute(sql);
  while(cursor.next())
    # Process the returned result set.
  cursor.close();
  conn.close();
```

Figure 3: Optimized Nested Query Pattern Example

The addition of the combined method breaks the already established functional encapsulation. The code is cluttered into function overlapping methods and the unit testing suite is complicated with more unintelligible tests. Thus, the code becomes more unmaintainable and inextensible.

Another point that we observed was the "predicates" that control whether the nested method is executed or not. The "if" boolean clause consists of two branches glued together with an "and" logical operator. The first branch may exhibit different results for each fetched outer tuple, while the outer method's result set is traversed. In contrast, the second branch should probably return the same result for a specific installed instance of the application. Thus, our application's behavior, i.e. whether the nested method (query) is executed all the times, some times or not at all depends on the aforementioned selectivity predicates. For different data value distributions of table "table1" and for different installed instances these selectivity predicates may vary greatly. If the nested method is not executed most of the times then the initial modular design is preferable. However, if the nested query is executed most of the times then a method like "method3" would be more efficient. For which of these two cases should we develop our application? The answer is that "we do not know". Therefore, any manual optimizations like "method3" do not justify the cluttering of the code. This is where a runtime optimization comes in. If we could track this nested query pattern and record not only the actual predicate selectivities but also the available memory, CPU, network latencies we could apply runtime optimizations, like issuing a combined query on-the-fly, fully transparently to the application developer.

Furthermore, if the nested query is submitted for execution most of the times, then the application essentially performs a nested loops join, on the client side. For the aforementioned reasons this is most of the times inefficient and hides from the back end database server the actual join. Thus, the selection of a runtime optimization, like the combined query, may reveal to the database server more scope amenable to further optimizations. The database optimizer may be able to choose a more efficient plan than a nested loops join. For example, the query optimizer may use common subexpressions like [1] or a query plan that makes extensive use of pipelining and parallelization. This more efficient execution may allow for more efficient usage of the server resources and increase the throughput of concurrently executed queries.

In the rest of the thesis we have explored techniques that allow effective and efficient runtime optimizations. The proposed techniques have been evaluated experimentally through a common benchmarking suite. This suite has been developed with all the alternative methods in mind. All the proposed methods are applicable on this suite and their benchmarking has allowed as to compare them and draw very useful conclusions.

All the aforementioned optimization techniques along with various other features have been implemented as a library like engine. This engine is called Sqlprefetch. Although out prototype is inspired by the work of [4], [5] it is implemented from scartch and bears some major differences with their implementation. Sqlprefetch is written in Java. In addition, a number of synthetic benchmarks have been added that allowed as to observe critical performance features of the proposed solution. A number of technical details regarding Sqlprefetch have been given at chapters 6, 7 on pages 46, 49.

2. NESTED PATTERN DETECTOR

Sqlprefetch operates in two modes, a so called "training" mode and a so called "running" mode. Initially, Sqlprefetch is set to function in the training mode. Under the training mode Sqlprefetch identifies the various nesting patterns that exist in the monitored application and materializes them in a data structure called context tree. This context tree is essentially an n-ary tree where the nodes represent the contexts and the edges represent the queries that result to any child contexts. Each context tree node is accompanied by a large number of attributes that serve either operational or cost related operations. The context tree and its associated metadata are described in detail in the rest of this chapter.

2.1 Context Tree

While operating under the training mode each JDBC application request is intercepted with the help of the corresponding JDBC training API wrapper. The currently submitted SQL query is parsed, analyzed and stored in a custom data structure. Each context contains an attribute that points to this data structure. Along with the context tree, Sqlprefetch maintains a simple linked list of the queries that have been intercepted so far. If a query is resubmitted in another context then it is not duplicated in this globally visible query list. In this case there may be two or more contexts that point to the same internal object that represents the query. Once a query is encountered for the very first time, a new context is created as a child of the current context tree node. If at some point this query is resubmitted in the same parent context, then the already existing corresponding context is revisited so as to update any necessary bookkeeping information. If a previously submitted query is resubmitted under a different parent context then a new context tree node is created so as to distinguish between its multiple uses. Each context tree node is uniquely identified. The goal is to allow Sqlprefetch to accurately follow any query requests during the running mode and act as prescribed in its query plan. In essence, a context tree node can be uniquely identified by the ordered list of queries that are open when a request is submitted.

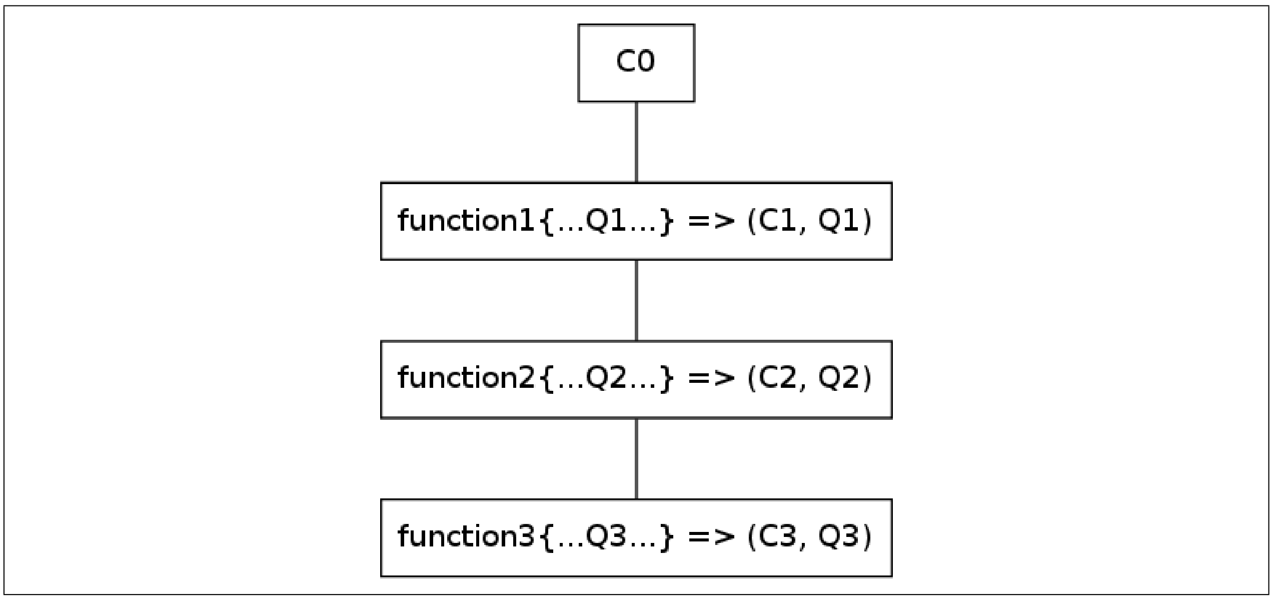
C0
function1{...Q1...} => (C1, Q1)
function2{...Q2...} => (C2, Q2)
function3{...Q3...} => (C3, Q3)

Figure 4: Example of a Context tree

An example of a context tree is given at figure 4 on page 21. Each function in the figure represents a method in the application code, that submits a query to the back end RDBMS. Each function, loops over the returned result set of its submitted query and for each fetched tuple, invokes its child method which in its turn follows the same nested pattern. In many cases, there is an "if" applciation predicate between each "fetch tuple" request and the invocation of inner function.

2.2 Tracking Parameter Correlations

During the training mode, Sqlprefetch tracks, for each context-query pair, both the lastly fetched tuple in an object attribute called "lastoutput" and the lastly used input parameter values in an object attribute called "lastinput". Each time a context tree node is created its possible correlation sources are recorded so as to be monitored each time this context is revisited. For example, if a context is at depth 3, i.e. it has two ancestor contexts, then upon its instantiation these two ancestor contexts are included in the possible contexts correlation list. This list defines the correlation scope of the current context. The "lastinput" attribute of the current context at depth 3 is compared with the "lastoutput" and "lastinput" attributes of its ancestor contexts. This comparison will result in any initially holding correlations. Subsequent revisits to the same context will allow Sqlprefetch to verify these correlations and discard any of them that do not hold anymore.

2.2.1 Correlation Detection Overhead

Unfortunately, the previously correlation detection algorith is featured by a polynomial cost function, regarding the necessary number of comparisons, as can be seen by equation 1.

$$O(D^2P(C+P)) \tag{1}$$

The following list describes each term of equation 1 in detail.

- D is the average context tree depth.
- C is the average number of projected columns.
- P is the average number of input parameters.

However, for all the benchmarks that we have run, the correlation detection overhead was not prohibitive. As is also noted by [4], [5] this overhead remains affordable for the vast majority of the real world applications. In a production system, where resources are limited and there is a demand for speed, a more sophisticated algorithm could be used that eliminates the polynomial nature of this overhead.

2.3 Application Predicates Selectivity

In the code example of figure 2 at page 16 we notice an if statement that acts as a filtering predicate for the inner query. This so called "application predicate" is a boolean expression that consists of two parts glued together with and "AND" logical operator. The first part of this boolean expression depends on the currently fetched tuple's specified attribute. This implies that the execution or not of the inner query depends on the fetched values of each outer tuple. This kind of predicate selectivity is accounted for in our cost model that is explained in greater detail at chapter 5 on page 37. Essentially, what we need to consider for this predicate is its selectivity, i.e. the ratio of the times that the inner query is submitted divided by the number of tuples fetched by the parent query. This fraction is the probability that the inner query will be submitted. In order for this probability to range between zero and one, we have made the assumption that each inner query is opend at most once for each outer row. This probability plays an important role in our cost model. For example, if this probability is close to one, then for each outerly fetched tuple it is almost certain that the inner query will be submitted. In this case, our query optimizer may consider as a possible rewriting the combination of the two queries. On the other hand, if this ratio is close to zero, then the optimizer is informed that the inner query is not submitted most of the times and it may consider the initial nested strategy as the most appropriate one for the runtime mode.

Another ratio that is being tracked during the training mode is the times that the inner query is opened for each open of the parent query. For example, if the inner query opens for each outer query open (not fetch) then the resulted ratio will be 1. If the inner query is opened for half of the times of the outer query opens, then this ratio is 0.5. This probability depends strongly on the second part of the boolean expression. The second part of the boolean expression represents either different application instances or different application configurations. This ratio is used by the optimizer for one the proposed rewritings so as to estimate its efficiency. For example, if this ratio is close to zero, then this means that the inner query is generally not submitted at all, regardless of the values of the outerly fetched tuples. In this case, one of the proposed strategies (client hash join) will save Sqlprefetch from submitting a combined query that will do more unnecessary work. Submitting a combined query is more expensive regarding both the client and the server side. The less round trips (decreased communication latency) is the benefit of the combined query. But if the inner query is not submitted most of the times then avoiding the submission of a combined query at the parent will save the overhead of prefetching unneeded result sets. A combined query is more complex to execute and involves more CPU usage at the back end RDBMS. In addition, it involves more CPU usage at the client side, for the interpretation of the combined result set. Thus, if an inner query is only rarely submitted, then the optimum strategy is to use the initial fine-grained pattern, than to employ a combined query that will cause more CPU overhead at both the client and the server side.

The above probabilities play a crucial role during the query plan optimization phase. Their importance will be more clear at chapter 3 on page 24, where the alternative query rewriting strategies are discussed in great detail.

3. QUERY REWRITING STRATEGIES

3.1 Nested Execution

By the term "nested execution" it is meant the submission of the application queries without any itervention and modification, i.e. as they were originally written in the application source code. Each resubmission of the same query is characterized by the same cost as the very first time. It is apparent that the number of inner query opens is proportional to the tuples fetched by their immediate ancestor query. The proportional relationship of the outer query's number of fetched rows and the inner query's openings entails that this proportionality accounts for the corresponding round trips. The more the outerly fetched tuples, the more the inner query openings and finally the more the round trips that will occur. However, in application parts where any predicates prevent the execution of the inner queries most of the times, the nested execution strategy is possibly the optimimum one. This is because the inner query will be submitted very few times and the cost model has concluded that the overhead of the combined query is larger than the cost of the original nested operation. However, in the case where the inner queries are submitted most of the times then an alternative strategy is possibly the best one to employ. These, alternative strategies are comprehensively discussed and presented in the following sections.

3.2 Unified Strategies

The most simple way of combining a parent query with a child query is to unnest their relationship and either "inner" or "outer" join them, like the example at figure 5 on page 25. Even with this simple join rewriting the application may benefit from the fact that a single query will be submitted and therefore less round-trips will occur. This way the application client join is moved to the backend RDBMS server, where its excution will be likely more efficient. All, the following SQL queries have been colorized so as to facilitate the effective understanding of their semantics. In a parent-child relationship the parent (outer) query has been colorized with the blue color. The child (inner) query has been colorized with the red color. The parts of the combined query that have been colorized with the cyan color represent the correlated parts of the parent query with its children queries. A correlation is identified as an **"output correlation"** when some projection list attribute of the parent query is used as an input parameter value by a child query. Furthermore, a correlation is identified as an **"input correlation"** when some input parameter value of the outer query is also used as an input parameter value by a child query. Finally, a correlation is identified as a **"constant correlation"** when some input parameter binding of a child query is always fed with the same value.

```
SELECT t1.column1, t1.column2, t2.column1, t2.column2
FROM
t1 LEFT [INNER|OUTER] JOIN t2
ON t2.column1 = t1.column1
WHERE t1.column1 = 1 AND t2.column2 <= 10;
```

Figure 5: Flattened Example Query

However, a simple join like this is still hiding from the database server the nesting relationship of the initial queries. Here we could add the fact the many times this nesting includes some form of value correlation. Correlated queries have been studied exhaustively by the database academic community. In our case, we could benefit even more if we could submit to the backend database server a combined join query that keeps the nested correlation relationship of the parent-child queries. Conveying to the database server intact the application semantics we provide to its query optimizer more scope. Modern query optimizers are equipped to understand nested correlated queries and optimize them as much as possible.

An alternative way of combining the parent-child queries and keeping the nested correlation semantics is that of figure 6 on page 25. Unfortunately, this query is not legal, as it includes an outer reference (p.column1) inside the inner query and outside the outer query's scope.

```
SELECT p.column1, p.column2, c.column1, c.column2
FROM
(
    SELECT column1, column2
    FROM t1
    WHERE column1 = 1
) AS p
(
    SELECT column1, column2
    FROM t2
    WHERE column2 <= 10 AND
    column2 = p.column1
) AS c
```

Figure 6: Non Legal Example Query

Shanmugasundaram [8] proposed a combined rewriting that keeps the nested, correlated information. An example of his proposal is given at figure 7 on page 26. However, this suggestion works only for cases where the correlation value is used inside the ON clause. If the correlation, is used inside some other, possibly more complicated expression, it does not work at all.

```
WITH p AS
(
    SELECT column1, column2
    FROM t1
    WHERE column1 = 1
) p,
c AS
(
    SELECT column1, column2
    FROM t2
    WHERE column2 <= 10
) AS c
SELECT p.column1, p.column2, c.column1, c.column2
FROM p LEFT JOIN c ON (c.column2 = p.column1)
```

Figure 7: Shanmugasundaram's suggestion, used outer (correlated) references within the ON condition

SQL99 [9] introduced a new concept called **lateral derived tables**. The new **LATERAL** keyword allows the use of an outer reference inside the inner query which is now considered a derived relation. A very abstract example of the lateral derived tables is given at figure 8 on page 26.

The initial use of this new feature in SQL was not only to allow for the use of outer refences inside a nested correlated query but at the same time to provide to the RDBMS query optimizer more scope for optimization. We could say that the LATERAL keyword acts as a query optimization hint that instructs the optimizer on how it should consider the execution of the query. An example of a lateral inner join is given at figure 9 on page 27.

The use of the LATERAL keyword informs the RDBMS optimizer that the inner query, semantically, needs to be re-evaluated for each fetched tuple of the outer query. Sqlprefetch employes the lateral derived tables so as to generate a single combined SQL query that submits to the back end RDBMS instead of two or even more queries (this is the case where a parent query has more than 1 children queries). This way, Sqlprefetch conveys to the database optimizer the application semantics. For this thesis we have considered 3 commercial RDBMS. One of them implements the LATERAL concept as specifed by the ISO organization. The other two have implemeted the same concept with distinct syntax. The final prototype has been completed with ONLY one of these RDBMSs and some representative benchmarking results have been presented on chapter 7.

The SQL standard has not catered for the quite common case where the outer query's

```
SELECT <parent.proj-list>, <child.proj-list>
FROM
<parent-query> LATERAL <child-query>
ORDER BY <parent-order-by>
```

Figure 8: Lateral Derived Relations

```
SELECT c1.column1, c1.column2, c2.column1, c2.column2
FROM
(
  SELECT t1.column1, t1.column2
  FROM table1 as t1
  WHERE
  t1.column1 >= 1 AND t1.column1 <= 5
) c1 lateral
(
  SELECT t2.column1, t2.column2
  FROM table2 as t2
  WHERE t2.column2 <= 10 AND t2.column1 = c1.column1
) c2
ORDER BY c1.column1;
```

Figure 9: Unified Lateral Inner Join

tuples that do not match with an inner tuple must be preserved. This case corresponds to an outer join. Tow of the three RDBMSs, that use distinct product specific syntax, have foreseen this need and have implemented a small variation that can handle the case where the outer query's tuples must be preserved. As expected, the outer tuples that have not been joined with 1 or more inner tuples are accompanied by NULL values in the corresponding attributes of the combined result set. The third RDBMS does not directly support the **LEFT OUTER LATERAL** concept, but even in this product we can workaround this limitation by using the lateral combined query into a standards compliant SQL, like a union for example.

When a unified execution strategy is used, a LATERAL combined query is submitted only by the parent context tree node and returns a combined result set. The combined result set contains the expected result set of the outer query, appropriately clustered with the result sets of the participating child queries. It is important to mention that a unified query is always submitted when the parent context issues an open request. Any participating child queries do not issue an actual query when they submit an open request. All these are performed transparently to the client application code. On the next sections we will present two rewriting strategies that we have considered and the conditions that must be held, so as to consist them applicable.

3.2.1 The Outer Join Strategy

Sqlprefetch produces a combined query by using the "LEFT OUTER LATERAL" concept. This is done so as to ensure that all the rows of the original outer query are included in the result. The rewriting procedure appends any initial "ORDER BY" clause of the outer query to the rewritten one. A generalized form of the outer join rewiting is given at figure 10 on page 28.

```
SELECT <parent.proj-list>, <child.proj-list>
FROM
<parent-query> OUTER LATERAL <child-query>
ORDER BY <parent-order-by>
```

Figure 10: General Lateral Outer Join Example

At figure 11 on page 28 a more specific example is given that may clarify some obscure details of this rewriting strategy.

```
SELECT c1.column1, c1.column2, c2.column1, c2.column2
FROM
(
  SELECT column1, column2
  FROM table1
  WHERE column1 >= 1 AND column1 <= 5
) c1 OUTER LATERAL
(
  SELECT column1, column2
  FROM table2
  WHERE column1 = c1.column1
) c2
ORDER BY c1.column1;
```

Figure 11: Lateral Outer Join Example

The function that produces the combined query (i.e. performs the rewriting) operates on a single node in the context tree at a time. Before the rewriting phase is launched the context tree is traversed and an Sqlprefetch pre-optimization function, chooses the most appropriate correlation (if more that one correlations have held). For the purposes of the current thesis, we considered that because the most interesting correlation was the output correlation, if there were more than one correlations and one of them was an output correlation, we chose that one. This way we could stress the usage of the LATERAL derived tables to the maximum. One of the objectives of this thesis is to evaluate the usefulness of the LATERAL derived tables and conclude whether its usage is mere syntactic sugar or it serves a more essential purpose when it comes to query optimization. Afterwards, the context tree is traversed and each node is rewritten, if applicable.

Sqlprefetch applies the join strategy when it has inferred that the inner query (or queries) will return at most one tuple for each tuple of the outer query (parent context). Sqlprefetch uses a specifically constructed method that concludes whether the aforementioned criterion holds. Although, this method can produce false negatives (i.e. that the at most one tuple requirement does not hold when it is) it will never produce false positives. For example, if an outer fetched attribute, that is correlated with an input binding of the inner query, is a candidate key for the inner relation and the inner query's "WHERE CLAUSE" is consisted of a single equality predicate, where this candidate key is used, then the at most one criterion is held. Any false negatives will miss the opportunity to use a join strategy that would otherwise be possible, but they do not cause any inconsistencies.

Furthermore, the at most once criterion, along with the use of the LEFT OUTER LATERAL, ensures that the combined query produces a result set with the same cardinality as the result set of the original unmodified query. In addition, the use of any original ORDER BY clause ensures that the combined result set will keep the same ordering with the result set of the original outer query. The interpretation of the combined result set is done as follows. Each fetch request of the parent query advances the combined result set cursor to the next row. If a NULL object reference is returned then there are no more rows for the parent query and the children queries. If a non NULL row was returned then any subsequent get value request will retrieve a column attribute that corresponds to the outer query. When an inner query is opened there is no actual query submission to the backend database server. Each fetch request by any immediate inner query is essentially a NOP API mehtod call (i.e. No Operation), since there is nothing to do. Finally, each get value request retrieves from the current tuple (of the parent context tree node) the correspondigly specified attributes. Therefore, the decoding procedure of the combined result set is straightforward.

3.2.2 The Outer Union Strategy

In the outer union strategy the returned combined result set is characterized by clustering at both the attribute and the tuple level. This is quite obvious from the general and specific examples given at figures 12, 13 on pages 29, 30. Each participating query is represented by distinct columns in the combined result set. Each tuple of the combined result set would have been fetched by a single participating query. Each tuple is actually populated with column values ONLY for the query to which it corresponds. The rest of the columns are populated with NULLS. For conveniece reasons, so as to facilitate the decoding phase an extra column has been used that takes integer values and helps to specify to which query a tuple corresponds. This metadata attribute is called "type".

```
SELECT type, <parent.proj-list>, <child.proj-list>
FROM
( <parent-query> )
LATERAL
(
        SELECT 0,
        <parent-proj-list>,
        NULLS<child-proj-list> UNION ALL
        SELECT 1,
        NULLS<parent-proj-list>,
        <child-proj-list>
        <remaining-child-query>
)
ORDER BY <parent-order-by>,
type, <child-order-by>
```

Figure 12: General Lateral Outer Union Example

```
SELECT type, c1_column1, c1_column2, c2_column1, c2_column2
FROM
(
  SELECT column1, column2
  FROM table1 WHERE column1 >= 1 and column1 <= 5
) c1 LATERAL
(
  SELECT 0, c1.column1, c1.column2, NULL, NULL
  UNION ALL
  SELECT 1, NULL, NULL, column1, column2
  FROM table2 WHERE column1 = c1.column1
) AS inner_contexts
(
  type, c1_column1, c1_column2, c2_column1, c2_column2
)
ORDER BY c1.column1, type, c2_column1
```

Figure 13: Lateral Outer Union Example

In the "order by" clause of the combined query the main idea is that first we use any ordering columns of the parent query, then we use the "type" column and finally we use the "order by" clauses of the inner queries with the same order that they were encountered during the training phase. If the inner queries are encountered with different orders during the training phase, because of the use of application predicates, then the "LATERAL OUTER UNION" strategy is not applicable to these specific parent-children queries and Sqlprefetch considers ONLY the other rewriting strategies as candidates.

3.3 Partitioned Strategies

In contrast to the unified strategies where the combined query is issued upon the opening of the original outer query, in the partitioned strategies the combined query is submitted when the inner query is opened for the first time for each distinct opening of the parent query.

3.3.1 The Client Hash Join Strategy

In this rewriting the parent query is combined with a single child query. The LATERAL derived tables result into a combined query that is submitted ONLY at the very first time that the inner query is opened for each individual open of the outer query. The combined query is submitted by the child context. As can be seen at figure 14 on page 31 the outer derived table is filtered with its DISTINCT output attributes that participate as input parameters in the inner query. After the DISTINCT keyword we could have more columns if more of them were participating as input parameters in the inner query. In the general case we could assume a tuple of such attributes. Each outer distinct tuple may correspond to multiple inner query tuples. The retrieved combined result set is stored locally in a main memory hash table where as keys we use these input parameters that are provided by the outer relation. Since, each unique combination of input parameter values from the outer context may result in multiple inner context rows, the value part of the <key, value> hash table element is a result set class instance.

```
SELECT
c1.column1 as c1_column1, c2.column1, c2.column2
FROM
(
  SELECT DISTINCT column1 FROM
  (
    select column1, column2, column3, column4
    from table1
    where column1 >= 1 and column1 <= 5
    order by column1 asc
  ) AS intermediate_relation
) c1 LATERAL
(
  SELECT column1, column2
  FROM table2 WHERE column1 = c1.column1
) c2
```

Figure 14: Lateral Client Hash Join Example

For each subsequent fetch request of the outer context that the inner context is reopend, instead of submitting a new query, it makes a hash table lookup, in order to retrieve any corresponding result set. Upon a close cursor request by the parent context, the inner context's hash table is discarded. The hash join rewriting is very similar with the outer join rewriting. Nevertheless, they do have some major differences. Firstly, the hash join rewriting combines a parent cotnext with ONLY one child context. Secondly, only the output columns of the outer context that are used as input parameters in the inner context are included in the projection list of the lateral combined query. Thirdly, in the current case we do not need any LATERAL OUTER JOIN semantics since any input parameter combinations that result in empty result sets are left out of the maim memory resident hash table.

It should be noted that under the hash join strategy the inner query is actually opened and submitted only once (for every parent open) independently to the cardinality of the outer query. Nonetheless, this rewriting alternative uses more main memory than the others and is more CPU intensive. These drawbacks and especially the extra memory consumption should be taken under serious consideration before choosing to apply this strategy. In coarse grained query workloads (where large result sets are common), this alternative may be prohibitively expensive. In this case one could suggest some disk resident hash table, but disk I/O is expensive as well and threfore it may invalidate the rewriting's usefulness.

4. QUERY PROCESSOR

Our prototype implementation bears some major differences from that of [4], [5] in various parts. Some of the major differences are located within the query processor egnine. In both prototypes the query processing component is found under both the "training" and the "running" modes. During the "training" mode Sqlprefetch performs 3 major functions in the following order, the pre-optimisation, the main optimization and the rewriting procedure of the entire context tree. On ther other side, during the "running" mode Sqlprefetch's query processing is found within the prefetecher. At the following sections we describe each part in more depth.

4.1 Pre-Optimization

The pre-optimization component lays the foundation upon which the next query processing components are built. As has already been said Sqlprefetch intercepts all the database requests during the "training" mode and maintains a list of holding correlations for each input paramter of a context query. Upon the termination of the training period, the pre-processing procedure is launched. The entire context tree is traversed in a pre-order fashion and for each context tree node the following operations take place. Firstly, it is ensured that all the input parameters of the current context are fully predicted. This means that Sqlprefetch has identified at least one valid correlation for each input parameter binding. If this is not the case then the optimization process is terminated for the current context and it is marked as not fully predicted. A context that is not fully predicted cannot be optimized further and subsequently cannot be rewritten during the run time. Once again, Sqlprefetch needs to know not only the nesting pattern of a query stream but also the actual values that will be used for each rewritten query so as to be able to submit a combined query that will prefetch the correct result sets.

There are three different correlations tracked for an input parameter. Initially, a constant correlation is considered (i.e. that the specific input binding is always assigned the same initial value). In addition, we add these output attributes from the ancestor contexts that match the specific input parameter. The same is done for any input parameters of the ancestor contexts. Althought, a correlation may hold initially, it may not hold at some future request, thus Sqlprefetch monitors the validity of all the initially considered correlations during the entire training period. If the training period lasts long enough it is highly possible that only the valid correlations will have remained. Even, if there are false positives, Sqlprefetch is designed to detect any correlation invalidities during runtime and it will submit the original unmodified query. Thus, even though it may do some extra work, so as to prefetch a clustered result set, there will not be any data inconsistencies for the client code. If for an input parameter all three correlation types hold with multiple ancestors output attributes and input parameters, then we have devised the following priority ordering (which is different from the [4], [5] prototype).

1. If applicable choose from bottom up an output attribute correlation.
2. If applicable choose from bottom up an input attribute correlation.
3. If applicable choose the constant correlation.

This way we were able to stress the rewriting mechanism to the maximum and study the performance behavior of the most expensive correlation type. Next, the pre-optimizer concludes whether the current query will return at most one tuple for each fetched tuple from the parent context. There are possibly many alternatives that can be employed to determine whether the at-most-one restriction holds. We used as our sole criterion whether the where clause contains only one equality condition on a candidate key. If this simple criterion is satisfied we consider that the query of the currently examined context fullfils the at-most-one restriction and the outer join strategy can be applied on this context. Nevertheless, there is always the possibility that Sqlprefetch may produce false negatives, i.e. conclude that the at-most-one criterion does not hold when it is. This misestimation does not pose any inconstistency problems. It is merely preventing Sqlprefetch from utilizing a possibly more efficient rewriting strategy. After the preoptimization phase is completed Sqlprefetch uses two critical components, the optimizer and the cost model that will help it conclude to the most efficient query plan.

4.2 Optimizer

The optimizer is the component that determines which is the most efficent query plan. In order to do that it makes heavy use of another component, that of the cost model, that is described in great detail at chapter 5 on page 37. Our query optimizer is much more simplified than that of the [4], [5]. The main idea is that we rewrite the entire context tree with the same strategy. Thus, we need to consider only 4 alternative strategies, i.e. the nested, the outer join, the outer union and the client hash join strategies. For each candidate strategy the optimizer traverses the context tree in a pre-order fashion. For the immediate children of the root (sentinel) context tree node, Sqlprefetch considers only the nested strategy. This is because they do not have an ancestor context with which they can be combined. For any other context, Sqlprefetch applies the currently evaluated strategy, if it is applicable. For example, if the outer join strategy is examined (for the entire context tree) and some nodes do not satisfy the at-most-one criterion, then thsese nodes will not be rewritten and will be annotated with the nested strategy. However, the rest of the nodes where the outer join strategy is applicable will be annotated with that.

Nevertheless, by simply annotating a context tree node with a specific strategy is not enough. For example, under the outer join strategy the leaf context tree nodes will only need to decode the currently fetched tuple from their parent contexts. In contrast, the non-leaf nodes will need to submit a combined query that will cater for their children, subsequently loop over the combined result set so as to allow the children contexts to retrieve their corresponding attributes and at the same time retrieve any column values for themeselves from their parent node. A similar pattern applies for the outer union strategy. Nonetheless, this is not the case for the nested and the client hash join strategies which require the same actions, independently of a node's position in the tree. For this reason, we have chosen to associate each rewrite strategy with one or more actions that guide the prefetcher on how to act when processing the requests of a specific context. All the considered strategies along with their associated actions are given at table 1 on page 35. More specifically, we annotate each context tree node with the chosen strategy and one or more actions that are associated with this strategy. The actions that are chosen each time somehow describe what is necessary to be done and in what order, so as to allow Sqlprefetch to process successfully all the possible requests on the currently active context.

After the completion of the context tree annotation with a specific strategy, the optimizer traverses once again the context tree in a post-order fashion so as to generate the appropriate actions for each context. Finally, a new pre-order traversal is launched, which estimates the cost of each context tree node and upon its completion it has caclulated the total sum. Each alternative strategy is associated with one cost number. The above procedure is repeated for all the alternative strategies. Therefore, by the end of this estimation algorithm Sqlprefetch may reach the optimum strategy. The optimizer makes use of the cost estimation component in order to determine the cost of a proposed rewriting for a specific context tree node. When Sqlprefetch has concluded to the optimum strategy, it traverses the context tree two times. The first time, it annotates each context tree node with the selected strategy. The first traversal is performed in a pre-order fashion. The second time, it generates the necessary actions for each context-strategy pair and uses them to annotate appropriately the contex tree node. Each action object, if applicable, is accompanied by a corresponding rewriting. The second pass is performed in a post-order fashion. The second step is performed by a specialized component, the rewriter, which is described in depth at section 4.3 on page 34.

4.3 Rewriting the Context Tree

At table 1 on page 35 we have presented all the alternative strategies and their associated actions. This scheme is exactly the same with the one used by [4], [5]. On the next paragraphs a short description will be given for each action and its applicability.

Strategy	Context Action Type
nested	parent, child ⇒ submit-nest
outer-join	parent ⇒ interpret-join child ⇒ decode-join
outer-union	parent ⇒ interpret-union child ⇒ decode-union
hash-join	parent, child ⇒ submit-hash

Table 1: Strategies with associated Actions

The nested and the client hash join strategies are the simplest regarding their associated actions. Each context that is annotated with the "nested" strategy is also annotated with a "submit-nest" action. Each "submit-nest" action is equipped with a reference pointer to the original unmodified query. A similar pattern is followed with the client hash join strategy. The rewriter traverses the context tree in a post-order fashion and for each context annotated with the "hash-join" strategy it adds a "submit-hash" action. Each "submit-hash" action is populated with a rewritten query that combines the current child query with its immediate parent query. In all the alternative strategies the immediate children of the root context tree node are annotated with the "nested" strategy. These children cannot be rewritten since they do not have an outer query. When the "outer join" strategy is used the leaf contexts are annotated with the "decode-join" action, whereas the non-leaf contexts are annotated with the "interpret-join", "decode-join" actions. An "interpret-join" action is populated with a combined query that includes the current context and all the qualified children. A "decode-join" action is not populated with some query since it does not involve one. The "decode-join" action is used to retrieve the current context's corresponding attributes from the currently fetched tuple of its parent context. The same pattern is followed with the "interpret-union", "decode-union" actions that are used for the implementation of the "outer-union" strategy.

4.4 Prefetcher

The Prefetcher is the only component of the query processor that is used during the "running" mode. If the context tree is annotated with the nested strategy, then the prefetcher merely submits the original, unmodified queries. If the context tree is annotated with the client "hash join" strategy, then each context is also annotated with the "submit-hash" action, except for the immediate children of the root node. The immediate children of the root node are annotated with the "nested" strategy and the "submit-nest" action. The first time that a context (annotated with the "hash-join" strategy and the "submit-hash" action) is opened for each open of its parent context the combined query, that accompanies the "submit-hash" action, is submitted to the back end RDBMS. The combined result set is consumed and stored locally in a main memory hash table. Thus, the next times that the same context will be opened for the same open of its parent cotnext there will be no query submission. In theses other opens the anticipated result sets are already stored within the aforementioned hash table.

If the context tree is annotated with the "outer-join" strategy then we have three distinct cases, that of the leaf context tree nodes, that of the immediate children of the root context and that of the rest of the non-leaf context tree nodes. The leaf context nodes do not submit any query but merely consume their corresponding attributes from the clustered result set of the parent cotnext. This action is prescribed by the "decode-join" annotation. Each fetch request on a leaf context results in a retrieval of the corresponding columns from the currently fetched tuple of the parent context. The second kind of contexts are annotated with a single "interpret-join" action which submits a combined query and returns a clustered result set that is used to satisfy the fetch requests of both the current context's and its immediate children[1]. The rest of the non-leaf context tree nodes are annotated with two actions, in the following strict order, the "interpret-join" and the "decode-join" actions. First is executed the "interpret-join" action, followed by the execution of the "decode-join" action. These two actions are executed with the exact same manner as in the first two cases. Somehow, the third case is a combination of the first two cases.

If the context tree is annotated with the "outer-union" strategy then we similarly have three distinct cases, that of the leaf context tree nodes, that of the immediate children of the root context and that of the rest of the non-leaf context tree nodes. In the first case, the leaf context nodes do not submit any query but merely consume a result set that is created on the fly from the clustered result set of the parent context. It should be noted here that in the "outer-union" strategy the combined result set exhibits clustering properties at both the row level and the column level. In addition, each parent row may result in more than one inner rows. In this case the child context will use the derived result set to loop over all the returned rows. In the second case, the immediate children of the sentinel context node are annotated with the "interpret-union" action, which is the one that submits the combined query. In this case, the combined query involves the current (parent) context and ALL its children[2]. Furthermore, the combined result set will be consumed by the current (parent) context and the participating children contexts. The third case (as in the "outer-join" strategy), involves both of the above actions, in the following strict order. First we find the "interpret-union" action and then we execute the "decode-union" action. These two actions are treated with the exact same fashion as in the first two cases. Apparently, as in the "outer-join" strategy the third case is a combination of the first two cases.

[1]By immediate children we mean these children that participated in the combined query. There is always the case that some children did not satisfied the "at-most-one" restriction.

[2]Unless some child is not fully predicted. This exception holds for the "outer-join" rewriting as well.

5. COST MODEL

5.1 Application Predicates Selectivity

Est-P is an estimate of the probability that an inner query will be opened for each outer query row. Suppose that C is a context with inner query Q and the C_P is C's parent in the context tree, that is C is C_P/Q. EST-P(C) is defined as the number times Q is opened within C_P, divided by the number of rows fetched from the inner query of C_P. The fraction that computes the probability EST-P(C) is given below.

$$\text{EST-P}(C) = \frac{\text{\# of opens of the child query}}{\text{\# of rows fetched from the parent query}} \tag{2}$$

The above equation is translated as follows in the application code:

$$\text{EST-P}(C) = \frac{\text{currctx.numopens}}{\text{currctx.parent.numrows}} \tag{3}$$

For C $=$ C_P/Q, EST-P0(C) is an estimate of the probability that Q will be opened at least once whenever context C_P is entered.

$$\text{EST-P0}(C) = \frac{\text{\# of opens of the parent query for which the child query opened at least once}}{\text{\# of opens of the parent query}} \tag{4}$$

The above equation is translated as follows in the application code:

$$\text{EST-P0}(C) = \frac{\text{currctx.numprtopens}}{\text{currctx.parent.numopens}} \tag{5}$$

5.2 Ranking Execution Plans

To estimate the cost of an execution plan, Sqlprefetch uses the statistics shown in table[3] 2 on page 37.

Quantity	Source	Description
$EstTotal(Q)$	Analytical	Estimated total cost for Q in seconds
$\|Q\|$	Analytical	Estimated # of rows returned by Q
$InterpretCost(C)$	Calibration	The cost to interpret results for context C
$Est - P0, Est - P$	Training	Selectivity of client predicates

Table 2: Statistics used for Ranking Plans

EstTotal(Q) refers to the estimated response time for a query Q that appears in the exe-

[3]This table is verbatim from [4].

cution plan. (Each node in the plan is annotated with two queries: the original application query and a rewritten query. It is the rewritten queries that are used to rank execution plans). $|Q|$ is the estimated number of rows returned by Q. $\text{InterpretCost}(C)$ is an estimate of Sqlprefetch's costs for interpreting the application's Open, Fetch and Close requests in context C of the plan. This reflects the cost of decoding the result sets of rewritten queries. $\text{InterpetCost}(C)$ depends on the strategy annotation at C, for example, the interpretation cost is higher at a "H" node than at a "N" node. Sqlprefetch calibrates $\text{InterpetCost}(C)$ for each strategy prior to training.

Sqlprefetch recursively estimates $\text{Rows}(C)$ which is the total number of tuples produced in context C.

$$\text{Rows}(C) = \text{Open}(C) \times |Q| \tag{6}$$

$\text{Opens}(C)$ is the number of times $C's$ query is opened.

$$\text{Opens}(C) = \text{Opens}(P) \times \text{Rows}(P) \times \text{Est} - P(C) \tag{7}$$

$\text{OneOpens}(C)$ is the number of opens of P for which $C's$ query is opened at least once.

$$\text{OneOpens}(C) = \text{Opens}(P) \times \text{Est} - P0(C) \tag{8}$$

The $\text{Opens}(C)$ and the $\text{Rows}(C)$ are defined to be 1 for the root context.

Weighted Cost for the Nested Strategy

$$\text{Cost}(C) = \text{Opens}(C) \times (\text{EstTotal}(Q) + \text{InterpretCost}(C)) \tag{9}$$

Weighted Cost for the Partitioned Strategies

Under the partitioned strategy, the rewritten query is issued only if the application requests open the inner query at least once after the outer has been opened. Therefore, for contexts annotated with a partitioned strategy (i.e. 'H' or 'M') the following formula provides the cost of the specified rewriting.

$$\text{Cost}(C) = \text{OneOpens}(C) \times \text{EstTotal}(Q) + \text{Opens}(C) \times \text{InterpretCost}(C) \tag{10}$$

Weighted Cost for the Unified Strategies

If a unified strategy is used at C, then no query is issued to the server when C is entered.

Instead, Sqlprefetch decodes prefetched results from the rewritten outer query. So for contexts with a unified strategy 'J' or 'U'), we use:

$$\text{Cost}(\text{C}) = \text{Opens}(\text{C}) \times \text{InterpetCost}(\text{Q}) \tag{11}$$

5.3 Estimating Query Costs and Sizes

This section deals with either the recording or the calculation of the average query times and the average number of rows fetched per query. Sqlprefetch needs to determine EstTotal(Q) and $|Q|$ for two different kinds of run time queries that appear in execution plans. The application queries have been observed by Sqlprefetch during its training period. The applicable query rewritings per context have not been observed, therefore there are no experimental measurements for their costs (i.e. average latency per query rewriting) and the average number of tuples fetched. For queries that were observed during training, query cost and query size are relatively easy to estimate, since Sqlprefetch measures their actual execution times during training.

Table[4] 3 on page 39 lists the statistics that Sqlprefetch uses for query costing and their sources.

Quantity	Source	Description
$AvgTot(Q)$	Training	Observed average total cost for Q in seconds
$AvgServ(Q)$	Training	Observed average server cost for Q in seconds
$AvgRows(Q)$	Training	Observed average # of rows returned by Q
$U_0 = AvgTot(Q0)$	Calibration	Query-independent overhead

Table 3: Statistics used to estimate Query Costs and Sizes

For a query Q that has been observed during training, we use the averages as our estimate. For the actual application queries, the execution time EstTotal(Q) and the number of rows |Q| are estimated using the following equations. In addition, Sqlprefetch measures the average server latency **AvgSrv**$(\mathbf{Q})$ of each context-query pair. That is the time spend on the RDBMS server excluding, the network related latencies.

$$\text{EstTotal}(Q) = \text{AvgTot}(Q) \tag{12}$$

$$|Q| = \text{AvgRows}(Q) \tag{13}$$

[4]This table is verbatim from [4].

5.4 Estimating Per-Request Overhead U_0

Using an assessment of the per-request overhead that is based on [2] results to a graph that is unsatisfactorily regressed linearly. Therefore, the simple query of 15 on page 40 is used to estimate the query independent overhead U_0. For this simple query, the server execution costs (apart from per-request overhead) are negligible. Actually, U_0 is the minimum cost of executing a query. Before Sqlprefetch enters the "training" phase, it first performs a calibration step (that involves the calibration query) to estimate U_0. Sqlprefetch executes the calibration query a number of times and uses the average execution time as the estimate of U_0 for the current configuration of the tier. Because of the extreme simplicity of the calibration query, it's server side processing cost (D interval - server executes Q) is considered negligible. In addition, due to lack of returned rows the E, F and G terms are also considered to be the approximately the same with their corresponding intervals A, B and C. Therefore, the following equations hold for the calibration query.

$$\text{AvgTot}(Q_0) = U_0 = 2(A + B + C) \tag{14}$$

$$\text{AvgSrv}(Q_0) = 2C \tag{15}$$

```
select t.x from (values(1)) t(x)
```

Figure 15: Calibration Query.

5.5 Cost Estimation for Result Set Interpretation

When prefetching occurs the returned result set encompasses the combination of the result sets of the participating queries. During run-time, this encoded result set is processed and decoded to the correspondingly expected result sets. This way Sqlprefetch operates in a transparent manner from the view of a client application. This decoding process bears a significant cost which is proportional to both the number of the combined result set columns (p) and the number of the returned rows (r). Thus, the decoding procedure will be represented by a formula of the form $EstInterpret(p, r)$. This interpretation is an extra overhead for the client process. The interpretation cost formula that is used is the result of the multiple linear regression on the experimental resutls. These results are produced by a number of test scenarios for different columns and rows. The same group of test scenarios is executed for each case (i.e. Nested, Outer-Join, Outer-Union, Client-Hash-Join). The entire benchmarking was performed in a local shared memory configuration and the resulted interpretation coefficients are presented at table 4 on page 41.

Action	Columns Coefficient	Rows Coefficient	Intercept
submit-nest	15.0710	17.4710	-9.1110
interpret-join	1.5030	24.5790	66.1490
decode-join	19.4110	7.1980	-20.1840
interpret-union	2.2080	20.6700	20.2220
decode-union	22.4390	9.2910	-73.3130
submit-hash	31.1600	11.1700	-139.0600

Table 4: Interpretation Coefficients in LSM Configuration (microsecond)

5.6 Cost Estimation for Alternative Strategies

The estimates of the U_0 and the $EstInterpret(p, r)$ are not adequate by themselves. The query optimizer needs to estimate both the cost of the queries along with the number of rows they will return. These estimations are needed both for the original queries and for the re-written ones.

The methodology that we followed was similar to [4], [5]. For the part of the application that it is feasible we have taken experimental measurements, i.e. for the original queries we track various cost related parameters like average total execution time and selectivities. In contrast, for the run-time mode where some queries will be re-written we have used an analytical model that takes into account measurements taken for the original queries during the training mode and RDBMS specific cost functions. More specifically, during the training mode we observe the average running time of an unmodified query ($AvgTot(Q)$) and the average number of rows ($AvgRows(Q)$) returned for this query. The observed average values are used in the analytical model as adequately accurate estimations.

$$\text{EstCost(Q)} = \text{AvgTot}(Q) \tag{16}$$

$$\text{EstRows(Q)} = \text{AvgRows}(Q) \tag{17}$$

The chosen analytical model calculates predicted costs for any possible re-writing of the original queries, since it is infeasible to execute all the possible re-writings and benchmark them. If a more accurate estimation was desired we would have to measure any cost parameters during the execution of the application with all the possible re-writing combinations. Although, this could be a feasible option in our case, it would not be an option for a more sophisticated plan enumeration algorithm like the one used by [4], [5].

Essentially, the 3 rewritings are based on the same concept of the lateral derived tables. The simplest one is the client hash join which is a lateral based inner join. For simplification reasons the classic symbol of the inner join relation operator will be overloaded with the lateral semantics as well. Thus, the lateral inner join of Q_1, Q_2, ... , Q_n is expressed with the $Q_1 \bowtie Q_2 \bowtie ... \bowtie Q_n$. Similarly, the outer join rewriting is a lateral enhanced outer join and is expressed with the $Q_1 \ltimes Q_2 \ltimes ... \ltimes Q_n$. Applying the same reasoning, the lateral outer union strategy is expressed with the $Q_1 \uplus Q_2 \uplus ... \uplus Q_n$.

5.6.1 Estimating Client Hash Join

A client hash join is essentially a lateral inner join. This means that the combined result set contains columns in each row from both joined queries. However, if an outer row (that provides the outer bindings to the inner query) does not match with an inner row, then it is omitted from the combined result set. This rewriting is characterized by clustering at the column level. A simple and naive evaluation technique for this essentially inner join is the nested loops join. We chose to use this as an upper bound for the estimate of the total cost of the rewriting. This estimation along with the estimated number of rows in the returned combined result set are given in equations 18, 26 and use experimental measurements taken during the training mode. The cost U_0 is subtracted from the $EstCost(Q_j)$ in equation 18 because the inner query is not submitted separately but combined with the outer derived relation.

$$JnlCost(Q_i \bowtie Q_j) = EstCost(Q_i) + EstRows(Q_i) \times (EstCost(Q_j) - U_0) \qquad (18)$$

$$JnlRows(Q_i \bowtie Q_j) = EstRows(Q_i) \times EstRows(Q_j) \qquad (19)$$

In addition, we considered a correction factor for the case where this inner join is handled more efficiently by the query processor of the back-end RDBMS. Equation 20 calculates the relative gain the server may accomplish by executing the lateral inner join with a better plan than the simple nested loops join.

$$SrvSavings(Q_i \bowtie Q_j) = \frac{AvgCost(Q_i \bowtie Q_j)}{AvgCost(Q_i) + AvgRows(Q_i) \times AvgCost(Q_j)} \qquad (20)$$

The function $AvgCost(Q)$ is RDBMS specific and takes into account only the server part of a query execution, i.e. it omits any communication costs and client side query processing overhead. In addition, its measurement unit neither corresponds nor fits with the operating system perception of the time. This cost estimation function is internally used by the RDBMS to merely provide a convenient unit for the ranking of all the considered query execution plans. Consequently, its usage is performed in a way that can provide an estimation of the relative gain that can be accomplished if a more sophisticated plan is chosen compared to the naive nested loops join. A value close to 1 signifies that the chosen execution plan is similar to the nested loops join and therefore no extra gain may be accomplished. However, a value close to zero denotes that a more efficient execution plan has been chosen that will outperform the nested loops join.

As has been mentioned before, the average cost of a query could be represented by a vector of three elements $AvgClntCost(Q)$ (client processing cost), $AvgCommCost(Q)$ (communication cost), $AvgSrvCost(Q)$ (server side execution cost) and expressed like the equation 21. All these 3 costs may overlap (but we do not consider this case) and are measured in time units.

$$AvgCost(Q) = AvgClntCost(Q) + AvgCommCost(Q) + AvgSrvCost(Q) \tag{21}$$

The $SrvSavings(Q)$ correction factor should be applied ONLY on the $AvgSrvCost(Q)$, but this is not possible in our case since we have an estimation of the total average cost for a query, i.e. for all 3 terms and we cannot decompose it to its compositing terms. As a consequence, we apply the correction factor to all 3 terms and since this is a significant approximation we correct it by using a weighted sum of the experimentally based estimation of Sqlprefetch (originated by measurements taken during the training mode) and the database based estimation delivered by the RDBMS itself. We denote the weighting term with K and its values range between 0 and 1, i.e. $0 \leq K \leq 1$.

$$J = JnlCost(Q_i \bowtie Q_j) - U_0 \tag{22}$$

$$S = SrvSaving(Q_i \bowtie Q_j) \tag{23}$$

$$EstCost(Q_i \bowtie Q_j) = K \times S \times J + (1 - K) \times J + U_0 \tag{24}$$

The equation 24 provides a blending of estimates from the Sqlprefetch performance module tracking and the RDBMS back-end. The main assumption in this reasoning is that the worst query execution plan that the RDBMS can come up with is the nested loops join. A value of 0 zero rejects the contribution of the RDBMS estimation, whereas a value of 1 considers it in its entirety. The K parameter weights the credence of the RDBMS server cost estimator.

5.6.2 Estimating Outer Join Queries

The lateral outer join rewriting is an extension of the client hash join with the major difference that keeps the rows of the outer derived relation that have not matched with some row from the inner relation. These unmatched outer rows fill the inner query's corresponding columns with NULLs. Although, these NULLs bear an extra cost during both result set transmission and interpretation, we have chosen to not account this extra cost, as it is insignificant compared with other approximations. As far as the estimated total query cost is considered we have used the same formula with the client hash join (as can be seen by equation 25). However, the estimated rows formula of the combined result set is altered so as to take into consideration the fact that the outer join preserves all the tuples of the outer relation. This difference is expressed with the equation 26.

$$JnlCost(Q_i \ltimes Q_j) = JnlCost(Q_i \bowtie Q_j) \tag{25}$$

$$JnlRows(Q_i \ltimes Q_j) = max(JnlRows(Q_i \bowtie Q_j), EstRows(Q_i)) \tag{26}$$

Another major difference is that the outer join rewriting can combine more than two queries. For example, if a parent context has n children contexts then the combined query will place the parent query in the derived outer relation and the children queries in the inner places. This difference results in equations 27, 28.

$$JnlCost(Q_0 \ltimes Q_1 \ltimes ... \ltimes Q_n) = EstCost(Q_0) + \sum_{i=1}^{n} [|Q_0| \times (EstCost(Q_i) - U_0)] \tag{27}$$

$$|Q_0 \ltimes Q_1 \ltimes Q_2 \ltimes ... \ltimes Q_n| = max(|Q_0 \bowtie Q_1|, ..., |Q_0 \bowtie Q_n|, |Q_0|) \tag{28}$$

5.6.3 Estimating Outer Union Queries

The combined result set contains a type column and the columns of each participating query. The total number of columns in the result set affects to overall cost of the result set interpretation. Similarly, to [4], [5] we chose to ignore this extra cost because its omission is insignificant compared to the other approximations that have been made. Thus, we estimate the average cost of a lateral outer union and the average number of rows returned by the combined query as if it was a lateral inner union. The difference between a lateral outer union and a lateral inner union is that the former is characterized by clustering at both the row level and the column level. In contrast, a lateral inner union does not use clustering at the column level. The following equations represent the aforementioned reasoning.

$$EstCost(Q_i \uplus Q_j) = EstCost(Q_i) + EstCost(Q_j) - U_0 \tag{29}$$

$$EstRows(Q_i \uplus Q_j) = EstRows(Q_i) + EstRows(Q_j) \tag{30}$$

Similarly, with the outer join strategy the outer union can involve one parent and n of its children. Thus, the above equations become.

$$EstCost(Q_0 \uplus Q_1 \uplus ... \uplus Q_n) = EstCost(Q_0) + \sum_{i=1}^{n} [|Q_0| \times (EstCost(Q_i) - U_0)] \tag{31}$$

$$EstRows(Q_0 \uplus Q_1 \uplus ... \uplus Q_n) = \sum_{i=0}^{n} EstRows(Q_i) \tag{32}$$

6. PROTOTYPE IMPLEMENTATION

Sqlprefetch is implemented in Java and is written in a highly modular and layered architecture. It consists of an SQL parser, the core engine, a command line processor, various convenience utilities, along with an extensive automatic unit test suite based on JUnit ([13]). The SQL parser has been implemented with the help of the ANTLR framework ([14]) and the ANTLRWorks IDE ([15]). The foundation software layers of Sqlprefetch are RDBMS agnostic. The implemented optimizations have targeted a specific commercial RDBMS. However, because of the agnosticity of the low level layers it is quite easy to extend Sqlprefetch so as to support another RDBMS.The modular part means that some layers can be easily provided by different implementations. For example, although, there is one cost model implementation, if another cost model was devised and we needed to test it, then it could be easily plugged in. The various layers abstract the functionality that they serve. A simplified bottom-up list of the various software layers and in which mode they are used is given in the following list.

- JDBC Vendor Abstraction Layer (both)
- JDBC plain wrapper (both)
- JDBC training wrapper (training)
- JDBC running wrapper (running)
- Patter Detector (both)
- Pre-Optimizer (training)
- Optimizer (training)
- Prefetcher (running)

6.1 JDBC Vendor Abstraction Layer

This layer is used under both running modes. This layer attempts to hide from the upper layers any RDBMS vendor specific behavior as far as the JDBC layer is concerned. Furthermore, this layer has been developed and completed successfully for 4 prominent commercial RDBMS vendors. One of these DBMS is PostgreSQL 9.0.3 ([16]), which has not implemented the LATERAL feature. Sqlprefetch uses PostgreSQL so as to persistenly store any bookkeeping information and benchmarking results. The other three RDBMSs implement the LATERAL semantics either with the same syntax as the ISO's specification or with some vendor specific syntax. Finally, as all the other layers it includes a number of convenience APIs.

6.2 JDBC Plain Wrapper

This layer is used under both running modes. As has been mentioned before Sqlprefetch intercepts all the JDBC API calls. This layer allows for JDBC requests to be submitted to the back-end RDBMS without being intercepted. It is a convenience API that facilitates the out of band submission of SQL queries. This API has been used in various places and quiet extensively in a large number of unit, integration test cases.

6.3 JDBC Training Wrapper

This is the JDBC wrapper API that is used to monitor all the JDBC method invocations during the training mode so as to construct the context tree and update various performance related counters per context, like total number of one opens, total number of opens, total number of rows fetched, total interpretation cost, total query execution cost, selectivity predicates and other bookkeeping information. An additional major task of this API is to track any parameter correlations and verify that they hold for the entire training period.

6.4 JDBC Running Wrapper

This is the JDBC wrapper API that is used to monitor all the JDBC method invocations during the running mode so as to keep the current context pointer on the correct context. If a submitted query has never been observed during the training mode then the current context pointer moves to the sentinel node and points it. This API updates almost the same performance counters as the JDBC training wrapper.

6.5 Pattern Detector

This layer is used under both running modes. The pattern detector is used extensively within both training and running JDBC wrappers. Its main task is to construct the context tree (in the training mode) and traverse the context tree (in the running mode).

6.6 Pre-optimizer

This layer is used only under the training mode. The pre-optimizer traverses the context tree in a pre-order fashion and ensures that a context-query pair is fully predicted. In addition, it specifies whether the current context-query will return at most one tuple for each fetched tuple of its parent context-query pair. The fulfillment of the at-most-one requirement determines whether the outer join strategy is applicable or not for the currently examined context-query pair. Finally, it chooses for each input parameter of a context-query pair the optimum correlation value.

6.7 Optimizer

This layer is used only under the training mode. The optimizer applies sequentially each strategy to the entire context tree and for each rewritten context tree it estimates its cost. The strategy with the minimum cost is the optimum one and is the one that is chosen to be used during the running mode.

6.8 Prefetcher

The prefetcher is used during the running mode, as expected. If a context tree node is annotated with the nested strategy it submits the original unmodified query. If a context is annotated with a unified strategy then it either submits a combined query (of the current query with its children queries) or it decodes the prefetched combined result set of its parent context or both. If a context is annotated with a partitioned strategy then the first time that is opened for each open of its parent context it submits a combined query (of itself with its parent query). The returned combined result set is stored locally in a hash table and used by the child context as long as the parent context's cursor is open. When the parent context closes its cursor then the locally stored hash table is discarded.

7. EXPERIMENTAL EVALUATION AND CONCLUSIONS

So as to identify our prototype's strengths and wicknesses, we developed a comprehensive benchmarking suite where the maximum depth is usually 3, except for one case where it is 4. 100 distinct queries were employed corresponding to 100 distinct tables. Some of the queries were reused in different contexts resulting in 106 distinct contexts. The benchmarked client code contained no application predicates since we wanted to explore what could be the maximum benefit from applying the proposed rewitings. Each fetched tuple from an outer query results to at most one inner tuple. This restriction was enforced on the underlying database so as to allow us to use and compare all the alternative strategies.

In addition, we executed the same benchmark for two different scaling factors of the underlying database. This way we could observe any performace differences between two quite different workloads regarding the scale size. On all the following bar diagrams SF1 means "scaling factor 1" and this is the reference populated database, where each query will return a result set of 100 tuples. Apparently, SF10 means "scaling factor 10" and each context query returns 1000 tuples. Once again with these different scaling factors we attempted to observe the perfromance implications on both the interpretation cost of a combined result set and the query execution time spend on the back end RDBMS. These two runs were conducted under a local shared memory (LSM) configuration, i.e. the client code, the Sqlprefetch library and the RDBMS server were running on the same workstation.

Furthermore, we run the same benchmark for 4 different network settings given at table 5 on page 49. All these 4 runs where performed for SF0.1, i.e. each original context-query pair returned 10 tuples during an entire individual run. For each run, we recorded the total execution time of the entire benchmark. Each network setting is characterized by a different nominal throughput rate and therefore by a different latency. Thus, in network configurations where the latency is higher, the alternative rewriting strategies are expected to pay off more for the increased CPU usage of their combined queries. This expectation is confirmed by the resulted total duration bar diagrams.

Network Configuration	Communication Rate (Mbps)
LCL	inf
LAN	100
WIFI	10
WAN	1

Table 5: Different Network Settings

For all the experiments we used one of the commercial RDBMSs whose name will not be disclosed because of license restrictions. We suspect that the produced perfromance results will be proportionally the same with the results that we would have seen from the other two commercial RDBMSs. For all the RDBMSs we installed and used the developer edition which was locked on 1 single core. All the experiments were conducted with an Intel Core i7 CPU 960 3.20GHz hyper threaded processor, 8 GB memory and running a 64-bit Debian Wheezy (testing) 3.0.0-1 kernel. All the performance results were taken during the "running" mode and the entire database schema was cached inside the main memory, so as to eliminate any disk I/O related time overheads.

7.1 Local Shared Memory (LSM)

We chose to track the total interpretation cost, the query ONLY execution cost on the database server and the total query cost. The 3 different metrics were taken for a leaf context tree node and for a non-leaf context tree node. This interpretation of the benchmarking results allowed us to adequately understand and explain any unexpected performace bottlenecks and any possible performance benefits that a future work could consider.

At figure 16 on page 51 we observe that the interpretation cost is always higher for the nested strategy and when the number of fetched rows increases the nested strategy interpretation cost increases much more faster than the other alternative strategies. We should note here that the recorded interpretation cost refers to the total interpretation cost for a specific leaf node during the running of the corresponding benchmarking scenario. The interpretation costs for the alternative strategies remain close at the same levels. This is because all of them "fetch" each tuple from the main memory where it has been buffered in some Sqlprefetch internal data structure. We should recall here the associated actions for each alternative strategy. In the outer join strategy a leaf context tree node is annotated with a single "decode-join" action that retrieves the inner query's attributes from the currently fetched tuple of the parent context. In the outer union strategy a leaf context tree node is annotated with a single "decode-union" action that retrieves the inner query's tuples and attributes from a buffered resutl set that was created from the clustered result set of the parent cotnext. Finally, in the client hash join strategy a leaf context tree node is annotated with the "submit-hash" action that retrieves the inner query's tuples and attributes from the locally stored in main memory hash table. As can be seen, in all the alternative rewiting strategies there is no network communication to which we can attribute the lower interpretation, retrieval costs. In contrast, the nested strategy has to perform more round trips since it fetched the result set rows from the back end database server.

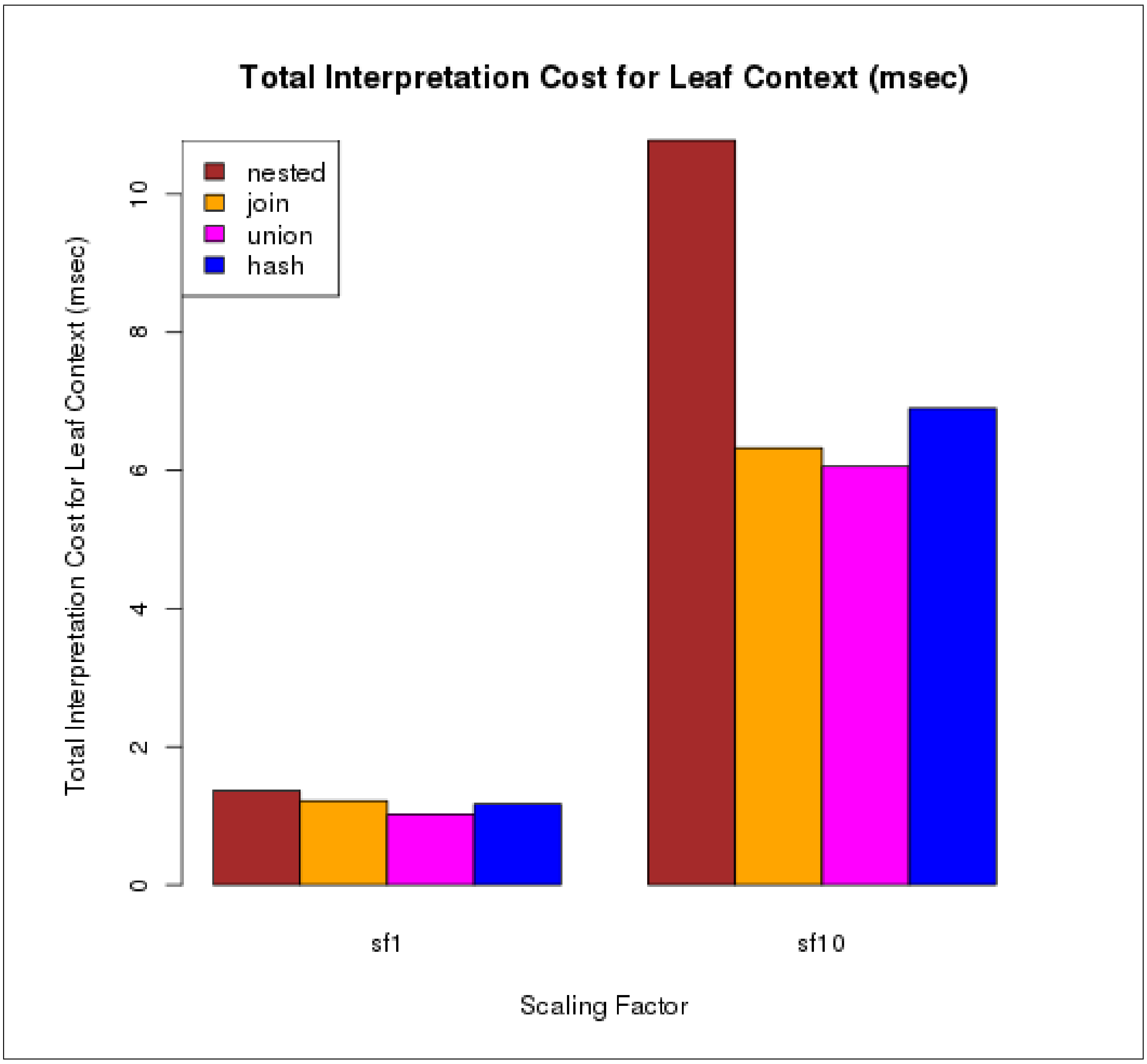

Figure 16: Total Interpretation Cost for a Leaf Context (msec)

Most database access APIs provide the capability to set a batch number of result set rows that should be fetched with a single round-trip. In the worst case this parameter is zero and each fetch request results in a distinct round trip. This parameter cannot be extremely large so as to cater for huge result sets, since it would risk to exhaust the memory resources of the client machine. In all our experiments, we keep the default value of the corresponding JDBC driver vendor. Someone could set this JDBC protocol parameter high enough so as to greatly limit the occuring round trips. We chose no to do that, because generally, an application cannot make safe assumptions for the size of the returned result sets. Setting this value high could exhaust the client's main memory as it would have to store locally a huge result set. Actually, this problem is similarly dealt with by two of the three suggested alternatives. In contrast, the third alternative rewriting, the "client-hash-join" strategy, stores on the client machine the entire combined result set, preassuming that there is enough main memory.

An additional note that needs to be made is that since each outer tuple results in EXACTLY one inner tuple, then under the hash join rewriting each open request at the child query (creation, population of the hash table) results in one fetch tuple (hash table lookup). And this pattern is repeated for each fetched tuple. Although, the interpretation cost seems to be low, this is somehow misleading, since as can be seen by the bar diagram 17 on page 52 the execution cost at the back end database server is the highest of all the strategies. This cost does not represent ONLY the execution cost of the combined query but includes the additional cost of the creation and population of the main memory resident hash table.

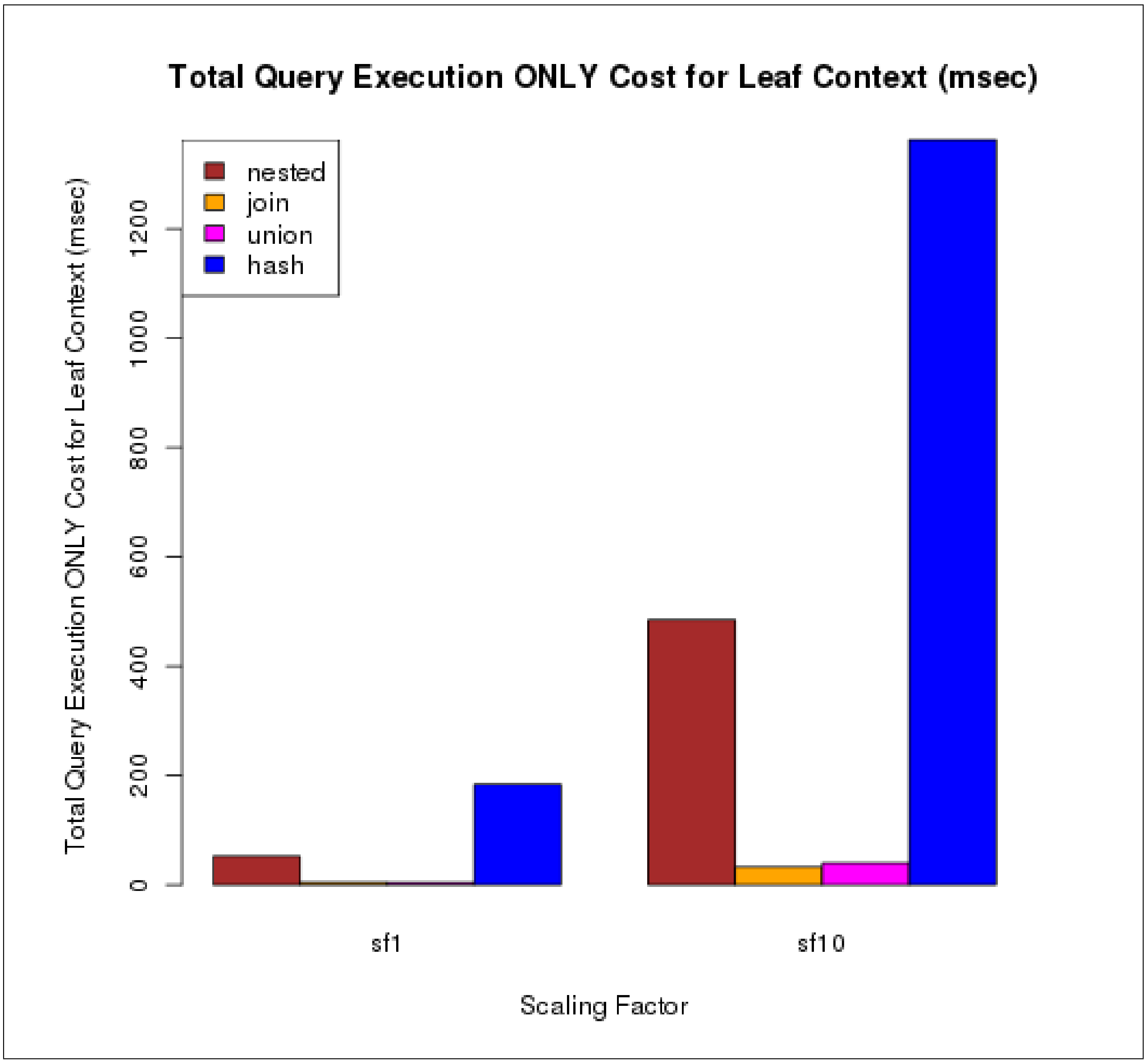

Figure 17: Total Query Execution ONLY Cost for a Leaf Context (msec)

Therefore, we have concluded that the client hash join rewriting is very expensive and the ONLY possibility to payoff is to be used in cases where one outer tuple results in a large number of inner tuples. This way it may amortize the initially high cost of the hash table creation and population with a large enough number of fetch requests.

The very low costs of the query executions on the database server for both unified strategies are due to the fact that there is not actual query submission and execution! The parent context has catered for its children contexts by prefetching a clustered result set. Moving from the scaling factor 1 to 10 seems to have little effect on the query execution costs of the unified strategies. Since, these costs in fact represent interpretation costs we may assume that the interpretation costs of the unified strategies are not essentially contributing to the prefetching cost. This is also the case for the partitioned strategy if we could separate the query submission, execution and hash table creation from the hash table lookups. Because of practical reasons this was not possible but we may safely assume that the hash table lookups are quite cheap and do not contribute significantly to the hash join cost. These conclusions are verified by the diagram at 18 on page 54. It is apparent that the total cost is overwhelmingly contributed by the query execution cost.

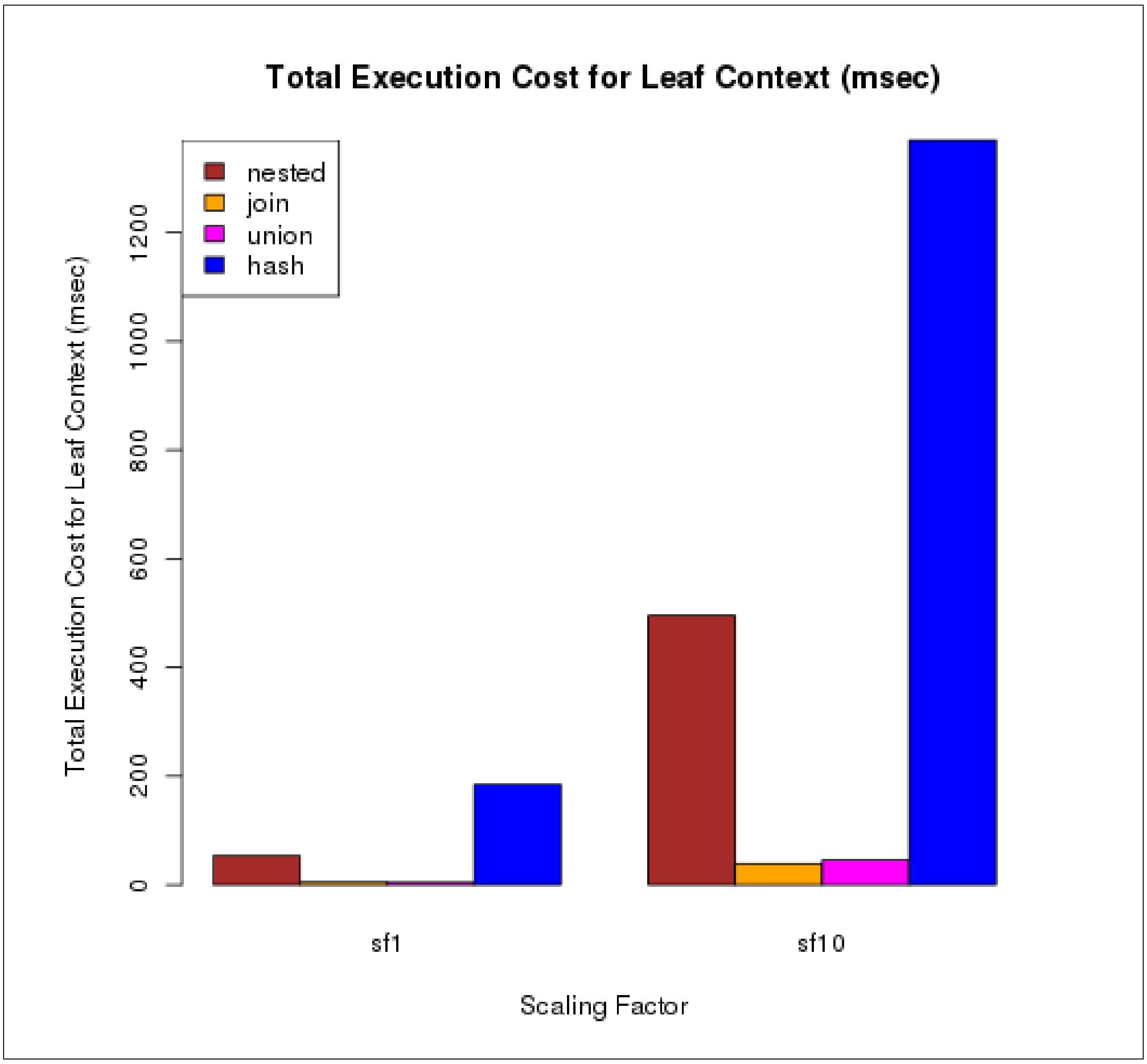

Figure 18: Total Execution Cost for a Leaf Context (msec)

Apart from the hash join strategy, the nested execution strategy remains more expensive from the unified strategies. Once again due to the fact that both the outer join and the outer union strategies do not submit and execute an actual query.

In the next diagrams we move to the study of the same metrics for a specific non-leaf context tree node. At bar diagram 19 on page 55 we can see the interpration costs for two different scaling factors and all the alternative strategies. A non-leaf context is annotated with an "interpret-join" action and a "decode-join" action for the outer join rewriting. This means that it submits a combined query (with the "interpret-join" action) that it will cater for its children and it uses the "decode-action" so as to retrieve the currently fetched columns from its parent context's row. Thus, this time the outer join strategy does not avoid the network communication. In fact, its network communication follows the same pattern with that of the nested strategy. This similarity explains the approximately similar communications costs. This is the case for the outer union strategy as well. In this diagram the less interpretation cost is attributed to the client hash join. This is expected, as the hash join in this case is not performing any network communications but only in main memory hash table lookups.

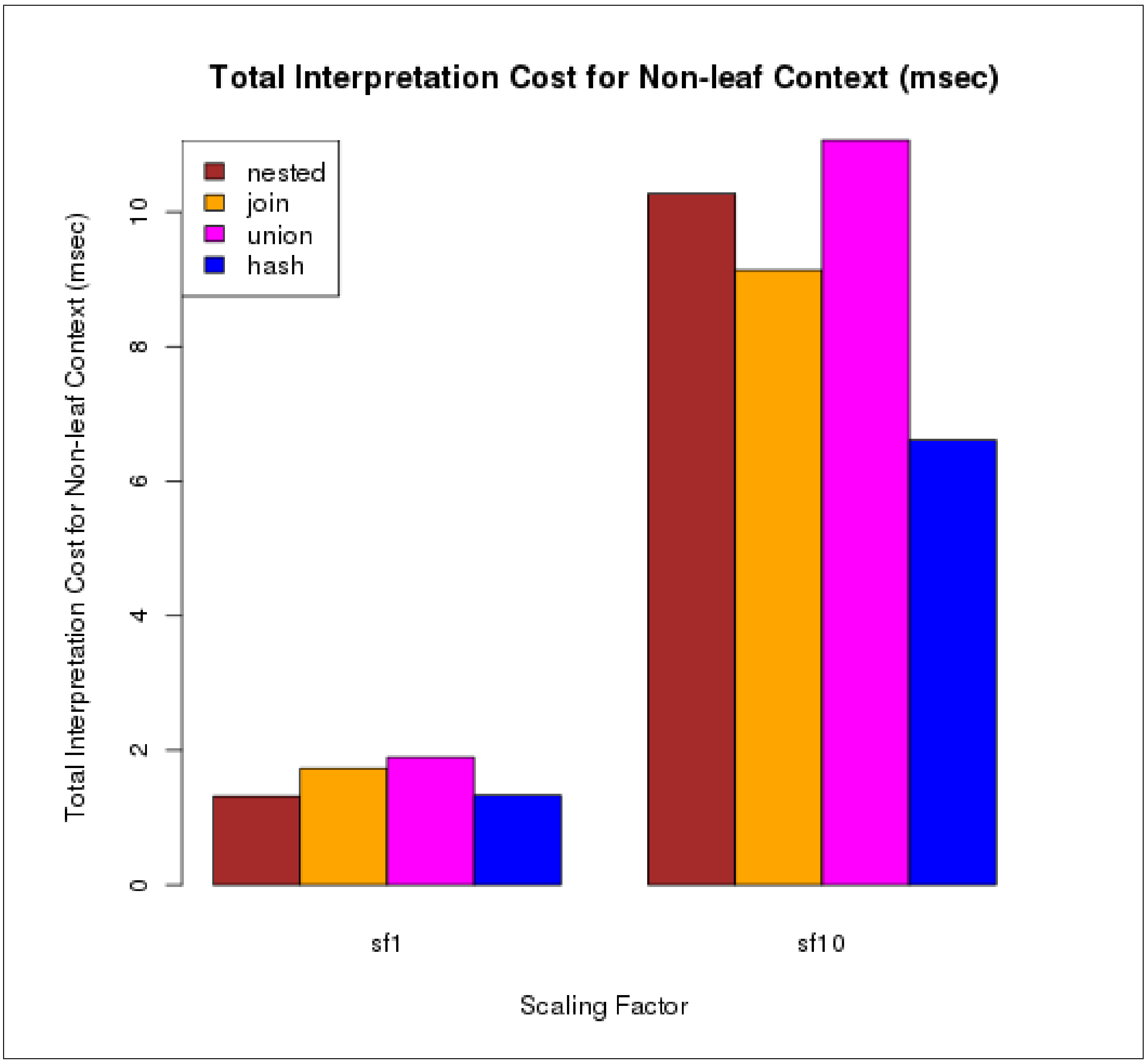

Figure 19: Total Interpretation Cost for a Non-Leaf Context (msec)

The total query only execution cost for a non-leaf context is depicted at diagram 20 on page 56. These query execution costs have revealed two game changing factors. Firstly, the query submission and execution costs of the unified strategies are by far the most expensive. In part this may be attributed to the fact that these queries are much more complex than the original unmodiffied query. Despite the fact that the hash join is more complex that the original query, it is not that complex as the unified queries. This explains why the hash join is more expensive than the nested strategy but by far more efficient than the unified strategies.

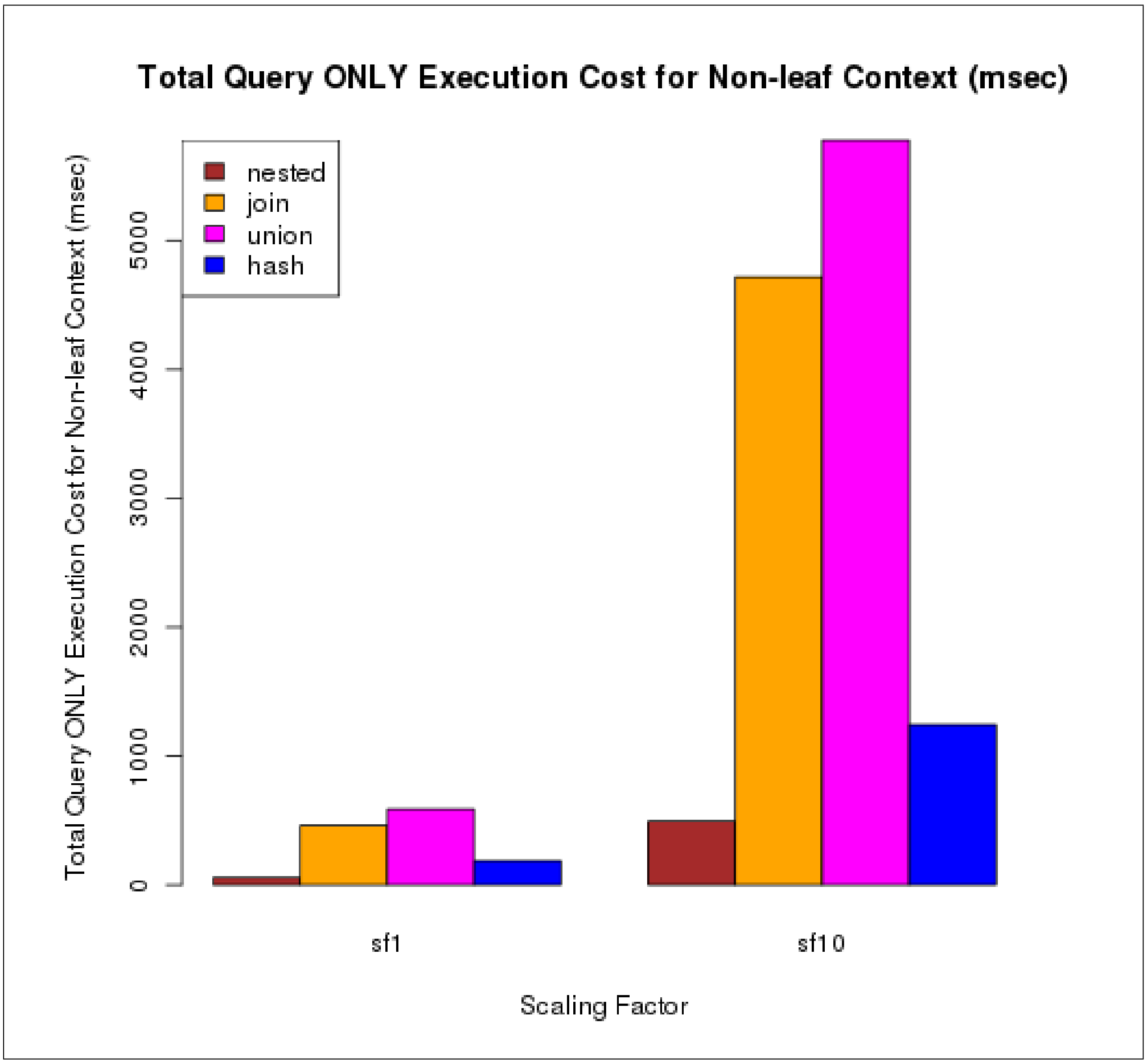

Figure 20: Total Query ONLY Execution Cost for a Non-leaf Context (msec)

Secondly, as has been mentioned in a previous point all the used RDBMs were "developer" editions. Thus, they were restricted to use limited system resources even if more hardware resources were available. For example the database server that was used for the experiments was "locked" on a single CPU core. Thus, the database engine could not employ any parallelization where ever this could have been both possible and beneficial. This statement is backed by the observation that during the benchmarking period one of the cores, of the 4-core processor, was experiencing extremely high usage, above 90%, whereas the other 3 cores were used at maximum 20%.

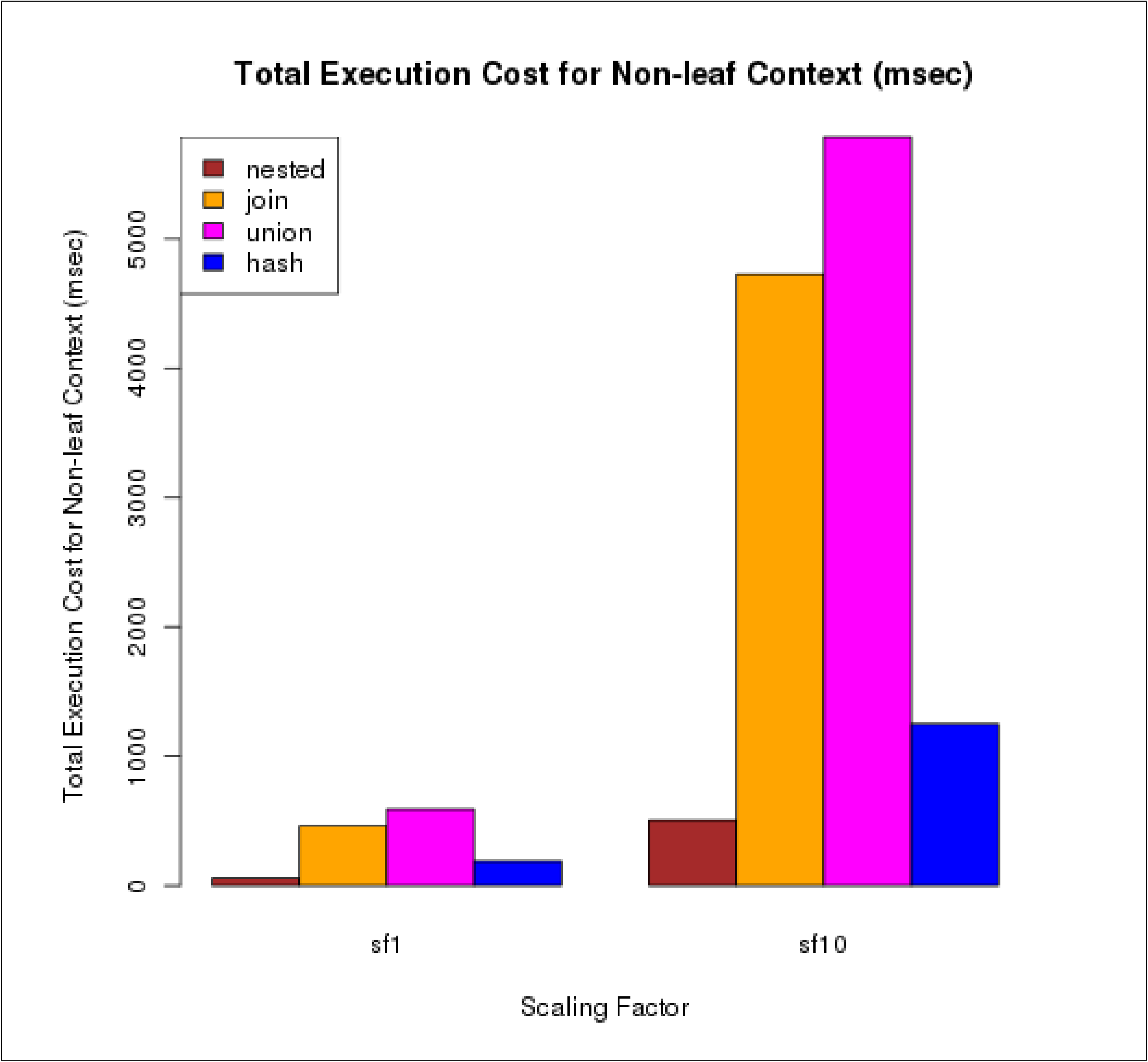

Figure 21: Total Execution Cost for a Non-Leaf Context (msec)

The bar diagran at 21 on page 57 unveils an already met pattern. As was the case with the leaf context tree node also in the non-leaf context tree node the total execution is dominated by the query execution cost on the back end database server. This conclusion signifies that the interpretation cost is not a key factor in the determination of the total cost.

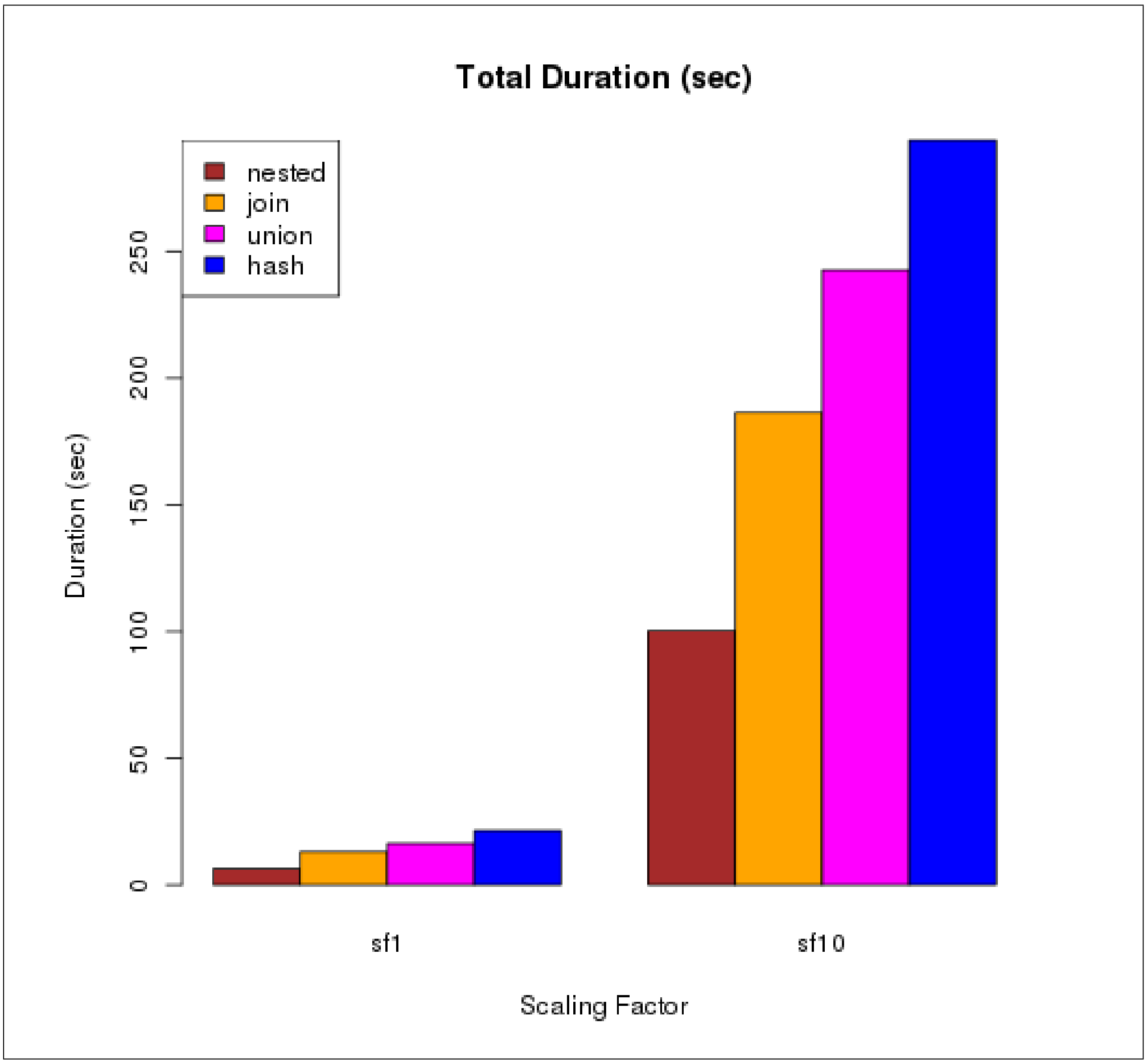

Figure 22: Total Duration under the LSM Network Configuration (sec)

In order to provide an aggregated view of each strategy's performance, we recorded the time that was needed by the benchmarking suite to run under each alternative strategy. The recorded times are depicted at the bar diagram 22 on page 59. The most important conclusion is that the nested strategy is always the cheapest. Although, these unexpected results refute the experimental results of [4], [5], they could be attributed to the fact that our database server was a developer, trial edition limited to use a single core. Thus, the RDBMS could not take advantage of any parallelization opportunities. Another observation is that the unified strategies exhibit similar perfromance behavior which is attributed to the fact that they create and consume their clustered result sets in a very similar fashion. In fact, the outer union strategy is always more expesnive because it is desgined to return many more rows clustered at both the tuple and the column level. The hash join rewriting seems to be the most expensive. Somehow, this was expected since it stores locally, in main memory the entire result set and then performs a hast table lookup for each inner query open request. These two operations cause significant memory consumption and CPU unitlization. As has been mentioned before, the high client CPU cost of the hash join strategy could be amortized with more inner context tuples for each outer cotnext tuple. Another reason that may explain the bad performance of the unified, partitioned strategies versus the original nested strategy is that the entire benchmarking experiment was run under a local shared memory configuration. This means that the client, the Sqlprefetch library and the back end RDBMS were running on the same workstation. Even the prototype of [4], [5] experienced only marginal total latency benefits unfer the same configuration. Client-server communication on the same workstation is characterized by extremely low latency round-trips. The increased CPU usage of the combined queries is not payed off by a large number of round-trips with significant latency. Thus, the problem that the above algorithms attempt to alleviate is not significant under the deployed configuration.

7.2 Different Network Configurations

So far, the experimental data of figure 22 on page 59 have demonstrated that under a local shared memory network configuration, the original, unmodified queries outerperform all the alternative rewriting strategies. This result is mainly attributed to the fact that the network latency is not large enough so as to pay of for the increased CPU usage of the combined queries. Although, this experimental result was the outcome of a benchmark on a real system, we chose to simulate the network latency of various other network settings with smaller bandwidth specifications. The networks whose latency we attempted to simulate are given at table 5 on page 49. In order to do that we developed a trivial network latency simulator in Java (a Java API) that we invoked within every Sqlprefetch API what was communicating with the back end RDBMS through some JDBC API call. The "injected" latency was calculated with the following reasoning. We assumed that all the JDBC API calls that would hit the network were sending 1 TCP packet and receiving 1 TCP packet with both packets having MSS (Maximum Segment Size) 1500 bytes in size. This assumption is quite reasonable, especially if we assume that each result set tuple requires one round-trip so as to be fetched. If we symbolize the bandwidth specification of each network setting with R, the total "injected" latency of each wrapped JDBC API call (D) is calculated with the help of the equation 33 on page 60.

$$D = 2 \times \frac{1500 \times 8}{R}(\text{sec}) \tag{33}$$

With the aforementioned network latency simulator we repeated our benchmarking test for 3 different network settings and for each case we recorded the total duration of the benchmark. The results are given at figure 23 on page 61. In the aforementioned figure we have included the "total duration" for the Local Shared Memory (LSM) configuration as well. In this figure, we present not only the performance of each alternative rewriting under some specified network configuration, but also the impact of each network latency on all the considered strategies.

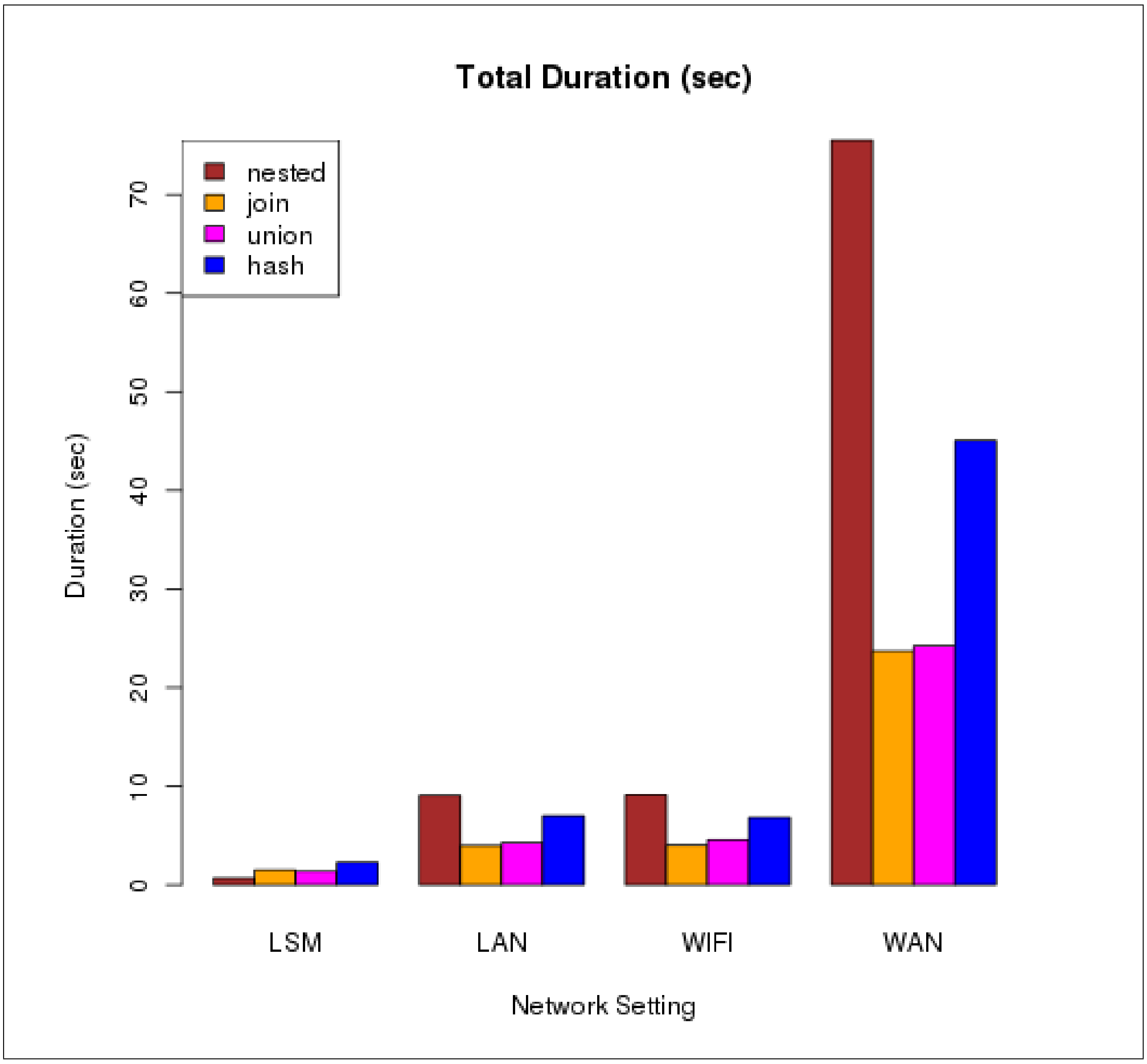

Figure 23: Total Duration under Different Network Configurations (sec)

As can be seen from figure 23 at page 61, all the network configurations except for the LSM network setting, exhibit proportionaly the same behavior. More specifically, for the LAN, WIFI and WAN network settings the unified strategies are four times faster than the nested strategy, whereas the client-hash-join strategy is two times faster than the nested strategy. In addition, the LAN and WIFI settings exhibit almost identical behavior, although the latter's bandwidth is ten times faster than the formers. However, both bandwidths are usually, categorized under very similar cost ranges. The client-hash-join stragegy is two times slower than the unified strategies and this is attributed to the high CPU usage that it exhibits at the initial creation and population of the result sets hash table storage (in main memory). A very useful conclusion that can be drawn from figure 23 at page 61 is that optimizing a relational query stream with semantic prefetching will pay off only only for networks with adequate latency, like LANs, WIFIs and WANs. In other cases it can even hurt performance. This is were the optimizer comes in, who takes into account runtime parameters like the network's latency, the CPU overhead of the combined queries and it is capable to conclude to the most optimum query plan for the current hardware setting.

8. FUTURE WORK

Ivan Bowman presented in [4], [5] three relational query patterns that they thought were commonly found in a variety of enterprise applications. These are the **Nested Request Pattern**, the **Batch Request Pattern** and the **Data Structure Correlated Pattern**. They proposed a number of alternative run time optimizations for the first two. In the third pattern a client code method submits a query on the database server and stores the retrieved results within some custom data structure. Afterwards, the method code loops over the collection data structure with the help of an iterator and for each item it submits a new query on the back end RDBMS. Despite the fact that the third pattern resembles the second, it is not a trivial procedure to observe it during some training period and then optimize it appropriately during runtime. However, it seems that this query stream pattern is used more often than not and it could payoff significantly if instead of "nested" like queries, the intermediate middleware was submitting combined queries that would aim to minimize the total round trips.

Nowdays, the object oriented paradigm is most popular than ever. Nevertheless, most back end persistent storage engines are relational. Both the academic and the research community have attempted to integrate more closely these quite different worlds with the help of the **Object Relational Mappers** (**ORM**). An ORM hides from the application developer the SQL featured database access API (e.g. ODBC, JDBC) and presents him application specified classes which correspond to database relations. Subsequently, a collection class with items belonging to an aforementioned class represents a retrieved result set from the back end database server. The query submission and the retrieval of the produced result set are fully transaprent to the application programmer. For small result sets some optimized ORMs prefer to use some main memory buffering, whereas for larger result sets the keep an open cursor through which they fetch any remaing tuples. ORMs are notorious for their badly formed and inefficient queries. This is where an Sqlprefetch like component library could come in and provide some sophisticated runtime SQL query rewriting.

Our prototype has been developed to operate with ACID semantics under a single user environment. More specifically, it supports multiple users but it does not provide any concurrency egnine that will enforce the known ACID properties of a mature RDBMS. Thus, we cannot gurantee any kind of serializability, which can lead to the observation of data view inconsistencies. A multi-process or multi-threaded Sqlprefetch like middleware could enforce any of the ANSI SQL's isolation levels. This would require some enhacements on the context tree with a number of well defined locking primitives. Providing a number of serializability levels through some concurrency manager operating on the context tree is certain that it would exhibit the classic headaches of RDBMS concurrency managers. These would be lock contention, CPU overhead, handling of deadlocks and livelocks. This enhancement would be critical for enterprise customers requiring some strictly defined concurrency levels.

Sqlprefetch is equipped to identify interesting nested patterns, that may result into more efficient combined queries. These queries could be transformed into materialized views on the back end database server. Although, this is not a trivial process, especially when it comes to keeping these materialized views up to date under the precense of intensive write dominated periods, it could payoff significantly for read dominated workloads. In addition, a future work could consider to move the materialized views functionality on the middleware side and implement it with some form of caching, resulting in partially materialized views. A resembling approach has been researched extensively by [10], [11] where they developed a peer-to-peer middleware that is installed at the edges of a core CDN network caching entire result set queries. Their setting is consisted of a relational back end database server, that acts as the master and a number of peripheral result set caching slave database servers. They attempt to handle various mixes of read and write workloads. Nonetheless, they do not employ materialized views but instead they are based on a sophisticated invalidation algorithm that is supported by a peer-to-peer publish-susbscribe multicast network. An architecturally similar example on the web servers field is the Apache Traffic Server [12]. The Apache Traffic Server attempts to shift the web request hot spots on the edges of a CDN core network so as to alleviate the back end web servers from any excessively high load and distribute client requests more evenly and closer to the actual requesters. In a similar fashion, observing the parts of a candidate materialized view that receive the vast majority of the read requests, an enhanced Sqlprefetch engine could materialize them locally on the edges of a core network, closer to the clients, thus providing both individual latency optimizations and throughput gains. Although, this architectural design is quite common on various application servers it poses significant difficulties in a relational database setting, where the ACID properties are expected to be strictly enforced. A possible solution could be to propagate the write requests directly to the master, back end database server, like [10], [11] and at the same time, through some multicasting infrastructure, publish them on all the slave middlewares, so as to incrementally update any already cached materialized views.

TERMINOLOGY TABLE

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Semantic Prefetching	Σημασιολογική Προανάκληση

ABBREVIATIONS, INITIALS AND ACRONYMS

TCP	Transmission Control Protocol
MSS	Maximum Segment Size
LAN	Local Area Network
MAN	Metropolitan Area Network
API	Application Programming Interface
RDBMS	Relational Database Management System
LSM	Local Shared Memory
ODBC	Open Database Connectivity
JDBC	Java Database Connectivity
ORM	Object Relational Mapper
CDN	Content Distribution Network
SSD	Solid State Disk
ERP	Enterprise Resource Planning
CRM	Customer Relationship Management
CMS	Content Management System
BPM	Business Process Management

BIBLIOGRAPHY

[1] Timos K. Sellis ``Multiple-Query Optimization,'' *ACM Trans. Database Syst.*, [Online]. Available: http: //doi.acm.org/10.1145/42201.42203,db/journals/tods/Sellis88.html. [Accessed: Oct 10, 2011].

[2] Philip A. Bernstein, Shankar Pal, David Shutt ``Context-Based Prefetch for Implementing Objects on Relations,'' *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, [Online]. Available: http://www.vldb.org/conf/1999/ P33.pdf. [Accessed: Oct 10, 2011].

[3] Ivan Bowman, Kenneth Salem ``Optimization of query streams using semantic prefetching.,'' *University of Waterloo*,CM SIGMOD International Conference on Management of Data (SIGMOD'04) [Online]. Available: http://www.cs.uwaterloo.ca/~kmsalem/pubs/BowmanSIGMOD04.pdf. [Accessed: Oct 10, 2011].

[4] Ivan Bowman ``Scalpel: Optimizing Query Streams Using Semantic Prefetching.,'' *ACM Transactions on Database Systems, 2005*, [Online]. Available: http://www.cs.uwaterloo.ca/~kmsalem/pubs/ scalpelTODS.pdf. [Accessed: Oct 10, 2011].

[5] Ivan Bowman ``Optimization of Query Streams Using Semantic Prefetching.,'' *PhD thesis, University of Waterloo, 2005*, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.73. 6386&rep=rep1&type=pdf. [Accessed: Oct 10, 2011].

[6] Ivan Bowman, Kenneth Salem ``Optimization of Query Streams Using Semantic Prefetching.,'' *University of Waterloo, 2006*, [Online]. Available: http://www.cs.uwaterloo.ca/research/tr/2006/CS-2006-43. pdf. [Accessed: Oct 10, 2011].

[7] Ivan Bowman, Kenneth Salem ``Optimization of Query Streams Using Semantic Prefetching.,'' *Proc. International Conference on Data Engineering (ICDE'07)*, [Online]. Available: http://www.cs. uwaterloo.ca/~kmsalem/pubs/BowmanICDE07.pdf. [Accessed: Oct 10, 2011].

[8] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, Berthold Reinwald ``Efficiently Publishing Relational Data as XML Documents,'' *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, [Online]. Available: http://www.vldb.org/conf/2000/P065.pdf. [Accessed: Oct 10, 2011].

[9] ``ANSI. Information Systems Database Language SQL, September 1999.,'' *ISO/IEC 9075-2:1999*, [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm? csnumber=26197. [Accessed: Oct 10, 2011].

[10] Christopher Olston, Amit Manjhi, Charles Garrod, Anastassia Ailamaki, Bruce M. Maggs, Todd C. Mowry ``A Scalability Service for Dynamic Web Applications.,'' *2005*, [Online]. Available: http://www. cidrdb.org/cidr2005/papers/P05.pdf. [Accessed: Oct 10, 2011].

[11] Charles Garrod, Amit Manjhi, Anastasia Ailamaki, Bruce M. Maggs, Todd C. Mowry, Christopher Olston, Anthony Tomasic ``Scalable query result caching for web applications.,'' *2008*, [Online]. Available: http://www.vldb.org/pvldb/1/1453917.pdf. [Accessed: Oct 10, 2011].

[12] ``Apache Traffic Server.,'' *2011*, [Online]. Available: http://trafficserver.apache.org/. [Accessed: Oct 10, 2011].

[13] ``JUnit,'' *2011*, [Online]. Available: http://www.junit.org/. [Accessed: Oct 10, 2011].

[14] ``ANTLR,'' *2011*, [Online]. Available: http://www.antlr.org. [Accessed: Oct 10, 2011].

[15] ``ANTLRWorks,'' *2011*, [Online]. Available: http://www.antlr.org/works/index.html. [Accessed: Oct 10, 2011].

[16] ``PostgreSQL,'' *2011*, [Online]. Available: http://www.postgresql.org/. [Accessed: Oct 10, 2011].