NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Database Techniques for Ontology-based Data Access

Dimitris S. Bilidas

ATHENS

DECEMBER 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Τεχνικές Βάσεων Δεδομένων με Εφαρμογή στην Ανάκτηση Δεδομένων Βάσει Οντολογιών

Δημήτριος Σ. Μπηλίδας

ΑΘΗΝΑ

ΔΕΚΕΜΒΡΙΟΣ 2020

PhD THESIS

Database Techniques for Ontology-based Data Access

Dimitris S. Bilidas

SUPERVISOR: Manolis Koubarakis, Professor UoA

THREE-MEMBER ADVISORY COMMITTEE:

Manolis Koubarakis, Professor UoA

Yiannis Ioannidis, Professor UoA

Vassilis Christophides, Professor Un. of Crete

SEVEN-MEMBER EXAMINATION COMMITTEE

Manolis Koubarakis,
Professor UoA

Yiannis Ioannidis,
Professor UoA

Vassilis Christophides,
Professor Un. of Crete

Dimitris Plexousakis,
Professor Un. of Crete

Giorgos Stamou,
Associate Professor NTUA

Ian Horrocks,
Professor Un. of Oxford

Martin Giese,
Professor Un. of Oslo

Examination Date: 15/12/2020

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Τεχνικές Βάσεων Δεδομένων με Εφαρμογή στην Ανάκτηση Δεδομένων Βάσει Οντολογιών

Δημήτριος Σ. Μπηλίδας

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Εμμανουήλ Κουμπαράκης, Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Εμμανουήλ Κουμπαράκης, Καθηγητής ΕΚΠΑ

Γιάννης Ιωαννίδης, Καθηγητής ΕΚΠΑ

Βασίλης Χριστοφίδης, Καθηγητής Παν. Κρήτης

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Εμμανουήλ Κουμπαράκης,
Καθηγητής ΕΚΠΑ

Γιάννης Ιωαννίδης,
Καθηγητής ΕΚΠΑ

Βασίλης Χριστοφίδης,
Καθηγητής Παν. Κρήτης

Δημήτρης Πλεξουσάκης,
Καθηγητής Παν. Κρήτης

Γιώργος Στάμου,
Αναπληρωτής Καθηγητής ΕΜΠ

Ian Horrocks,
Καθηγητής Παν. Οξφόρδης

Martin Giese,
Καθηγητής Παν. Όσλο

Ημερομηνία Εξέτασης: 15/12/2020

ABSTRACT

Ontology-based Data Acess (OBDA) is a method for linking an ontology, which encodes knowledge about the classes and properties of entities for a given application domain, to underlying data sources. These data sources, managed by specialized systems, can be in various forms and usually reside in pre-existing repositories. The linking is accomplished through declarative mappings, which are used to generate ontology terms from information in the data sources. Instead of materializing all the ontology terms, the user of the relevant application can pose a query over the ontology, and then a process of query transformation is carried out, which has as a result a query in the native language of the underlying data sources. This resulted query is then executed, and the results are presented to the user, transformed as ontology terms. This approach, also known as virtual knowledge graph approach, has the advantage that provides the user with a familiar vocabulary over which he can pose a query, concealing details about the underlying data sources, such as complex schemas and storage particularities. On the other hand, the process of transforming the initial query over the ontology into a query over the underlying sources, leads in many cases to complex and large queries.

In this thesis, we study the problem of efficient query answering for OBDA systems from a database perspective, concentrating on the ontology language OWL 2 QL, which is a dialect of the OWL family specifically tailored for the case where massive data are stored in an external data source. We are also concentrating on the case where the initial query posed over the ontology is in the form of a union of conjunctive queries. As expected, this issue heavily depends on the exact kind of the underlying data source. For this reason we make a distinction between three different commonly encountered scenarios. In the first scenario we consider that the underlying system is a single relational database management system. In the second scenario we consider that we have a federation of different relational systems. Finally, in the third scenario we consider the case where data are stored in a specialized triple store in the form of RDF statements.

For the first scenario, we identify redundant processing as a key problem in OBDA query execution over a relational system. Examples of such processing are duplicate answers obtained during query evaluation, which must finally be discarded, or common expressions evaluated multiple times from different parts of the same complex query. Many optimizations that aim to minimize this problem have been proposed and implemented, mostly based on semantic query optimization techniques, by exploiting ontological axioms and constraints defined in the database schema. However, operations that introduce redundant processing are still generated in many practical settings, and this is a factor that impacts query execution. To handle this issue, we propose a cost-based method for query translation, which starts from an initial default translation and uses information about redundant processing in order to come up with an equivalent, more efficient translation. The method operates in a number of steps, by relying on certain heuristics indicating that we obtain a more efficient query in each step. Through experimental evaluation using the

Ontop system for ontology-based data access, we exhibit the benefits of our method.

For the second scenario we have developed a system that acts as a mediator between the OBDA system and the federated databases. This system, built using the Exareme engine, decomoses the produced query into different fragments, sends these fragments for evaluation in the external databases and imports the intermediate results that correspond to these fragments. Finally, these intermediate results are combined in order to produce the final query result. During this process, we have adapted techniques and methods from database literature for usage in the context of OBDA. These methods cover areas such as data integration, common subexpression identification, caching of intermediate query results and distributed processing. The developed mediator system has been integrated into the platform of the Optique reseacrh project and has been successfully deployed in a demanding real world use case, federating seven different databases which contain geological data.

For the last scenario, we have developed PARJ, a specialized in-memory RDF store which takes into consideration ontological hierarchies during join processing with very low performance overhead, using on-the-fly computation of the inferences regarding class and property hierarchies. In the spirit of the OBDA apporach, PARJ avoids expensive preprocessing and materialization of implications. PARJ is also amenable to straightforward parallelization. Specifically, we present a join implementation that allows to achieve any desired degree of parallelism on arbitrary join queries and RDF graphs stored in memory using compact vertical partitioning. We use an adaptive join processing approach, such that we take advantage of complete or even partial ordering of RDF data, which is compactly stored in order to increase spatial locality and keep memory consumption low, coupled with an ID-to-Position vector index used when ordering does not allow for efficient scanning of the input relation. Finally, we experimentally show the efficiency and scalability of our proposal.

SUBJECT AREA: Databases

KEYWORDS: OBDA, Query Optimization, Semantic Web, Knowledge Graphs, Data Integration

ΠΕΡΙΛΗΨΗ

Στην επιστημονική περιοχή της Αναπαράστασης Γνώσης και Συλλογιστικής, οι οντολογίες διαδραματίζουν καθοριστικό ρόλο στη μοντελοποίηση γνώσης για έναν τομέα εφαρμογών. Μια οντολογία κωδικοποιεί πληροφορίες σχετικά με τις κατηγορίες αντικειμένων του τομέα και τις σχέσεις μεταξύ τους, παρέχοντας στους χρήστες της εφαρμογής με μια οικεία μοντελοποίηση του τομέα. Επιπλέον, αξιώματα εκφρασμένα σε λογική και κωδικοποιημένα στην οντολογία, μπορούν να χρησιμοποιηθούν για την απόκτηση νέας γνώσης μέσω συμπερασμού. Η Κοινοπραξία του παγκόσμιου ιστού (World Wide Web Consortium-W3C) συνιστά τη χρήση της γλώσσας οντολογιών OWL, ως την οικογένεια γλωσσών για αναπαράσταση γνώσης στον παγκόσμιο ιστό.

Η ανάκτηση δεδομένων βάσει οντολογιών (Ontology-based Data Access-OBDA) είναι μια μέθοδος διασύνδεσης οντολογιών με υποκείμενες εξωτερικές πηγές δεδομένων μέσω δηλωτικών αντιστοιχίσεων (mappings). Οι συγκεκριμένες αντιστοιχίσεις μπορούν να θεωρηθούν ως κανόνες που δημιουργούν αντικείμενα της οντολογίας βάσει επερωτήσεων στα εξωτερικά δεδομένα. Στη συνέχεια, ένας χρήστης μπορεί να θέσει μια επερώτηση στην οντολογία και αυτή η επερώτηση μπορεί να μεταφραστεί στη γλώσσα επερωτήσεων του υποκείμενου συστήματος διαχείρισης δεδομένων, ή αλλιώς της εξωτερικής πηγής δεδομένων, χρησιμοποιώντας τις αντιστοιχίσεις και αποστέλλεται για εκτέλεση, παρέχοντας στο χρήστη τα επιθυμητά αποτελέσματα, σαν τα δεδομένα να αποτελούσαν εξαρχής μέρος της οντολογίας. Παρά το ότι από άποψη πολυπλοκότητας υπάρχουν αποτελεσματικοί αλγόριθμοι για απάντηση επερωτήσεων σε συστήματα που πραγματοποιούν ανάκτηση δεδομένων βάσει οντολογιών (OBDA-συστήματα), το τελικό ερώτημα που προκύπτει και που πρέπει να εκτελεστεί στις εξωτερικές βάσεις δεδομένων είναι σε πολλές πρακτικές περιπτώσεις πολύπλοκο και μεγάλο. Για παράδειγμα, δεν είναι ασυνήθιστο σε ένα τυπικό OBDA σενάριο, σε περίπτωση που στην οντολογία ορίζονται μεγάλες ιεραρχίες κλάσεων και ιδιοτήτων, μια αρχική συζευκτική επερώτηση πάνω στην οντολογία να μεταφραστεί σε μια ένωση συζευκτικών επερωτήσεων, που μπορεί να περιέχει εκατοντάδες ή χιλιάδες υποερωτήματα.

Η παρούσα διδακτορική διατριβή προσφέρει τεχνικές για την αντιμετώπιση του προαναφερθέντος ζητήματος από την πλευρά των βάσεων δεδομένων. Συγκεκριμένα, επικεντρωνόμαστε στην διάλεκτο OWL 2 QL της οικογένειας γλωσσών για οντολογίες OWL, η οποία διάλεκτος είναι ειδικά προσαρμοσμένη για την περίπτωση μεταγραφής επερωτήσεων όταν έχουμε μαζικά δεδομένα σε εξωτερικές πηγές. Σε σχέση με τις αντιστοιχίσεις ανάμεσα στην οντολογία και τις εξωτερικές πηγές, επικεντρωνόμαστε στην γλώσσα αντιστοιχήσεων R2RML, η οποία αποτελεί επίσημη σύσταση του W3C, ή άλλες γλώσσες παρόμοιας εκφραστικότητας. Σε αυτό το πλαίσιο, κάνουμε μια διάκριση μεταξύ τριών διαφορετικών σεναρίων OBDA, ανάλογα με το είδος του υποκείμενου συστήματος διαχείρισης δεδομένων, και υλοποιούμε αποδοτικές τεχνικές για κάθε ένα από τρία αυτά σενάρια. Συγκεκριμένα, κατά το πρώτο σενάριο θεωρούμε ότι τα δεδομένα αποθηκεύονται σε μια εξωτερική σχεσιακή βάση δεδομένων με αυθαίρετο σχεσιακό σχήμα. Κατά το

δεύτερο σενάριο τα δεδομένα αποθηκεύονται σε πολλές εξωτερικές βάσεις δεδομένων, όπου η καθεμία έχει το δικό της σχήμα και μπορεί να βρίσκεται σε διαφορετική τοποθεσία. Τέλος, στο τρίτο σενάριο τα υποκείμενα δεδομένα βρίσκονται υπό την μορφή γράφου, ακολουθώντας το μοντέλο RDF, και είναι αποθηκευμένα σε ένα εξειδικευμένο σύστημα διαχείρισης αυτού του είδους των δεδομένων. Το RDF είναι ένα μοντέλο δεδομένων ευρέως χρησιμοποιούμενο για την ενσωμάτωση δεδομένων από διαφορετικές πηγές, ακολουθώντας ένα απλό σχήμα γράφου κατά το οποίο τα δεδομένα μοντελοποιούνται ως τριπλέτες που έχουν τη μορφή υποκείμενο-κατηγόρημα-αντικείμενο. Σε κάθε ένα από αυτά τα σενάρια προτείνονται λύσεις και αναπτύσσονται συστήματα τα οποία βελτιώνουν τους χρόνους εκτέλεσης επερωτήσεων, ενώ η συνεισφορά της παρούσας διατριβής επιβεβαιώνεται εμπειρικά με την διεξαγωγή εκτεταμένων πειραμάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Βάσεις Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ανάκτηση Δεδομένων Βάσει Οντολογιών, Βελτιστοποίηση Επερωτήσεων, Σημασιολογικός Ιστός, Γράφοι Γνώσης, Ενοποίηση Δεδομένων

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ ΤΗΣ ΔΙΔΑΚΤΟΡΙΚΗΣ ΔΙΑΤΡΙΒΗΣ

Εισαγωγή

Στην επιστημονική περιοχή της Τεχνητής Νοημοσύνης οι οντολογίες αποτελούν ένα φορμαλισμό με στόχο την εννοιολογική αναπαράσταση γνώσης για κάποιο πεδίο ενδιαφέροντος. Σε μία οντολογία τα αντικείμενα του πεδίου καταχωρούνται σε κλάσεις αντικειμένων. Για παράδειγμα, αν θέλουμε να μοντελοποιήσουμε την πληροφορία σχετικά με ένα σχολείο, θα μπορούσαμε να ορίσουμε σαν κλάσεις αντικειμένων τις εξής: ΚΑΘΗΓΗΤΗΣ, ΜΑΘΗΤΗΣ, ΜΑΘΗΜΑ, ΑΝΘΡΩΠΟΣ. Έτσι θα μπορούσαμε να πούμε ότι ο Γιώργος είναι μαθητής, ο κύριος Γερασίμου είναι καθηγητής και η Μουσική είναι μάθημα. Έπειτα, στην οντολογία ορίζονται ιδιότητες (ή ρόλοι) μέσω των οποίο συσχετίζονται συγκεκριμένα αντικείμενα του πεδίου. Για παράδειγμα, θα μπορούσαμε μεταξύ άλλων να έχουμε την ιδιότητα ΔΙΔΑΣΚΕΙ, για να πούμε ότι ο κύριος Γερασίμου διδάσκει μουσική. Η οργάνωση των κλάσεων και των ιδιοτήτων σε μια οντολογία συνήθως έχει ιεραρχική δομή. Για παράδειγμα μπορούμε να πούμε ότι η κλάσεις ΚΑΘΗΓΗΤΗΣ και ΜΑΘΗΤΗΣ είναι υποκλάσεις της κλάσης ΑΝΘΡΩΠΟΣ, με την έννοια ότι κάθε καθηγητής είναι επίσης και άνθρωπος, και το ίδιο ισχύει για κάθε μαθητή. Ένα ακόμα σημαντικό χαρακτηριστικό των οντολογιών είναι ότι χρησιμοποιούνται για την εξαγωγή συμπερασμών, δηλαδή γνώσης η οποία δεν έχει αναπαρασταθεί ρητά, αλλά μπορεί να εξαχθεί από τους κανόνες της οντολογίας. Για παράδειγμα, με βάση όσα αναφέραμε προηγουμένως, μπορούμε να εξάγουμε ως συμπερασμό το γεγονός ότι ο Γιώργος και ο κύριος Γερασίμου ανήκουν στην κλάση ΑΝΘΡΩΠΟΣ.

Για το σκοπό της αναπαράστασης γνώσης, κατά τη δεκαετία του 1970 χρησιμοποιήθηκαν συστήματα όπως τα Πλαίσια (Frames) και τα Σημασιολογικά Δίκτυα (Semantic Networks), τα οποία δεν όριζαν κάποιο τυπικό σύστημα για συμπερασμό. Αργότερα, κατά τη δεκαετία του 1980 η έρευνα επικεντρώθηκε σε συστήματα βασισμένα στη μαθηματική λογική, και γρήγορα έγινε αντιληπτό ότι υπήρχε άμεση αντιστοιχία ανάμεσα στο πόσο εκφραστική είναι η χρησιμοποιούμενη λογική και στο πόσο υπολογιστικά αποδοτικές είναι οι τεχνικές συμπερασμού. Αυτή η διαπίστωση οδήγησε από τα τέλη της δεκαετίας του 1980 στην ανάπτυξη των Περιγραφικών Λογικών, οι οποίες εξετάζουν αυτές τις λογικές από άποψη εκφραστικότητας σε σχέση με την υπολογιστική πολυπλοκότητα των διαδικασιών συμπερασμού. Ως βασικό εγχειρίδιο για τη μελέτη των Περιγραφικών Λογικών έχει καθιερωθεί το [9]. Σχετικά με την εκφραστικότητα της εκάστοτε λογικής, σαν παράδειγμα μπορούμε να θεωρήσουμε το κατά πόσο μπορεί κάποιος κατά τη μοντελοποίηση του πεδίου να χρησιμοποιήσει αξιώματα που αναφέρουν ότι κάποια ιδιότητα είναι μεταβατική ή συναρτησιακή, ή ότι μια ιδιότητα είναι η αντίστροφη μιας άλλης ιδιότητας, ή ότι μια κλάση είναι η ένωση κάποιων άλλων ξένων μεταξύ τους κλάσεων. Έτσι για παράδειγμα, σε κάποιες Περιγραφικές Λογικές μπορούμε να μοντελοποιήσουμε το γεγονός ότι κάποιος είναι μετεξεταστέος αν και μόνο αν έχει πάρει σε ένα τουλάχιστον μάθημα κάτω από τη βάση, ενώ σε άλλες όχι. Τέλος, στις Περιγραφικές Λογικές γίνεται μια διάκριση ανάμεσα

στα αξιώματα, όπως αυτα που αναφέραμε ή αυτά που ορίζουν την ιεραρχία των κλάσεων και των ιδιοτήτων, και στα απλά γεγονότα που δηλώνουν ότι ένα αντικείμενο του πεδίου ανήκει σε μια κλάση ή ότι ένα αντικείμενο συνδέεται μέσω μιας ιδιότητας με ένα άλλο αντικείμενο. Το μέρος το οποίο περιλαμβάνει τα αξιώματα ονομάζεται $\mathcal{T}$-box (terminology), ενώ το μέρος που περιλαμβάνει τα γεγονότα ονομάζεται $\mathcal{A}$-box (assertional knowledge).

Κατά τη δεκαετία του 2000 η Κοινοπραξία του Παγκοσμίου Ιστού (WWW Consortium, συντ. W3C) ανέπτυξε την οικογένεια γλωσσών OWL για αναπαράσταση γνώσης στον παγκόσμιο ιστό. Η συγκεκριμένη οικογένεια γλωσσών βασίζεται στις περιγραφικές λογικές, και χρησιμοποιεί το μοντέλο δεδομένων RDF (Resource Description Framework). Το μοντέλο RDF προσφέρει έναν απλό τρόπο για την αναπαράσταση πληροφορίας με βάση τριπλέτες της μορφής υποκείμενο-κατηγόρημα-αντικείμενο, κατασκευάζοντας έτσι ουσιαστικά ένα γράφο γνώσης. Σε αυτόν το γράφο ορίζονται επίσης αξιώματα μέσω της OWL και μπορούν να χρησιμοποιηθούν τεχνικές συμπερασμού. Η τελευταία έκδοση της οικογένειας γλωσσών OWL ονομάζεται OWL 2 και περιλαμβάνει 3 προφίλ με διαφορετική εκφραστικότητα. Ένα από αυτά τα προφίλ, το OWL 2 QL, βασίζεται στην οικογένεια περιγραφικών λογικών DL-Lite [22, 23], έχει σχεδιαστεί έτσι ώστε να είναι υπολογιστικά αποδοτική η απάντηση συζευκτικών επερωτήσεων. Έτσι για παράδειγμα, το συγκεκριμένο προφίλ είναι πρακτικό να χρησιμοποιηθεί όταν έχουμε μεγάλο όγκο δεδομένων, τα οποία μπορεί να είναι αποθηκευμένα σε μια εξωτερική σχεσιακή βάση δεδομένων, η οποία έχει δημιουργηθεί ανεξάρτητα από την οντολογία. Η συγκεκριμένη μέθοδος για εκτέλεση επερωτήσεων ονομάζεται ανάκτηση δεδομένων βάσει οντολογιών (Ontology-based Data Access, συντ. OBDA), ενώ χρησιμοποιείται και ο όρος εικονικός γράφος γνώσης (virtual knowledge graph) για να περιγράψει την προσέγγιση.

Ανάκτηση Δεδομένων Βάσει Οντολογιών

Στο [77] μελετάται ανάκτηση δεδομένων βάσει οντολογιών για την περίπτωση που τα δεδομένα βρίσκονται αποθηκευμένα σε μία εξωτερική σχεσιακή βάση δεδομένων και για οντολογίες της οικογένειας DL-Lite. Κατά τη συγκεκριμένη διαδικασία δημιουργούνται δηλωτικές αντιστοιχίσεις (mappings) για τη δημιουργία γεγονότων της οντολογίας με βάση τα δεδομένα που είναι αποθηκευμένα στη βάση δεδομένων και μελετάται η διαδικασία μετάφρασης μίας αρχικής συζευκτικής επερώτησης πάνω στην οντολογία σε μία τελική επερώτηση εκφρασμένη στη γλώσσα SQL, έτοιμη να εκτελεστεί στη βάση δεδομένων. Η συγκεκριμένη διαδικασία χωρίζεται σε δύο διακριτά βήματα. Πρώτα έχουμε τη μεταγραφή (rewriting) ή αναδιαμόρφωση (reformulation) της επερώτησης λαμβάνοντας υπόψιν τα αξιώματα της οντολογίας, και έπειτα την ανάπτυξη (unfolding) της επερώτησης με βάση τις αντιστοιχίσεις. Το πρώτο βήμα βασίζεται στον αλγόριθμο PerfetRef[23] και έχει σαν αποτέλεσμα μια ένωση συζευκτικών επερωτήσεων πάνω στην οντολογία, ενώ το δεύτερο βήμα βασίζεται σε μερική αποτίμηση (partial evaluation)[61] λογικών προγραμμάτων.

Σύντομα παρατηρήθηκε ότι η ένωση συζευκτικών επερωτήσεων που παράγεται κατά την παραπάνω διαδικασία είναι σε πολλές περιπτώσεις απαγορευτικά μεγάλη προκειμένου να γίνει αποτελεσματική εκτέλεση του αποτελέσματος της μετάφρασης από την εξωτερική βάση δεδομένων. Για παράδειγμα, σε πολλές πρακτικές περιπτώσεις μπορεί να περιέχει

χιλιάδες συζευκτικές υποερωτήσεις. Σαν συνέπεια της συγκεκριμένης παρατήρησης προτάθηκαν βελτιστοποιημένες μέθοδοι που παράγουν επερωτήσεις με λιγότερα υποερωτημάτα, βασιζόμενες σε σημασιολογική βελτιστοποίηση [66, 27], οι οποίες σε πολλές περιπτώσεις παράγουν καλύτερο αποτέλεσμα, ωστόσο η αποδοτική αποτίμηση από την εξωτερική βάση δεδομένων παραμένει ένα πρόβλημα. Προκειμένου να επιλυθεί αυτό το πρόβλημα, προτάθηκαν ορισμένες μέθοδοι μεταγραφής με στόχο την παραγωγή πιο συμπαγούς μεταγραφής υπό τη μορφή μη αναδρομικού προγράμματος Datalog, αντί για ένωση συζευκτικών επερωτήσεων[86, 52], όμως η αποτελεσματική αποτίμηση τέτοιου είδους προγραμμάτων από τα υπάρχοντα συστήματα διαχείρισης βάσεων δεδομένων παραμένει ένα ανοιχτό ζήτημα. Στο [20] πραγματοποιείται σύγκριση διαφορετικών μεταγραφών με βάση το κόστος και στη γενική περίπτωση η τελική μεταγραφή θα είναι πάλι μια ένωση συζευκτικών επερωτήσεων. Το Semantic Index [82] περιέχει μια αριθμητική κωδικοποίηση των ιεραρχιών κλάσεων και ιδιοτήτων και αποθηκεύει δεδομένα RDF σε σχεσιακές βάσεις με συγκεκριμένο σχήμα, χρησιμοποιώντας κατάλληλα ευρετήρια B-tree, έτσι ώστε η ιδιότητα μέλους κλάσης και ιδιότητας να μπορεί να προσδιοριστεί από ερωτήματα εύρους για αυτά τα ευρετήρια, αποφεύγοντας ένα μεγάλο αριθμό υποερωτημάτων.

Σχετικά με την υλοποίηση συστημάτων που πραγματοποιούν ανάκτηση δεδομένων βάσει οντολογιών, έχει παρατηρηθεί ότι στην πράξη είναι πιο αποδοτικό να κωδικοποιείται η πληροφορία που αφορά τις ιεραρχίες κλάσεων και ιδιοτήτων στις αντιστοιχίσεις. Με αυτό τον τρόπο, οι συγκεκριμένες ιεραρχίες μπορούν να αγνοηθούν κατά το βήμα της αναδιαμόρφωσης, και να ληφθούν υπόψιν μόνο κατά το βήμα της ανάπτυξης. Έτσι, για παράδειγμα το σύστημα Ontop [21], χρησιμοποιεί τα λεγόμενα $\mathcal{T}$-Mappings που ακολουθούν αυτήν ακριβώς την πρακτική, ενώ με παρόμοια λογική χρησιμοποιούνται οι κορεσμένες αντιστοιχίσεις (saturated mappings) από το σύστημα Ultrawrap-OBDA [92]. Σχετικά με την αρχιτεκτονική του συστήματος Ontop, όπως αυτή περιγράφεται στο [82], χρησιμοποιεί τη μέθοδος tree-witness [52] για το βήμα της αναδιαμόρφωσης, και τη μερική αποτίμηση λογικών προγραμμάτων για το βήμα της ανάπτυξης μέσω των $\mathcal{T}$-Mappings. Πρόσφατα ωστόσο[105], έχει εισαχθεί η έννοια του ενδιάμεσου επερωτήματος (intermediate query) για να πραγματοποιηθεί η μετάφραση, εγκαταλείποντας τη μέθοδο ανάπτυξης μέσω μερικής αποτίμησης. Όσον αφορά την αρχική επερώτηση πάνω στην οντολογία, δεδομένου ότι το Ontop παρουσιάζει τα δεδομένα ως έναν εικονικό RDF γράφο, αυτή εκφράζεται στη γλώσσα SPARQL[37], μία γλώσσα επερωτήσεων για το μοντέλο RDF.

Στην παρούσα διατριβή μελετάμε το πρόβλημα της αποδοτικής αποτίμησης επερωτήσεων που παράγονται από συστήματα που επιτελούν ανάκτηση δεδομένων βάσει οντολογιών κυρίως από τη σκοπιά των βάσεων δεδομένων, βασιζόμενοι σε μοντέλα κόστους εκτέλεσης. Επικεντρωνόμαστε στην περίπτωση που η αρχική οντολογία είναι εκφρασμένη στη γλώσσα OWL 2 QL και μελετάμε τη μετάφραση αρχικών επερωτήσεων πάνω στην οντολογία που έχουν τη μορφή της ένωσης συζευκτικών επερωτήσεων. Καθώς το συγκεκριμένο πρόβλημα εξαρτάται άμεσα από το είδος των εξωτερικών πηγών δεδομένων, κάνουμε μια διάκριση σε τρία διαφορετικά σενάρια εφαρμογής. Κατά το πρώτο σενάριο θεωρούμε ότι τα δεδομένα είναι αποθηκευμένα σε μία σχεσιακή βάση δεδομένων. Κατά το δεύτερο σενάριο θεωρούμε ότι υπάρχουν πολλές διαφορετικές

σχεσιακές βάσεις δεδομένων, και τέλος στο τρίτο σενάριο θεωρούμε ότι τα δεδομένα είναι αποθηκευμένα στη μορφή RDF και τα διαχειρίζεται ένα εξειδικευμένο σύστημα αποθήκευσης και διαχείρισης τέτοιου είδους δεδομένων. Επίσης, κατά τη συγκεκριμένη διατριβή χρησιμοποιούμε το Ontop ως το σύστημα που επιτελεί την ανάκτηση δεδομένων βάσει οντολογιών, και τροποποιούμε διάφορα χαρακτηριστικά του όπου αυτό χρειάζεται από τις μεθόδους που προτείνουμε, ώστε να επιτύχουμε βελτιωμένη απόδοση για την κάθε περίπτωση. Το σύστημα Ontop έχει αναπτυχθεί από το Ανοιχτό Πανεπιστήμιο του Μπόζεν-Μπολζάνο και θεωρείται από τα πλέον εξελιγμένα και αξιόπιστα συστήματα για τη συγκεκριμένη εργασία τόσο από άποψη λειτουργικότητας, όσο και αποδοτικότητας. Στη συνέχεια παρουσιάζουμε τη συνεισφορά της διατριβής σε κάθε ένα από τα προαναφερθέντα σενάρια.

Διαχείριση της περιττής επεξεργασίας κατά την εκτέλεση επερωτήσεων

Κατά το πρώτο σενάριο, μελετάμε την περίπτωση κατά την οποία τα δεδομένα αποθηκεύονται σε μια εξωτερική σχεσιακή βάση δεδομένων με αυθαίρετο σχεσιακό σχήμα, κατά την οποία η αρχική επερώτηση πρέπει να μεταγραφεί σε μία επερώτηση εκφρασμένη στη γλώσσα επερωτήσεων σχεσιακών βάσεων SQL. Σε αυτό το σενάριο, η περιττή επεξεργασία εντοπίζεται ως το βασικό πρόβλημα το οποίο καθιστά αναποτελεσματική την εκτέλεση. Ως περιττή επεξεργασία εννοούμε i) διπλότυπες απαντήσεις και ii) επαναλαμβανόμενη πρόσβαση στα ίδια δεδομένα από διαφορετικά σημεία μίας πολύπλοκης επερώτησης ακόμα και ελλείψει διπλότυπων απαντήσεων. Σχετικά με τις διπλότυπες απαντήσεις, αυτές προέρχονται από διαφορετικούς τρόπους που κάποιο γεγονός της οντολογίας μπορεί να προκύψει από τα δεδομένα. Για παράδειγμα, μπορεί ένα διπλότυπο να προκύψει από δύο διαφορετικά υποερωτήματα (συζευκτικά υποερωτήματα) μίας ένωσης συζευκτικών ερωτημάτων. Η επαναλαμβανόμενη πρόσβαση στα ίδια δεδομένα ακόμα και ελλείψει διπλότυπων απαντήσεων αναφέρεται σε ανάγνωση του ίδιου σχεσιακού πίνακα πολλές φορές σε μία επερώτηση, όπως για παράδειγμα από διαφορετικά υποερωτήματα.

Για την αντιμετώπιση του συγκεκριμένου ζητήματος διερευνώνται διαφορετικές μεταγραφές των επερωτήσεων χρησιμοποιώντας έναν αλγόριθμο βασισμένο στο κόστος εκτέλεσης, για να επιτευχθεί μία ισοδύναμη, αλλά πιο αποτελεσματική μεταγραφή, σε σύγκριση με την προκαθορισμένη μεταγραφή που λαμβάνεται μέσω της μερικής αποτίμησης λογικών προγραμμάτων. Συγκεκριμένα, αρχικά εκτελείται ο αλγόριθμος της μερικής αποτίμησης για τη λήψη της προκαθορισμένης μεταγραφής. Κατά τη συγκεκριμένη εκτέλεση κρατώνται πληροφορίες που αφορούν τις αντιστοιχίσεις που χρησιμοποιήθηκαν. Έπειτα, οι αντιστοιχίσεις που αφορούν το ίδιο στοιχείο της οντολογίας εξετάζονται σχετικά με το αν θα είναι προτιμότερο από άποψη κόστους να δημιουργήσουμε μια αντιστοίχιση που να περιλαμβάνει όλες τις υπόλοιπες που αφορούν το ίδιο στοιχείο της οντολογίας. Για αυτό το λόγο χρησιμοποιούμε την έννοια της συνδυασμένης αντιστοίχισης (combined mapping) και δείχνουμε ότι το λογικό πρόγραμμα που χρησιμοποιεί τη συνδυασμένη αντιστοίχιση είναι ισοδύναμο με το αρχικό, και το αποτέλεσμα της μετάφρασης είναι σωστό. Έτσι, μπορούμε να υπολογίζουμε και να

σώσουμε σε ένα προσωρινό αποτέλεσμα τις επερωτήσεις που αφορά η συνδυασμένη αντιστοίχιση και να εξετάσουμε ποια από τις δύο επιλογές είναι προτιμότερη από άποψη κόστους. Για το σκοπό αυτό εισάγουμε συγκεκριμένες ευρετικές σχετικά με την απαλοιφή διπλότυπων και την επαναλαμβανόμενη πρόσβαση σε πίνακες. Έπειτα εκτελούμε μια άπληστη αναζήτηση όσον αφορά τις επιλογές για το ποιες συνδυασμένες αντιστοιχίσεις θα χρησιμοποιηθούν, επιλέγοντας σε κάθε βήμα όποια δίνει το μεγαλύτερο κέρδος. Έπειτα υπολογίζουμε τη μερική αποτίμηση με την επιλεχθείσα συνδυασμένη αντιστοίχιση και εξετάζουμε τις υπόλοιπες, έως ότου δεν υπάρχει κάποια που να δίνει κέρδος.

Η προτεινόμενη μέθοδος έχει ως στόχο να ελαχιστοποιήσει την περιττή επεξεργασία από την πλευρά του σχεσιακού συστήματος βάσεων δεδομένων σχετικά με διπλότυπες απαντήσεις που προκύπτουν κατά την εκτέλεση, καθώς και σχετικά με επαναλαμβανόμενη πρόσβαση στα ίδια δεδομένα πολλαπλές φορές κατά την εκτέλεση της επερώτησης. Κατά τη διδακτορική διατριβή, η συγκεκριμένη μέθοδος υλοποιήθηκε ως μια επέκταση του OBDA-συστήματος Ontop, και επιβεβαιώθηκε πειραματικά ότι προσφέρει καλύτερους χρόνους εκτέλεσης σε διαφορετικές ομάδες δεδομένων και επερωτήσεων, σε σύγκριση με άλλες πλέον πρόσφατα δημοσιευμένες μεθόδους μεταγραφής.

Συνοπτικά, η συνεισφορά του συγκεκριμένου μέρους της διατριβής έχει ως εξής:

- Προτείνουμε μια καινοτόμο επέκταση πάνω σε προηγουμένως δημοσιευμένη μέθοδο για μετάφραση επερωτήσεων όπως αυτή πραγματοποιείται από συστήματα που πραγματοποιούν ανάκτηση δεδομένων βάσει οντολογιών. Η επέκταση βασίζεται σε μερική αποτίμηση λογικών προγραμμάτων και παρέχει μία πλήρη διαδικασία βασισμένη στο κόστος εκτέλεσης για τη μετάφραση των συγκεκριμένων επερωτήσεων
- Προτείνουμε συγκεκριμένες εκτιμήσεις κόστους για την προαναφερθείσα επέκταση, οι οποίες στοχεύουν στην ελαχιστοποίηση της περιττής επεξεργασίας, τόσο υπό τη μορφή των διπλότυπων απαντήσεων, όσο και υπό τη μορφή της επαναλαμβανόμενης πρόσβασης στα σχεσιακά δεδομένα.
- Υλοποίηση των συγκεκριμένων επεκτάσεων στο σύστημα Ontop και εκτενής πειραματική αξιολόγηση που επιβεβαιώνει την αποτελεσματικότητα της προσέγγισής μας.

Ανάπτυξη Συστήματος-Διαμεσολαβητή

Στο δεύτερο σενάριο, τα δεδομένα αποθηκεύονται σε πολλές εξωτερικές βάσεις δεδομένων, όπου η καθεμία έχει το δικό της σχήμα και μπορεί να βρίσκεται σε διαφορετική τοποθεσία. Σε αυτήν την περίπτωση αναπτύξαμε ένα σύστημα-μεσολαβητή, το οποίο είναι σε θέση να εκτελέσει την επερώτηση που προέκυψε από το αρχικό OBDA-σύστημα, έχοντας πρόσβαση σε όλες αυτές τις βάσεις δεδομένων. Κατά την ανάπτυξη του συστήματος επεκτάθηκαν και προσαρμόστηκαν δημοσιευμένες τεχνικές βάσεων δεδομένων σε θέματα σχετικά με ταυτοποίηση κοινών υποεκφράσεων (common subexpression identification), την αποθήκευση και ανάκτηση επί μέρους αποτελεσμάτων από

προσωρινή μνήμη (query caching), τη βελτιστοποίηση επερωτήσεων για ενοποίηση δεδομένων (data integration) και την κατανεμημένη επεξεργασία επερωτήσεων. Το συγκεκριμένο σύστημα που αναπτύχθηκε σε αυτό το μέρος της διδακτορικής διατριβής βασίζεται στο Exareme, ένα σύστημα για κατανεμημένη επεξεργασία ελαστικών ροών δεδομένων [55] και είναι το πρώτο δημοσιευμένο σύστημα που χρησιμοποιεί ταυτόχρονα όλες αυτές τις τεχνικές και δρα ως μεσολαβητής για την εκτέλεση OBDA επερωτήσεων σε ένα τέτοιο περιβάλλον.

Συγκεκριμένα, ακολουθώντας τη μέθοδο που παρουσιάζεται στο [87], αναπαριστούμε τους τελεστές κάθε ενός υποερωτήματος που περιλαμβάνεται στην ένωση συζευκτικών επερωτησεων που παράγεται από τη μετάφραση της αρχικής επερώτησης πάνω στην οντολογία σε ένα γράφο AND-OR. Σε αυτό το γράφο οι κόμβοι AND αναπαριστούν τελεστές (όπως για παράδειγμα μια ζεύξη) με τις ακμές που καταλήγουν στον κόμβο να αναπαριστούν τις εισόδους του αντίστοιχου τελεστή. Οι κόμβοι OR αναπαριστούν ενδιάμεσα αποτελέσματα, τα οποία μπορούν είτε να αποθηκευτούν σε ένα προσωρινό πίνακα στο δίσκο, είτε να προωθηθούν ως είσοδος στον επόμενο AND τελεστή ανά πλειάδα καθώς παράγονται. Οι είσοδοι ενός OR κόμβου είναι διαφορετικοί τρόποι με τους οποίους μπορεί να παραχθεί ένα ενδιάμεσο αποτέλεσμα, όπου στο τέλος ένας από αυτούς τους τρόπους πρέπει να επιλεχθεί. Έτσι, κάθε AND κόμβος έχει σαν εισόδους ακμές προερχόμενες από OR κόμβους, και κάθε OR κόμβος έχεις σαν εισόδους ακμές προερχόμενες από AND κόμβους. Αφού αναπαραστήσουμε όλα τα υποερωτήματα στο γράφο, εφαρμόζουμε μετασχηματισμούς, όπως αντιμεταθετικότητα και προσεταιριστικότητα των ζεύξεων, για να εξερευνήσουμε διαφορετικά πλάνα εκτέλεσης [35]. Κατά τη συγκεκριμένη διαδικασία οι κόμβοι OR που αναπαριστούν λογικά το ίδιο ενδιάμεσο αποτέλεσμα αναγνωρίζονται και ενοποιούνται, μέσω χρήσης μιας συνάρτησης κατακερματισμού για κάθε κόμβο.

Επίσης, εισάγονται φυσικοί τελεστές που αφορούν τη διαμέριση των πινάκων στο κατανεμημένο περιβάλλον εκτέλεσης, και εκτελείται αναζήτηση στο χώρο των πιθανών πλάνων εκτέλεσης, με χρήση τακτικών κλαδέματος πλάνων που δε θεωρείται ότι θα οδηγήσουν σε καλό πλάνο με βάση ευρετικές μεθόδους. Κατά την αναζήτηση, υποστηρίζεται η ανάκτηση ενδιάμεσων αποτελεσμάτων από προσωρινή μνήμη (cache) στην οποία αποθηκεύονται αποτελέσματα από προηγούμενες επερωτήσεις. Εφόσον κάποιο αποτέλεσμα βρεθεί στην προσωρινή μνήμη, μπορούμε να αποφύγουμε το κόστος του να το ξαναυπολογίσουμε στο τρέχον ερώτημα, έχοντας απλά το κόστος της ανάγνωσης του. Αφού οριστεί το τελικό πλάνο εκτέλεσης, όσα ενδιάμεσα αποτελέσματα αφορούν δεδομένα που βρίσκονται στις εξωτερικές βάσεις δεδομένων στέλνονται για εκτέλεση εκεί, μέσω ειδικών τελεστών που χειρίζονται την εισαγωγή των συγκεκριμένων ενδιάμεσων αποτελεσμάτων στο Exareme, επιλέγοντας να γίνει εκεί η επεξεργασία διαφόρων τελεστών όπου είναι δυνατόν.

Το σύστημα που αναπτύχθηκε στο συγκεκριμένο μέρος της διατριβής χρησιμοποιήθηκε με επιτυχία κατά το ευρωπαϊκό ερευνητικό έργο Optique [50] για την εκτέλεση απαιτητικών επερωτήσεων σε μεγάλο όγκο γεωλογικών δεδομένων από επτά διαφορετικές σχεσιακές βάσεις με πολύπλοκα σχήματα [49].

Συνοπτικά, η συνεισφορά του συγκεκριμένου μέρους της διατριβής έχει ώς εξής:

- Έχουμε αναπτύξει ένα σύστημα-διαμεσολαβητή για ανάκτηση δεδομένων βάσει οντολογιών πάνω από ένα σύνολο ανεξάρτητων μεταξύ τους σχεσιακών βάσεων δεδομένων, προσαρμόζοντας τεχνικές βάσεων δεδομένων για ταυτοποίηση κοινών υποεκφράσεων, χρήση προσωρινής μνήμης για ενδιάμεσα αποτελέσματα, βελτιστοποίηση επερωτήσεων για ενοποίηση δεδομένων και κατανεμημένη επεξεργασία δεδομένων.

- Εγκαταστήσαμε και λειτουργήσαμε επιτυχώς το σύστημα-διαμεσολαβητή σε ένα πραγματικό περιβάλλον εκτέλεσης στο πλαίσιο του ερευνητικού έργου Optique, στις υπολογιστικές εγκαταστάσεις της εταιρείας Statoil, πάνω από ένα σύνολο επτά διαφορετικών σχεσιακών βάσεων εγκατεστημένων σε διαφορετικούς εξυπηρετητές, με ετερογενή πολύπλοκα σχεσιακά σχήματα. Εκτελέσαμε τις περισσότερες από τις απαιτητικές επερωτήσεις που είχαν προκύψει από την ανάλυση των απαιτήσεων των χρηστών με μέσο χρόνο εκτέλεσης περίπου 100 δευτερόλεπτα.

Παραλληλοποίηση ζεύξεων πάνω σε γράφους RDF που περιέχουν οντολογικές ιεραρχίες

Στο τρίτο και τελευταίο σενάριο μελετάμε την περίπτωση που έχουμε RDF δεδομένα αποθηκευμένα σε ένα εξειδικευμένο σύστημα διαχείρισης αυτού του είδους των δεδομένων. Το RDF είναι ένα μοντέλο δεδομένων ευρέως χρησιμοποιούμενο για την ενσωμάτωση δεδομένων από διαφορετικές πηγές, ακολουθώντας ένα απλό σχήμα γράφου κατά το οποίο τα δεδομένα μοντελοποιούνται ως τριπλέτες που έχουν τη μορφή υποκείμενο-κατηγόρημα-αντικείμενο. Για αυτό το τρίτο σενάριο, αναπτύχθηκε ένα σύστημα διαχείρισης μεγάλων RDF γράφων που αποθηκεύονται στην κύρια μνήμη, ικανό να παραλληλοποιήσει αποτελεσματικά τις επερωτήσεις που προέρχονται από ένα OBDA-σύστημα, κατά το οποίο η οντολογία μπορεί να περιέχει μεγάλες ιεραρχίες από κλάσεις αντικειμένων ή ιδιότητες. Το σύστημα που ονομάστηκε PARJ, χρησιμοποιεί πρωτότυπο τρόπο αποθήκευσης των δεδομένων στην κύρια μνήμη, ο οποίος έχει μικρές απαιτήσεις σε όγκο, καθώς και πρωτότυπο τρόπο εκτέλεσης επερωτήσεων που περιέχουν ζεύξεις πάνω στο γράφο των δεδομένων, καθώς επίσης λαμβάνει υπόψιν του και τις ιεραρχίες της οντολογίας.

Η αποθήκευση των δεδομένων ακολουθεί τη μέθοδο της κάθετης διαμέρισης (vertical partitioning)[1], σύμφωνα με το οποίο για κάθε διακριτό κατηγόρημα που εμφανίζεται στα δεδομένα, δημιουργείται ένας πίνακας με δύο στήλες που αντιστοιχούν στον υποκείμενο και το αντικείμενο κάθε τριπλέτας με το συγκεκριμένο κατηγόρημα. Αρχικά γίνεται κωδικοποίηση λεξικού (dictionary encoding) η οποία αντιστοιχεί σε κάθε όρο των δεδομένων ένα μοναδικό ακέραιο αριθμό. Η αποθήκευση στην κύρια μνήμη γίνεται σε πίνακες που περιλαμβάνουν ταξινομημένους τους συγκεκριμένους ακέραιους για κάθε πίνακα της κάθετης διαμέρισης, ακολουθώντας αποθήκευση κατά στήλες όπως γίνεται σε συστήματα βασισμένα σε παρόμοια αποθήκευση (column stores) [98, 43]. Η εκτέλεση μίας επερώτησης που περιλαμβάνει πολλές ζεύξεις σε ένα πολυνηματικό περιβάλλον γίνεται με βάση ένα αριστεροβαθύ πλάνο εκτέλεσης. Ένα διαφορετικό μέρος του πιο

αριστερού πίνακα στο πλάνο εκτέλεσης ανατίθεται στο κάθε νήμα. Έπειτα, για κάθε πλειάδα και για κάθε ζεύξη, το κάθε νήμα εκτελεί αναζήτηση στους πίνακες των επόμενων σχέσεων του ερωτήματος. Με αυτό τον τρόπο το κάθε νήμα εκτελείται παράλληλα, χωρίς να χρειάζεται καθόλου επικοινωνία ή συγχρονισμός μεταξύ τους.

Σχετικά με την αναζήτηση σε κάθε πίνακα, αυτή γίνεται με έναν προσαρμοστικό αλγόριθμο, ο οποίος κατά τη διάρκεια της εκτέλεσης αποφασίζει για κάθε τιμή αν θα εφαρμοστεί δυαδική αναζήτηση ή σειριακή αναζήτηση. Η απόφαση λαμβάνεται με βάση την απόσταση στην οποία εκτιμάται ότι θα βρεθεί η συγκεκριμένη τιμή, εάν αυτή υπάρχει, από το σημείο του πίνακα στην οποία έχει μείνει η προηγούμενη αναζήτηση. Για τη σωστή λειτουργία του αλγόριθμου, κατά την έναρξη λειτουργίας του συστήματος διενεργείται μια διαδικασία βαθμονόμησης, η οποία επιστρέφει μία απόσταση που σε επαναλαμβανόμενες αναζητήσεις σε έναν πίνακα, οι δύο αυτές μέθοδοι αναζήτησης έχουν σχεδόν την ίδια απόδοση. Έτσι, κατά τη διάρκεια της εκτέλεσης, αν η εκτιμώμενη απόσταση είναι μεγαλύτερη από την προκαθορισμένη απόσταση που έχει ορίσει η διαδικασία βαθμονόμησης, εκτελείται δυαδική αναζήτηση. Διαφορετικά εκτελείται σειριακή αναζήτηση. Με αυτό τον τρόπο εκμεταλλευόμαστε την ολική ή ακόμα και μερική διάταξη με την οποία παράγονται οι τιμές σε κάθε πλειάδα, από την αρχική ταξινομημένη αποθήκευση των πινάκων, χωρίς να διακόπτεται ο διασωληνωμένος τρόπος εκτέλεσης. Επίσης, ένα ευρετήριο που περιέχει την αντιστοίχιση από την αριθμητική τιμή της κωδικοποίησης λεξικού προς τη θέση της τιμής σε κάθε πίνακα μπορεί να χρησιμοποιηθεί βοηθητικά με κάποια μικρή επιβάρυνση στον αποθηκευτικό όγκο.

Το σύστημα PARJ επεκτάθηκε κατάλληλα ώστε να λειτουργήσει ως εξωτερική βάση για το Ontop. Αρχικά, κατά την εκκίνηση του Ontop δημιουργούνται αυτόματα αντιστοιχίσεις με βάση το σχήμα της κάθετης διαμέρισης, και εισάγονται τα κατάλληλα μεταδεδομένα στο μοντέλο που δημιουργεί το Ontop. Έπειτα, στο PARJ δημιουργούνται ειδικές δομές με σκοπό την αποδοτική αποτίμηση των επερωτήσεων που παράγονται από το Ontop. Αυτές οι δομές ονομάζονται περικαλύμματα ενώσεων (union wrappers), και κάθε μία αντιστοιχεί σε μία $\mathcal{T}$-αντιστοίχιση του Ontop που αφορά κάποια ιεραρχία κλάσεων ή ιδιοτήτων. Ουσιαστικά, κάθε περικάλυμμα παρουσιάζεται στο Ontop σαν ένας εικονικός πίνακας που περιέχει όλα τα δεδομένα της εκάστοτε ιεραρχίας, ενώ στην πραγματικότητα οι συγκεκριμένοι πίνακες δεν αποθηκεύονται, αλλά κατά τη διάρκεια της εκτέλεσης γίνεται αναζήτηση σε κάθε πίνακα που συμμετέχει στο περικάλυμμα, και χρησιμοποιώντας μία ουρά προτεραιότητας, η ένωση των αποτελεσμάτων παράγεται αποδοτικά κατά την εκτέλεση με ταυτόχρονη απαλοιφή διπλότυπων τιμών. Έτσι, αποφεύγουμε τον κοστοβόρο υπολογισμό και την αποθήκευση των συμπερασμών που αφορούν τις ιεραρχίες, και ταυτόχρονα αποφεύγουμε τις επερωτήσεις που περιέχουν μεγάλο αριθμό υποερωτημάτων.

Η πειραματική αξιολόγηση του συστήματος έδειξε ότι το PARJ εκτελεί πιο αποδοτικά επερωτήσεις πάνω σε RDF γράφους από άλλα συστήματα με παρόμοια λειτουργικότητα, τόσο στην περίπτωση που οι επερωτήσεις τίθενται απευθείας, όσο και στην περίπτωση που προέρχονται από μεταγραφή μέσω ενός OBDA-συστήματος.

Συνοπτικά, η συνεισφορά του συγκεκριμένου μέρους της διατριβής έχει ώς εξής:

- Παρουσιάζουμε το σύστημα PARJ, ένα σύστημα αποθήκευσης RDF γράφων στην κύρια μνήμη, και εκτέλεσης επερωτήσεων πάνω σε αυτούς, το οποίο αποθηκεύει τα δεδομένα με συμπαγή τρόπο, έχοντας χαμηλές απαιτήσεις σε αποθηκευτικό χώρο και παράλληλα αυξάνοντας τη χωρική τοπικότητα των δεδομένων. Επίσης χρησιμοποιεί ένας προσαρμοστικό αλγόριθμο αναζήτησης κατά την εκτέλεση των ζεύξεων πάνω στα δεδομένα, ο οποίος μπορεί να παραλληλοποιήσει σε ένα πολυνηματικό περιβάλλον επερωτήσεις που περιλαμβάνουν πολλαπλές ζεύξεις, χωρίς επικοινωνία ή συγχρονισμό ανάμεσα στα νήματα.
- Μία δομή που αναπαριστά τις ιεραρχίες κλάσεων και ιδιοτήτων της οντολογίας στο σύστημα PARJ, η οποία παρέχει πλήρεις απαντήσεις με βάση τα αντίστοιχα αξιώματα της οντολογίας, χωρίς να υπολογίζει ή να αποθηκεύει τη συγκεκριμένη πληροφορία σε ξεχωριστά τμήματα μνήμης.
- Μία υλοποίηση που δένει το PARJ με το σύστημα Ontop για αποδοτική ανάκτηση δεδομένων βάσει οντολογιών όταν τα εξωτερικά δεδομένα είναι RDF γράφοι. Σύμφωνα με την πειραματική αξιολόγηση, η υλοποίησή μας έχει καλύτερη απόδοση σε σχέση με άλλα συστήματα παρόμοιας λειτουργικότητας.

Συμπεράσματα και μελλοντικές επεκτάσεις

Στην παρούσα διατριβή προτείνουμε λύσεις όσον αφορά το πρόβλημα της αποδοτικής εκτέλεσης επερωτήσεων σε ένα περιβάλλον ανάκτησης δεδομένων βάσει οντολογιών, σε τρία βασικά σενάρια ανάλογα με το είδος των υποκείμενων εξωτερικών πηγών δεδομένων. Πιθανές μελλοντικές επεκτάσεις αφορούν άλλες μορφές εξωτερικών δεδομένων, όπως για παράδειγμα τα λεγόμενα NoSQL συστήματα, όπως συστήματα αποθήκευσης δεδομένων εγγράφων (document stores) και αποθήκευσης δεδομένων της μορφής κλειδί-τιμή (key-value stores). Επίσης, σε αυτή την περίπτωση είναι επιθυμητή και η επέκταση του συστήματος-διαμεσολαβητή ώστε να υποστηρίζει και τέτοιου είδους δεδομένα ως πηγές, συνδυάζοντάς τα με τις σχεσιακές βάσεις.

Ένα άλλο ενδιαφέρον θέμα έχει να κάνει με την υποστήριξη γεωχωρικών δεδομένων. Σε αυτή την περίπτωση, το αρχικό ερώτημα πάνω στην οντολογία μπορεί να εκφραστεί στη γλώσσα GeoSPARQL[72], που είναι μια επέκταση της γλώσσας SPARQL και είναι πρότυπο της Ανοιχτής Γεωχωρικής Κοινοπραξίας (Open Geospatial Consortium συντ. OGC). Το σύστημα Ontop-spatial [11] είναι μια επέκταση του συστήματος Ontop που πραγματοποιεί μετάφραση επερωτήσεων από τη γλώσσα GeoSPARQL σε επερωτήσεις εκφρασμένες στη γλώσσα SQL, εμπλουτισμένη με γεωχωρικούς τελεστές. Η υποστήριξη του συγκεκριμένου συστήματος μπορεί να αποτελέσει μελλοντική δουλεία τόσο από το σύστημα-διαμεσολαβητή που αναπτύχθηκε πάνω στο Exareme για την περίπτωση που γεωχωρικές σχεσιακές βάσεις αποτελούν έναν από τους τύπους εξωτερικών πηγών δεδομένων, όσο και απο το σύστημα PARJ στην περίπτωση που έχουμε γεωχωρικά RDF δεδομένα, όπως αυτά της γλώσσας stSPARQL[57].

CONTENTS

1 INTRODUCTION **31**

1.1 Contributions . . . 32

1.2 Publications . . . 33

1.3 Thesis structure . . . 34

2 BACKGROUND AND RELATED WORK **35**

2.1 RDF and SPARQL . . . 35

2.2 Ontology-based Data Access (OBDA) . . . 36

2.3 Systems and approaches for RDF Storage and SPARQL query processing . . . 37

2.4 In-memory data processing . . . 39

3 HANDLING REDUNDANT PROCESSING IN OBDA QUERY EXECUTION OVER RELATIONAL SOURCES **41**

3.1 Introduction and Outline . . . 41

3.2 Preliminaries . . . 46

3.2.1 Databases. . . . 46

3.2.2 Queries. . . . 46

3.2.3 Ontology and Mappings. . . . 50

3.2.4 Logic Programs . . . 50

3.3 Unfolding Queries Through Partial Evaluation . . . 52

3.4 Offline Duplicate Elimination With Materialized Views . . . 57

3.5 Pushing Duplicate Elimination Before IRI Construction . . . 60

3.6 Cost-Based Selection of Query Translation . . . 62

3.6.1 Analyzing External Tables . . . 63

3.6.2 Early Duplicate Elimination of Intermediate Results . . . 63

3.6.3 Cost-based Translation . . . 65

3.7 Implementation and Experimental Evaluation . . . 68

3.7.1 Experiments with NPD and LUBM Benchmarks . . . 68

3.7.1.1 Queries and Mappings . . . 69

3.7.1.2 Overhead in Setup and Optimization . . . 69
3.7.1.3 Results . . . 69
3.7.2 Comparison with the JUCQ Approach . . . 71
3.7.3 Performance gain . . . 72
3.7.4 Evaluating the Duplicate Elimination Heuristic . . . 74
3.8 Related Work and Conclusions . . . 75

4 FEDERATED OBDA QUERY EXECUTION **77**
4.1 Introduction . . . 77
4.2 Background . . . 78
4.2.1 The Optique Platform . . . 78
4.2.2 The Exareme System . . . 80
4.2.2.1 Language and Optimization: . . . 81
4.2.2.2 Execution Engine: . . . 82
4.2.2.3 Worker Pool: . . . 82
4.2.2.4 Data / Stream Connector: . . . 82
4.2.2.5 Data Import: . . . 83
4.2.2.6 Query Execution: . . . 83
4.3 Overview . . . 83
4.3.0.1 Federated Analyzer . . . 84
4.3.0.2 Common Subexpression Identification . . . 84
4.3.0.3 Pushing processing to endpoints . . . 84
4.3.0.4 Caching Intermediate Results . . . 85
4.4 Query Optimization in the OBDA Mediator . . . 86
4.4.1 Transformation-Based Optimization . . . 86
4.4.2 Incorporating Partitioning Information in the Search and Pruning . . . 89
4.4.3 Adapting Volcano-style Search in Exareme . . . 93
4.4.4 Search with Materialized Results . . . 97
4.4.5 Improving Common Subexpression Identification . . . 98
4.5 Experimental Evaluation . . . 100
4.5.1 Experiments in the Statoil Optique Use-Case . . . 100
4.5.2 Experiments using Canonical IRIs . . . 101

4.6 Conclusions . . . 102

5 IN-MEMORY PARALLELIZATION OF JOIN QUERIES OVER LARGE ONTOLOGICAL HIERARCHIES **105**

5.1 Introduction . . . 105

5.2 Physical Data Storage and Execution Model . . . 107

5.3 Query Processing . . . 110

5.3.1 Adaptive Join Processing . . . 111

5.3.2 ID-to-Position Index . . . 113

5.4 Query Execution Over Ontological Hierarchies . . . 116

5.4.1 System Design . . . 116

5.4.2 Union Wrappers for Ontology Hierarchies . . . 118

5.4.3 Join Ordering and Selectivity Estimation . . . 120

5.5 Experiments . . . 122

5.5.1 Setup . . . 122

5.5.2 Results . . . 123

5.5.2.1 Effect of Runtime Join Optimization . . . 125

5.5.2.2 Effect of ID-to-Position Index . . . 128

5.5.2.3 Scalability . . . 129

5.5.2.4 Results for Query Execution over OWL2 QL Ontologies . . . 130

5.5.2.5 Comparison With Distributed RDF Stores . . . 132

6 CONCLUSIONS AND FUTURE WORK **133**

ABBREVIATIONS - ACRONYMS **135**

APPENDICES **136**

A NPD Queries 31-34 **137**

REFERENCES **144**

LIST OF FIGURES

3.1 SLD Tree 47
3.2 Example Mappings 48
3.3 Example $\mathcal{T}$-Mappings 48
3.4 UCQ over the database 49
3.5 Combined Mapping for Mapping Assertions $m1$, $m1'$ and $m1''$ 49
3.6 Combined Mapping for Mapping Assertions $m5$ and $m5'$ 49
3.7 SLD Tree 2 58
3.8 SLD Tree 3 59
3.9 Performance gain for varying number of mappings per predicate 73
3.10 Performance gain with respect to number of results 73

4.1 The general architecture of the Optique OBDA system 79
4.2 General architecture of the Exareme component within the Optique System 81
4.3 Query Plan. 85
4.4 Simple Join Example 87
4.5 Expanded DAG 88
4.6 Pruned Path in the DAG 90
4.7 Adding Partitioning Information 91
4.8 Execution Times for Federated Scenario (With and Without Query Cache) . 101
4.9 Execution time and statistics for the queries in the federated setting at Statoil 102
4.10 Comparison of execution time for sameAs and Canonical IRI approaches . 104

5.1 Example of Physical Data Storage for a Property Partition 109
5.2 $\mathcal{T}$-Mappings entry for class Professor 118
5.3 Union Wrapper for Class Professor 121
5.4 LUBM 32 threads execution times in ms for different dataset sizes 130
5.5 LUBM 10240 execution times in ms for different number of threads 131

LIST OF TABLES

3.1 Results for NPD scale 100 (Times in ms) . . . 70
3.2 Results for LUBM scale 100 (Times in ms) . . . 71
3.3 Average execution time (ms) for Wisconsin Benchmark . . . 72
3.4 Results for NPD queries from [59] (scale 100-Times in ms) . . . 72
3.5 Query Results for Different Duplicate Elimination Strategies . . . 75

4.1 Mappings . . . 99

5.1 Example of Dictionary Encoding . . . 108
5.2 Results for LUBM 10240 (times in ms) . . . 125
5.3 Results for YAGO2 (times in ms) . . . 125
5.4 Results for WatDiv Basic Workload scale 1000 (times in ms) . . . 126
5.5 Results for WatDiv Incremental and Mixed Linear Workloads scale 1000 (times in ms) . . . 127
5.6 Impact of Adaptive Processing for LUBM 10240 and WatDiv 1000 (times in ms) for 1 thread . . . 128
5.7 Number of binary searches and sequential searches for LUBM10240 chosen by out adaptive join method and comparison of binary search with ID-to-Position index with respect to total execution cycles and L1, L2 and L3 cache misses . . . 129
5.8 Results for $\text{LUBM}^{\exists}_{20}$ 1000 (times in ms) . . . 132

1. INTRODUCTION

In the area of *Knowledge Representation and Reasoning*, *ontologies* play a crucial role in modeling knowledge about an application domain. An ontology encodes information about the classes of objects of the domain and the relationships between them, providing the users of the application with a familiar conceptualization of the domain. On top of that, logic based constructs encoded in the ontology can be used to derive new knowledge through inference. The *World Wide Web Consortium* (W3C) recommends the usage of the *Web Ontology Language* (OWL) as the family of ontology languages used to represent knowledge in the web.

Ontology-based Data Acess (OBDA) is a method for linking an ontology to underlying external data sources through declarative mappings. The mappings can be thought of as rules that populate ontology objects based on queries over the the external data (*global as view*-GAV mappings in the data integration terminology). Then, a user can pose a query over the ontology, and this query can be translated into the query language of the underlying data sources using the mappings and sent for execution, providing the user with the desired results, as if the data were part of the ontology. In this thesis, unless otherwise stated, we deal with ontologies in the OWL 2 QL dialect of the OWL family, a dialect specifically tailored for the case of having massive data in external sources, for which there exist effective sound and complete query translation methods. In a similar manner, the mappings considered here belong are GAV mappings of the R2RML language which is a W3C standard, or other languages of similar expressibilit, and the initial queries over the ontology have the form of unions of conjunctive queries.

Despite the efficient, in terms of complexity, algorithms for query answering in OBDA systems, the resulted query that must be executed in the external databases is in many practical cases complex and large. As an example, it is not unusual in a typical OBDA setting where long property and class hierarchies are defined in the ontology, an initial conjunctive query over the ontology to be translated into a union of conjunctive queries, which can contain hundreds or thousands of subqueries. The aim of this thesis is offer database techniques in order to alleviate the aforementioned issue.

In this context we make a distinction between three different OBDA scenarios. In the first scenario we have an OBDA setup such that the data are stored in an external relational database with an arbitrary schema. In this case we explore different query translations using a cost-based algorithm in order to come up with an equivalent, but most efficient translation, compared to the translation obtained by other state of the art methods. In the second scenario, the data are stored in several external databases, where each one has its own schema. In this case we have developed a mediator which is able to efficiently execute the resulted query over all of these databases, by employing and adapting state of the art techniques regarding common subexpression identification, query caching, query planning for data integration and distributed query processing. In the third and final scenario, we have RDF data stored in a specialized triple store. RDF is a data model widely used in order to integrate data from different sources, following a simple schema of triples

in the form of subject-predicate-object statements. For this third scenario, we have developed an in-memory RDF store, able to efficiently parallelize OBDA queries over large ontological hierarchies.

1.1 Contributions

The contributions of the work described in this thesis are summarised as follows:

- We propose a novel enhancement over previous state of the art method for query translation from an initial query over the ontology, to an SQL query over an external database. Our enhancement provides a full cost-based method for OBDA query translation.
- We propose cost estimates for the previous method, aiming to deal with redundant processing in both forms of duplicate answers and repeated operations.
- We have implemented our method in the state of the art OBDA system Ontop and have perfomed extended experimental evaluation that confirms the efficiency of our method.
- We have implemented a mediator system for OBDA over multiple external relational databases, by employing and adapting database techniques for common subexpression identification, query caching, query planning for data integration and distributed query processing.
- We have successfully deployed the OBDA mediator in the real world use case of Statoil from the Optique research project, in a setting that contains 7 different realational data sources with complex schemas. We have executed most of the demanding use case queries with an average execution time of almost 100 seconds.
- We present PARJ, an in-memory triple store which compactly stores RDF data in main memory, in order to increase spatial locality during join processing, and employs a cache-friendly adaptive join processing approach with low memory consumption, able to efficiently parallelize evaluation of arbitrary multijoin BGPs without any communication.
- We propose a method that takes into consideration information about ontological hierarchies during join processing in PARJ, in order to provide complete answers with respect to such axioms.
- We present PARJ-Ontop, a system that couples PARJ with Ontop, providing a complete system for efficient OBDA query answering over RDF graphs. According to the experimental evaluation, our system outperforms all other state of the art systems.

1.2 Publications

The content of the present thesis is partially covered in the following publications:

- Dimitris Bilidas and Manolis Koubarakis. In-memory parallelization of join queries over large ontological hierarchies. Distributed and Parallel Databases, pages 1–38, 2020.
- Dimitris Bilidas and Manolis Koubarakis. Scalable parallelization of RDF joins on multicore architectures. In Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019, pages 349–360, 2019.
- Dimitris Bilidas and Manolis Koubarakis. Efficient duplicate elimination in SPARQL to SQL translation.In Proceedings of the 31st International Workshopon Description Logics co-located with 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018), Tempe, Arizona, US, October 27th - to - 29th, 2018, volume 2211 of CEUR Workshop Proceedings. CEUR-WS.org, 2018.
- Evgeny Kharlamov, Dag Hovland, Martin G Skjæveland, Dimitris Bilidas, Ernesto Jiménez-Ruiz, Guohui Xiao, Ahmet Soylu, Davide Lanti, Martin Rezk, Dmitriy Zheleznyakov, et al. Ontology based data access in Statoil. Journal of Web Semantics, 44:3–36, 2017.
- Evgeny Kharlamov, Ernesto Jiménez-Ruiz, Dmitriy Zheleznyakov, Dimitris Bilidas, Martin Giese, Peter Haase, Ian Horrocks, Herald Kllapi, Manolis Koubarakis, Özgür Özçep, et al. Optique: Towards OBDA systems for industry. In Extended Semantic Web Conference, pages 125–140. Springer, 2013.
- Evgeny Kharlamov, T Mailis, Konstantina Bereta, Dimitris Bilidas, Sebastian Brandt, Ernesto Jiménez- Ruiz, Steffen Lamparter, Christian Neuenstadt, O Özçep, Ahmet Soylu, et al. A semantic approach to polystores. In 2016 IEEE International Conference on Big Data (Big Data), pages 2565–2573. IEEE, 2016.
- Herald Kllapi, Dimitris Bilidas, Ian Horrocks, Yannis Ioannidis, Ernesto Jiménez, Evgeny Kharlamov, Manolis Koubarakis, Dmitriy Zheleznyakov, et al. Distributed query processing on the cloud: the optique point of view (short paper). 2013.
- Guohui Xiao, Dag Hovland, Dimitris Bilidas, Martin Rezk, Martin Giese, and Diego Calvanese. Efficient ontology-based data integration with canonical IRIs. In European Semantic Web Conference, pages 697–713. Springer, 2018.

Part of the work described in this thesis is also covered in the following submitted journal paper:

- Dimitris Bilidas and Manolis Koubarakis. Handling redundant processing in OBDA query execution over relational sources (currently under review). 2020.

1.3 Thesis structure

We first present related work and preliminaries in Chapter 2. The next three chapters are in direct correspondence with the three different OBDA scenarios that we described. Specifically, in Chapter 3 we consider the case where we have one external relational database and we provide a cost-based translation method for obtaining an optimized query aiming to handle redundant processing. Then, in Chapter 4 we consider the case of multpile external relational databases and we present a mediator system able to efficiently perform the task of data integration over them. In Chapter 5 we consider the case where external data are stored as RDF in a specialized triple store and we present the PARJ system for efficient execution of OBDA queries in this setting. Finally, in Chapter 6 we conlude this thesis and present future research directions.

2. BACKGROUND AND RELATED WORK

In this chapter we introduce basic background knowledge and related work that are required to understand this thesis. We start with a description of the RDF data model and the SPARQL query language (Section 2.1. Then we present important notions related to OBDA (Section 2.2. In section 2.3 we describe several systems and approaches for centralized and distributed storage and processing of RDF graphs. Finally, in Section 2.4 we present systems and approaches for in-memory data processing.

2.1 RDF and SPARQL

The *Resource Description Framework* (RDF) [1] is a data model recommended by the W3C for semantic data integration, sharing and linking across different organizations and applications on the Web. RDF provides flexible modeling of data coming from heterogeneous domains in the form of triples forming subject-predicate-object statements, facilitating the construction of Knowledge Graphs. Every component of such a triple is a resource uniquely identified by an IRI or a data value in the form of a literal. The latter can only be present in the object position. A set of such statements can be considered an RDF graph, where subjects and objects are nodes and there exists an arc labeled with the property name, connecting corresponding subject and object for each statement. Several organizations publish data in the RDF model, leading to interlinking information from different sources and automatic processing using software agents. As a result, as of 2019 the *Linked Open Data* (LOD) cloud [90] contains more than 1200 datasets and 60 billion triple statements, with DBpedia [8], a dataset that contains semantic information extracted from Wikipedia, taking up a central position with 3 billion triples and around 50 million links to other datasets.

The SPARQL query language is the W3C recommendation for querying RDF graphs. The basic building block of SPARQL queries are triple patterns. A triple pattern is similar to an RDF statement, with the exception that each component (subject, predicate or object) can be either a resource or a variable. The evaluation of a single triple pattern over an RDF graph consists of finding matches of the pattern on the graph such that variables are substituted by RDF resources. A *Basic Graph Pattern* (BGP) is a set of triple patterns. During evaluation of a BGP all triple patterns are matched to an RDF statement and common variables between triple patterns are substituted by the same resource. If we consider RDF storage on a single relational triples table, a BGP with n triple patterns corresponds to $n-1$ self joins of the triples table.

On top of that, RDF can be extended with the ability to encode ontological knowledge which is useful when describing the domain of a knowledge graph. RDF Schema (RDFS) as well as more expressive ontological languages like OWL-2 QL define ontological constraints on top of RDF graphs, such that SPARQL query answering must be extended by

[1] https://www.w3.org/TR/rdf11-concepts/

taking into consideration the corresponding semantics in order to provide the user with the complete answers, as it is the case of ontology-based data access that we describe in the following section.

2.2 Ontology-based Data Access (OBDA)

Ontologies can be used on top of RDF graphs in order to enrich the semantic information by providing a vocabulary that facilitates the conceptual modeling of a specific domain. For example, one can use an ontological statement to declare that if someone teaches a university course, then he is a professor. Then, for a given individual such that the data contain the information that he teaches a course, a query answering system under the appropriate entailment regime can deduce that this individual is a professor, even if this knowledge is not explicitly stated in the data.

Ontology-Based Data Access (OBDA) is a database technique in which an ontology is linked to underlying data sources through mappings. An end user can pose queries over the ontology, which we assume to represent a familiar vocabulary and conceptualization of the user domain. The OBDA system automatically translates the query and sends it for execution to the underlying data sources, providing the end user with a convenient abstraction over possibly complex schemas and details about the data storage and query processing.

A system can follow two main approaches to provide complete answers under such entailment regimes. The first one is query rewriting, which is similar in spirit with backward chaining of datalog evaluation. Under this approach the original query is rewritten in order to provide complete answers when posed over the incomplete data. This method has the advantage that it does not need data preprocessing, but on the other hand it produces more complicates queries. On the other hand, the second approach that is similar with forward chaining, uses materialization during data loading in order to add all the implicit knowledge to the data. This usually achieves better query performance, but it involves expensive data preprocessing, leading to increased database size and making things complicated during data updates or when the ontology changes.

There are several ontology languages aiming to different trade-offs between expressivity and efficiency of reasoning tasks, but regarding conjunctive query answering over RDF graphs, W3C recommends the usage of a specific profile of the OWL language which is called OWL2-QL. Under this entailment regime, it is guaranteed that it always exists a first-order query rewriting of a conjunctive query such that, when this rewriting is executed over the incomplete data, it provides the full answers implied by the OWL2-QL ontology. Indeed, several methods in order to obtain such a rewriting have been proposed, assuming that data are stored in an external, usually relational, database. In many cases, this rewriting has the form of a union of conjunctive queries, as for example in the PerfectRef [77] which was the first such proposed rewriting. It was soon observed that the result union of conjunctive queries was in many cases prohibitively large in order to be evaluated efficiently (for example it could contain thousands conjunctive queries), and as a result,

optimized methods that produce queries with fewer unions were proposed, based on query containment [66, 27]. Nevertheless, the result could still be extremely large. In order to solve this problem, some rewriting methods were proposed aiming to produce more compact rewritings in the form of nonrecursive datalog instead of union of conjunctive queries [86, 52], but efficient evaluation of such rewritings in existing database engines is still an issue. In [20] a cost-based comparison of different reformulations is carried out and in general, the final rewriting will be a join of unions of CQs. Semantic Index [82] contains an arithmetic encoding of class and property hierarchies and stores RDF data into relational back-ends using the appropriate B-tree indexes, such that class and property membership can be determined by range queries over these indexes, avoiding a large number of union subqueries.

On the other hand, a complete materialization of all the implicit knowledge defined by an OWL2-QL ontology in the general case may not be even possible, as the canonical model of the ontology and data may be infinite. As a result, regarding methods based on forward chaining, there have been proposed a combined approach [56] relying on materialization with respect to certain axioms combined with query rewriting, and an extension of that which uses finite materialization coupled with a filtering procedure in order to discard spurious answers[63].

Commercial RDF stores that support query answering over OWL2 QL ontologies include Stardog [2] that is based on query rewriting, and GraphDB [3] that relies mostly on materialization. Our work, following the architecture of [82], is also based on query rewriting, using PARJ as the execution engine, instead of an external *relational database management system* (RDBMS), as we describe in Section 5.4.

2.3 Systems and approaches for RDF Storage and SPARQL query processing

RDF storage and processing can be distinguished in three perspectives: i) a relational perspective, (ii) an entity perspective and (iii) a graph-based perspective [62]. Our work is mainly following the first perspective, as using relational technology for RDF processing has been a subject of research since the proposal of the RDF data model with prominent results. BGP evaluation using a single triples table that contains the whole RDF graph involves expensive self joins over this large table. As a solution, some systems like Jena [103] proposed the usage of “flattened“ property tables, which contain a larger number of columns, in an effort to simulate a relational schema and avoid joins as much as possible. Nevertheless, this design has some drawbacks, like for example a lot of NULL values for wide tables, the need for UNION during a single BGP processing and difficulty to handle multi-values attributes. [5] aims at efficient evaluation using an object-relational DBMS including a two-column representation for properties. Vertical partitioning[1] uses this representation in order to treat the drawbacks of the property tables. In this approach a separate two-column table is created for every property of the RDF graph. In this case,

[2]https://www.stardog.com
[3]http://graphdb.ontotext.com/

the number of joins is not reduced in comparison to the single triples table, but each join is between smaller tables and also tables not relevant to the query do not need to be accessed at all. Column stores are ideal candidates for RDF processing using vertical partitioning, as they provide compact storage and compression over each column. Our physical design is based on vertical partitioning as in [1], combined with techniques from column stores adapted for efficient in-memory RDF data storage.

Hexastore enhances the vertical partitioning by replicating the data through six different indexes, corresponding to all possible permutations of subject, predicate and object [102]. RDF-3X [70] also uses extensive indexing such that an index is created not only for all possible permutations but also for aggregated values, resulting in 15 indexes stored as clustered B+ trees. This schema along with several optimizations, such as skipping large parts of irrelevant data during merge joins using a form of *sideways information passing*, made RDF-3X one of the most efficient disk-based RDF stores, despite conceptually using the single triples-table approach. As in Hexastore, we keep two different replicas for each property in different sort orders (corresponding to POS and PSO indexes) with respect to our vertical partitioning data storage, and we also compactly store only distinct subjects and objects. Also, our adaptive join optimization (Section 5.3.1) can be considered a way of skipping irrelevant data as in RDF-3X.

Regarding SPARQL query processing using cloud technologies, [47] provides an overview and classification of systems and approaches in different categories regarding several characteristics. Here we briefly mention the most relevant research. An initial approach using the MapReduce framework is presented in [84, 85]. In this work, the authors describe the query evaluation of Basic Graph Patterns of SPARQL using an iterative algorithm, such that every join in the query requires a separate MapReduce job. The RDF data is stored in plain files in the distributed file system. A similar iterative approach is also used in [67], but here the authors note that more than one triple patterns that share a variable can be joined together in the same MapReduce job. They use a greedy selection algorithm that chooses in every step the variable that appears in more triple patterns and they employ reduce-side joins to get the results. In [31] predicate-based hash partitioning is employed. The query is decomposed to subqueries using the same partitioning and in every node a local Sesame RDF store is used to evaluate each subquery. Instead of hash partitioning, in [42] the authors use a graph partitioning algorithm to assign triples to nodes and also they employ data replication for triples that are on the boundaries of each partition, in order to maximize the number of subqueries that can be executed without communication between the nodes. They stress the usefulness of a heuristic that finds the minimal number of subqueries because this corresponds to a minimal number of Hadoop jobs, and they split each query to a number of such subqueries using a brute-force method, which is suitable only for queries with few triple patterns.

A number of approaches store the RDF data into an existing system that has its own declarative language and then they transform the SPARQL queries into that language. For example, [88] uses Pig Latin[71] and performs some well known optimizations to the SPARQL query, like the early execution of filters and some selectivity estimations based on variable counting. During the translation to Pig Latin, [88] just uses multi joins when

consecutive joins on the same variable are found, as this is an option that Pig Latin offers. RAPID+, a system which is also based on Pig Latin, is presented in [81]. Here the authors propose an intermediate algebra which is called Nested TripleGroup Algebra, in order to facilitate the grouping of join operators during the translation of the query to the execution plan in Pig Latin. The result is that each star join involving two or more triple patterns can be executed in one Map-Reduce job, using vertical partitioning.

H_2RDF+ [73] uses HBase[4] to store the RDF data. It takes advantage of the HBase key ordering for each table and it uses six tables, each one corresponding to an RDF triple permutation. In this way there is replication of the data, so that the system can perform fast merge joins when all triples are part of the initial RDF data. When some data is result of an intermediate step, the system first performs a sorting on this intermediate data. Another key feature of the system is that during the query planning it examines the option that the query will be executed in a centralized system. The rationale behind this is that if the query is simple, its evaluation in a centralized system can be preferable, because one can avoid the overhead of the MapReduce jobs and network communication. The system uses a greedy planner to decide about the order of the joins, based on a cost model and some index statistics that it has. In a similar manner, the system named Rya[79] uses Accumulo[5], to store indexes for permutations of subject, predicate and object in the row ID field of each corresponding table, but it only uses three indexes instead of all the possible ones. Rya supports range queries and regular expressions, multi-threaded join execution and also provides some limited inferencing capabilities. S2RDF [89] uses the in-memory system Spark to store the RDF data using vertical partitioning combined with semi-join materialization and then translates the SPARQL query to Spark SQL [7].

2.4 In-memory data processing

Regarding in memory join processing, a lot of research has been concentrated on cache friendly methods, such as the radix hash join [64], and also into taking advantage of hardware features such as the SIMD vectorized instructions for efficient parallel sort-merge joins [4, 53]. These works consider the setting of relational data with arbitrary number of columns, where a single join has to be performed on previously non indexed columns and sorting or hashing is a serious overhead that has to be performed in parallel. Instead, our work is tailored for RDF graphs, as it exploits initial ordering of both subject and object RDF columns and partial ordering of subsequent joins for pipelining multiway joins, such that it completely avoids hashing or sorting during query execution. Exploiting partial ordering of values in a column has been used by main memory systems in the form of zone-maps [96, 80] where additional statistics about each such zone have to be maintained in order to skip scanning certain areas. On the contrary, our join processing does not need to rely on extra statistics as in zone-maps. Adaptivity during run-time regarding the decision of scanning a base relation or use a secondary index has been studied in

[4] http://hbase.apache.org/
[5] http://accumulo.apache.org/

[17, 18] for disk-based systems.

Regarding centralized parallel in memory RDF processing, to the best of our knowledge there is no work concentrating on query processing. RDFox [68] and Inferray [99] are both systems that aim at parallel in memory computation and materialization of RDF inferences. This can be thought of as a preprocessing step prior to querying. Although RDFox offers query evaluation, it seems that is not the focal point of the system and for such queries there is no support for intra-query parallelism, that is each query is evaluated in a single thread. In [36] several variations of the disk based RDF-3X are presented, such that they allow parallel join evaluation. From the experimental results it is shown that depending on the query, there is no clear variation that has better performance, whereas for some queries the original version is better, as parallel evaluation prohibits the usage of the sideways information passing optimization in RDF-3X. Also, their approach works by parallelizing each join separately and demands communication and synchronization costs.

3. HANDLING REDUNDANT PROCESSING IN OBDA QUERY EXECUTION OVER RELATIONAL SOURCES

This chapter describes an optimized query translation method for obtaining efficient SQL queries from an initial SPARQL query posed over an ontology, though declarative mappings to a relational database. The results of this chapter are included in publications [12] and [14].

3.1 Introduction and Outline

As we described in Section 2.2, the query translation in OBDA typically involves *query rewriting* and *query unfolding*. During query rewriting, an initial query over an ontology is rewritten in order to take into consideration the ontological axioms. The result of this process is a query, that when posed over the ABox alone (that is, by disregarding all the ontological axioms), will return the same answers as the initial query posed over the ontology. This is done using the notion of *certain answers*, that is answers present in every model of the ontology. During query unfolding the rewritten query is transformed into another query expressed in the query language of the underlying data sources. In what follows we consider an OBDA setting, where an OWL 2 QL ontology is linked through mappings to data stored in a *relational database management system* (RDBMS). This method provides the user with access to a virtual RDF graph. The original query is expressed over this virtual RDF graph in the SPARQL query language, and the result of rewriting and unfolding is a SQL query.

Example 1. *As an example of OBDA setting consider a relational schema that contains the relational tables A_1, A_2, A_3, C_1 and C_2 and the mappings from Figure 3.2. In these mappings P_1, Q_1, R_1, P_2, P_3 and Q_3 are properties defined in the ontology, whereas f, g, h and k are functions roughly corresponding to string concatenation. These functions are responsible for constructing an object that acts as an ontology instance out of values occurring in the database. In our setting, they construct an RDF term represented by an IRI. Also, consider the query $ans(x, y, w, z) \leftarrow P_1(x, y), P_2(x, w), P_3(y, z)$ posed over the ontology.*

The notion of OBDA as we describe it, was presented in [77]. There, the result of query rewriting of an initial *conjunctive query* (CQ) over the ontology is a *union of conjunctive queries* (UCQ) over the ontology. Then, the authors define a faithful representation of this UCQ, along with the mappings and database instance in terms of a logic program. Query unfolding is based on partial evaluation of such logic programs, and as final result it produces an SQL query. More details about this process are given in Section 3.3. Subsequent research was focused on more efficient rewritings in the form of UCQs over the ontology [52, 27, 76]. The main aim of these approaches was to produce a UCQ with as few subqueries as possible, as it was observed that the number of union subqueries

in the result of query rewriting could be very large. A different approach was followed in [20], where a cost-based comparison of different reformulations is carried out, considering that no mappings are used and the ABox is directly stored in the external database.. In general, the final query in this case will be an SQL query that contains joins over UCQs (JUCQs). An extension of this work for arbitrary relational schemas, so that it also takes into consideration the unfolding step with arbitrary mappings, is presented in [59].

Regarding the implementation of OBDA systems, it has been observed that in practice it is more efficient to compile ontological knowledge regarding class and property hierarchies into the mappings, and ignore such axioms during query rewriting. For this reason, Ultrawrap-OBDA[92] uses the notion of saturated mappings and Ontop[21] uses the so called $\mathcal{T}$-Mappings [82]. For example, consider the setting of Example 1 and an ontology that contains the following axioms: $Q_1 \sqsubseteq P_1, R_1 \sqsubseteq P_1$ and $Q_3 \sqsubseteq P_3$. We can ignore the ontology axioms regarding property or class hierarchies during query rewriting (in our case all three ontology axioms), and perform query unfolding with the enhanced $\mathcal{T}$-mappings shown in Figure 3.3.

In [82] three main reasons are specified for the presence of a large number of union subqueries in the result of query translation: i) ontological queries with existentially quantified variables that can lead to rewritings of exponential size, ii) large ontological hierarchies and iii) multiple mappings for each ontology term. Also, the authors observe that the first reason is rarely observed in real world ontologies and queries. As a result, when compiling ontological information about hierarchies into the mappings, as for example in the Ontop $\mathcal{T}$-mappings, the last two important reasons that lead to a large number of subqueries are encountered during query unfolding. As an example, consider the query from Example 1 posed over the previously specified OBDA setting and $\mathcal{T}$-mappings. The unfolding method from [77] will produce a UCQ over the database that contains six union subqueries as shown in Figure 3.4. Each subquery corresponds to a different combination of the three mappings defined for P_1 with the two mappings defined for P_3. One can easily see that in case of queries with many atoms posed over large hierarchies, the final UCQ can contain hundreds or thousands of subqueries. On the other hand, a different unfolding method could choose to first compute as intermediate results the queries that correspond exactly to the first and third atoms of the initial query. In the specific example, the first temporary result would be a union query over tables A_1, A_2 and A_3 and the second temporary result would be a union query over tables C_1 and C_2. The final result would be a join of UCQs. Finally, one could choose an intermediate strategy, that would compute only one of these two intermediate results. Clearly, a cost-based decision should be made by the OBDA system regarding which exactly of these intermediate results should be computed, and if the overhead from computing and saving these results is counterbalanced from the gain in the final query.

Unfortunately, uncertainty about query execution costs is an inherent problem in *data integration*, where the *mediator* system (in our case the OBDA system) operates outside the database engine[45], as knowing all the factors that affect query execution is difficult or even impossible. For example, these factors include the exact execution plan that will be chosen by the RDBMS, including the access methods for each base relation and the

join order in a join query, hardware characteristics like for example the amount of available memory and disk throughput, the disk block size, the exact details of the database physical design, like the existing indexes and the kind of each index and several other factors.

On the other hand, one could expect that the RDBMS is capable of optimizing the produced query, since it performs query planning and optimization by taking into consideration the aforementioned parameters. Unfortunately, database engines focus on optimization of certain aspects of queries, including join ordering of multi-join queries, optimization of aggregate functions, access methods for each relation, etc. Queries produced by OBDA systems have some characteristics that are not regularly encountered on human-written queries for database applications. One such characteristic is the occurrence of common subexpressions in different parts of the query, for example in different subqueries of a union query. As we saw, the number of these subexpressions and subqueries can be very large. Although *common subexpression identification* (and in the case of multiple queries the related *multi-query optimization* area) have long been investigated in database research and implemented in database prototypes [91, 74, 87], to the best of our knowledge these methods have not become integral part of commercial RDBMSs, due to the increase in optimization time and the complexity introduced to the query optimizer. But since these common subexpressions are created during query translation, the OBDA system has the knowledge about them that can be taken into consideration to produce the final SQL query. Furthermore, it has been observed [59] that by using knowledge from the mappings, we can compute during system setup some parameters that will help us obtain more accurate selectivity estimations. For example, in our approach, a crucial factor that must be used when deciding about the exact form of the final SQL query, is the number of duplicates contained in the mappings used during unfolding for each ontology predicate. For example, for predicate P_1 of the query given in the previous example, it is crucial to know the number of duplicates rows in tables A_1, A_2 and the table obtained by selecting the first two columns of table A_3. The OBDA system knows from the mappings for which such columns and tables it should collect such information as an one-time task prior to query execution. On the other hand, an RDBMS cannot accurately estimate the number of duplicates in seemingly unrelated tables and columns during query execution.

Given the previous observations, in this chapter we propose a cost-based method employed by the OBDA system during unfolding, for choosing the final form of the SQL query to be executed by the RDBMS. This method relies on heuristics that in turn rely only on factors known to the OBDA system, such as sizes of the relations, duplicates introduced by the mappings for each ontology term and selectivity estimation for simple CQs over the database, that are not affected by issues such as join ordering or access methods, and thus can be performed even from a system operating outside the RDBMS as long as some basic statistics about the tables have been obtained prior to the deployment of the system. Specifically, our method starts with the “fully” unfolded query produced by the method of [77] as the baseline, and uses the heuristics in order to “fold” back specific paths, when this is expected to be more effective. Each such fold corresponds to the creation of an intermediate table, as explained in the previous example. These heuristics are based on

the notion of *redundant processing* between the union subqueries. We make a distinction between two kinds of redundant processing: i) duplicate answers and ii) repeated operations (disk reads and writes on the same data) from different union subqueries of the same query even in the absence of duplicate answers.

Regarding duplicates, using the standard set semantics for queries over ontologies, the final answer should be duplicate-free, but since RDBMSs operate using the bag semantics, duplicates are often introduced during query evaluation. Duplicates can be introduced as different ways to obtain the same fact from the data, for example the same tuple may be produced from different mappings used for the same property or class assertion. Using the unfolding method from [77], this will result in duplicate answers coming from different union subqueries. But duplicates can be introduced even from a single mapping, in case the database relation already contains duplicate rows, or due to the projection operator in the SQL query in the body of the mapping. In this setting, duplicates are redundant answers whose impact can be detrimental for query evaluation, as the size of intermediate results can increase exponentially in the number of joins in the query. Even if the final SQL query produced by an OBDA system dictates that the result should be duplicate free using the SQL DISTINCT or UNION keyword, relational systems rarely consider early duplicate elimination in order to limit the size of intermediate results, but only perform the task on the final query result. This behavior is justified by the fact that duplicate elimination is a costly blocking operation [16] and also that the SQL queries are usually formulated by expert users who take into consideration the integrity constraints of normalized relational schemas. Under these assumptions, considering early duplicate elimination options during optimization is not usually regarded worthy. Contrary to this situation for SQL queries, it has been ascertained [48] that in real world OBDA settings, duplicate answers frequently dominate query results and also that this appears as “noise” to end users that might be using a visual query formulation tool. In the previous version of this work [12], we introduced a heuristic regarding early duplicate elimination, for duplicates introduced from a single mapping. In this version we extend this heuristic for the case of duplicates that show up in different union subqueries, and use it to help us decide when to “fold” back specific branches of the unfolded query.

Regarding the second kind of redundant processing, this depends heavily on the exact execution plan that will be chosen by the RDBMS. As an example, consider the UCQ from Figure 3.4 and let us suppose that there are no duplicates (each fact for each ontology predicate can be obtained only from a single mapping). Also suppose that the RDBMS chooses to perform all the joins using index-based nested loops, using for the first three subqueries the table C_1 as the leftmost table and for the next three subqueries the table C_2 as the leftmost table. In this case, the redundant processing is equal to the two scans of table C_1 plus the two scans of table C_2 (ignoring the possible impact of the memory cache). If there was no redundant processing, then it is reasonable to assume that this form of the query would be the most efficient translation, as it consists of simple CQs which the RDBMS can efficiently optimize and probably evaluate in parallel. But since we have redundant processing, one would expect that it would be more efficient to first compute and save the temporary union table corresponding to the three mappings for P_1, if the

RDBMS will again choose to perform index-based nested loops and the cost for creating and saving the temporary result is smaller than the cost of the initial redundant processing. As all these possible execution plans cannot be known to the OBDA system, for this case of redundant processing, we use a criterion according to which temporary tables are created in a "conservative" manner, only when it is almost certain that this decision will lead to smaller execution cost.

In this chapter we present efficient solutions to the problem of handling redundancy, considering ontologies belonging to the OWL 2 QL language[1], as the W3C recommendation for query answering against datasets stored in relational back-ends. Nevertheless, several aspects of this work can be considered for other ontology languages as well. As mentioned, an early version of this work was presented in [12], where a heuristic was presented for early duplicate elimination in duplicates introduced from a single mapping (that is for each union subquery of the final SQL query separately). This heuristic was evaluated over four different RDBMSs and it was shown that its usage is justified and that for query mixes from two different used benchmarks, such that low selectivity queries do not dominate execution time, it can lead to overall improvement of up to 25% compared to the strategy of always performing duplicate elimination. The main contributions of the present work, extending this previous version in several aspects, are as follows:

- We enhance the unfolding step previously described in the literature with cost-based decisions regarding the redundant processing, obtaining a full cost-based method for OBDA query translation (Section 3.3).
- We extend the heuristic in order to deal with duplicate answers coming from different union subqueries (Section 3.6).
- We take into consideration other forms of redundant processing in the form of repeated operations (Section 3.6).
- We implement our method for cost-based translation by modifying the state of the art OBDA system Ontop [82] and we perform extended experimental evaluation (Section 3.7).

The organization of this chapter is as follows. We start by providing some preliminaries regarding ontologies, mappings, relational databases and logic programs (Section 3.2). In Section 3.3 we modify the unfolding method from [77], which is based on partial evaluation of logic programs, in order to explore equivalent results given that certain mappings have been replaced by a *combined mapping* which we define. In Section 3.4 we present a methodology that computes the mapping assertions that are possibly responsible for the introduction of duplicate answers and in Section 3.5 we discuss how to perform duplicate elimination over the database values by pushing the duplicate elimination before the IRI construction. In Section 3.6 we describe the cost-based decisions and we present the algorithm that incorporates them in the unfolding process. In Section 3.7 we present

[1] `https://www.w3.org/TR/owl2-profiles/`

experimental evaluation of our implementation using the Ontop OBDA framework and the NPD and LUBM benchmarks. We also use the Wisconsin benchmark to compare our results with the results of [59]. In Section 3.8 we present relevant work and conclusions.

3.2 Preliminaries

We consider the following pairwise disjoint alphabets: Σ_O of ontology predicates, Σ_R of database relation predicates, $Const$ of constants, Var of variables and Λ of function symbols where, each function symbol has an associated arity. We also consider that $Const$ is partitioned into DB_{Const} of database constants and $\mathcal{O}_{Const}$ of ontology constants.

As in [77], we use functions with symbols from Λ in order to solve the so called *impedance mismatch problem* of constructing ontology objects from values occurring in the database. These functions roughly correspond to IRI templates specified in the R2RML[2] language. We also consider that $\forall \lambda_1, \lambda_2 \in \Lambda$, where $\lambda_1 \neq \lambda_2$, the range of function with symbol λ_1 and the range of function with symbol λ_2 are disjoint. That is, the same ontology object cannot be produced from different functions.

3.2.1 Databases.

We start by giving definitions for database instances and queries over them, following the bag semantics from [26].

A *bag* B is a pair (A, μ), where A is a set called the underlying set of B and μ is a function from elements of A to the positive integers, which gives the multiplicities of elements of A in B. The *underlying set* of a bag B will be denoted by US_B.

A *relation instance* is a bag of tuples of fixed arity using constants from DB_{Const}.

A *source schema* S is a set of relation names from Σ_R.

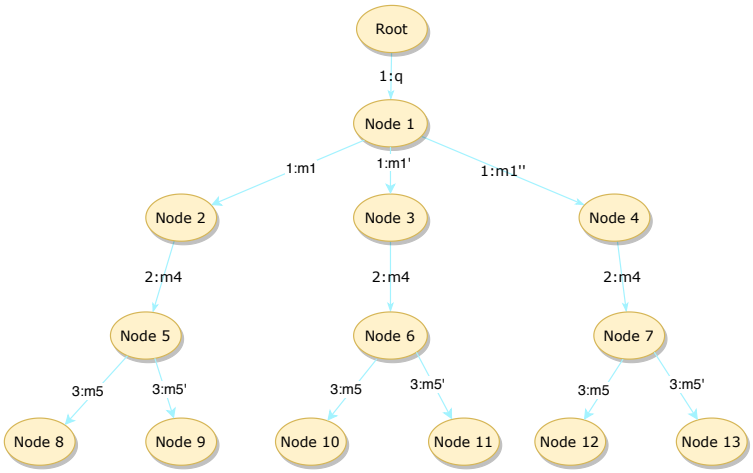
A *database instance* D for a source schema S is a mapping from relation names in S to relation instances.

3.2.2 Queries.

We define queries following the bag semantics of [26]. In our definitions we use the term "SQL query" although the syntax of our formulas is that of first-order logic. Similarly, relation instances are viewed as bags of *ground atoms* (i.e., with no variables) of first-order logic.

A *SQL query* over a relational schema S is an expression that has the form: $Query(\vec{x}) \leftarrow \alpha$, where α is a first order expression containing predicates from Σ_R, which are among the

[2] `https://www.w3.org/TR/r2rml/`



$$Root : ans(x,y,z)\theta_0 \leftarrow ans(x,y,z)$$
$$\theta_0 = \{\}$$
$$Node1 : ans(x,y,z)\theta_0\theta_1 \leftarrow P_1(x,y), P_2(x,h(A)), P_3(y,z)$$
$$\theta_1 = \{\}$$
$$Node2 : ans(x,y,z)\theta_0\theta_1\theta_2 \leftarrow A_1(v_1^{m1}, v_2^{m1}), P_2(f(v_1^{m1}), h(A)), P_3(g(v_2^{m1}), z)$$
$$\theta_2 = \{x/f(v_1^{m1}), y/g(v_2^{m1})\}$$
$$Node3 : ans(x,y,z)\theta_0\theta_1\theta_3 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), P_2(f(v_1^{m1'}), h(A)), P_3(g(v_2^{m1'}), z)$$
$$\theta_3 = \{x/f(v_1^{m1'}), y/g(v_2^{m1'})\}$$
$$Node4 : ans(x,y,z)\theta_0\theta_1\theta_4 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), P_2(f(v_1^{m1''}), h(A)), P_3(g(v_2^{m1''}), z)$$
$$\theta_4 = \{x/f(v_1^{m1''}), y/g(v_2^{m1''})\}$$
$$Node5 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), P_3(g(v_2^{m1}), z)$$
$$\theta_5 = \{v_1^{m_4}/v_1^{m1}, v_3^{m_4}/A\}$$
$$Node6 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), P_3(g(v_2^{m1'}), z)$$
$$\theta_6 = \{v_1^{m_4}/v_1^{m1'}, v_3^{m_4}/A\}$$
$$Node7 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), P_3(g(v_2^{m1''}), z)$$
$$\theta_7 = \{v_1^{m_4}/v_1^{m1''}, v_3^{m_4}/A\}$$
$$Node8 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_8 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), C_1(v_2^{m1}, v_2^{m5})$$
$$\theta_8 = \{v_1^{m5}/v_2^{m1}, z/k(v_2^{m5})\}$$
$$Node9 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_9 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), C_2(v_2^{m1}, v_2^{m5'})$$
$$\theta_9 = \{v_1^{m5'}/v_2^{m1}, z/k(v_2^{m5'})\}$$
$$Node10 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6\theta_{10} \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), C_1(v_2^{m1'}, v_2^{m5})$$
$$\theta_{10} = \{v_1^{m5}/v_2^{m1'}, z/k(v_2^{m5})\}$$
$$Node11 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6\theta_{11} \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), C_2(v_2^{m1'}, v_2^{m5'})$$
$$\theta_{11} = \{v_1^{m5'}/v_2^{m1'}, z/k(v_2^{m5'})\}$$
$$Node12 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7\theta_{12} \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), C_1(v_2^{m1''}, v_2^{m5})$$
$$\theta_{12} = \{v_1^{m5}/v_2^{m1''}, z/k(v_2^{m5})\}$$
$$Node13 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7\theta_{13} \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), C_2(v_2^{m1''}, v_2^{m5'})$$
$$\theta_{13} = \{v_1^{m5'}/v_2^{m1''}, z/k(v_2^{m5'})\}$$

Figure 3.1: SLD Tree

$$
\begin{aligned}
m1 &: A_1(v_1^{m1}, v_2^{m1}) \rightarrow P_1(f(v_1^{m1}), g(v_2^{m1})) \\
m2 &: A_2(v_1^{m2}, v_2^{m2}) \rightarrow Q_1(f(v_1^{m2}), g(v_2^{m2})) \\
m3 &: A_3(v_1^{m3}, v_2^{m3}, v_3^{m3}) \rightarrow R_1(f(v_1^{m3}), g(v_2^{m3})) \\
m4 &: A_3(v_1^{m4}, v_2^{m4}, v_3^{m4}) \rightarrow P_2(f(v_1^{m4}), h(v_3^{m4})) \\
m5 &: C_1(v_1^{m5}, v_2^{m5}) \rightarrow P_3(g(v_1^{m5}), k(v_2^{m5})) \\
m6 &: C_2(v_1^{m6}, v_2^{m6}) \rightarrow Q_3(g(v_1^{m6}), k(v_2^{m6}))
\end{aligned}
$$

Figure 3.2: Example Mappings

$$
\begin{aligned}
m1 &: A_1(v_1^{m1}, v_2^{m1}) \rightarrow P_1(f(v_1^{m1}), g(v_2^{m1})) \\
m1' &: A_2(v_1^{m1'}, v_2^{m1'}) \rightarrow P_1(f(v_1^{m1'}), g(v_2^{m1'})) \\
m1'' &: A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}) \rightarrow P_1(f(v_1^{m1''}), g(v_2^{m1''})) \\
m2 &: A_2(v_1^{m2}, v_2^{m2}) \rightarrow Q_1(f(v_1^{m2}), g(v_2^{m2})) \\
m3 &: A_3(v_1^{m3}, v_2^{m3}, v_3^{m3}) \rightarrow R_1(f(v_1^{m3}), g(v_2^{m3})) \\
m4 &: A_3(v_1^{m4}, v_2^{m4}, v_3^{m4}) \rightarrow P_2(f(v_1^{m4}), h(v_3^{m4})) \\
m5 &: C_1(v_1^{m5}, v_2^{m5}) \rightarrow P_3(g(v_1^{m5}), k(v_2^{m5})) \\
m5' &: C_2(v_1^{m5'}, v_2^{m5'}) \rightarrow P_3(g(v_1^{m5'}), k(v_2^{m5'})) \\
m6 &: C_2(v_1^{m6}, v_2^{m6}) \rightarrow Q_3(g(v_1^{m6}), k(v_2^{m6}))
\end{aligned}
$$

Figure 3.3: Example $\mathcal{T}$-Mappings

relations that belong to S, $Query \in \Sigma_R$, $Query \notin S$ and $\vec{x}$ is a vector of constants from DB_{Const} and variables from Var that appear in α.

A *conjunctive query* CQ over a relational schema S is a SQL query, where α has the form $R_1(\vec{x_1}) \wedge ... \wedge R_n(\vec{x_n})$, where $\vec{x_1}, ..., \vec{x_n}$ are vectors of constants from DB_{Const} and variables from Var, and $R_1, ..., R_n \in S$. Variables from $\vec{x_1}, ..., \vec{x_n}$ that do not appear in $\vec{x}$ are existentially quantified, but we omit the quantifiers in order to simplify the reading. CQs roughly correspond to SQL Select-From-Where queries.

Let q be the SQL query $Query(\vec{x}) \leftarrow \alpha$, we will denote by $\prod_i(q)$ the query resulting from the projection of the i-th answer term of q, that is the query: $Query(x_i) \leftarrow \alpha$

An *assignment mapping* of a conjunctive query Q into a database instance D is an assignment of values from DB_{Const} belonging to D to the variables of Q such that every atom in the body of Q is mapped to a ground atom in D. Let θ be an assignment mapping of Q into database instance D and let X be a variable in Q. We denote by $\theta(X)$ the constant in DB_{Const} to which θ maps X and we denote by $\theta(R_i(\vec{x_i}))$ the ground atom to which $R_i(\vec{x_i})$ is mapped.

$$
\begin{aligned}
ans(f(x), g(y), h(w), k(z)) \leftarrow \\
A_1(x, y), A_3(x, v_1, w), C_1(y, z) \vee \\
A_2(x, y), A_3(x, v_1, w), C_1(y, z) \vee \\
A_3(x, y, v_2), A_3(x, v_1, w), C_1(y, z) \vee \\
A_1(x, y), A_3(x, v_1, w), C_2(y, z) \vee \\
A_2(x, y), A_3(x, v_1, w), C_2(y, z) \vee \\
A_3(x, y, v_2), A_3(x, v_1, w), C_2(y, z)
\end{aligned}
$$

Figure 3.4: UCQ over the database

$$
\begin{aligned}
cm_1 : A_1(v_1^{m1}, v_2^{m1}) \vee A_2(v_1^{m1}, v_2^{m1}) \vee A_3(v_1^{m1}, v_2^{m1}, v_3^{m1''}) \rightarrow P_1(f(v_1^{m1}), g(v_2^{m1})) \\
\theta = \{v_1^{m1'}/v_1^{m1}, v_1^{m1''}/v_1^{m1}, v_2^{m1'}/v_2^{m1}, v_2^{m1''}/v_2^{m1}\}
\end{aligned}
$$

Figure 3.5: Combined Mapping for Mapping Assertions $m1$, $m1'$ and $m1''$

$$
\begin{aligned}
cm_2 : C_1(v_1^{m5}, v_2^{m5}) \vee C_2(v_1^{m5}, v_2^{m5}) \rightarrow P_3(g(v_1^{m5}), k(v_2^{m5})) \\
\theta = \{v_1^{m5'}/v_1^{m5}, v_2^{m5'}/v_2^{m5}\}
\end{aligned}
$$

Figure 3.6: Combined Mapping for Mapping Assertions $m5$ and $m5'$

Let μ_i denote the multiplicities $\mu(\theta(R_i(\vec{x_i}))), i = 1, ..., n$. The *result* due to θ of a conjunctive query Q over D is the atom $(\theta(\vec{x}), \mu_\theta)$ with the multiplicity $\mu_\theta = \mu_1\mu_2 \cdots \mu_n$.

The *result* of a conjunctive query Q over a database instance D denoted by $Q(D)$ is given by $\uplus_\theta r_\theta$, where θ is any assignment mapping of Q into D, r_θ is the result due to θ and $\uplus$ denotes bag union.

3.2.3 Ontology and Mappings.

A TBox is a finite set of ontology axioms.

An ABox is a finite set of membership assertions $A(\rho)$ or role filling assertions $P(\rho, \rho')$, where $\rho, \rho' \in \mathcal{O}_{Const}$ and $A, P \in \Sigma_O$ denote a concept name and role name respectively.

A DL ontology $\mathcal{O}$ is a pair $\langle \mathcal{T}, \mathcal{A} \rangle$ where $\mathcal{T}$ is a TBox and $\mathcal{A}$ an ABox.

A *mapping assertion* (or simply a mapping) m from a source schema S to a TBox $\mathcal{T}$ has the form: $\phi(\vec{x}) \rightarrow \psi_1 \wedge ... \wedge \psi_n$, where $\phi(\vec{x})$ will be denoted by *body(m)* and it is a SQL query over a database schema S, each ψ_i has the form $P_i(cc_i^1(\vec{x_i^1}), cc_i^2(\vec{x_i^2}))$ or $C_i(cc_i^1(\vec{x_i^1}))$ with P_i (respectively C_i) $\in \Sigma_O$ a property (respectively concept) name, and each $cc_i^j \in \Lambda$ is a function with arity equal to the length of $\vec{x_i^j}$ and range a subset of $\mathcal{O}_{Const}$. The conjunction in the right-hand side will be denoted by *head(m)*. A *mapping collection* $\mathcal{M}$ is a finite set of such mapping assertions.

In this chapter we consider mappings where all variables in $\psi_1 \wedge ... \wedge \psi_n$ also appear in $\vec{x}$. In this setting, $\mathcal{M}$ can be transformed into a set of equivalent mapping assertions where the head of each assertion consists of a single atom [77]. In what follows we assume that the head of every mapping assertion consists of a single atom.

Let $\mathcal{M}$ be a mapping collection, we will use the symbol $\mathcal{M}_{CQ}$ to denote the assertions from $\mathcal{M}$ whose body is a CQ over the database schema.

In correspondence with CQs over a relational schema, we define a CQ over an ontology $\mathcal{O}$ as an expression of the form: $Query(\vec{x}) \leftarrow P_1(\vec{x_1}) \wedge ... \wedge P_n(\vec{x_n})$ where $\vec{x_1}, ..., \vec{x_n}$ are vectors of constants from $\mathcal{O}_{Const}$ and variables from Var, $\vec{x}$ is a vector of constants from $\mathcal{O}_{Const}$ and variables from Var that appear in $\vec{x_1}, ..., \vec{x_n}$, and $P_1, ..., P_n \in \Sigma_O$ are ontology predicates that appear in $\mathcal{O}$.

A *union of conjunctive queries* UCQ over an ontology $\mathcal{O}$ is an expression of the form $Query(\vec{x}) \leftarrow CQ_1(\vec{x}) \vee ... \vee CQ_n(\vec{x})$, where each CQ_i for $i = 1, ..., n$ is an expression of the form $P_1^i(\vec{x_1^i}) \wedge ... \wedge P_n^i(\vec{x_n^i})$ as in the previous definition.

3.2.4 Logic Programs

Following [77], we use partial evaluation of logic programs in order to translate a UCQ over the ontology into a UCQ over the data sources, using the SQL language. In this section we present basic notions from logic programs[60] regarding partial evaluation [61]. As we

are interested in the translation of UCQs, we do not deal with negation, and as a result we only present notions related to definite logic programs.

A *definite logic program* is a set of statements that have the following form: $\forall \vec{x}(A \leftarrow A_1 \wedge ... \wedge A_n)$, where $A, A_1, ..., A_n$ are atoms as in standard first order logic definitions and $\vec{x}$ are all the variables occurring in $A, A_1, ..., A_n$. Each such statement is also called a (definite) *program clause*, or a (definite) *rule*, with A being the *head* of the rule, and $A_1 \wedge ... \wedge A_n$ the *body* of the rule.

A definite *goal* is a definite clause such that the head is empty. Following the standard convention in logic programming, we omit the existential quantifiers and use the syntactic form $A_1, ..., A_n$ for the body, instead of $A_1 \wedge ... \wedge A_n$, both in clauses and goals.

A *substitution* θ is a finite set of the form: $\{x_1/t_1, .., x_n/t_n\}$, where each x_i is a variable, each t_i is a term distinct from x_i, variables $x_1, ..., x_n$ are pairwise distinct and no variable x_i occurs in some term t_i. Let Exp be an expression. The application of a substitution θ on Exp is denoted $Exp\theta$ and is the expression obtained by Exp after replacing each occurrence of x_i with t_i, for $i = 1, ..., n$.

Let Exp_1 and Exp_2 be expressions. A *unifier* for Exp_1 and Exp_2 is a substitution θ such that $Exp_1\theta = Exp_2\theta$.

Let $\theta_1 = \{x_1/s_1, ..., x_m/s_m\}$ and $\theta_2 = \{y_1/t_1, ..., y_n/t_n\}$ be substitutions such that no variable from $x_1, ..., x_m$ occurs in θ_2. The *composition* of θ_1 with θ_2 is the following substitution: $\{x_1/s_1\theta_2, ..., x_m/s_m\theta_2, y_1/t_1, ..., y_n/t_n\}$.

The *most general unifier (mgu)* of two expressions Exp_1 and Exp_2, is a unifier ξ such that for every unifier ν of Exp_1 and Exp_2 there exists a substitution θ such that ν is the composition of ξ with θ.

A *computation rule* is a function from a set of goals to a set of atoms, such that the value of the function for a goal is always an atom, called the *selected atom*, in that goal.

Let G be $\leftarrow A_1, ..., A_m, ..., A_k$, C be $A \leftarrow B_1, ..., B_q$ and R be a computation rule. Then G' is *derived* from G and C using the mgu θ via R if the following conditions hold:

- A_m is the selected atom in G given by R.
- θ is an mgu of A_m and A.
- G' is the goal $\leftarrow (A_1, ..., A_{m-1}, B_1, ..., B_q, A_{m+1}, ..., A_k)\theta$

A *resultant* is a first order formula of the form $Q_1 \leftarrow Q_2$, where each of Q_1, Q_2 is either absent or a conjunction of atoms. Any variables in Q_1 or Q_2 are assumed to be universally quantified at the front of the resultant.

Let P be a definite program, G' be a goal with body G and R a computation rule. Then, the *SLD-tree* of $P \cup \{G'\}$ via R is the tree defined as follows:

- Each node is a resultant (possibly with an empty body)

- The root node is $G_0\{\} \leftarrow G_0$, where $G_0 = G$.
- Let $G\theta_0...\theta_i \leftarrow A_1, ..., A_m, ..., A_k$ be a node in the tree with $k \geq 1$ and suppose that A_m is the selected atom of the derivation given by R. Then, this node has a descendant for each input clause of $A \leftarrow B_1, ..., B_q$ of P such that A_m and A are unifiable. The descendant is $G\theta_0...\theta_{i+1} \leftarrow (A_1, ..., B_1, ..., B_q, ..., A_k)\theta_{i+1}$, where θ_{i+1} is an mgu of A and A_m.
- Nodes which are resultants with empty bodies have no descendants.

Each branch of the SLD-tree is a derivation of G'. A branch which ends in a node such that the selected atom does not unify with the head of any program clause is called a *failure* branch. A branch which ends in the empty clause is called a *success* branch. An SLD-tree is *complete* if all of its branches are either failure or success branches. An SLD-tree that is not complete is called *partial*.

In general an SLD-tree can contain branches that correspond to infinite derivations, but we will not deal with this case, as the logic programs that we will construct do not contain recursion.

The *computed answer* θ for a node $Q\theta_0, ..., \theta_i \leftarrow Q_i$ of an SLD-tree is the restriction of $Q\theta_0, ..., \theta_i$ to the variables in the goal G'.

Let P be a definite program, A an atom and R a computation rule and T an SLD-tree for $P \cup \{\leftarrow A\}$ via R. Then:

- any set of nodes such that each non-failing branch of T contains exactly one of them is a *Partial Evaluation* (PE) of A in P;
- the logic program obtained from P by replacing the set of clauses in P whose head contains A with a PE of A in P is a PE of P with respect to A.

The semantics of a definite logic program P can be defined by two different ways, proved to be equivalent. The first one is the declarative, that uses the model-theoretic semantics of first-order logic, where the semantics are given by the *least Herbrand model*, which contains the facts that are true in every model of P. The second way is the procedural, where the SLD-tree is used, and the semantics are given by the *success set* of P, that is all the facts A such that the SLD-tree of $P \cup \{\leftarrow A\}$ has a success branch. Also, it is known that the semantics of a program P coincide with the semantics of any partial evaluation of P.

3.3 Unfolding Queries Through Partial Evaluation

In this section we describe the process of unfolding queries over the ontology, into queries over the external relational database using declarative mappings. We are following the approach of [77], with the following modifications:

- We enforce that during each step of the SLD_Derive process, the algorithm employs the computation rule that chooses for unification the leftmost possible atom in the right-hand side of the resultant.
- We make a distinction between mapping assertions whose body is a CQ over the database and the rest of the mapping assertions.
- We define a step that "folds" back specific branches of the PE tree based on the notion of combined mapping, and we show that the SQL query that is obtained based on this form of the PE tree has exactly the same answers with the SQL query obtained using the initial form of the tree.

As in [77], we start by defining the logic program for a $UCQ\ Q(\vec{x}) \leftarrow CQ_1(\vec{x}) \vee ... \vee CQ_n(\vec{x})$ over: (i) an ontology $\mathcal{O}$ (ii) a database instance D over a database schema S and (iii) a mapping collection $\mathcal{M}$ from source schema S to the vocabulary of $\mathcal{O}$. The main modification is that we use auxiliary predicates only for mapping assertions in $\mathcal{M} \setminus \mathcal{M}_{CQ}$.

The program for Q, D and $\mathcal{M}$, denoted $P(Q, \mathcal{M}, D)$ is the logic program defined as follows:

- $P(Q, \mathcal{M}, D)$ contains the clause $Q(\vec{x}) \leftarrow CQ_i(\vec{x})$ for each CQ_i in the right-hand side of Q.
- $P(Q, \mathcal{M}, D)$ contains the fact $R(\vec{t})$ for each tuple $\vec{t}$, such that $\vec{t} \in R$, for each relation instance R in D.
- $P(Q, \mathcal{M}, D)$ contains each mapping assertion $m \in \mathcal{M}_{CQ}$.
- For each mapping assertion $m \in \mathcal{M} \setminus \mathcal{M}_{CQ}$, $P(Q, \mathcal{M}, D)$ contains the clause $head(m) \leftarrow Aux_m(\vec{x})$, where Aux_m is an auxiliary predicate associated to m, whose arity is the same as $head(m)$.
- For each mapping assertion $m \in \mathcal{M} \setminus \mathcal{M}_{CQ}$, $P(Q, \mathcal{M}, D)$ contains the fact $Aux_m(\vec{t})$, where $\vec{t} \in body(m)(D)$.

We now present the function SLD-Derive defined in [77], with the extra condition that we enforce use of the computation rule that chooses for unification the leftmost possible atom. The SLD-Derive$(P(Q, \mathcal{M}, D))$ takes as input $P(Q, \mathcal{M}, D)$ and returns a set Res of resultants constituting a PE of $q(\vec{x})$ in $P(Q, \mathcal{M}, D)$, by constructing an SLD-tree for $P(Q, \mathcal{M}, D) \cup \{\leftarrow q(\vec{x})\}$ as follows:

- it start by selecting the atom $q(\vec{x})$,
- it continues by selecting the atoms whose predicates belong to the alphabet of $\mathcal{T}$, as long as possible, using the computation rule R which selects each time the leftmost such atom
- it stops the construction of a branch when no atom with predicate in the alphabet of $\mathcal{T}$ can be selected.

The partial evaluation $PE(Q, \mathcal{M}, D)$ of $P(Q, \mathcal{M}, D)$ with respect to $q(\vec{x})$ is obtained by dropping the clauses for q in $P(Q, \mathcal{M}, D)$ and replacing them with the result of SLD-Derive$(P(Q, \mathcal{M}, D))$.

Example 2. *Consider the query* $ans(x, y, z) \leftarrow P_1(x, y), P_2(x, h(A)), P_3(y, z)$*, the mapping collection ($\mathcal{T}$-mappings) shown in Figure 3.3 and a database instance over a schema that contains the relation names* A_1, A_2, A_3, C_1 *and* C_2 *with tuples of appropriate arities according to Figure 3.3. The SLD-tree for* $P(Q, \mathcal{M}, D) \cup \{\leftarrow ans(x, y, z)\}$ *is shown in Figure 3.1.*

In [77] the *virtual ABox* given by a mapping collection $\mathcal{M}$ over a database instance D for a database schema S is defined as the set of ABox assertions generated by applying each mapping assertion in $\mathcal{M}$ over D and it is shown that for each tuple of constants $\vec{t}$, $P(Q, \mathcal{M}, D) \cup \{\leftarrow q(\vec{t})\}$ is unsatisfiable if and only if $\vec{t}$ belongs to the result of executing Q over the database instance that stores exactly the assertions contained in the virtual ABox. Here we omit the formal definitions and the proof, but we note that it is straightforward to see that the specific result carries over to our modified definition of $P(Q, \mathcal{M}, D)$. It is also shown that the result of SLD-Derive is independent of the database instance D (i.e., independent of facts of the form $Aux_m(\vec{t})$ for some auxiliary predicate Aux_m, and facts of the form $R(\vec{t})$ for some relation instance R) and the algorithm *UnfoldDB* is defined, which, given an UCQ Q over an ontology $\mathcal{O}$ with a mapping collection $\mathcal{M}$, translates the set of resultants returned by SLD-Derive$(P(Q, \mathcal{M}, D))$ into queries over the database instance D. Again, we omit the details and we note that in our case the resulted query will be a UCQ over S that has the form

$$Query(\vec{x}) \leftarrow Q_1(\vec{x}) \vee ... \vee Q_n(\vec{x}) \tag{3.1}$$

where each Q_i for $i = 1, ..., n$ is the translation given by UnfoldDB that corresponds to a resultant returned by SLD-Derive$(P(Q, \mathcal{M}, D))$, and it is an expression of the form

$$Q_i(\vec{f}_i(\vec{x_i})) \leftarrow \\ Aux_{i_1}(\vec{x_{i_1}}) \wedge ... \wedge Aux_{i_l}(\vec{x_{i_l}}) \wedge R_{i_{l+1}}(\vec{x_{i_{l+1}}}) \wedge ... \wedge R_{i_m}(\vec{x_{i_m}}) \tag{3.2}$$

where each $f_i^j \in \vec{f}_i$ is a function whose function name belongs in Λ and whose variable arguments are among the variables of $\vec{x_{i_1}}, ..., \vec{x_{i_m}}$, each Aux_{i_j} for $j = 1, ..., l$ corresponds to $body(m)$ for some $m \in \mathcal{M} \setminus \mathcal{M}_{CQ}$ and each R_{i_k} for $k = l + 1, ..., m$ is a relation name from the database schema. Note that on the original definition of UnfoldDB semantic query optimization (SQO) with respect to the database schema S is not performed. Nevertheless, in subsequent research, the role of SQO with respect to this context was proved crucial [93, 83]. In this work we consider that SQO, like for example self-join elimination, is performed in the result of UnfoldDB, that is in each Q_i for $i = 1, ..., n$ in (3.1). Furthermore, by overloading the definition of UnfoldDB, we consider a version of the function that takes as input an SLD-tree resulted from the application of the SLD-Derive$(P(Q, \mathcal{M}, D))$, and operates as described to produce a query that has the aforementioned form.

We now proceed with some definitions that will be used when we “fold back” the SLD-tree

produced by SLD-Derive. For each edge e of the SLD-tree we define $source(e)$ to be the node at the beginning of e, $target(e)$ to be the node the node at the end of e, $TM(e)$ to be the predicate symbol of the atom selected by computation rule R at $source(e)$, $M(e)$ to be the clause (mapping assertion) used in the specific derivation, $sub(e)$ to be the substitution used in the specific derivation and $pos(e)$ to be the set of integers corresponding to the positions of atoms affected by the derivation in the right-hand side of the resultant in $target(e)$.

Let $m_1, ..., m_n$ be mapping assertions of the form $\phi_i \rightarrow \psi_i$ where no variable is repeated in ψ_i, for $i = 1, ..., n$ and there exists a unifier θ such that the expressions $\psi_1\theta, ..., \psi_n\theta$ are all syntactically equal (obviously the predicate symbol at the head of each assertion is the same). Then, the *combined mapping* of $m_1, ..., m_n$ is the following expression:

$\phi_1\theta \vee ... \vee \phi_n\theta \rightarrow \psi_1\theta$

If a variable z is repeated multiple times in ψ_i for a mapping assertion, then we modify ψ_i by keeping only the first occurrence and we replace all other occurrences with fresh variables $z_1, ..., z_k \in Var$. Then, we compute the unifier θ, and finally we add to the head of the combined mapping the conditions $z = z_1\theta, ..., z = z_n\theta$. In this case, the combined mapping has a slightly different form from the definition of mappings as given in Section 3.2.3, as it now also contains a set of equalities in the head. These are simply translated to SQL by projecting the same column multiple times with different renamings, in the corresponding union subquery.

Essentially, the combined mapping introduces a mapping assertion whose body is the union the input mappings, with the appropriate renaming. Two examples of combined mappings for the example mappings shown in Figure 3.3 are presented in Figures 3.5 and 3.6.

Proposition 1. *Let T be an SLD-tree resulted from SLD-Derive with input $P(Q, \mathcal{M}, D)$, m_c be the combined mapping of mappings $m_1, ..., m_n \in \mathcal{M}$ and $\mathcal{M}_c = (\mathcal{M} \setminus \{m_1, ..., m_n\}) \cup \{m_c\}$. The semantics of $PE(Q, \mathcal{M}, D)$ coincide with the semantics of $PE(Q, \mathcal{M}, D_c)$.*

Proof. We need to show that for every tuple $\vec{t}$ of constants, $q(\vec{t})$ is true in $PE(Q, \mathcal{M}, D)$ if and only if $q(\vec{t})$ is true in $PE(Q, \mathcal{M}, D_c)$, which follows directly from the construction of m_c. □

Let T be the tree resulted from SLD-Derive$(P(Q, DB, \mathcal{M}))$ and e_0 an edge in T. Also, let $e_1, ..., e_n$ be edges in T with $source(e_0) = source(e_1) = ... = source(e_n)$ and $TM(e_0) = TM(e_1) = ... = TM(e_n)$ such that there exists a combined mapping $m_c : \phi_0\theta \vee ... \vee \phi_n\theta \rightarrow \psi_1\theta$, with $\theta = sub(e_0)$ and $TM(e_0)$ be equal to the predicate at the head of m_c. A *fold of T into* e_0 is the tree T_2 that is resulted from T by replacing in each descendant node of $target(e_0)$ (including $target(e_0)$) the atoms at positions $pos(e_0)$ with the atom $\psi_1\theta$, and deleting all the sub-trees starting from $target(e_1), ..., target(e_n)$. Moreover, let $f_1, ..., f_m$ be all the edges in T (including e_0) such that $M(f_1) = ... = M(f_m) = M(e_0)$. Then, the *fold of T based on* m_c is the tree that is obtained if we sequentially apply the process of obtaining the fold of T into f_i for $i = 1, ..., m$ ensuring that for each f_k, f_l with k, l in $1, ...m$, if the

depth of $target(f_k)$ in T is smaller than the depth of $target(f_l)$, then the fold of T into f_k is obtained after obtaining the fold of T into f_l.

Figures 3.7 and 3.8 show the fold of the SLD-tree of example 2 based on mappings from Figures 3.5 and 3.6 respectively, where Aux_{cm1} and Aux_{cm2} are regular auxiliary predicates used for mappings in $\mathcal{M} \setminus \mathcal{M}_{CQ}$ according to the construction of $P(Q, \mathcal{M}, D)$, that correspond to combined mappings $cm1$ and $cm2$ respectively.

Proposition 2. *Let T be an SLD-tree resulted from SLD-Derive with input $P(Q, \mathcal{M}, D)$, m_c be the combined mapping of mappings $m_1, ..., m_n \in \mathcal{M}$ and $\mathcal{M}_c = (\mathcal{M} \setminus \{m_1, ..., m_n\}) \cup \{m_c\}$. The fold of T based on m_c is exactly the tree returned by SLD-Derive with input $P(Q, \mathcal{M}, D_c)$.*

Proof. Let T_{fold} be the fold of T based on m_c and T_c be the SLD-tree resulted from SLD-Derive with input $P(Q, \mathcal{M}, D_c)$. We need to show that T_{fold} and T_c consist of the same resultants. Clearly the two trees have the same root. Then, given a resultant $ans(\vec{x})\theta_0...\theta_k \leftarrow A_1(\vec{x_1}), ..., A_i(\vec{x_i}), ..., A_w(\vec{x_w})$ at depth k which is the same for the two trees, it is sufficient to show that the children of this resultant are the also the same for the two trees. Let $node_{T_c}$ and $node_{T_{fold}}$ be the nodes in T_c and T_{fold} respectively that contain the specific resultant. Suppose that A_i is the leftmost atom in the body of the resultant with predicate that belongs to the alphabet of $\mathcal{T}$ and will be chosen by the computation rule R. Also, for now, let's suppose that there is only one node $node_T$ in the initial tree T such that for every edge e in T with $TM(e)$ equal to the predicate symbol at the head of m_c, then $source(e) = node_T$. According to the construction of the fold of T into e_0, if $node_T$ is different from $node_{T_{fold}}$, then the children of $node_{T_{fold}}$ are the same with the children of $node_{T_c}$, as they are not affected by the combined mapping. If $node_T$ is equal to $node_{T_{fold}}$, then $node_T$ has n children affected by the combined mapping, plus a number of children not affected (possibly 0). The second kind of children are also children of $node_{T_c}$, whereas the first kind have been replaced in T_{fold} with the child $ans(\vec{x})\theta_0...\theta_k\theta_{k+1} \leftarrow A_1(\vec{x_1}), ..., CM(\vec{z}), ..., A_w(\vec{x_w})$, which is also a child of $node_{T_c}$, and these are the only children of both $node_{T_c}$ and $node_{T_{fold}}$. Now, if there are more nodes in T affected by the fold of T based on m_c, then from the construction of the fold, where descendant nodes are always modified prior to their predecessors, and from the fact that R chooses always the leftmost possible atom, it is straightforward to see that the result of the case where only one node is affected by the combined mapping is carried over to this case.

□

A direct consequence of Propositions 1 and 2 is that if we consider the SLD-tree tree T resulted from SLD-Derive with input $P(Q, \mathcal{M}, D)$ and we apply the UnfoldDB algorithm on the resultants contained in the the fold of T based on the combined mapping m_c, then the SQL query that will be produced has exactly the same answers with the SQL query produced by applying the UnfoldDB algorithm on the resultants of the original tree T. This gives us the ability to choose a sequence of folds, in order to obtain an equivalent

translation that can be more efficient, by using a cost-based search in the initial SLD-tree, which we describe in detail in the Section 3.6.3.

An issue that arises in these translations has to do with the way the combined mapping is treated. One way is to be treated as a regular mapping assertion, as the body of this mapping is simply a union query over the original mapping assertions. This union query will be computed as many times as the combined mapping is used in the produced query. Obviously a better choice would be to create a temporary table that holds the specific result as an intermediate result of the main query, in the same database connection. This is the solution that we follow in this work, as it also avoids the overheads of creating a permanent materialized view in the database. A second issue that has to be handled is the decision regarding which folds should be used, if any, for a specific query. As we will describe in the following section, the process of taking the specific decision heavily depends on the size of the SQL query, the size of each combined mapping in comparison to the size of the final SQL query and the number of duplicate answers contained in them.

3.4 Offline Duplicate Elimination With Materialized Views

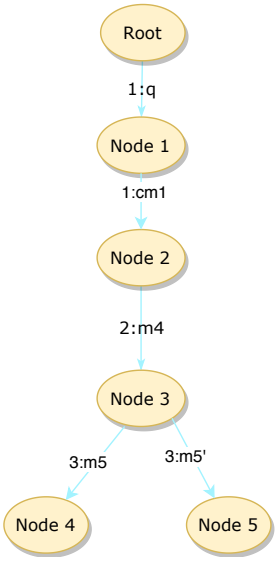
One solution to the problem of duplicates, is to track down the mapping assertions which are responsible for duplicates, create materialized views with the distinct results, possibly with indexes, and then use these views instead of the original assertions during query unfolding. It is reasonable to expect that this solution will give the best performance during query execution, but on the other hand this incurs expensive preprocessing and also, using materialized views in the database increases the database maintenance load, especially for frequently updated tables, as well as the the database size. Also, this solution will not take into consideration duplicates due to projections in the SPARQL query and finally, it is not in line with the overall approach of providing the end user with access to several underlying data sources, without the need to modify data, and on a practical level, such access may not be even possible.

Nevertheless, even in the case where one chooses to use materialized views, it is not straightforward exactly which of the mapping assertions should be chosen. In the rest of this section we describe a process to find the exact assertions for this setting, whereas in the following sections we consider the case where no materialization happens and all processing needs to be done during query execution.

Given a mapping $\mathcal{M}$ and a database instance D over a schema S, a straightforward solution is to materialize all $m \in \mathcal{M}$ such that $DTR_{head(m)(D)} > 1$, but as the query produced after rewriting takes into consideration the ontology axioms, implied assertions may be used, such that a specific variable has been projected out from the outputs of the body of an existing assertion due to reasoning for class instances with respect to domain or range of a property like in Example 4, where the modified mapping implied assertion is the following:

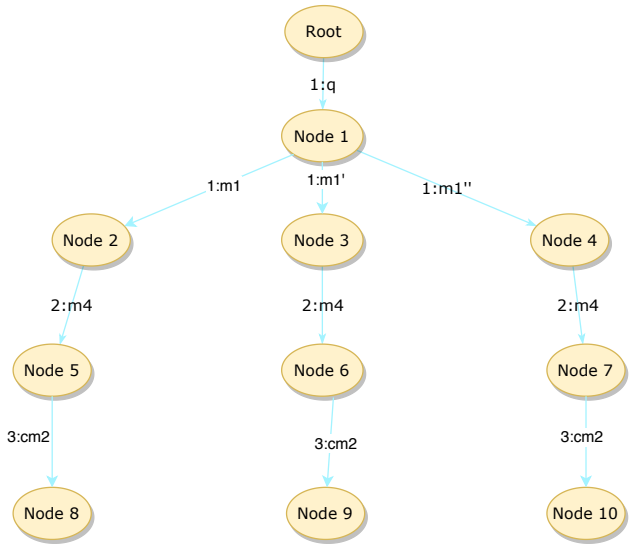
$$movies(t, d) \rightarrow Director(d)$$

The exact way that this implied assertion will be used depends on the rewriting method,



$$Root : ans(x,y,z)\theta_0 \leftarrow ans(x,y,z)$$
$$\theta_0 = \{\}$$
$$Node1 : ans(x,y,z)\theta_0\theta_1 \leftarrow P_1(x,y), P_2(x,h(A)), P_3(y,z)$$
$$\theta_1 = \{\}$$
$$Node2 : ans(x,y,z)\theta_0\theta_1\theta_2 \leftarrow Aux_{cm_1}(v_1^{m1}, v_2^{m1}), P_2(f(v_1^{m1}), h(A)), P_3(g(v_2^{m1}), z)$$
$$\theta_2 = \{x/f(v_1^{m1}), y/g(v_2^{m1})\}$$
$$Node3 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5 \leftarrow Aux_{cm_1}(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), P_3(g(v_2^{m1}), z)$$
$$\theta_5 = \{v_1^{m_4}/v_1^{m1}, v_3^{m_4}/A\}$$
$$Node4 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_8 \leftarrow Aux_{cm_1}(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), C_1(v_2^{m1}, v_2^{m5})$$
$$\theta_8 = \{v_1^{m5}/v_2^{m1}, z/k(v_2^{m5})\}$$
$$Node5 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_9 \leftarrow Aux_{cm_1}(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), C_2(v_2^{m1}, v_2^{m5'})$$
$$\theta_9 = \{v_1^{m5'}/v_2^{m1}, z/k(v_2^{m5'})\}$$

Figure 3.7: SLD Tree 2



$$Root : ans(x,y,z)\theta_0 \leftarrow ans(x,y,z)$$
$$\theta_0 = \{\}$$
$$Node1 : ans(x,y,z)\theta_0\theta_1 \leftarrow P_1(x,y), P_2(x,h(A)), P_3(y,z)$$
$$\theta_1 = \{\}$$
$$Node2 : ans(x,y,z)\theta_0\theta_1\theta_2 \leftarrow A_1(v_1^{m1}, v_2^{m1}), P_2(f(v_1^{m1}), h(A)), P_3(g(v_2^{m1}), z)$$
$$\theta_2 = \{x/f(v_1^{m1}), y/g(v_2^{m1})\}$$
$$Node3 : ans(x,y,z)\theta_0\theta_1\theta_3 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), P_2(f(v_1^{m1'}), h(A)), P_3(g(v_2^{m1'}), z)$$
$$\theta_3 = \{x/f(v_1^{m1'}), y/g(v_2^{m1'})\}$$
$$Node4 : ans(x,y,z)\theta_0\theta_1\theta_4 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), P_2(f(v_1^{m1''}), h(A)), P_3(g(v_2^{m1''}), z)$$
$$\theta_4 = \{x/f(v_1^{m1''}), y/g(v_2^{m1''})\}$$
$$Node5 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), P_3(g(v_2^{m1}), z)$$
$$\theta_5 = \{v_1^{m_4}/v_1^{m1}, v_3^{m_4}/A\}$$
$$Node6 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6 \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), P_3(g(v_2^{m1'}), z)$$
$$\theta_6 = \{v_1^{m_4}/v_1^{m1'}, v_3^{m_4}/A\}$$
$$Node7 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7 \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), P_3(g(v_2^{m1''}), z)$$
$$\theta_7 = \{v_1^{m_4}/v_1^{m1''}, v_3^{m_4}/A\}$$
$$Node8 : ans(x,y,z)\theta_0\theta_1\theta_2\theta_5\theta_8 \leftarrow A_1(v_1^{m1}, v_2^{m1}), A_3(v_1^{m1}, v_2^{m_4}, A), Aux_{cm_2}(v_2^{m1}, v_2^{m5})$$
$$\theta_8 = \{v_1^{m5}/v_2^{m1}, z/k(v_2^{m5})\}$$
$$Node9 : ans(x,y,z)\theta_0\theta_1\theta_3\theta_6\theta_{10} \leftarrow A_2(v_1^{m1'}, v_2^{m1'}), A_3(v_1^{m1'}, v_2^{m_4}, A), Aux_{cm_2}(v_2^{m1'}, v_2^{m5})$$
$$\theta_{10} = \{v_1^{m5}/v_2^{m1'}, z/k(v_2^{m5})\}$$
$$Node10 : ans(x,y,z)\theta_0\theta_1\theta_4\theta_7\theta_{12} \leftarrow A_3(v_1^{m1''}, v_2^{m1''}, v_3^{m1''}), A_3(v_1^{m1''}, v_2^{m_4}, A), Aux_{cm_2}(v_2^{m1''}, v_2^{m5})$$
$$\theta_{12} = \{v_1^{m5}/v_2^{m1''}, z/k(v_2^{m5})\}$$

Figure 3.8: SLD Tree 3

but in any case, in the resulted SQL query column $title$ will not be in the $Select$ clause.

Situations like this can be identified offline, by analyzing the ontology and the original mapping. The first step is to find the ontology axioms of the form $\exists P.\top \sqsubseteq C$ and $\exists P^{-}.\top \sqsubseteq C$ (or equivalently $\top \sqsubseteq \forall P.C$) that define the domain and range of some property P to be a class C in our ontology. Then we identify the mapping assertion in $\mathcal{M}$ that generates RDF triples which have as predicate the property P and we modify the target SQL query of the mapping, by projecting out the columns used for the subject or object respectively. At this point, we can skip certain mappings that are covered by the corresponding $rdf{:}type$ mapping for a given class, as OBDA systems eliminate the usage of the property triple pattern for these cases based on foreign key relationships [82]. Consider again the case of Example 4: if we had one more database table $director\_info$ which has a primary key id and we also know that $director$ in $movies$ is a foreign key that references this primary key, then if we also had the mapping:

$$director\_info(id, ...) \rightarrow Director(f(id))$$

the previously obtained mapping can be skipped as it is redundant and will not be used by the OBDA system.

The method is described in Algorithm 1. $ComputeDTR$ is a function that returns the DTR for the query passed as argument. If $DTR = 1$ according to proposition 1, then access to data is avoided altogether, otherwise the actual DTR is computed by sending two count queries: with and without the distinct modifier. Later, when the DTR needs to be determined during query optimization, an estimation based on data summarization is used instead (Section 3.6). Function $ExistsFK$ returns true if a foreign key exists between the output column of query $query_1$ and the output column of query $query_2$. The result of this algorithm is a set of mapping assertions, possibly annotated with information about the projection of a column. Modification of the produced SQL query in order to take into consideration the views created for these mappings, instead of the original body of the mapping, can simply be performed in the final step of the query unfolding, where each Aux_j is replaced by the corresponding SQL, and as a result it is independent of the query rewriting method.

3.5 Pushing Duplicate Elimination Before IRI Construction

In SPARQL to SQL approaches, pushing joins inside unions is a well known *structural* optimization, so that joins over IRIs are avoided and relational columns, whose values are possibly indexed, are used instead. Methods for unfolding based in partial datalog evaluation like the one we consider here produce such queries, where additionally, union subqueries that contain joins between incompatible IRIs, that when evaluated will produce an empty result, are completely discarded. Also, the *safe separator*[3] of the R2RML mapping language can be used to ensure that concatenation of multiple columns cannot produce the same value with that of a single column [83]. In a similar manner, it can be very useful to perform duplicate elimination before IRI construction. In this section we dis-

[3]https://www.w3.org/TR/r2rml/#dfn-safe-separator

Algorithm 1: Track down SQL queries that contain duplicates

MappingsWithDuplicates (mapping $\mathcal{M}$, ontology $\mathcal{O}$, database schema S, database instance D);
Output: Mapping assertions possibly annotated with a projected column
Uses : $ComputeDTR(query, db\_schema, db\_instance)$
Uses : $ExistsFK(schema, query_1, query_2)$
$result := \emptyset$;
for $m \in \mathcal{M}$ **do**
  if $head(m)$ *is Class assertion* **then**
    if $ComputeDTR(body(m), S, D) > 1$ **then**
      add m to result;
    end
  else
    /* $head(m)$ is Property assertion with predicate P */
    if $\mathcal{O}$ *contains* $\exists P.\top \sqsubseteq C$ **and** $ComputeDTR(\prod_1(body(m)), S, D) > 1$ **and not** *(*$\exists m2 \in \mathcal{M}$ *s.t. predicate of* $head(m2)$ *is* C **and** $ExistsFK(S, \prod_1(body(m)), body(m2))$ **then**
      add m to $result$ for projection of 1st column;
    end
    if $\mathcal{O}$ *contains* $\exists P^-.\top \sqsubseteq C$ **and** $ComputeDTR(\prod_2(body(m)), S, D) > 1$ **and not** *(*$\exists m2 \in \mathcal{M}$ *s.t. predicate of* $head(m2)$ *is* C **and** $ExistsFK(S, \prod_2(body(m)), body(m2))$ **then**
      add m to $result$ for projection of 2nd column;
    end
  end
end
return $result$;

cuss the process of transforming the unfolded query that has the form shown in formula (3.1) at page 54, into an equivalent one, such that duplicate elimination is performed on database values.

To do so, we must group together union subqueries that have the same select clause up to variable (column name) renaming. In our case the situation is more complicated, as we want to ensure that tuples produced from different IRI templates cannot possibly have equal values. Consider for example the following query:

```
SELECT ':Person' || alias1.id AS x
FROM table1 alias1
UNION
SELECT ':Person' || alias1.key AS x
FROM table2 alias1
UNION
SELECT ':Person' || alias1.id ||
   '/' || alias1.name AS x
FROM table3 alias1
```

Given that the '/' character is a safe separator, the third subquery cannot produce any result tuple that will be the same with a result tuple coming from the first two subqueries. On the other hand, there is a possibility that the first two subqueries may produce the same answer. The following rewriting of this query can be used:

```
SELECT ':Person' || var1 AS x,
FROM
(SELECT
   alias1.id as var1
```

```
   FROM
   table1 alias1
 UNION
 SELECT
   alias1.key AS var1
   FROM
   table2 alias1
 )
UNION ALL
SELECT DISTINCT ':Person' ||
   alias1.id || '/' || alias1.name AS x,
FROM table3 alias1
```

Note that UNION ALL operator is simply concatenating the results. When the UNION ALL is the outer operator of a query, it is reasonable for the RDBMS to start sending the results in a pipelining fashion, as they are produced from each subquery without saving or waiting for all the results to be produced. In this sense, it can be considered a "cheap" operator in contrast to UNION. If we know that column id of $table3$ is a primary key, then the distinct keyword of the last subquery can be eliminated. Even if this is not the case, the resulted query has several advantages over the initial. First, the duplicate elimination process has been separated over two distinct result sets and also each tuple is smaller in size. This gives to the RDBMS the opportunity to better utilize available memory, as it now has smaller datasets to perform duplicate elimination, or even parallelize the process. Available indexes on the columns can be used. Also, as discussed, when there is no blocking outer operator, results are produced in a pipelined fashion. This way the first results can be obtained very quickly and, as IRI construction is an expensive operation, the difference can be impressive when we have large results and the processing for each subquery is relatively cheap.

3.6 Cost-Based Selection of Query Translation

In this section we consider a cost-based algorithm in order to choose a specific sequence of folds and obtain the SQL translation of the initial query. During this process we take into consideration the two kinds of redundant processing that we described in Section 3.1. Regarding the first kind (redundancy due to duplicates), we will employ a heuristic about early duplicate elimination of intermediate results during query evaluation that we first described in [12]. In order to describe the heuristic, we first consider a single subquery that has the form shown in formula 3.2 of Section 3.3. We will describe our algorithm operating on the complete query that has the form shown in formula (3.1) in Section 3.6.3. Our method relies on an estimation of the final result size of each union subquery. To obtain this estimation we should gather some statistics from the database in the form of data summarization for all the columns that can be possibly referenced from a query, that is all the columns in the SQL query of some mapping assertion. As making an estimation for an arbitrary FOL query is an involved process, we make a distinction between assertions in $\mathcal{M}_{CQ}$ ($R_{i+l+1}, ..., R_{i+m}$ in formula 3.2) and assertions in $\mathcal{M} \setminus \mathcal{M}_{CQ}$ ($Aux_{i_1}, ..., Aux_{i_n}$ in formula 3.2). We consider that the latter are primitive tables as if they were virtual views, and we collect statistics only for the output columns, whereas the former are parsed and

we collect statistics for all the referenced columns. We will refer to each conjunct in the right-hand side of (3.2) as an *input table* of query $Q_i(\vec{f}_i(\vec{x_i}))$.

Let q be a query as in (3.2) and $I_i(\vec{x_i})$ be an input table of q. The query $ans(\vec{x_{c_i}}) \leftarrow I_i(\vec{x_i})$, where $\vec{x_{c_i}}$ contains exactly the variables of $\vec{x_i}$ that appear more than once in q, will be called the *projection query* of input table $I_i(\vec{x_i})$ from q. Additionally, let D be a database instance (which will be implied).

3.6.1 Analyzing External Tables

As we operate outside the RDBMS engine, in order to extract the needed information we should import all the corresponding data, something that is clearly not practical. Luckily we have several other options. One such option is to only import a random sample and extract the needed information from that, as most database vendors support ordering the results by a random function. Another option is to obtain the data summarization directly from the RDBMS, if it provides a way to access this information. This option is likely to give the most accurate results, but it is highly dependent on the specificities of each database vendor. One third option is to build a simple single-bucket histogram for each column, by sending for execution queries that ask for the number of values, number of distinct values, minimum and maximum value. Simple histograms like this are known to give imprecise selectivity estimations for filter and join results of attributes that exhibit skewness [44], but on the other hand their construction and usage is faster in comparison to more elaborate kinds of histograms. For our experiments we have chosen the last option, as it is fast and simple and can be applied to any underlying RDBMS. This is an one-time offline process, that needs to be done before query execution, similar to an analyze command in a database schema, as it only depends on mappings and data. Also, as it is crucial to have an accurate estimation of the number of duplicate answers that come from different mappings for the same predicate, we execute queries counting exactly the distinct number of answers for queries in bodies of mappings that can possibly formulate a combined mapping assertion. These mapping assertions can simply be identified offline as the subsets of mappings whose heads can be unified during the partial evaluation. Regarding duplicates coming from a single mapping, adopting the commonly used value independence assumption between the result attributes and the uniformity of values in an attribute [100], we estimate the distinct tuples of the relation to be the product of the distinct values of its attributes. In case this value is larger than the number of tuples in the relation, we assume that there are no duplicate tuples in the relation.

3.6.2 Early Duplicate Elimination of Intermediate Results

First, we define the *duplicate-tuple ratio* DTR_R of a relation instance R is equal to $\frac{\sum_{t \in US_R} \mu(t)}{|US_R|}$. A relation instance with DTR equal to 1, will be called a *duplicate-free* relation instance. Now, let us suppose that we have a single SQL subquery coming from the unfolding step and we have to take the decision regarding a single input table (either

"real" primitive table or virtual view) used in this subquery; we will take into consideration different union subqueries in Section 3.6.3. In this case, it may be advantageous to dictate the RDBMS to perform the duplicate elimination on projection query of the specific input table at the beginning of query execution, store the duplicate-free intermediate result in a temporary table and use it for the specific query. This can be done in several ways depending on the exact SQL dialect and capabilities of the underlying system. For example one can use (non-recursive) common table expressions or temporary table definitions. Of course the exact decisions as to when this should happen depend on several factors, including the exact query, the DTR of the projection query of the input table, the number of uses of the specific input table in the query, the choice to save the temporary table in disk or keep it in memory and several other factors that depend on the database physical design, database tuning parameters, the exact query execution plan and the evaluation methods chosen by the optimizer of the RDBMS. As mentioned, it is difficult for all these factors to be estimated outside the database engine. For this reason, in what follows, we propose to take this decision according to a heuristic that depends only on the size of the data and the DTR of the input table, whose estimation can be obtained using data summarization.

The main assumption that we make regarding duplicate elimination states that the impact of an input table with DTR equal to a constant number n in the number of tuples of the final query result is proportional to n. As a result of this assumption, the selectivity of the query plays the most important role regarding the duplicate elimination decisions. Intuitively, a query whose result size is much larger than the size of the intermediate result for which we examine the duplicate elimination option, it is expected to be faster if we first perform the elimination, as each tuple of the intermediate result has as impact the creation of a large number of tuples in the final result. On the other hand, when we have very selective queries with few results, whereas the size of the intermediate result under consideration is much larger, one would expect that each tuple of the intermediate result does not add that much to the total cost of the query in order to counterbalance the cost of a duplicate elimination, especially when expecting the optimizer to limit the sizes of intermediate query results as soon as possible.

A Heuristic Regarding Duplicate Elimination. Given a database instance D, a query q that has the form (3.2) and whose result over D is the relation instance Q and an input table $I_i(\vec{x_i})$ of q, then perform duplicate elimination on input table $I_i(\vec{x_i})$ prior to execution of q if

$$Size_Q - \frac{Size_Q}{DTR_{Ans}} > \frac{Size_{Ans}}{DTR_{Ans}}$$

where relation instance Ans is the result of the projection query of $I_i(\vec{x_i})$ from q on D and $Size_Q$ and $Size_{Ans}$ are the estimated sizes (in bytes) of relation instances Q and Ans respectively. That is, duplicate elimination should be performed if it is expected that the reduction on the size of the final result will be bigger than the size of the intermediate result with duplicate elimination.

3.6.3 Cost-based Translation

In this section we present the algorithm $GetTranslation$ (Algorithm 2), which, given a UCQ Q over an ontology $\mathcal{O}$ and a mapping collection $\mathcal{M}$ from $\mathcal{O}$ to a database instance D over a database schema S, it returns a SQL query over D, including a set of temporary views that should be created (contained in $CM_{temporary}$). Each one of these temporary views corresponds to a SQL query on the body of a combined mapping that exists in the SDL-tree produced by SLD-Derive$(P(Q, \mathcal{M}, D))$. In other words, the algorithm chooses a sequence of folds based on one of these combined mappings each time, that are performed repeatedly in a corresponding sequence of trees, starting from the initial SLD-tree. The $T_{current}$ variable holds the current tree at each point of execution. In each step, the fold that is expected to provide the largest gain is chosen, and this process is continued until no fold that provides gain exists. In this sense, the algorithm proceeds in a greedy way, in order to avoid examining all the combinations. The gain for each possible combined mapping is estimated based in the redundant processing that we avoid by materializing and using the specific mapping with respect to i) duplicate answers and ii) repeated operations even in the absence of duplicate answers.

Regarding duplicate answers, in correspondence with the observations made in Section 3.6.2, here the main factors that determine the behavior of the algorithm are the query selectivity and the size of the result of the SQL query in the body of each combined mapping. The difference here is that we consider the final query that is the result of UnfoldDB, instead of a single union subquery, and a combined mapping that contains many input mappings which can produce duplicate results between them, instead of a single input table of one subquery. Let cm be the combined mapping $\phi_1 \vee ... \vee \phi_n \rightarrow \psi$ in this context, for simplicity we will denote by $Size_{cm}$ and DTR_{cm} the size and DTR of the relation instance that is the result of executing the query $\phi_1 \vee ... \vee \phi_n$ over the database instance D, given that duplicate elimination is not performed. Computing and saving the combined mapping is expected to be more efficient, if the reduction on the size of the final SQL query will be bigger than the size of the temporary table resulting from the materialization of the combined mapping with duplicate elimination ($Size_{cm}/DTR_{cm}$). Using the quantity $Size_{SQL_{cm}}$ to denote the size of the result of the final SQL query when the combined mapping cm has been chosen for materialization with the duplicates eliminated, which is equal to $Size_{SQL_{current}}/DTR_{cm}$, we have that the result of UnfoldDB with input the fold of T into cm (SQL_{cm}) is preferred over the result of UnfoldDB with input T ($SQL_{current}$) if:

$$Size_{SQL_{current}} - Size_{SQL_{cm}} > \frac{Size_{cm}}{DTR_{cm}} \tag{3.3}$$

Regarding repeated operations even in the absence of duplicate answers, as discussed in Section 3.1, in order to obtain an exact cost model we should be aware of the exact execution plan and the choice of access methods for each relation in order to estimate the amount of data read and written to disk for each CQ. As this is not viable for the OBDA system that operates outside the database engine, we base our estimation on the sizes of the input relations and the size of the result. Specifically, we consider that the smaller table

in each CQ is fully scanned once, and all other tables are either probed using an index as many times as the number of final query results or are fully scanned once, depending on which of the two options has the lowest cost. In order to find the smaller table, table sizes in this context are compared by taking into consideration the filters that appear in each table in the CQ, that is tables are compared according to the size of each corresponding projection query. Also, as we do not want to take into consideration duplicates introduced from the combined mapping under consideration, for each input table that participates in the combined mapping, we take its size after we divide it by DTR_{cm}.

Let SQL be an SQL query of the form 3.1 that is the result of UnfoldDB, we will denote by RR_{SQL} the estimation for the size in bytes of redundant reads in the absence of duplicates as described. In other words, RR_{SQL} holds the sum of redundant reads for every disjunct (CQ) in the right-hand side of (3.1). Then, the result of UnfoldDB with input the fold of T into cm (SQL_{cm}) is preferred over the result of UnfoldDB with input T ($SQL_{current}$) if the estimated reduction in redundant reads from $SQL_{current}$ to SQL_{cm} is larger than than the size of the temporary table resulting from the materialization of the combined mapping with duplicate elimination ($Size_{cm}/DTR_{cm}$):

$$RR_{SQL_{current}} - RR_{SQL_{cm}} > \frac{Size_{cm}}{DTR_{cm}} \tag{3.4}$$

If we want to take both kinds of redundant processing into consideration concurrently, we simply have to add the left-hand side parts of (3.3) and (3.4):

$$Size_{SQL_{current}} - Size_{SQL_{cm}} + RR_{SQL_{current}} - RR_{SQL_{cm}} > \frac{Size_{cm}}{DTR_{cm}} \tag{3.5}$$

In Algorithm 2 we are considering the heuristic as a quantity giving the expected gain, with negative values meaning that we have loss instead of gain, as shown in Line 11 of the algorithm, since we want to compare the different options and choose the one that gives the biggest gain at each step. So the final formula used is:

$$Size_{SQL_{current}} - Size_{SQL_{cm}} + RR_{SQL_{current}} - RR_{SQL_{cm}} - \frac{Size_{cm}}{DTR_{cm}} \tag{3.6}$$

Regarding some implementation issues, we should note that we do not need to make selectivity estimation for all the results each time, but only for those that are affected by the combined mapping, that is, the disjuncts in the result of UnfoldDB that correspond to resultants in the SLD-tree which are descendants of nodes which use some of the

input mappings of the combined mapping examined each time. As a matter of fact, we can modify the gain formula so that only these disjuncts are taken into consideration in the computation of $RR_{SQL_{current}}, RR_{SQL_{cm}}, SQL_{current}$ and SQL_{cm}. Also, the queries that correspond to the temporary tables for the combined mappings contain only database values and not ontological terms. As a result, joins on the final result of the UnfoldDB are performed as in the original unfolding algorithm, only over database values, and not over IRIs. In a similar manner, the duplicate elimination during the computation of the temporary results corresponding to combined mappings is also performed over generally smaller and possibly indexed values, and finally, given that the final query result that will be sent for execution should also be duplicate free, we again perform duplicate elimination over the database values by pushing the duplicate elimination before the IRI construction as described in Section 3.5.

Algorithm 2: Translation Process

```
1  GetTranslation (M, Q, D);
   Input  : Mapping Collection M, Query Q, Database D
   Output: SQL query over D
2  CM_temporary = ∅;
   // The combined mappings that should be used as temporary tables
3  T_current = SLD-Derive(P(Q, M, D)); // The SLD-tree at each step. Initially equal to the
     result of SLD-Derive(P(Q, M, D))
4  SQL_current = UnfoldDB(T_current);
5  Add to CM_used all the combined mappings that exist in T_current;
6  MaxGain = 0;
7  do
8      foreach cm ∈ CM_used do
9          T_cm: the fold of T_current based on cm;
10         SQL_cm = UnfoldDB(T_cm);
11         Compute Gain from SQL_current to SQL_cm according to Formula 3.6 ;
12         if Gain > MaxGain then
13             MaxGain = Gain;
14             T_best = T_cm;
15             SQL_best = SQL_cm;
16             BestCm = cm;
17         end
18     end
19     if MaxGain > 0 then
20         SQL_current = SQL_best;
21         T_current = T_best;
22         Remove BestCm from CM_used;
23         Add BestCm to CM_temporary;
24     end
25 while MaxGain > 0;
26 return SQL_current;
```

3.7 Implementation and Experimental Evaluation

We have implemented our translation in an prototype extension of Ontop version 1.18.1. This version of Ontop normally uses the default unfolding method of [77] over the $\mathcal{T}$-Mappings [82] in order to emulate H-complete ABoxes, as we mentioned in Section 3.1, and employs the tree-witness query rewriting [52] on such ABoxes. We follow the same architecture, using the tree-witness approach for query rewriting and we modify the unfolding step over the $\mathcal{T}$-Mappings as described here.

Newer versions of Ontop use a different query unfolding method that employs the notion of *intermediate query* (IQ) [105]. We discuss the relevance of our method to this in Section 3.8. For this reason, we compare our method with both the default translation based in partial evaluation of logic progrmams obtained by version 1.18.1, but also with the new translation method obtained from the latest Ontop versions 3.0.1 and 4.0.2. In general, version 3.0.1 outperforms version 4.0.2, so we only report times for version 3.0.1 here, but all the execution times for version 4.0.2 are also available along with all other material in [4].

Our aim in this section is to perform an experimental comparison of our approach with other methods using well-known benchmarks. For this reason, we present experiments using the NPD and LUBM benchmarks in Section 3.7.1, comparing our approach with the translation performed by the two aforementioned Ontop versions. Then in Section 3.7.2 we compare our approach with the JUCQ approach using the datasets and queries from [59] and in Section 3.7.3 we study the performance of our method in comparison to the default translation, for different query characteristics. Finally, in order to obtain an empirical analysis of our heuristic regarding duplicate elimination, in Section 3.7.4 we perform an experimental evaluation using a micro benchmark with specific query fragments coming from queries used in the general evaluation.

3.7.1 Experiments with NPD and LUBM Benchmarks

We have performed an experimental evaluation of our techniques using the LUBM [38] and NPD [58] benchmarks, with the ontology and mappings that are publicly available at the Ontop repository on github[5] and with existential reasoning enabled. Both datasets were generated for scale 100.

The experiments in this section were carried out on a machine with an Intel Core i7-3770K processor with 8 cores and 32 GB of RAM running UBUNTU 18.04, using PostgreSQL version 11.3 as a backend. PostgreSQL was setup and tuned for usage in a machine with 32GB RAM. The schema and data in all systems were identical and all the proposed indexes were created. The database size was about 1.1 GB for LUBM and about 5.8 GB for NPD.

[4] `http://cgi.di.uoa.gr/~dbilid/experiments-obda/`

[5] `https://github.com/ontop/iswc2014-benchmark/tree/master/LUBM` and `https://github.com/ontop/npd-benchmark`

3.7.1.1 Queries and Mappings

For LUBM benchmark in total **84** mapping assertions were produced as $\mathcal{T}$-Mappings from Ontop. For LUBM we used the original 14 queries. For NPD we used a subset of 19 out of the original 30 queries: queries 1-12, 22-25 and 28-30, excluding the queries that use GROUP BY, as it is not supported by the used Ontop version, queries that contain OPTIONAL and queries with empty translation due to incompatible IRIs. To these queries we added four more, in order to showcase the advantage of duplicate elimination coming from a single mapping. The reason for this addition is that despite the fact that many mappings introduce duplicates, the existing queries are only using a small subset of the mappings that mostly avoid this problem. We believe that the four added queries are sensible and simple, yet their evaluation proved very hard. This showcases that the problem we are dealing with is also present in the NPD benchmark. These new queries are numbered 31 to 34 and presented in Appendix A. All SPARQL queries were executed using the DISTINCT modifier.

3.7.1.2 Overhead in Setup and Optimization

The time needed to gather all the necessary statistics and analyze tables prior to the first deployment of the system as described in Section 3.6 was 48 seconds for LUBM and 3 minutes and 10 seconds for NPD. Total optimization time for the 14 LUBM queries total time increased from **325** ms to **360** ms, whereas for the 23 NPD queries the increase was from **1115** ms to **1380** ms. The given times include the total time from parsing each SPARQL query to outputting the corresponding SQL query. The first time is the time needed by the original Ontop version 1.18.1, whereas the second time is the time needed by our modified version.

3.7.1.3 Results

For each query we used a timeout of 1000 seconds. For each setting, all queries were executed sequentially according to their numbering, after a full system reboot. The given times measure the total time needed for each query including the optimization time in Ontop, the execution time in the relational back-end and the time to obtain the results in Ontop. All the results were obtained, but they were not saved or processed otherwise. The combined mappings chosen by our method were materialized as temporary tables during execution in the same session as the main query and unique indexes were created on those tables. All times are in milliseconds. All results and the produced SQL queries, as well as all the necessary material to reproduce the experiments are available in the link given in the beginning of this section.

Results are presented in Table 3.1 for NPD queries and in Table 3.2 for LUBM queries. Results in column *v1 Default* contains the execution times obtained by the Ontop version 1.18.1, column *v1 Opt.* contains the times obtained by the modified Ontop version accord-

Query	v1 Default	v1 Opt.	v3	#Results
NPD 1	4899	5258	13696	1627744
NPD 2	4189	4142	5015	172751
NPD 3	1155	1119	1535	83737
NPD 4	20542	20899	27159	1627744
NPD 5	54	66	128	193
NPD 6	33234	23533	36128	1231564
NPD 7	1438	1377	1489	180
NPD 8	307	303	ERROR[1]	5974
NPD 9	2354	2222	1537	12750
NPD 10	4243	3649	3800	79512
NPD 11	86773	7650	8523	418056
NPD 12	122712	14376	16824	838430
NPD 22	6373	3247	8003	1113200
NPD 23	6565	3304	44340	763400
NPD 24	2437	498	ERROR[1]	147400
NPD 25	10055	9324	12106	1725400
NPD 28	32343	22815	167362	2141968
NPD 29	90271	17212	26400	419834
NPD 30	163276	26661	58143	705984
NPD 31	TIMEOUT	29771	54641	2979400
NPD 32	1085	318	746	8000
NPD 33	77139	19545	24509	148037
NPD 34	5443	3329	18678	486000
Avg.	30768[2]	9592	25274[2]	

[1] Error during unfolding

[2] Excluding timeouts and errors

Table 3.1: Results for NPD scale 100 (Times in ms)

ing to our approach and column *v3* contains the times obtained by Ontop version 3, the latest stable Ontop release. The average execution times for each case are also shown in the bottom of each table, excluding errors and timeouts. For the case of NPD queries, there was 1 timeout from v1 Default for query 31, and two errors during unfolding from Ontop v3. The exact error message for each error can be found at our result repository. According to the results, our approach outperforms on average both Ontop version 1.18.1 and version 3. For the NPD benchmark the decrease in average execution time obtained by our method is **69%** and **62%** in comparison to version 1.18.1 and version 3 respectively, while for the LUBM benchmark the decrease is **31%** and **12%** respectively. Also, with very few exceptions, our method outperforms the other two approaches on every single query.

Query	v1 Default	v1 Opt.	v3	#Results
LUBM 01	543	587	685	4
LUBM 02	1283	1272	1377	264
LUBM 03	129	87	101	6
LUBM 04	149	125	438	34
LUBM 05	69	98	71	719
LUBM 06	17086	8868	29419	1048532
LUBM 07	259	306	334	67
LUBM 08	393	301	1079	7790
LUBM 09	47126	33518	16539	27247
LUBM 10	16	16	13	4
LUBM 11	191	187	192	224
LUBM 12	132	134	245	15
LUBM 13	112	111	138	472
LUBM 14	3096	2826	4406	795970
Avg.	5042	3460	3931	

Table 3.2: Results for LUBM scale 100 (Times in ms)

3.7.2 Comparison with the JUCQ Approach

In this section we compare our method with the approach from [59]. As this implementation is not part of the Ontop release, we directly use the queries produced by this approach, which are available at the Ontop examples github repository [6]. For this reason, in all the experiments presented in this section we only report the time for executing the SQL queries in PostgreSQL, omitting the time for query unfolding. For measuring the execution times of the JUCQ approach, we used the scripts provided in the aforementioned github repository. As in the previous section, we also include the times obtained using the versions 1.18.1 and 3 of Ontop. The execution environment is the same as in the previous section.

We use the exact benchmark and queries that were also used in [59]. Specifically, we use the OBDA version of the Wisconsin benchmark [30], with the same ontology and mappings, for which we have created 24 instances of the base relational table, each one with 1 million tuples. This is the exact setting used in [59]. The results of the Wisconsin benchmark are presented in Table 3.3, where there are two different query sets, one that contains queries consisting of 3 atoms, and the other with queries consisting of 4 atoms. Each query set contains 84 queries, and the average execution time for each approach is shown. Our approach outperforms all other translations, followed by the JUCQ approach, whereas the worst performance is obtained from the default translation of version 1, which is the only approach such that timeouts occur. One other observation has to do with the execution times for the UCQ (default translation of Ontop 1.18.1) and JUCQ translations reported in [59]. Specifically, our execution times for these two sets of approaches seem

[6] `https://github.com/ontop/ontop-examples/tree/master/iswc-2017-cost/`

Query Set	v1 Default	v1 Opt.	v3	JUCQ
3 atoms	73133	22589	53624	30528
4 atoms	223684[1]	30922	64048	43926

[1] Excluding 24 timeouts

Table 3.3: Average execution time (ms) for Wisconsin Benchmark

Query	v1 Default	v1 Opt.	v3	JUCQ [59]	#Results
NPD 6*	91083	21700	132787	295445	2150854
NPD 11*	169833	9189	20070	204426	734214
NPD 12*	74001	5415	11471	14699	734214
NPD 31*	224925	18201	3980	ERROR[1]	1718
AVG	139960	13626	42077	171523[2]	

[1] Error during execution after 221 seconds

[2] Excluding Errors

Table 3.4: Results for NPD queries from [59] (scale 100-Times in ms)

to be much better. For example, in their reported times, timeouts of 20 minutes occurred in every setting, and the average execution time for the JUCQ approach was 160 seconds for the 3 atoms query set, whereas in our experiments the corresponding time is only 30.5 seconds. These differences can possibly be attributed to different versions of the PostgreSQL database (they used version 9.6) and different tuning parameters of the database engine. Other than that, our findings are consistent with theirs. Specifically, we observed the largest improvement of JUCQ with respect to the default UCQ translation for queries with more mappings and redundancy. The behavior of our approach is similar, exhibiting large improvement for these queries in comparison to all other three approaches.

Finally, we use the same modified NPD queries NPD 6*, NPD 11*, NPD 12* and NPD 31* as in [59], executed over the scale 100 of the NPD benchmark. This is different from [59], where these queries were executed only over the original NPD dataset (scale 1). The results are presented in Table 3.4. Again our approach outperforms all other approaches. Also, regarding the JUCQ translation, the results here show a different situation in comparison to the Wisconsin benchmark, as it exhibits the worst performance and also a timeout occurs for query NPD 31*. The queries produced by the JUCQ approach seem in general more complicated from the ones produced from the other three approaches.

3.7.3 Performance gain

In this section, we study the performance gain of our optimized method over the default translation which is obtained by partial evaluation of logic programs and leads to generation of UCQs. Following the setup of [59], we use the Wisconsin benchmark, generating

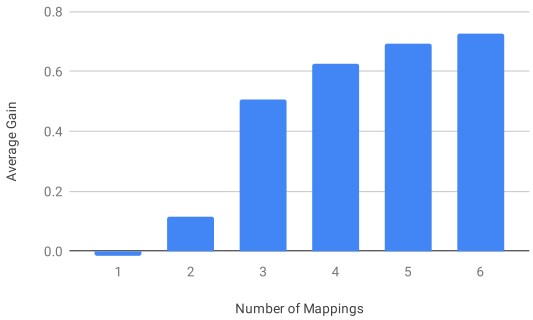


Figure 3.9: Performance gain for varying number of mappings per predicate

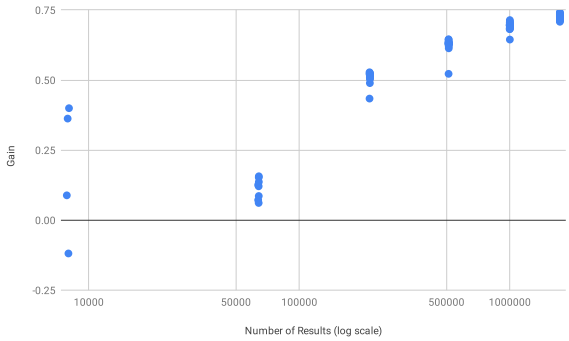


Figure 3.10: Performance gain with respect to number of results

24 tables with 1 million tuples per table, executing 84 queries with 3 atoms each, with a varying number of mappings used for each query (from 1 to 6) and we compute the performance gain using the formula $1 - (\text{Opt. Time}/\text{Default Time})$. In Figure 3.9 we present results for each number of mappings per predicate. The figure presents the average gain for all the queries per case (1 to 6 mappings). As expected, when there is only 1 mapping per predicate, our method does not generate any temporary table, and as a result, it performs roughly the same as the default translation. Starting from two predicates, our methods begins to outperform the default translation, reaching an average gain of more than 0.7.

In Figure 3.10, we present a scatter chart with the performance gain with respect to the number of results for the 84 queries of the benchmark. As shown, the queries are partitioned in visually distinct groups with respect to the number of their results. The effect of query selectivity is evident in this chart, with our method becoming increasingly efficient compared to the default, as the number of results grows larger, achieving a gain of 0.75 for the queries with about 1.7 million results. On the contrary, for the first group of queries, with low number of results, we cannot see a consistent behavior in comparison with the default.

3.7.4 Evaluating the Duplicate Elimination Heuristic

In this section we present experimental justification for the use of our heuristic regarding duplicate elimination. For this purpose, we have chosen four query fragments from the LUBM benchmark and four from NPD, such that duplicate elimination is applicable on them, as it was found during the previously described experiments. The experiments of this section were carried out on a machine with an Intel Core i7-3770K processor with 8 cores and 16 GB of RAM running UBUNTU 16.04. As our intention was to examine how our optimizations perform in different underlying systems, we used four different back-ends: PostgreSQL (version 9.3), MySQL (version 5.7) and two of the most widely used proprietary RDBMSs, which due to their license we will call *System I* and *System X*. All systems were setup and tuned for usage in a machine with 16GB RAM.

Each query fragment consists of a single select-from-where subquery. The fragments were chosen such that they have varying characteristics regarding the execution time, the number of results and the DTR of the mapping assertion under consideration. In order to test these queries with different selectivities, we applied to them extra filters. As LUBM100 contains information about exactly 100 universities, we used a filter on the university ID attribute in direct correspondence to the percentage of selectivity, whereas for NPD we used different filters for each fragment. We used filters that result in selectivity percentage of 1, 5, 10, 30 and 60, resulting in a total of 40 queries per system. We executed each of these 40 queries with and without duplicate elimination performed, resulting in a total of 240 runs for all systems. The results were obtained with warm caches.

In the upper part of Table 3.5 (one-time) we present the total execution times for these queries per system, depending on the duplicate elimination strategy. The titles of the first three columns are self explanatory. The fifth column gives the total time, if always the best strategy were chosen for each system. The fourth column gives the best time, if for each query and each selectivity, the best common strategy was chosen for all systems. This way, the difference between the fourth and fifth column can give an indication of how similar the behaviors of the systems are, whereas comparison of third and fourth columns can give a measure of how well our heuristic takes advantage of this common behavior.

One can observe that the strategy of always performing duplicate elimination is much better than never performing, and that even the strategy of always choosing the best approach is not extremely better. The reason for this result is that for queries with low selectivity, the execution time is much larger and dominates the total time. For these queries, performing duplicate elimination is preferable and sometimes gives up to two orders of magnitude better results. In order to simulate a query mix such that low selectivity queries do not dominate execution time, we also computed results where we give very selective queries a weight, such that queries with 1% selectivity have been executed 60 times, queries with 5% selectivity have been executed 12 times, etc. We present the total execution time under this setting in the lower part of Table 3.5. As before, exact times and queries are available at the same location [7].

[7] `http://cgi.di.uoa.gr/~dbilid/experiments-obda/`

	System	Always	Never	Heuristic	Best (common)	Best(Separate)
one-time	PostgreSQL	13345	168785	12854	12638	12353
	MySQL	281598	-	281685	279522	279265
	SystemI	10733	143616	9906	9693	9502
	SystemX	20558	27479	8588	8803	7280
query-mix	PostgreSQL	167116	618328	144984	146406	143191
	MySQL	1129311	-	1066499	1056659	1056145
	SystemI	135790	520408	102724	101984	99989
	SystemX	167761	220408	93660	90557	83045

Table 3.5: Query Results for Different Duplicate Elimination Strategies

3.8 Related Work and Conclusions

Regarding related work, [59] constitutes the most relevant research, as it also deals with cost-based translation. The authors extend the cover-based translation of [20], in order to take into consideration the mappings to arbitrary relational schemas. The authors analyze the database as a preprocessing step, in order to extract useful statistics, such as the cardinality of join results between queries in bodies of mapping assertions whose heads can be joined. Using these statistics, the authors can obtain accurate selectivity estimations for the produced queries. Unfortunately, despite the accurate selectivity estimations, the cost model used to compare the different cover-based reformulations is not realistic, as it assumes that all joins in a CQ are performed using hash joins, which is highly unlikely, and also it is assumed that every input relation is completely scanned. Also, the join order is not taken into consideration at all, something that can have a huge impact in the cost of the query. As we have discussed, this is an inherent problem of a system that operates outside the database engine. The difference with our method is that we use heuristics that apply to different execution plans and database engines, and also, at each step of our method, we compare highly relevant queries, where apart from the relations affected by the combined mapping under consideration, all other input relations and joins between them are the same, such that query selectivity plays the most important role in our decision. Also, we avoid running the query translation process multiple times, whereas in [59] for each different query cover, the rewriting, unfolding and estimation process has to be performed independently. Finally, the authors only consider mappings whose the body is always a CQ over the relational schema.

Since version 3, the Ontop system has departed from the usage of partial evaluation of logic programs for query unfolding. Specifically, it now relies on a query representation which is called intermediate query [105], in order to represent both SPARQL and SQL queries, facilitating the translation of SPARQL query operators like OPTIONAL [104] and GROUP BY. Instead, in this work we concentrate only on CQs over the ontology. We have experimentally shown that our method performs better on average for CQs in comparison with the latest Ontop versions. We believe that it is an interesting topic for future research to also apply cost-based methods to other operators present in SPARQL, possibly combining our results with the line of research carried out in [105, 104].

[92] is also relevant, as it uses a cost model in order to materialize specific views prior to query execution. This solution in many cases provides efficient query execution, but incurs expensive preprocessing and also, using materialized views in the database increases the database maintenance load, especially for frequently updated tables, as well as the the database size. Also, it is not in line with the overall OBDA approach of providing the end user with access to several underlying data sources, without the need to modify data, and on a practical level, such access may not be even possible. In contrast, we compute specific temporary views during query execution, when we estimate that this will result in lower execution cost, without affecting the original database schema.

In [46] the authors adopt a logic which enables them to avoid mappings when using an *object-relational* back-end and a combination of data completion and query rewriting. During this process primary keys are used for object identification, removing the need for duplicate elimination. Also, the authors use disjointness axioms in the ontology to further remove the need of duplicate elimination between unions. [33] presents query rewriting and optimization techniques that eliminate redundant atoms during the application of a resolution based algorithm. To do so, they employ a method that takes into consideration the *tuple-generating dependencies* (TGDs) of the ontological language they consider, which unlike the DL-Lite languages, considers atoms of arbitrary arity, thus it's conceptually closer to the relational model and does not need separate mappings, so a separate unfolding phase is not needed.

We provided a complete cost-based method for unfolding an initial query over an ontology into an SQL query with a number of temporary views ready to be executed over the external data source. According to the experimental evaluation, our method produces more efficient queries compared with other statr of the art methods.

4. FEDERATED OBDA QUERY EXECUTION

In the previous chapter we focused on the case where the declarative mappings of the OBDA setup access a single relational database. In this chapter we present research related to the scenario where mappings contain references to multiple databases, or in other words, to a database federation. Unlike the previous scenario, where the result of the query translation can be directly executed to the database, in this case we need a middleware module, that decomposes the result of query translation into different fragments, such that each fragment contains references to data residing in a single database, it imports the results of these fragments and combines them in order to produce the final query result. Here, we present the development of a module that performs this task, as it was carried out in the Optique project, by extending the Exareme system. The integration of our module in the Optique platform is described in [54], whereas a detailed overview of our system is presented in Chapter 5.4 of [49] and in Section VI-A of [51]. Experimental evaluation in the *Statoil* Optique use-case is presented in Section 7.4 of [49], and also in Section 6.1 of [104] in regards to the canonical table approach for identifying the same resource in different databases.

4.1 Introduction

Data integration refers to the process of accessing information from multiple and possibly heterogeneous data sources. In the context of OBDA this refers to scenarios where there are multiple relational databases, each one having its own schema, and we have declarative mappings that define virtual RDF data by accessing several of them. In contrast with the scenario that we explored in the previous chapter, where we only had a single relational database, here the OBDA system cannot issue the produced query obtained from the query translation process directly to these databases. Instead, an intermediate system is needed, which is responsible for decomposing the input query to different query fragments, such that each fragment can be evaluated in a single database. Then the intermediate results that correspond to the specific fragments must be combined and processed to produce the final result. In certain cases it may be preferable to take advantage of the processing capabilities of each endpoint, in order to ship fragments of the query for execution there and only import back an intermediate result, instead of the detailed data. Of course, such a decision should be *cost-based*. For instance a simple policy of sending the largest possible fragment to each end-point is often sub-optimal. Systems that accomplish these tasks are known as *mediators*, whereas the underlying database systems are known as *endpoints*.

In this chapter we present the development of a mediator system for OBDA. Our system was developed in the context of Optique project and it is based on the Exareme System, an *elastic* execution environment for complex data workflows on the cloud. These data workflows incorporate user computations in the form of *User-Defined Functions* (UDFs). Several extensions have been implemented in order for the system to be able to cope with

the demanding requirements of the federated OBDA scenario, whereas at the same time preserving its massively parallel processing capabilities:

(i) its optimizer has been re-designed in order to take into consideration common subexpressions coming from different parts of a complex query;
(ii) special data transfer operators have been implemented in order to be able to import data from endpoints;
(iii) a *federated analyzer* module has been implemented, which based on the OBDA mappings, gathers statistics about the external data;
(iv) *pushing data processing* to endpoints as a post-optimization step is considered;
(v) caching and reuse of intermediate results can be enabled to expedite query processing.

We start this chapter by providing background information regarding the Optique platform and the Exareme system (Section 4.2). Then we present an overview of the extensions made in Exareme in order to become an OBDA mediator (Section 4.3. After that we present a more detailed description of the developed optimizer (Section 4.4) and we finally present experimental evaluation (Section 4.5).

4.2 Background

In this section we give background information about the Optique platform and its deployment in the Statoil use-case, the Exareme system (formerly known as ADP), and the integration of these two components.

4.2.1 The Optique Platform

The Optique European project [1] [50] provides an end-to-end solution for scalable access to Big Data integration, where end users formulate queries based on a familiar conceptualization of the underlying domain. From the users' queries the Optique platform automatically generates appropriate queries over the underlying integrated data, optimizes and executes them on the Cloud. The efficient execution of complex queries posed by end users is an important and challenging task. The distributed query processing engine of the Optique platform (Exareme) aims at providing a scalable solution for query execution in the Cloud, and should cope with heterogeneity of data sources as well as with temporal and streaming data.

In Figure 4.1 we present the architecture of the Optique OBDA approach. The core elements of the architecture are an *ontology*, which describes the application domain in terms of user-oriented vocabulary of classes (usually referred as concepts) and relationships between them (usually referred as roles), and a set of *mappings*, which relates the terms in the ontology and the schema of the underlying data source. End-users formulate

[1] `http://www.optique-project.eu`

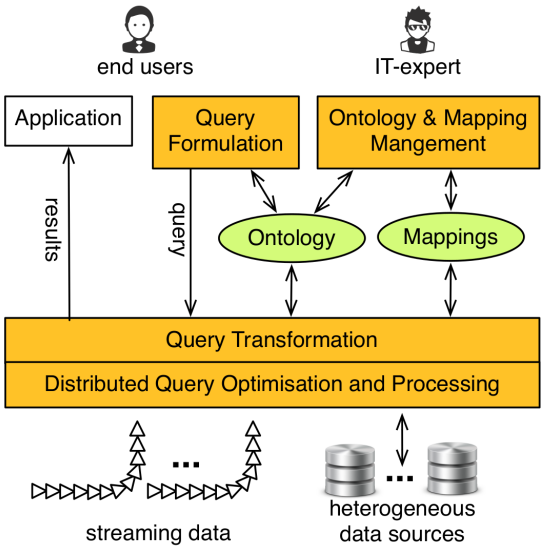


Figure 4.1: The general architecture of the Optique OBDA system

queries using the terms defined by the ontology, which should be intuitive and correspond to their view of the domain, and thus, they are not required to understand the data source schemata. The main components of the Optique's architecture are

- *the Query Formulation component* that allows end users to pose queries to the system,
- *the Ontology and Mapping Management component* that allows for bootstrapping of ontologies and mappings during the installation of the system and for their subsequent maintenance,
- *the Query Transformation component* that rewrites users' queries into queries over the underlying data sources,
- *the Distributed Query Optimisation and Processing component* that optimises and executes the queries produced by the Query Transformation component.

All the components will communicate through agreed APIs.

In order for the Optique OBDA solution to be practical, it is crucial that the output of the query rewriting process can be evaluated effectively and efficiently against the integrated data sources of possibly various types, including temporal data and data streams. This efficiency for Big Data scenarios is not an option – it is a necessity. We plan to achieve the efficiency by both *massive parallelism*, i.e., running queries with the maximum amount of parallelism at each stage of execution, and *elasticity*, i.e., by allowing a flexibility to execute the same query with the use of resources that depends on the the resource availability for this particular query, and the execution time goals. The role of the Distributed Query Optimisation and Processing component is to provide this functionality.

An important motivation for the Optique project are two demanding use cases that give to the project the necessary test-bed. The first one is provided by Siemens [2] and encom-

[2] `http://www.siemens.com`

passes several terabytes of temporal data coming from sensors, with an increase rate of about 30 gigabytes per day. The users need to query these data in combination with many gigabytes of other relational data that describe events. The second use case is provided by Statoil, a Norwegian state-owned multinational energy company, which recently has changed its name into Equinor [3] and concerns more than one petabyte of geological data. The data are stored in multiple databases which have different schemata and the user has to access many of them in order to get results for a single query. In general, in the oil and gas industry IT-experts spend 30–70% of their time gathering and assessing the quality of data [28]. This is clearly very expensive in terms of both time and money. The Optique project provides solutions that reduce the cost of data access dramatically. A bigger goal of the project is to provide a platform with a generic architecture that can be easily adapted to any domain that requires scalable data access and efficient query execution for OBDA solutions.

4.2.2 The Exareme System

The distributed query execution engine of Optique is based on the Exareme (formely known as Athena Distributed Processing-ADP) [101], a system for complex dataflow processing in the cloud. Exareme has been developed and used successfully in several European projects. The initial ideas came from Diligent[4]. Then Exareme was adapted and used in project Health-e-Child as a Medical Query Processing Engine. Subsequently, it was refined to support more execution environments, more operators, and a more query processing and optimization algorithms. Exareme has been used successfully at the University of Athens for large scale distributed sorting algorithms, large scale database processing, and also for distributed data mining problems.

The general architecture of the distributed query answering component within the Optique platform is shown in Figure 4.2. The system utilizes state-of-the-art database techniques: *(i)* a declarative query language based on data flows, *(ii)* the use of sophisticated optimization techniques for executing queries efficiently, *(iii)* operator extensibility to bring domain specific computations into the database processing, and *(iv)* execution platform independence to insulate applications from the idiosyncrasies of the execution environments, such as local clusters, private clouds, or public clouds.

The query is received through the gateway using JDBC API (Java Database Connectivity). This communication mainly involves interaction with the Query Transformation component. The Master node is responsible for initialization and coordination of the process. The Optimization Engine produces the execution plan for the query using techniques described in [55]. Next, the execution plan is given to the Execution Engine which is responsible for reserving the necessary resources, sending the operators of the graph to the appropriate workers, and monitor the execution.

The system uses two different communication channels between the different components

[3] `https://www.equinor.com/`
[4] http://diligent.ercim.eu/

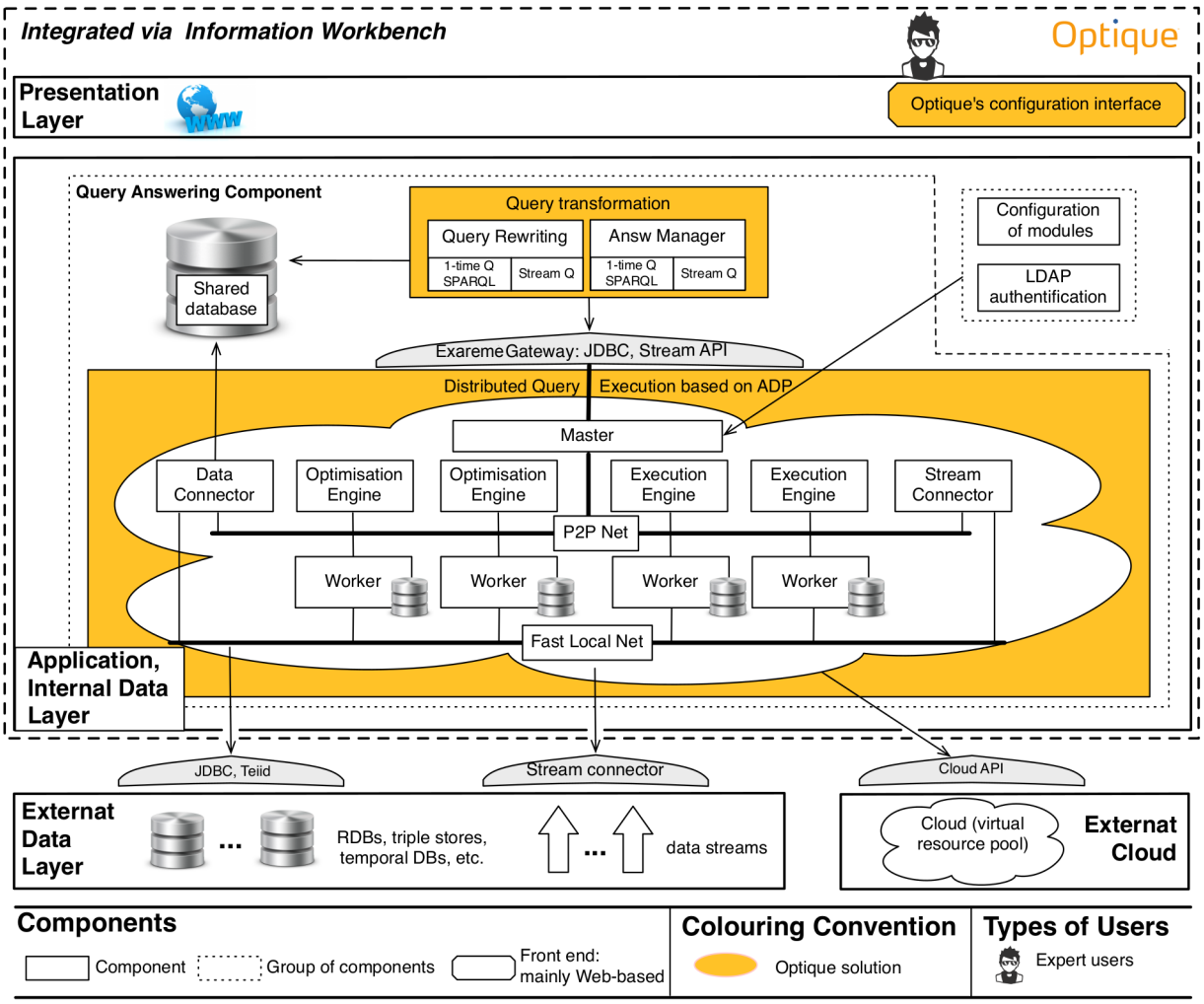


Figure 4.2: General architecture of the Exareme component within the Optique System

of the system. Data from the relational data sources, streams, and federated sources is exchanged between the workers using lightweight TCP connections and compression for high throughput. All the other communications (e.g., signals denoting that a node is connected, execution is finished, etc.), is done through a peer-to-peer network (P2P Net). For the time being, this network is a simple master-slaves using Java-RMI (Remote Method Invocation).

4.2.2.1 Language and Optimization:

The queries are expressed in SQL. Queries are issued to the system through the gateway. The SQL query is transformed to a data flow language allowing complex graphs with operators as nodes and with edges representing producer-consumer relationships. The first level of optimization is planning. The result of this phase is an SQL query script. We enhanced SQL by adding the table partition as a first class citizen of the language. A table partition is defined as a set of tuples having a particular property (e.g., the value of a hash

function applied on one column is the same for all the tuples in the same partition). A table is defined as a set of partitions. The optimizer produces an execution plan in the form of a directed acyclic graph (DAG), with all the information needed to execute the query. The following query is an example.

DISTRIBUTED CREATE TABLE lineitem_large TO 10 ON l_orderkey AS
SELECT * FROM lineitem WHERE l_quantity = 20

The query creates 10 partitions of a table with name lineitem_large with rows based on a selection condition. The partitioning is based on the column l_orderkey.

4.2.2.2 Execution Engine:

Exareme relies on an asynchronous execution engine. As soon as a worker node completes one job, it is sending a corresponding signal to the execution engine. The execution engine uses an asynchronous event based execution manager, which records the jobs that have been executed and assigns new jobs when all the prerequisite jobs have finished.

4.2.2.3 Worker Pool:

The resources needed to execute the queries (machines, network, etc.) are reserved or allocated automatically. Those resources are wrapped into containers. Containers are used to abstract from the details of a physical machine in a cluster or a virtual machine in a cloud. Workers run queries using a python wrapper of SQLite [5]. This part of the system, which is available [6], can also be used as a standalone single node DB. Queries are expressed in a declarative language which is an extension of SQL. This language facilitates considerably the use of user-defined functions (UDFs). UDFs are written in Python. The system supports row, aggregate, and virtual table functions.

4.2.2.4 Data / Stream Connector:

Data Connector and Stream Connector are responsible for handling and dispatching the relational and stream data through the network respectively. These modules are used when the system receives a request for collecting the results of executed queries. Stream Connector uses an asynchronous stream event listener to be notified of incoming stream data, whereas Data Connector utilizes a table transfer scheduler to receive partitions of relational tables from the worker nodes.

[5] http://www.sqlite.org
[6] https://code.google.com/p/madis/

4.2.2.5 Data Import:

The system provides the possibility to import data from several heterogenous sources. These data can be of many different types, including relational data, data in file formats like comma-separated values files or XML and streams. When the data is in the form of streams, the procedure is initiated through the Stream API in the Exareme Gateway, otherwise the JDBC API is used. In the first case, Master Node employs one or more Optimization Engines which produce a plan defining which worker nodes should be receiving each data stream. In the second case, the Optimization Engines also define how the data should be partitioned (number of partitions, partitioning column, etc.) and where each partition should be stored. The Master Node is notified when the execution plan is ready and then it employs one or more Execution Engines.

4.2.2.6 Query Execution:

In a similar manner, when Exareme Gateway receives a query, one or more Optimization Engines produce an execution plan which contains the resulted sequence of operators and the data partition upon which they should be applied. The Optimization Engines report back to the Master Node which then utilizes the Execution Engines who communicate with the Worker Nodes to execute the query. In the case of federated data, some Worker Nodes need to communicate with external databases. They ask queries and get back their results which, depending on the plan, need to be combined with the data that they have locally.

When the execution of the query has finished, the Master Node is notified and through the Gateway it can send a message to the external components. The results stay in the Worker Nodes, because the volume of data in the results may be prohibitive for them to be transferred in a single node. When an external component want to access the results, then it must do so by sending an extra request. When receiving such a request, the Master Node uses the Data Connector to collect the results or apply to them some aggregation functions (for example sum, average, etc.).

4.3 Overview

In this section we give an overview of the development of the OBDA mediator using the Exareme engine, considering as input query a query that has the form of the unfolded query produced by the default translation process of Ontop, as described in the previous chapter.

4.3.0.1 Federated Analyzer

Since the Exareme optimizer is cost-based, the first step in order to be able to make cost estimations for different federated query plans, is to analyze the columns of base tables residing in different endpoints. This is an offline process taking place before query answering. Initially, the OBDA mappings are parsed and a list of all base tables referenced there is obtained. This way Exareme avoids gathering statistics for tables that cannot show up in an unfolded SQL query. This number can be very large for the databases considered in the Statoil environment and this simple optimization saves a lot of computation. For each column of the obtained tables Exareme sends to the corresponding endpoint queries that ask for the different values, the minimum and maximum value and the column size. This way we can obtain basic statistical measures without having to resort to the often unfeasible task of importing all the data.

4.3.0.2 Common Subexpression Identification

Common subexpression identification refers to the process of identifying the same query fragment in different queries, or in different parts of the same query, and the equally important task of deciding if the specific subexpressions should be computed only once and reused or not. This last decision is not obvious, as reusing a subexpression includes the cost of materializing the intermediate result to disk, whereas if the tuples of the subexpression, as they are produced, can be pipelined to the next query operator, it may be preferable to compute it from the beginning. The decision becomes even more complicated, as when many common subexpressions exist, the choice for each one possibly affects the cost regarding the choice for the rest. In a parallel environment, the decision to reuse can be even less preferable, as independent query fragments can be computed simultaneously.

Using state of the art techniques in common subexpression identification proved to be crucial in evaluation of OBDA queries, as these contain highly correlated union subqueries. Consider for example the query shown in Figure 4.3. It consists of two different unions and accesses three different endpoints. Note that the join between tables SLEGGE_EPI.WELLBORE and SLEGGE.STRATIGRAPHIC_ZONE is a common subexpression for these two unions. Exareme includes a Volcano-style optimizer and models the different possible query plans using an AND-OR graph. The optimizer implements a greedy heuristic that was proposed in [87] in order to take the aforementioned decisions.

4.3.0.3 Pushing processing to endpoints

In a data integration setting where each endpoint is an RDBMS, a mediator can take advantage of the corresponding processing capabilities in order to “push” a query fragment for execution in the endpoint and obtain an intermediate result. One could think of a process of query decomposition where maximal fragments that can be executed

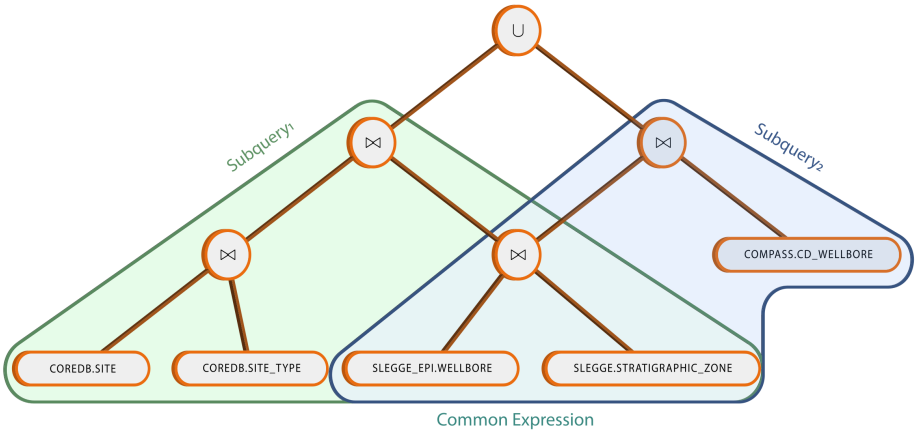


Figure 4.3: Query Plan.

in each endpoint are identified and sent for execution. Unfortunately, this approach often leads to inefficient execution plans, as very large intermediate result, that otherwise could be avoided, may be produced. Consider again the example from Figure 4.3. The join between SLEGGE_EPI.WELLBORE and SLEGGE.STRATIGRAPHIC_ZONE can be pushed to the EPDS endpoint, but if this join leads to a very large intermediate result, then maybe a better query plan would be to import each table separately and use a different join order, by first joining one of the tables with a result coming from another endpoint. The same situation arises for the joins between tables COREDB.SITE and COREDB.SITE_TYPE. For this reason, we examine opportunities for pushing processing towards the endpojnts as a post-optimization step. This is done after an optimized plan, possibly with common subexpression re-usage, has been obtained. In such a plan, we consider pushing fragments that only touch tables from a single endpoint, as long as there is no sub-plan marked as materialized for re-usage. If no plan with more than a single descendant table is found to be beneficial to be pushed to an endpoint, then separate requests for each base table are sent. These requests contain only possible filters and projections for the corresponding base table.

4.3.0.4 Caching Intermediate Results

Caching of intermediate results refers to the process of keeping results coming from evaluation of a query for future reuse. These results correspond to query fragments imported from endpoints, or results of processing that takes place inside Exareme, for example intermediate results chosen to be materialized from the common subexpression identification optimization or final results of a query. As a subsequent, possibly different query is coming for evaluation, the optimizer should choose a plan, by taking into consideration existing fragments in the cache, and estimate the cost of plans that reuse such fragments accordingly. In the context of data integration this can lead to important savings, as the need of data import can be completely avoided. An eviction policy that guarantees data

freshness can be applied, for example by specific timeouts. Also, it is important to provide the user with the option to enable or disable use of the cache on a per query basis.

4.4 Query Optimization in the OBDA Mediator

Query optimization in the Exareme mediator system is based on transformation-based optimizers, that also take into consideration common subexpression in different parts of the query and existence of intermediate results in cache.

In the next section we first describe the main ideas behind transformation-based optimization and the common subexpression identification procedure used. In order to apply these ideas to a distributed system like Exareme, in Section 4.4.2 we describe the required incorporation of repartitioning for distributed processing and the search process. In Section 4.4.5 we discuss ways to improve common subexpression identification for the queries produced by Ontop. Then we describe the main ideas of adding the consideration for federated execution into the optimization phase.

4.4.1 Transformation-Based Optimization

Transformation-based or *rule-based* optimizers start from an initial query plan and apply possible transformations in order to examine alternative plans and keep the one with the lower cost. This process is sometimes called *top-down* optimization, in contrast to classic System-R [25] like *bottom-up* optimization, which starts from the best access plans for each base relation and proceeds to combine the best of them in order to build the complete query plan. The work of [35] and later of [34] present an efficient way for exploring the search space of alternative plans using *memoization*. Each logical expression is kept in a hash table. The memo structure keeps the best plan for each equivalent class of expressions. During the application of the transformation rules, if a new expression is generated, then a look-up to the memo structure is made for its hash value and if the expression is there the optimizer avoids redundant work of re-optimizing this expression. The optimizer takes also into consideration *physical properties* of the relations; for each equivalent class it also keeps plans that result in a relation with a desirable physical property, e.g. sorting or partitioning in a specific column. These properties can be the result of the algorithm that was used for the implementation of the operator, or from a separate algorithm that was used to ensure the presence of the desirable property. In the latter case the algorithm is called an *enforcer*. Depending on the transformation rules that are examined, one can generate a specific set of plans, e.g. only left-deep joins or bushy joins [75].

We will give an example using the Logical Query DAG representation that is also used in [87] and is known as *AND-OR DAG*. In this representation, more than one query plan is plotted in the same DAG using two different kind of nodes, the OR-nodes, which refer to base or intermediate relations, and the AND-nodes, which refer to operators. Each OR-node can have many AND-nodes as children. Each of these operators will result in

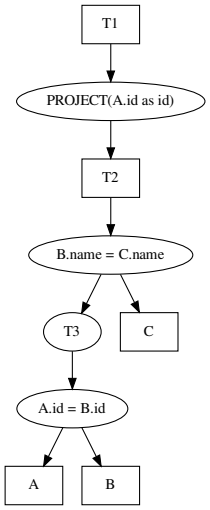


Figure 4.4: Simple Join Example

the same logical relation: their common parent. On the other hand, each AND-node has as children some OR-nodes, whose number equals to the cardinality of the operator. We will use rectangles to draw the OR-nodes and ellipses to draw the AND-nodes. Figure 4.4 presents an AND-OR DAG that holds a single query plan, whereas the DAG of Figure 4.5 holds two different query plans. The DAG of the second figure has been produced from the first one by applying the right join associativity operator: $(A \bowtie_{id} B) \bowtie_{name} C \rightarrow A \bowtie_{id} (B \bowtie_{name} C)$.

When a plan is examined, except from the application of the possible logical transformations, different implementations are also applied for each operator and different enforcers that create a specific physical property for the result. This causes the search space to become larger. Apart from the memoization of best plans for each equivalent class, the Volcano algorithm also employs a *branch and bound* style pruning by keeping the cost of the best solution found so far and discarding paths that will lead to a more expensive solution. In order for this pruning to be efficient, it is important that a good solution is found as early as possible and also to choose to expand each time the most prominent amongst all options. These choices heavily depend on the system and the execution environment.

Regarding multi-query optimization and common subexpression identification, [87, 109] use transformation-based optimizers to represent all queries in the same query tree and identify the same logical expressions between different queries. This method offers solutions to the problem of identifying possible common subexpressions as well as to the problem of deciding which of them it would be beneficial to materialize. Apart from that, these methods inherit the advantage of extensibility from the transformation-based optimizers. For each logical expression, a hash value is generated based on its operator and the input expressions. This way, equivalent logical expressions have the same hash value. During the application of the logical transformations, nodes that will be deduced as

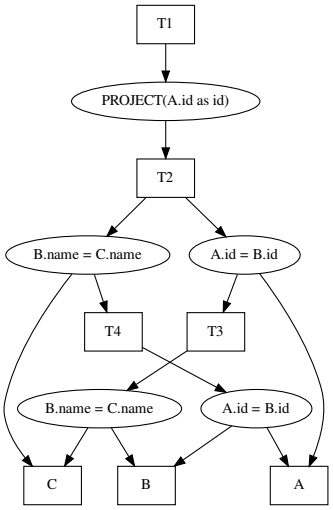


Figure 4.5: Expanded DAG

equivalent will be unified, which may lead to further unification of their ancestors.

Regarding the search process, three different methods are presented in [87]. The first one is called *Volcano-SH* and optimizes the DAG using the normal Volcano search, without taking into consideration common subexpressions that can be reused. After an optimal plan (ignoring common subexpressions) has been found, the algorithm examines what nodes of the plan are beneficial to materialize. This is not an easy decision, as the cost of each node depends on whether descendant nodes have been chosen for materialization and the number of uses of each node depends on whether its ancestor nodes will be materialized. As examining all the possibilities is exponential on the number of nodes, Volcano-SH traverses the DAG bottom-up and uses an underestimation in the number of uses for every node, such that if a decision to materialize the given node will be taken, it is certain that this will lead to a cheaper plan. On the other hand, the algorithm may also decide not to materialize nodes that should have been chosen.

Even if the optimal set of nodes would have been chosen, Volcano-SH still would not necessarily produce a globally optimal plan, as it only examines reuse possibilities between the independent optimal plans for each query. The second method, *Volcano-RU* tries to solve this problem, by providing information about what should be reused from the already optimized queries during the search process. In order to achieve this, during the optimization process, the algorithm keeps track of how many of the previous queries (best plan for each query) are using each node in the graph and whether, given the reuse and materialization costs, each node is worth to get materialized given that it is used once more. If it is worth, then the node is considered materialized and its cost for the currently optimized query is the reuse cost. After this process, the Volcano-SH is executed on the final plan to make the final decisions for the materialization. Again this method does not guarantee that the globally optimal plan will be found, as the resulting plan depends on the

order of the queries. Even if we try all the possible orders, something that it is intractable, still the final choice will be made from the Volcano-SH and will not be necessarily optimal. Apart from that, for some query we may have the case that a node that in the end will be materialized, will not be considered as such, thus leading in a suboptimal plan.

Finally, the third method is a greedy algorithm, which tries to find the best set of nodes that should be materialized. For each set, it computes the best plan given that the nodes are considered materialized from the beginning. The total cost for this set is the sum of the plan cost and the materialization costs. The method starts with an empty set of materialized nodes and a set of *sharable* nodes, i.e. nodes that can be shared between different queries in some global plan. The algorithm seeks to build a good set of materialized nodes by trying to add one node each time from the set of sharable nodes and selecting each time the one that gives the larger gain. The algorithm stops when adding any of the remaining sharable nodes to the materialized set does not lead to cost reduction. Clearly the greedy algorithm invokes the search process many times. The authors propose some optimizations in the algorithm in order to share work between different invocations of the search process and also in order to completely avoid the search process for some nodes, however, according to the experiments of [87, 109], the optimization time with this algorithm is up to an order of magnitude larger than that of Volcano-SH and Volcano-RU. Apart from that, although in most cases it finds a better plan, there is still no guarantee that this is the globally optimal plan, as at each step it chooses to materialize a single node that will give the larger improvement, despite the fact that the combination of two or more other nodes could give a better result.

4.4.2 Incorporating Partitioning Information in the Search and Pruning

In this section we discuss extensions that are required in order to apply the DAG representation in a distributed system like Exareme. We describe the incorporation of partitioning information in the DAG and discuss steps that are taken during the search process, which help reduce the optimization overhead. We also describe the system properties and the assumptions that we are making during the execution, as well as the modifications in the multi-query optimization algorithm that emanate from these properties.

We assume that each table is either partitioned to a fixed number of partitions, or is replicated in each node of Exareme as a whole. Using this partitioning scheme data shuffling can be avoided for joins between replicated tables, which can be performed on a single node, or joins between one partitioned table and one replicated table, which can be performed as broadcast joins. The first option is better for large tables, whereas the second may be proved useful for small ones.

As intermediate results that are transferred through the network are always written to disk, we take as granted that these are always materialized, and we avoid examining if it would be profitable or not to do so. This decision also reduces the search space, as we have to decide only about the input results of operators that can have them readily available locally. Query fragments that are sent for execution in the nodes of Exareme can be

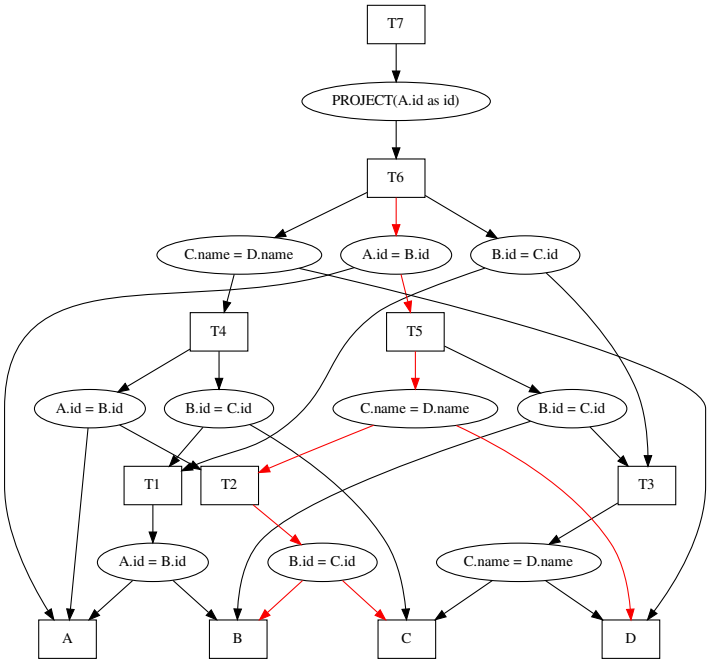


Figure 4.6: Pruned Path in the DAG

of arbitrary form, as long as the decomposer has ensured that they can be executed in parallel, without data transfer.

In order to get the cost for each operator, the decomposer asks the *cost estimator*, which gives estimations based on statistics for each column of each table. Apart from the cost, the decomposer also gets an estimation about the size of each intermediate relation.

```
SELECT A.id
FROM A, B, C, D
WHERE A.id=B.id AND B.id=C.id AND C.name=D.name
```

Listing 4.1: SQL query

Regarding the transformations, we want to examine bushy join plans because they can increase the parallelism of the execution plan, but at the same time we want to exclude cartesian products, as it is very unlikely that they can lead to a good execution plan, and their incorporation leads to a significant growth in the number of plans that needs to be considered. The join transformations that we use during the logical expansion of the DAG are the left join associativity and join commutativity, avoiding cartesian products. In Figure 4.6 we can see the DAG that corresponds to the Query 4.1, after the application of the join transformations. We do not show the commutativity, where for each join operation in the figure, there should also be a join operation with reverse operand order. We also use a transformation for pushing projections inside the joins that is not shown in the figure.

Regarding the common subexpression identification, we assume that the process is carried out in the spirit of Volcano-SH, that is each query is optimized in turn, knowing what

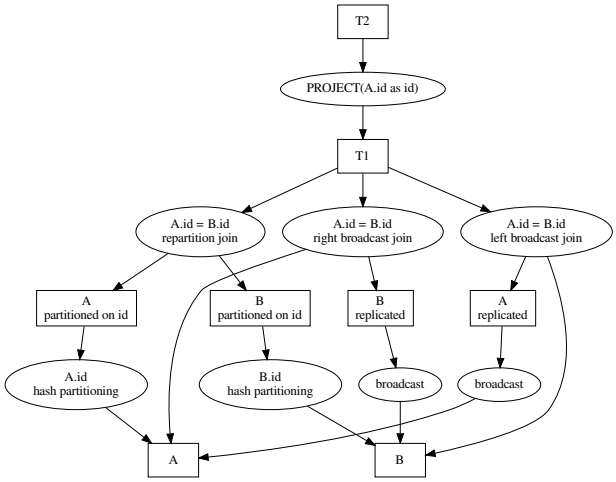


Figure 4.7: Adding Partitioning Information

previous queries have chosen for materialization and computing the cost of each node accordingly. After the optimal plan has been found for each query, the nodes in this plan which are the result of a repartition or broadcast operator are marked as materialized. This is based on the previously explained assumption, that data transferred through the network are always written to the disk of the destination node. In order to also take into consideration results that are processed locally by the nodes of Exareme and decide about their materialization status, we can again use the Volcano-SH decisions in order to avoid examining all the choices, the number of which is exponential in the number of nodes.

We treat partitioning as a physical property [35]. In [108] the authors present the incorporation of partitioning into a system for massive data analysis, which also uses a transformation based optimizer. The authors propose a single logical *data exchange* operator which covers all kinds of data transfer and can have several physical implementations. Repartition and broadcast operators could be such physical implementations. The authors reason about the combination of partitioning with other structural properties (grouping and sorting). Nevertheless, the authors, although they stress the need for heuristics in order to deal with the large number of alternatives, do not give details about the search process. In what follows we focus on the repartition of data and leave aside other structural properties, but as shown in [108], they can be integrated to the model.

The authors of [95] use a similar method as used in [109] and examine the problem of finding partitioning schemes that will result in a table that can act as input for different operators. This problem can arise when operators have as requirement the input table to be partitioned in more than one column, e.g., a group by operator on two columns. In order to achieve this, the authors propose a second optimization phase, after the normal optimization has finished.

For each AND node that needs its input partitioned, we examine two different enforcers, one that repartitions the input table on the specified column and one that replicates the

input table to all the nodes of Exareme. These correspond to physical implementations of the data exchange operator. Other implementations can be considered as well. With respect to the partitioning scheme of the input tables, we distinguish between three kinds of joins. The *repartition join* requires both input tables to be partitioned the same way (same hash function and same number of partitions) on the joining columns. The *left broadcast join* requires the left input table to be replicated to all the nodes of the system; it does not impose any requirements to the right input table, that is, the right input table can also be replicated, or it can be partitioned in any way and in any columns. The *right broadcast join* imposes exactly the opposite requirement. These three kinds of joins are treated as different physical implementations; for each join in the logical DAG all three options must be examined in order to explore the complete search space. Local join implementations can be added as a second level of physical implementation for each of the three joins.

Figure 4.7 shows an example of the resulting DAG for the query “Select A.id from A, B where A.id=B.id”, after the addition of the partitioning information. Note that the bottom nodes for tables A and B denote the tables with any valid partitioning scheme, that is partitioned on any of their columns or replicated. The same holds for tables T1 and T2. In the example, T1 can be partitioned on column A.id and also partitioned on B.id, if the path followed comes from its first child. It can be partitioned on whatever column table B is (or replicated if B is also replicated) if the path comes from its second child, or it can be partitioned on whatever column table A is (or replicated if A is also replicated) if the path comes from its third child.

An implementation detail that will be proved useful is that during the expansion of the DAG we keep information in the form of *column equivalence classes* for each path, as it has been described in [108]. Along a path we put in the same equivalence class columns that it has been deduced that they contain the same values for all the tuples in the specific path. This information is coming from the join conditions of the path. Also, selections can lead to all the columns in a class to be equal to a constant, and we can further take advantage of functional dependencies to add more columns in a class [108]. After each join, in order to get the partition history of the resulted node, the partition histories of the input nodes needs to be merged with respect to the newly acquired information that the columns that take part in the join condition now contain equal values.

Furthermore, for each path we also keep information about the repartition history. This information is related to the column equivalence classes. As soon as it becomes certain that a repartition operator on a specific column is contained in the path, the corresponding equivalence class is annotated as participating in the repartition history. The repartition history is cleared if we meet two consecutive nodes such that commutativity does not hold between them. This information will help us to prune the search space as we will describe in the next section.

Representing partitioning information in this way creates two problems. The first one is that on some paths we may end up with redundant repartition or broadcast operators. Consider for example that T1 in Figure 4.7 has as parent a repartition join operator A.id=C.id which requests T1’s result to be partitioned on A.id. During the addition of partitioning information, a repartition operator on A.id will be added above T1 and a new OR node T1

(partitioned on A.id) will be added between the new repartition operator and the operator A.id=C.id. The newly created operator is redundant for the path that passes through the first child of T1, it is not redundant for the path that pass through the second child of T1 and it may or may not be redundant for the path that pass through the third child of T1. In this last case it is redundant only if the bottom A node is partitioned on A.id. We can be sure that the operator is not needed once we meet the next repartition operator and it is on a column that does not belong to the same equivalence class. In order to deal with this problem we must examine for every child of T1 if i) it guarantees that its result will be partitioned in the required equivalent class, ii) it guarantees that its result will not be partitioned in the required equivalent class, and iii) it does not guarantee any of the previous two conditions. In the first case we don't have to insert the repartition operator, in the second case we have and in the third case we can push the repartition requirement to the child of the corresponding operator, as in [108]. Nevertheless, in the last case, this is not always the optimal solution, as it may add the repartition operator later in the tree, whereas adding the operator above T1 could be cheaper. The decision must be taken by reckoning the costs of possible choices into it. In the next section we present the search algorithm that settles this problem, integrating the solution into the branch and bound pruning strategy of the algorithm.

The second problem has to do with the fact that when an operator does not have any requirement about the partitioning of an input table, it will only examine the possibilities that are already existing in the plan. That is, the ways that this input tables can be partitioned according to its input tables if it is an intermediate table, or the ways it has been partitioned during the import to the system if it is a base table. Consider the right broadcast join operator in the example. It will consider table A partitioned only as it was imported into Exareme, even though in some other paths it can be partitioned on different columns or replicated. It is not a problem if an optimization does not take into consideration materialized results, since it is always cheaper to choose the table as it is already partitioned instead of applying an extra repartition operator, but if there are some other options materialized, maybe it is preferable to use one of these. To deal with this problem we can keep the information about which materialized nodes correspond to each logical node and and when no specific partition requirement is present, examine all of them.

4.4.3 Adapting Volcano-style Search in Exareme

As we mentioned in Section 4.4.1, Volcano-style optimizers depend on the developer to take into consideration the system particularities and use the appropriate heuristics in order to facilitate the search process. The pruning occurs either based on the cost in a branch and bound fashion or based directly on some heuristic in order to completely avoid specific paths. In the first case, it is also important to use some heuristic in order to reach quickly a complete solution with low cost, in order to prune as much as possible the remaining paths. Furthermore, a time limit could be set for each invocation of the search algorithm. After the specific time limit the algorithm returns the best solution that has been found up to that point. Again it is important that the most prominent solutions

Algorithm 3: getBestPlan

input : e: A Node in the expanded Logical DAG, c: The column that the result must be partitioned, $limit$: Cost limit, $repCost$: enforcer cost from parent
output: The best plan

```
1  Plan result;
2  if e.isMaterialized then
3  |  result=new Plan(new Path(), getReuseCostOf(e));
4  else if memo contains e partitioned on c with repartition cost less than repCost then
5  |  result=memo.getPlanFor(e, c);
6  else
7  |  result=searchForBestPlan(e, c, limit, repCost);
8  end
9  if result!=null AND result.getCost()<limit then
10 |  return result;
11 else
12 |  return null;
13 end
```

are examined within the specified limit.

Regarding the branch and bound pruning, in [94] it was proposed that apart from the cost of the path that it has examined so far, we can also consider a lower bound based on an estimation about the cost of the remaining of the path (or the actual cost, in case some of the children are already computed), so that pruning would be more efficient. In [29], the first case is referred as *accumulated-cost bounding*, whereas the second is referred as *predicted-cost bounding*. The authors note that branch and bound, especially in the case of accumulated-cost bounding where many cost computations may already have been done before a decision to stop searching is taken, may have a counter-effect of deteriorating the search efficiency, as it comes in contrast with the nature of memoization. A path may be searched many times, each time with a bigger budget, and if pruning is decided to be done each time, then work is being repeated. In [32] a solution to this problem is given, along with several other optimizations in the search process. Note that both [32] and [29] consider that the search space has not been generated using transformations, but some top down join enumeration method. Nevertheless, the search process is similar.

Regarding the cost estimation for each operator, as in all top down optimizers, we compute it before the computation of the cost for its inputs. The rationale behind this is that the cost of each operator depends on some properties of its input tables, independently of how these tables have been created. In other words, we estimate the size of an intermediate table and the histograms of its columns, without having to compute an estimation for its total cost and explore the search space below it.

The outline of the search algorithm is shown in Algorithm 4. In this outline we only show information regarding the basic accumulated-cost bounding that interacts with the repartition operator. Nevertheless, we also employ optimizations from [32] which are not shown in the algorithm for clarity of presentation. The search algorithm is based on the original Volcano search process [35] with several modifications regarding the incorporation of

Algorithm 4: searchForBestPlan

input : e:A Node in the expanded Logical DAG, c: The column that the result must be partitioned, $limit$: Cost limit, $repCost$ enforcer cost from parent
output: The best plan

```
1  Plan result=new Plan();
2  repartitionCost=getRepartitionCostFor(e, c);
3  foreach child o of e do
4      Plan e2Plan=new Plan();
5      opCost=getCostFor(o);
6      limit-=opCost;
7      foreach child e2 of o do
8          minRepCost=min(repCost, repartitionCost);
9          Column c2=getPartitionRequired(o, e2);
10         int c2RepCost=getRepartitionCost(e2, c2);
11         if o guarantees that its result is partitioned on Eq. Class of c then
12             e2Plan.append(getBestPlan(e2, c2, limit, c2RepCost));
13         else if o guarantees that its result is NOT partitioned on Eq. Class of c then
14             e2Plan.addToRepartitionHistory(c);        /* at this point we can be sure that a
                repartition operator will be added somewhere */
15             if repartitionCost<repCost then
16                 addRepartitionNodeAbove(e, c);
17                 limit-=repartitionCost;
18             end
19             e2Plan.append(getBestPlan(e2, c2, limit, c2RepCost));
20         else
21             e2Plan.append(getBestPlan(e2, c, limit, minRepCost));
22             if result is not ptned on c AND repartitionCost<repCost then
23                 addRepartitionNodeAbove(e, c);
24                 limit-=repartitionCost;
25             end
26         end
27         //the repartition op was not added during children calls;
28         if e2Plan is not partitioned on c2 then
29             addRepartitionNodeAbove(e2, c2);
30             limit-=c2RepCost;
31         end
32         int e2PlanCost=e2Plan.getCost();
33         if result is not ptned on c then
34             e2PlanCost+=minRepCost;      /* add a penalty of minRepCost to cost of e2 */
35         end
36         limit-=e2PlanCost;
37     end
38     if e2PlanCost<result.getCost then
39         result=e2Plan;
40         memo.put(e, result, c, repCost);
41     end
42     if e.getParent is root node then
43         setNodesInPlanMaterialized(e2Plan);
44     end
45     return result;
46 end
```

partitioning. The algorithm is invoked from Algorithm 3 which examines if the best plan for the input node has already been found or if the input node is materialized. If none of these happens, then it calls Algorithm 4. The algorithm takes as input a node of the logical DAG, a column c in which the node must be partitioned, which can be null if no specific partitioning is required and a cost limit such that the plan that will be found must have a cost less than the limit. If no such plan is found, the algorithm will return null. Furthermore, the algorithm also takes as input a maximum repartition cost $repCost$. Its meaning is that it should explicitly add a repartition operator that guarantees that the result will be partitioned on c only if the cost for that operator is less than $repCost$. As a result, the algorithm does not finally guarantee that its result will be partitioned on c. The check has to be done from the point that we called the algorithm. At that point we know if finally the result was partitioned as intended or not.

In each call of the algorithm, if the operator examined guarantees that the result will be partitioned on c it does not add the repartition operator. If the operator examined guarantees that the result will not be partitioned on c, then it only adds the repartition operator if the cost for that is less than $repCost$, otherwise it returns the result supposing that a proper repartition operator will be added at some time during the parent calls. Otherwise, if the operator examined does not guarantee any of the two conditions, it makes a recursive call using the minimum of $repCost$ and the cost for repartition in the current call. Upon the return of that call it will examine if the result is finally partitioned on c. If not, as in the second case it will add a repartition operator only if the cost for that is less than $repCost$. In the last case, the partition requirement $c2$ is null, so the partition in c is retained. Regarding the partition requirement in $c2$, if this is not null, after the recursive calls it is examined if $e2$ that was returned satisfies this condition. If not, that is none of the children added the repartition operator, it means that the cheapest repartition operator must be added at this point.

Once the best plan has been found, we put it in the memo, along with the minRepCost. This is needed, because the result is optimal only with respect to parent calls that have repartition cost equal or greater than the current repCost for which the optimization took place. This is because during the optimization it is possible that a repartition operator has been added with a cost R, where $R < repCost$, but in a future call we cannot guarantee that R is less than the next repartition cost. In this future call, the optimization needs to be redone only if the returned plan is partitioned on c, otherwise we can be sure that no repartition operator has been added. In order to prevent redundant optimization, we can keep the actual repartition cost that was added at some point during recursive calls and only reoptimize if the future $repCost$ is less than that.

We also employ a technique that increases the chances that the first plans examined are prominent, so that we provide quickly a low limit to the accumulated-cost bounding. The decision has to do with first following at each OR node the paths that do not need a repartitioning operator. At each step at which we must choose the next operator, if we can choose an operator that we can be sure it can be executed without repartition, this would be preferable. As a result, we first examine plans in which we can group more joins that can be executed in parallel and are more prominent to provide good results. For example,

consider the expanded DAG that is shown in Figure 4.6, which corresponds to query 4.1 (for ease of presentation we have not applied transformation regarding the projection). When we are in node T5, we know from the previous join that we will need the result to be partitioned in B.id. So, we first choose to follow the path that guarantees that T5 will be partitioned in B.id, which is the child that contains the join operator B.id = C.id.

The described heuristic is used in order to find good solutions as early as possible. We also use another criterion in order to completely ignore a specific path that contains more than one repartition operators for a given equivalence column class, as this can be derived from the repartition histories of the nodes in the path. This second criterion is complementary to the first one. The reasoning behind it has to do with the fact that we avoid plans that do not group together operators that can be executed with the same partitioning, since we can be sure there is another branch in the DAG which contains the specific operator grouped together. Operators are grouped together with respect to the parallel execution, if there is not a repartition nor a broadcast operator between them. In the example of Figure 4.6, if we suppose that all tables are partitioned and not replicated, we will decide to mark as redundant the path shown in red. Note that this decision can be taken at a point when we can be sure that a repartition operator will be added, that is when we are meeting an operator that guarantees that its result will not be partitioned in the desired column, even if we don't know exactly the point at which we will finally add it. This happens before the exploration of the whole subtree below the specific operator. In the given example we can take the decision before exploring the space below $B.id = C.id$.

We also use another criterion that stems from the assumption that it is always better to partition large tables and replicate smaller ones. We set a *repartition threshold* below which we do not examine repartitioning a table, and a *broadcast threshold* above which we do not examine replicating a table. These two thresholds can be equal, that is we examine exactly one option for each table, or the repartition threshold can be smaller than the broadcast threshold, that is for tables with estimated size between the two thresholds we examine all options. At each operator we examine if the input tables satisfy the corresponding threshold condition with respect to the partitioning scheme required by the operator. For example, if we have a left broadcast join and the estimated size of the left input table is above the broadcast threshold, we discard the specific path and do not explore the space below that operator.

4.4.4 Search with Materialized Results

When the optimal plan for a query has been found, we mark the nodes that occur after a repartition or broadcast operator as materialized. In the next query we have to invoke again the search process, with a new set of materialized nodes. As it is observed in [87], we can take advantage of previous invocations and compute only the differences in the costs for the nodes that are impacted by the differences in the set of materialized nodes. To do that, we must recompute the cost for all the parents of each node whose materialization status has changed. The authors of [87] present the incremental cost update algorithm in order to avoid redundant computations, as we can reach a node from many different

paths. The algorithm keeps a priority heap of nodes that are sorted based on a topological number that is computed for each node during the generation of the DAG, such that each node always has a smaller number from every of each ancestors. Each time, the node with the smallest number is extracted from the heap and its cost is computed. If the cost has changed then the cost of the parent AND nodes is recomputed and parent OR nodes are added to the heap. The process continues until the heap is empty.

In [87] the specific process takes place in the context of the greedy algorithm. But in our case, we know the nodes that must be added in the materialized set even though we proceed in the spirit of Volcano RU, that is we examine the queries in turn. This gives the opportunity for one more optimization. Each time, we only need to examine nodes that are used by the next query. To do that, we keep a list of nodes whose materialization status has changed and the impact of this change has not been recomputed. This list is global for all the invocations of the search algorithm. After the optimization has finished for a query, the nodes that have chosen to be materialized, i.e., the nodes of the optimal plan that are after a data transfer operator and are not materialized from previous queries, are added to the list. For the next query, we remove from the list only the nodes that are accessed by the specific query, as this information has been kept from the traversal described in Section 4.4.2. The heap is initialized to contain only the specific nodes. Furthermore, we do not examine at all the OR nodes that are already materialized.

4.4.5 Improving Common Subexpression Identification

In this section we present some optimizations that can increase the number of common subexpressions that will be identified in queries that contain self-joins and unary filter-like where conditions. These optimizations take place during the application of the logical transformations and are independent of the physical layer and the execution environment (distributed or centralized). The motivation came from the queries produced by ontop.

One of the characteristics of many queries that result from OBDA is the presence of self joins. Let us consider an example query, taken from [58] and simplified for ease of presentation. Listing 4.2 presents a SPARQL query that asks for wellbores and the lengths of their cores, where length is greater than 50 and its unit of measurement is meters. The query is first rewritten with respect to some ontology. We will omit this process as it is not relevant for our example and we will concentrate on the unfolding to SQL based on some mappings to a database schema. The relevant mappings for our case are given in Table 4.1

Based on these mappings, one of the unions of the resulting SQL query is shown in Listing 4.3. This query contains two self joins on the table core. Notice that these two self joins could be avoided if the OBDA system had information that the combination of columns id and core is unique for the table core. But assuming that we don't have this information, we have to evaluate the query as it is. First of all, in order to plot all queries in the same AND-OR DAG, we have to assign some global aliases to the base tables. As the query is parsed, we replace the local aliases with the global ones. If there is no global alias for

$$
\begin{aligned}
npdv : Wellbore(uri(id)) &\leftarrow wellbore(id, name) \\
npdv : coreForWellbore(uri(id, coreNumber), uri(id)) &\leftarrow wellbore(id, name), \\
&\quad core(id, coreNumber, coreLength, coreUom) \\
npdv : name(uri(name), uri(id)) &\leftarrow wellbore(id, name) \\
npdv : coreLength(uri(name), coreLength) &\leftarrow core(id, coreNumber, coreLength, coreUom) \\
npdv : coreUOM(uri(name), coreUom) &\leftarrow core(id, coreNumber, coreLength, coreUom)
\end{aligned}
$$

Table 4.1: Mappings

a base table, we generate a new one. If a query contains self joins, we generate more global aliases for the same base table. Supposing that the first query that we have to plot in the DAG is the query in Listing 4.3 and that the global aliases that have been generated are alias0, alias1, alias2 for base table core and alias3 for base table wellbore, we have to plot the query shown in Listing 4.4 in the same DAG. For table wellbore we choose alias3 that we have been generated for the first query, but for table core we can choose any of the three aliases that we have generated for the specific base table. Clearly, choosing the global alias that has been assigned to the alias QVIEW1 of the first query will result in identifying more common subexpressions in the DAG. But the choice will not be always that obvious, as common subexpressions may be identified later during the application of transformations and also the final decisions for the plan must be taken during the search phase. In order to be able to find the globally optimal plan we must try all possible cases, which are three in our example. We can easily do that by adding more than one child to the AND node that represents the result of the second query. In general, if the number of global aliases for a base table is n and the query that we have to add contains k instances of that table, we must add $n!/(n-k)!$ different choices. This obviously can lead to a significant growth of the size of the DAG. Luckily, usually most queries contain at max two or three different instances of a specific base table. For queries that contain a greater number of self joins and self joins on different tables, a trade-off between identifying more subexpressions and keeping the optimization process time reasonable must be found.

```
SELECT DISTINCT ?wellbore (?length AS ?lenghtM)
WHERE {
  ?wc npdv:coreForWellbore ?w.
  ?w  rdf:type npdv:Wellbore .
  ?w  npdv:name ?wellbore .
  ?wc npdv:coreLength ?length .
  ?wc npdv:coreUOM "m"^^xsd:string .
  FILTER(?length > 50)
 }
```

Listing 4.2: SPARQL query

```
SELECT
QVIEW1.`name` AS `wellbore`, QVIEW2.`wlbTotalCoreLength` AS `lenghtM`
 FROM
wellbore_core QVIEW1,
wellbore_core QVIEW2,
wellbore_core QVIEW3,
```

```
wellbore_npdid_overview QVIEW4
WHERE
QVIEW1.`id` = QVIEW4.`id` AND
QVIEW1.`coreNumber` = QVIEW2.`coreNumber`
QVIEW1.`id` = QVIEW2.`id`
QVIEW1.`coreNumber` = QVIEW3.`coreNumber`
QVIEW1.`id` = QVIEW3.`id` AND
QVIEW3.`wlbCoreIntervalUom` = 'm' AND
QVIEW2.`wlbTotalCoreLength` > 50
```

Listing 4.3: SQL query

```
SELECT
QVIEW2.`name` AS `wellbore`, QVIEW1.`wlbTotalCoreLength` AS `lenghtM`
 FROM
wellbore_core QVIEW1,
wellbore QVIEW2
WHERE
QVIEW1.`id` = QVIEW2.`id`
```

Listing 4.4: SQL query

Regarding unary conditions, like $NOT\ NULL$, ontop may produce queries that have these conditions on different columns, but by also taking into consideration the joins on each union, these conditions can be proved equivalent. In order to identify these cases, we add to each query all the implied conditions during the plotting at the AND-OR DAG.

4.5 Experimental Evaluation

We now present experimental evaluation of our OBDA mediator system built on Exareme. In Section 4.5.1 we present the execution results for the Optique Statoil use-case and we compare the execution with and without query cache for 81 queries of the query catalog of the use-case. In Section 4.5.2 we present complementary results from the Statoil use-case, using the Canonical IRIs approach as described in [104].

4.5.1 Experiments in the Statoil Optique Use-Case

In this section we present the query results for running the Statoil query catalog using our Exareme as the federation engine. Queries are accessing the following databases described in [49]: *(i)* EPDS *(ii)* Recall *(iii)* CoreDB *(iv)* GeoChemDB *(v)* OpenWorks *(vi)* Compass

EPDS contains two different database schemas, resulting in a total number of 7 different data sources.

In total, 81 queries were executed with a 1000 seconds timeout under two different setups: with and without caching of intermediate results. In the second setup, execution started with empty cache and queries were executed sequentially, according to their numbering in the query catalog. Out of all 81 queries, 66 were executed successfully within the time

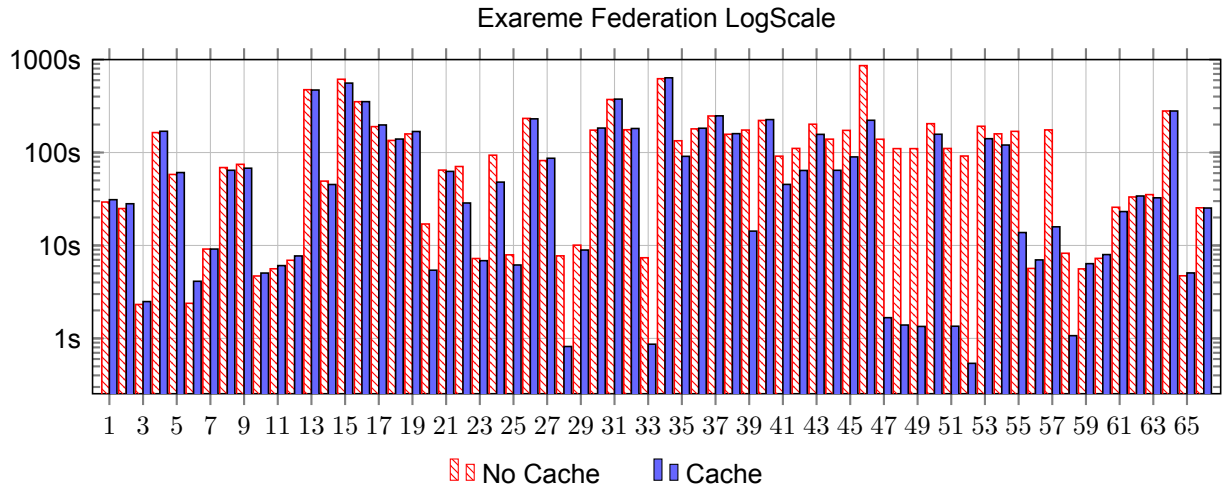


Figure 4.8: Execution Times for Federated Scenario (With and Without Query Cache)

limit and only seven of them needed more than four minutes. The average time for successful queries in the first setting was 135.6 seconds, whereas in the second setting 101.4 seconds. All results are presented in Figure 4.8 (Times are in seconds). For the experiments, Exareme was installed on a cluster of eight virtual machines, running in a protected subnet at Statoil. Each virtual machine contains 8 GB of RAM and two processing cores.

4.5.2 Experiments using Canonical IRIs

When multiple databases participate in a federation through OBDA, the same actual resource may have different IRIs in each different database. Canonical IRIs provide a way for efficiently identifying the same resource in different databases that participate in a federation through OBDA, by assigning to it an canonical IRI, which uniquely represents the resource. Normally, the sameAs property of OWL vocabulary can be used for this purpose, but as this property is transitive, in the context of OBDA, using this property would result in inability to rewrite the initial query over the ontology to non-recursive SQL [24]. For this reason, sameAs is not part of the OWL2 QL language. An initial approach for incorporation this functionality in OBDA is presented in [24], where a restricted use of sameAs is proposed. The canonical IRI approach is used instead in [104], in order to treat the large rewritings that result from this initial approach.

In this section we provide experimental results obtained by our OBDA mediator in the Statoil use case, that compare these two approaches, in order to show that our system with all the features and techniques described in this chapter, can take advantage of the optimized query form offered by the canonical IRIs approach, and reduce significantly the execution time over the initial sameAs approach. All necessary linking tables of these two approaches were used as internal Exareme tables, by importing and processing all the needed information from the endpoints.

As before, we integrated the 7 data sources (relational databases) used in Statoil, extend-

	sameAs	Canonical IRI
Total queries	76	76
Timeouts	31	11
Min exec. time	12s	0.50s
Mean exec. time	11m	4.3m
Median exec. time	11m	0.77m

Figure 4.9: Execution time and statistics for the queries in the federated setting at Statoil

ing an existing ontology and the set of mappings, and creating the tables necessary for `sameAs` and `canIriOf`. The queries and ontology are published in [41]. One of the data sources is the *slegge* database, which is also described in [41] together with the mappings toward this database.

The experiments in Statoil were run with a catalogue of 76 SPARQL queries constructed from information needs written down by geologists and geoscientists in the company. The domain of the queries is that of subsurface exploration, with a focus on wellbore information. The most complex query had 23 triple patterns, using object and data properties coming from 5 data sources. The queries were executed with a 20 minute timeout, both with sameAs approach and with the canonical IRI approach.

The *Ontop* rewriting engine and Exareme SQL federation engine all run on virtual machines deployed on the company intranet, as the data cannot be moved out. *Ontop* ran on a single machine, while the Exareme SQL federation ran on 8 other machines. The oracle databases are version 10g, and run on separate machines.[7]

We realize that this setup does not comply with the normal *clean* setup of a database experiment. However, the complexity (7 datasources) and realism (real questions and production databases) of the setup means the results have great value, although their precision is sub-optimal. Compare this with biology, where the *in vivo* experiments on live creatures, dealing with the full complexity of the organisms, may lead to results that cannot be seen in the *in vitro* experiments, and therefore are considered superior.

The minimum, mean and media query execution times, for both the sameAs and canonical IRI approaches are shown in Figure 4.9, whereas in Figure 4.10 we present a comparison of execution times for each query separately. The improvement from sameAs to canonical IRI is drastic. With the canonical IRI approach all queries, with three exceptions, are faster, there are fewer timeouts, and the majority of the queries execute within 3 minutes.

4.6 Conclusions

We presented database techniques for executing OBDA queries over a federation of relational data sources. We have implemented our proposed techniques in a new mediator system built using Exareme, and we have succesfully deployed the new system in the Sta-

[7]Typical machine: HP ProLiant Server, 24 Intel Xeon CPUs (X560@2.67GHz), 283 GB RAM.

toil use case of the Optique project, in a setting with 7 external relational legacy databases involving complex schemas and many hundrends of tables

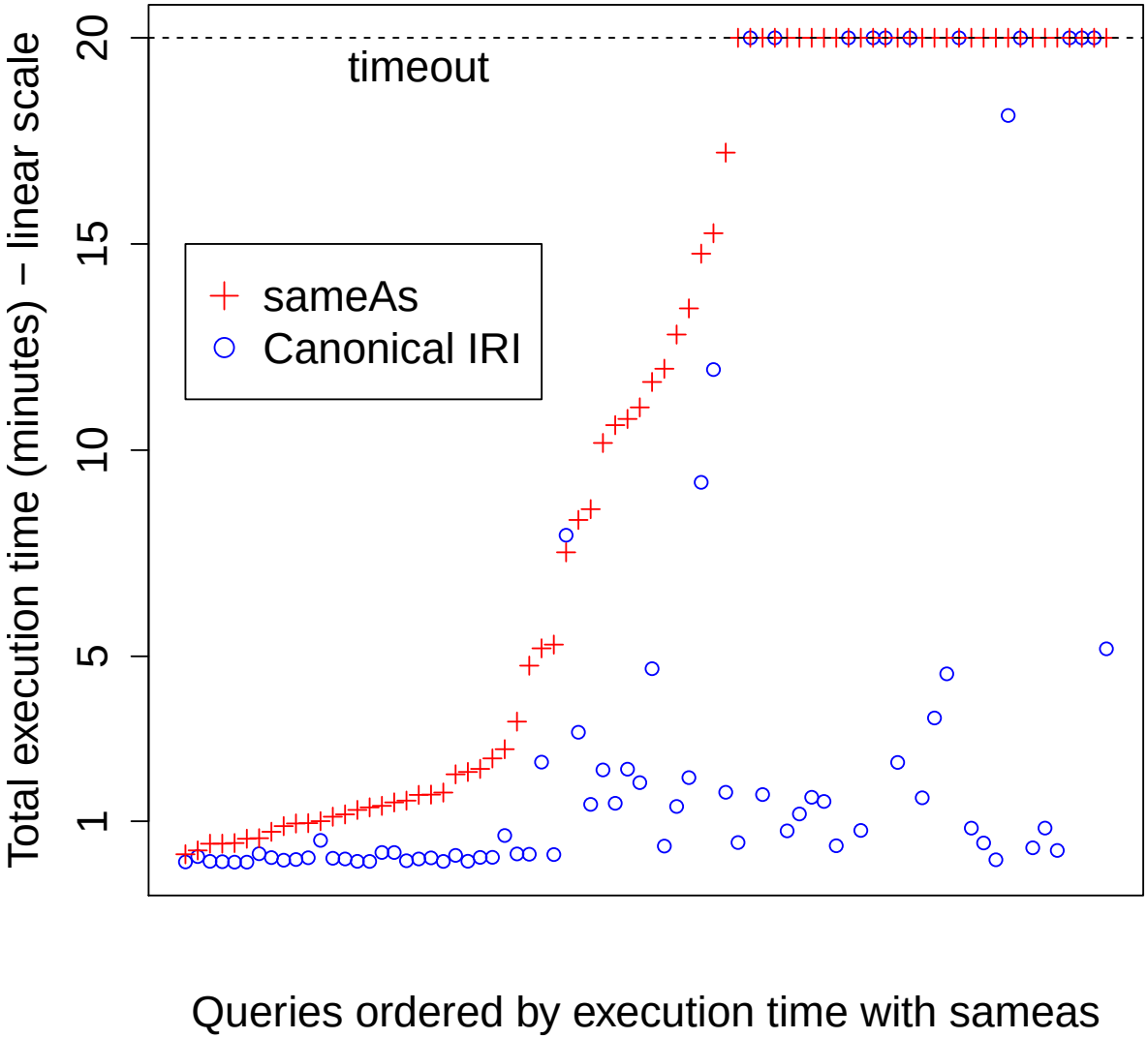


Figure 4.10: Comparison of execution time for sameAs and Canonical IRI approaches

5. IN-MEMORY PARALLELIZATION OF JOIN QUERIES OVER LARGE ONTOLOGICAL HIERARCHIES

This chapter presents the design and development of PARJ, an in-memory RDF store able to parallelize multi-join OBDA queries. Unlike Chapters 3 and 4, here we do not consider external relational databases and arbitrary mapping rules. Instead, the data are stored as RDF triples in PARJ. The results of this chapter are included in publication [15], which in turn extends our previous work [13].

5.1 Introduction

Since the adoption of the RDF data model numerous systems and research prototypes have been developed aiming at efficient SPARQL query evaluation, focusing mainly on the evaluation of BGPs which proved to be extremely demanding. Centralized systems explored different physical storage options and query execution techniques. Main storage schemas include a single triples table, denormalized property tables, vertical partitioning, graph-based storage and storage based on bit arrays. Details and references to such systems are presented in the next section. As scalability became an issue with the continuously increasing size of several datasets, distributed approaches came into play, assisted by cloud technologies such as the MapReduce framework, its implementation Apache Hadoop and several Big Data processing systems built on top of it. Most of these systems use optimizations in order to minimize the execution cycles, which correspond to Hadoop jobs and involve data transfer between the workers. This is due to the synchronous nature of the MapReduce paradigm. As a result, depending on data partitioning and replication one can achieve evaluation completely in parallel for some queries, but for queries that require communication the overhead is important due to the synchronization step.

A number of in-memory distributed systems were later proposed such that their communication is based on custom asynchronous methods, mostly on the Message Passing Interface (MPI) standard. Trinity.RDF [107] is based on graph exploration and it was the first system to follow this design. TriAD [39] and the extension of the centralized main memory RDF store RDFox with a dynamic data exchange operator [78] also use an asynchronous execution model (In what follows we will refer to the system described in [78] as the dynamic exchange operator approach), but unlike Trinity.RDF they use relational-style joins, increasing the level of parallelism for large intermediate results over the graph-based approach. In order to do so, both of these systems use expensive graph partitioning before data loading. AdPart [3] tries to overcome this problem by using simple subject-based hash partitioning and then adaptively, based on the query load, replicates specific data fragments to the workers. As a result of the initial subject-based partitioning, expensive broadcast of intermediate result occurs in case of joins on objects.

In the OBDA setting, deep and wide class and property hierarchies pose a serious per-

formance issue for all systems that perform query answering over RDF data with respect to entailment regimes that allow the definition of such hierarchies. Materializing all implied assertions with respect to these hierarchies, as it is the case in RDFS reasoning with forward chaining, is an expensive preprocessing step and it may lead to data size many times larger than the original, something that may not be viable especially for an in-memory system. On the other hand, using RDFS reasoning with backward chaining may lead to complicated queries. Many approaches exist that aim to treat these problems, mainly focusing on disk based storage.

In this chapter we present *PARJ*, an in-memory query processing system able to parallelize multi-join queries over large RDF graphs. Its name stands for Parallel Adaptive RDF Joins. Our query processing approach is inspired by the asynchronous execution model of main-memory distributed RDF stores, mainly of TriAD and the dynamic exchange operator approach. Both these approaches use expensive preprocessing in the form of graph partitioning in order to minimize communication between servers during query execution. Also, extra effort is needed in order to track the server that contains each resource. Most importantly, even in a centralized parallel environment these systems would require some form of inter-process or inter-thread communication and as a result some form of synchronization. For example, in case of rehashing, each worker of TriAD has to wait in order to receive and rehash all intermediate results from all other workers. Same kind of overheads occur in the dynamic exchange operator where each worker must hold a queue for each query atom, where incoming messages are put. This may lead to blocking execution until some other worker process results for a subsequent query atom. Also, in the dynamic exchange operator approach detecting termination is not trivial and requires a round of message exchanging. Our method ensures parallel execution of arbitrary multi-join BGPs without any form of communication or synchronization between the workers (in our case threads) while at the same time avoiding expensive preprocessing like graph partitioning. Furthermore, we adaptively decide to scan the corresponding partitions when it is preferable, instead of always using index-based nested loops as done by the dynamic exchange operator approach. This adaptive cache-friendly method can take advantage of existing (even partial) sorting of RDF triples, that further improves our join implementation. An auxiliary bit vector index can be used to avoid binary search and improve efficiency.

Regarding the physical data storage, our approach is inspired by *column-store* systems such as MonetDB [43] and C-Store [98], as we first use vertical partitioning [1] to create a separate table for each property, and then keep subjects and objects for each property in separate arrays so that each tuple can be reconstructed by relating entities at the same positions in these arrays, reminiscent of the virtual IDs of column stores. This physical design compactly stores RDF data in memory, in order to increase spatial locality during join processing. For example, for scale 10240 of the LUBM dataset with about 1.4 billion triples, excluding dictionary, the storage requirements are only 22 GB (50GB if we include the dictionary). Also, we allocate a single array position for each distinct subject or object as a simple form of column specific compression (reminiscent of the POS and PSO indexes used by Hexastore [102]) and we keep two replicas of each two-column table in different sort orders.

Finally, PARJ is able to perform scalable query answering with respect to large class and property hierarchies by providing *virtually* complete data over these hierarchies. Specifically, during join processing, we incorporate on-the-fly union computations over our physical data storage without impairment of the pipelined execution model. Our architecture is based on [82], modified in order to use our in-memory system as a triple-store, instead of a relational database, in order to perform query answering over OWL2-QL ontologies. Our approach does not require expensive preprocessing in the form of materializing ontology inferences via forward chaining, and at the same time it only has an 10-20% overhead in query execution time in comparison with complete materialization of ontological hierarchies. Our experimental evaluation provides the fastest execution times over the $LUBM_{\exists}$ [63] OWL benchmark, outperforming state of the art systems based on materialization or query rewriting.

This chapter is an extension of publication [13], where an initial version of PARJ had been presented, by adding the following novel contributions:

- An experimental evaluation with larger (LUBM 20480) and real-world (YAGO2) datasets that confirms the scalability and applicability of our approach
- A method for incorporating information about ontological type and property hierarchies during the join processing without any additions to the physical data storage layout and without affecting the scalability and effective parallelization of execution
- An implementation that couples PARJ with Ontop[21], a state of the art tool for Ontology-Based Data Access (OBDA), enabling PARJ to act as an efficient RDF store, answering queries over OWL2-QL ontologies. We also perform an experimental evaluation and comparison of our implementation with other state of the art systems for OWL2-QL query answering.

In Section 5.2 we present details of the physical data storage and execution model and in Section 5.3 we present details of the adaptive join method that allows for incorporating parallelism into processing. Query answering approach over ontological hierarchies for OWL 2 QL ontologies is described in Section 5.4. We present implementation details and experimental evaluation in Section 5.5.

5.2 Physical Data Storage and Execution Model

In this section we present our physical data storage and give an overview of the join method that allows incorporation of parallelism. First, following the common practice used by many systems, we use dictionary encoding, by assigning an integer value to each value encountered in the RDF data. We use common numbering for values appearing in the subject and object positions and a different numbering for values appearing in the property position, but for ease of presentation here we assume common numbering for all values. Thus, after parsing of an RDF dataset that contains N distinct values, our dictionary will

contain integer IDs from 1 to N. Then, we apply vertical partitioning [1] to create a separate two-column table for each property defined in the data. We keep two replicas of each two-column table, the first sorted on subject and then on object, and the second sorted first on object and then on subject. Given that a property P is assigned to integer i from our dictionary encoding, we will refer to the first replica of two-column table for P as $prop_i$ and to the second replica as $prop_{i^-}$ and we will call the tables first sorted on subject *S-O tables* and tables first sorted in object *O-S tables*.

Consider for example the following RDF data (IRIs are omitted):

```
ProfessorA teaches Mathematics
ProfessorB teaches Chemistry
ProfessorC teaches Literature
ProfessorA teaches Physics
ProfessorA worksFor University1
ProfessorB worksFor University2
ProfessorC worksFor University2
```

The dictionary encoding of the data is given in Table 5.1. Using this encoding, the two-column tables $prop_2$ and $prop_9$ that correspond to properties $teaches$ and $worksFor$ will be created.

Integer	Value
1	ProfessorA
2	teaches
3	Mathematics
4	ProfessorB
5	Chemistry
6	ProfessorC
7	Literature
8	Physics
9	worksFor
10	University1
11	University2

Table 5.1: Example of Dictionary Encoding

For each table, we store a sorted integer array with the distinct subjects (for S-O tables) or distinct objects (for O-S tables). We also store a second array of same length with the first. Each position of this second array contains a pointer to a sorted integer array and an integer denoting the length of this array. This is a pointer to the objects (for S-O tables) or subjects (for O-S tables) that correspond to the subject (respectively object) located at the same position of the first array. The reason that we keep two separate arrays has simply to do with compactly storing the integers of the first array and improving spatial locality during the join processing. We also keep track of the length of the first array, using an array of length $2 * (number\ of\ properties)$ that contains this information for all properties.

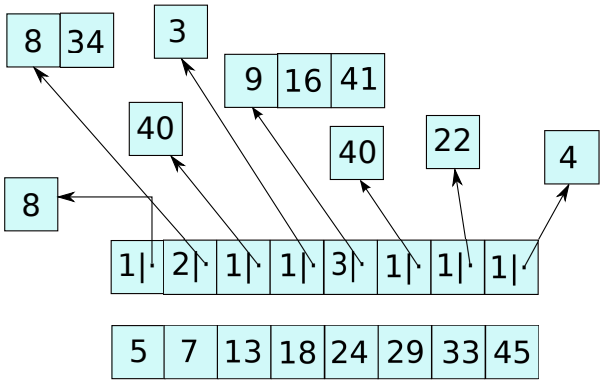


Figure 5.1: Example of Physical Data Storage for a Property Partition

Getting this information involves a simple lookup at a specific position, for example, to get the number of subjects for $prop_7$, we should look at position $2 * 7$, whereas to get the number of objects for $prop_{7^-}$ we should look at position $(2 * 7) + 1$.

Figure 5.1 contains an example of physical storage for a property table. Assuming that the specific table has been created for property $prop_3$, then it contains the following triples: $5\ prop_3\ 8$, $7\ prop_3\ 8$, $7\ prop_3\ 34$, $13\ prop_3\ 40$, $18\ prop_3\ 3$, $24\ prop_3\ 9$, $24\ prop_3\ 16$, $24\ prop_3\ 41$, $29\ prop_3\ 40$, $33\ prop_3\ 22$, $45\ prop_3\ 4$. Note that in order to avoid memory fragmentation, the different object arrays of this example can be allocated to a continuous memory area. In this case, instead of having different pointers for each position of the second array, we can keep a single pointer to the start of this memory area and only keep offsets in each position of the second array.

Our execution models targets multi-threaded environments, where each thread operates on the common data without any form of inter-thread communication. To achieve this, our join method resembles an index-based nested loops join (or merge join when possible - this will be discussed later), such that each thread is assigned a different shard of the first (leftmost) table in the join, and runs in parallel, by probing the next table to be joined for each tuple. Given a number of available threads, the first table of a join is virtually partitioned in an equal number of shards, such that every shard contains about the same number of tuples. In this way our method operates on left-deep query join trees as shown in Example 3.

Example 3. *Consider a SPARQL query:*

```
SELECT ?x ?y ?z
WHERE {
    ?x teaches ?z .
    ?x worksFor ?y .  }
```

Also suppose that the join order chosen by the optimizer (see Section 5.4.3) is the same with the order of the triples in the text of the query. This will be translated to a join $prop_2 \bowtie_{subject=subject} prop_9$*. If there are two available threads, our algorithm will start concurrently scanning two different shards of* $prop_2$*. For each tuple encountered during this*

process, it will probe, using binary search, table $prop_9$. This process can be decomposed into completely independent tasks that start from different shards and operate on read-only common data, and thus it straightforward to be implemented using threads (as it is our current execution model and implementation) or separate processes with shared memory. It is even straightforward to be implemented on different machines using complete data replication and parallelize the query across machines without any communication.

Note that for the given query, the degree of parallelism depends on the number of different shards of the first table. For more selective queries a different strategy may be needed as shown in Example 4.

Example 4. *Consider the following query, that contains an extra filter:*

```
SELECT ?x ?z
WHERE {
    ?x teaches ?z.
    ?x worksFor University1 .   }
```

In this case, suppose that the optimizer chooses the inverse join order, as it is reasonable that the filter will limit the results of the second triple pattern. In this case, table $prop_9$ should be scanned first. One first observation is that instead of scanning the whole table, we can search for tuples where object is equal to 10. To do so it is better to use the replica that is first ordered by object. After we search $prop_{9^-}$ for $object = 10$, we obtain the vector of subjects that correspond to $object = 10$ (in our case it is only value 1). Then we start scanning this vector and probing table $prop_2$ using these values. In this way we do not obtain any level of parallelism for this query, as we start from a specific value of the first table. It is easy though to recover the parallelism, if we start scanning concurrently different shards of the vector that corresponds to $object = 10$. If the query contains a triple pattern with variable in the predicate position, then a union over all properties will be needed, but this is rarely encountered in real world queries[1]. In any case, if the number of distinct predicates encountered in the dataset is very large, an ID-Predicate index similar to the one use in [106] can be useful. Also note that the exact number of threads that will be used is independent of our physical data storage and can be decided on a per query basis after data loading in memory. In our current implementation (Section 5.5) we choose to execute each query with the same number of threads (optimally this should be equal to the number of available processing cores or greater in case hyper-threading is supported as shown in Section 5.5.2.3), but an extension such that very simple and selective queries could be executed with fewer resources is possible.

5.3 Query Processing

The approach followed by RDF stores like RDF-3X and TriAD, is to take advantage of initial sorting of RDF triples, and perform merge joins when possible. Hash join is preferred when

inputs are not sorted on the join key. On the other hand, the dynamic exchange operator approach always uses index-based nested loops aiming at low memory consumption and avoiding blocking operators. Our system uses a combination of these two approaches, by taking into consideration the following points:

- When both inputs are already sorted on the join key, merge join is preferable over hash join.
- For main memory systems, index-based nested loops (in our case in the form of binary searches over the inner table stored as an array) does not exploit data locality and also it is not amenable to efficient data prefetching due to conditional branching. Nevertheless, for very selective joins, it may still be faster than merge join.
- For RDF data processing, where the initial triples are sorted in all three subject, predicate and object columns, even if the whole input is not sorted on the join key of a subsequent join, large portions of the input can still be sorted as it is demonstrated in the following example.

Example 5. *Consider the following SPARQL query:*

```
SELECT ?x ?y
WHERE {
    ?x prop1 ?y .
    ?x prop2 ?z .
    ?z prop3 ?w .   }
```

If the selected join order is as shown in text of the query, S-O tables will be used for all properties. As shards of prop1 are scanned, for each thread of execution, $prop2$ *will be probed for values sorted on* $?x$*, but for the second join, probing* $prop3$ *will not in general be sorted on* $?z$*. Nevertheless, for each distinct* $?x$*,* $prop3$ *probing will still be sorted on* $?z$ *and if each subject of* $prop3$ *is connected to many objects, it may be more efficient to avoid binary search on* $prop3$ *and switch to scanning for each distinct* $?x$*.*

A single join operator has been implemented in our system, that adaptively during run-time, for each search key, decides if it will switch to binary search (a behavior similar to index-based nested loops) or keep scanning the input in the form of sequential search, continuing from the position that the cursor has been left from a previous search (a behavior similar to merge join).

5.3.1 Adaptive Join Processing

Given a left-deep join tree produced from the optimizer, each worker starts scanning a shard of the first relation, or a specific shard of an object/subject vector of the first S-O/O-S relation in case a filter exists, and searching the subsequent relations for each produced

tuple. The search procedure is presented in Algorithm 5. The algorithm takes as input a pointer to current cursor position ($cursor\_position$), which corresponds to the position of the last accessed element for the array, and decides if it will use binary or sequential search. The $cursor\_position$ is updated each time for both successful and unsuccessful searches inside the functions $Sequential\_Search$ and $Binary\_Search$.

Obtaining an exact cost-model in order to take the correct decision is an involved process that needs to take into consideration factors such as the exact cache hierarchy, the size and bandwidth estimation for each cache level for both sequential access (scanning) and random access, cache line size, the replacement of cache entries from operations other than the join under consideration (for example subsequent joins of the same query) and the existence in cache of relevant entries from previous operations (for example scanning of the same relation in a previous query). Obtaining such cost models for hierarchical memory systems has been studied in [65], where cost functions are defined for basic access patterns and then combinations of these functions can be used to derive the cost of complex compound access patterns. As a prerequisite, specific hardware measurements should be known, which can be obtained through a separate calibration program that estimates cache and CPU characteristics.

In our case, decision has to be made during runtime for each produced tuple and each join of the query. Instead of using an analytical cost model, we opt for a fast and lightweight method using two assumptions: a uniform distribution of integers in the first array of each table and that existing cache contents have an impact proportional to the cost of either binary search or scanning. The second assumption simply denotes that existing cache contents can improve both methods, but they will not change which the methods is more efficient in each case. For example, if binary search is preferable with completely empty cache, it will remain so independently of the cache contents and vice versa. As a result we base our decision on the difference between the last accessed element and the element that we are currently searching for. Specifically, we pass as argument to the algorithm a threshold which is computed during data loading for each table. This threshold takes into consideration an estimation about the maximum distance of the position of the last accessed element and the position of the element to be found in the array, in order for sequential search to be preferable. To switch from distance in the array to the actual arithmetic distance of the two numbers, we use the uniform distribution assumption, which leads to an estimation that the difference between an element and its subsequent one is $(array[size-1]-array[0])/size$. Note that in Algorithm 5, if $Distance > Threshold$ then we could perform binary search using as starting position the position denoted by $CursorPosition$ instead of 0, and if $Distance < -Threshold$ we could use $CursorPosition$ as the end position instead of $size$. In theory this reduces the steps needed from binary search, but in practice it is not efficient, as always performing binary search on the whole array leads to the array positions visited during the first steps to frequently occur in cache.

Regarding the determination of the threshold, a calibration process shown in Algorithm 6 is used. This process takes place after data loading, prior to query execution, and tries to determine a distance (called $WindowSize$) such that when searching for a value $ToFind$ in the $Array$ and the position of $ToFind$ is at distance $WindowSize$ from the position of

the last accessed element ($CursorPosition$), then $BinarySearch$ and $SequentialSearch$ perform roughly the same. Specifically, the ratio of the larger to the smaller execution times of these two methods should be smaller than a value close to 1.0 which is specified in the input of the algorithm ($Threshold$). For each calibration step, each process is called $NoOfSearches$ times, each time searching for a value whose distance from the previous one is estimated to be equal to $CurrentWindowSize$. If the ratio is larger than the $Threshold$, calibration continues such that the window size is multiplied by this ratio (in case time spent on binary search is larger) or divided (otherwise). This calibration process is different from a calibration needed when using an analytical cost model, in the sense that we directly make an estimation for a value related to processing, instead of estimating values about several hardware characteristics. Once the calibration process terminates, we precompute the estimated value distance (corresponding to the position distance that we obtained) for each property, such that during query execution we only need to perform one integer subtraction, one absolute value computation and one comparison for each tuple (lines 2-3 of Algorithm 5).

Algorithm 5: Adaptively switching between binary and sequential search

```
1 Search (Array, Value, CursorPosition, Threshold, Size);
  Input  : Array: an array of integers (subjects of an S-O table or objects of an O-S table), Value:
           integer value to find, CursorPosition:pointer to current cursor position, Threshold: integer,
           Size: size of array
  Output: nonnegative integer corresponding to the position of Value in Array or a negative integer if
          Value is not present in the Array
  Uses   : Binary_Search(Array, Value, CursorPosition, Size), Sequential_Search(Array, Value,
           CursorPosition, Size)
2 Distance := Array[CursorPosition] − Value;
3 if |Distance| <= Threshold then
4 |   return Sequential_Search(Array, Value, CursorPosition, Size);
5 else
6 |   return Binary_Search(Array, Value, CursorPosition, Size);
7 end
```

5.3.2 ID-to-Position Index

Our join method takes advantage of initial sorting and performs cache-friendly joins even when only a partial order of input triples is possible, but when ordering does not help we must resort to binary search. In this section we describe the structure of an ID-to-Position index that is used to avoid binary search and directly locate the position of a given integer on the property array. A separate such ID-to-Position index must be built for each S-O or O-S table, but its usage is auxiliary, in the sense that our system can operate without all or some of these indexes. Given an RDF dataset with N distinct values and a corresponding dictionary with IDs from 1 to N, in order to directly locate the position of a given value in a table, we need to store an integer array of length N, such that the value at index p denotes the exact position at the table where it is located the resource whose ID value according

Algorithm 6: Calibration Process

```
1  Calibrate (Array, NoOfSearches,
     StartingWindowSize, Threshold);
   Input  : Array: an array of integers (subjects of an S-O table or objects of an O-S table),
            NoOfSearches: number of times to run sequential and binary search in each calibration step,
            StartingWindowSize: initial window size used in first step of calibration, Threshold: A
            threshold ratio to stop calibration
   Output: integer corresponding to the window size such that if two values in array are longer apart then
            binary search is preferable
2  NextWindowSize = StartingWindowSize;
3  AvgGap = (Array[Size − 1] − Array[0])/Size;
4  do
5      WindowSize = NextWindowSize;
6      TotalGap = AvgGap ∗ WindowSize;
7      PreviousSearchPosition = 0;
8      StartTime = getTimeNow();
9      ToFind = Array[0];
10     for K ← 0 to NoOfSearches do
11         Binary_Search(Array, ToFind, 0, &PreviousSearchPosition);
12         ToFind+ = TotalGap;
13     end
14     TimeBinary = getTimeNow() − StartTime;
15     toFind = Array[0];
16     PreviousSearchPosition = 0;
17     StartTime = getTimeNow();
18     for k ← 0 to noOfSearches do
19         Sequential_Search(array, toFind, &PreviousSearchPosition);
20         ToFind+ = TotalGap;
21     end
22     TimeScan = getTimeNow() − StartTime;
23     TimeDiff = |TimeBinary − TimeScan|;
24     if TimeBinary > TimeScan then
25         Fraction = TimeBinary/TimeScan;
26         NextWindowSize = WindowSize ∗ Fraction;
27     else
28         Fraction = TimeScan/TimeBinary;
29         NextWindowSize = WindowSize/Fraction;
30     end
31 while Fraction > Threshold;
32 return WindowSize;
```

to the dictionary is p, or a special value to denote absence of the specific resource from the table.

For example, given the property shown in Figure 5.1 and supposing that the maximum ID contained in the dictionary is 45, we would need an array of integers with length 45, such that at position 5 of the array we would have the value 0, at position 7 the value 1, at position 13 the value 2 and so on for positions $18, 24, 29, 33$ and 45, and all other position of the array would have a value denoting absence. If we use M-byte integers, then for each table the memory requirement would be $M * N$ bytes. In order to save space, we use a different layout on out ID-to-Position index, such that we only use an integer to denote the position of the property table at specific intervals, and for all other positions we use a bit value to simply denote presence or absence of value from the property table. Finding the exact position for a value requires reading the previous integer and then counting bits set to 1 up to the position of the ID-to-Position Index corresponding to the value. For example, if we choose the interval to be equal to 8, then our index will store the integer -1 at start, followed by bit values $0, 0, 0, 0, 1, 0, 1, 0$, then integer value 1 and bit values $0, 0, 0, 0, 1, 0, 0, 0$, then integer value 2 and bit values $0, 1, 0, 0, 0, 0, 0, 1$, then integer value 4 and bit values $0, 0, 0, 0, 1, 0, 0, 0$, then integer value 5 and bit values $1, 0, 0, 0, 0, 0, 0, 0$ and finally integer value 6 and bit values $0, 0, 0, 0, 1$. If we want to find the position of value 29 at the property we can directly check bit at position $((29 \div 8) + 1) * M * 8 + 29$. If bit is not set, then value is not present in property table. If bit is set we read integer value that starts at bit position $(29 \div 8) * M * 8 + (29 \div 8) * 8$ at the array and we add to this the number of bits that are set after this number for $29 \ mod \ 8$ positions. With this layout, given an interval A we only need $N/8 + ((N/A) * M)$ bytes. Also, given that the integer and the number of bits followed up to the next integer fit into a single cache line (with proper alignment of the index in the memory), we only need one memory access and some computation that can be done efficiently as a popcount operation in order to determine the position.

As an example, using the dataset LUBM 10240 described in Section 5.5, which contains about 1.4 billion triples, 17 distinct properties and about 336 million distinct resources, using 4-byte integers and choosing the interval to be 480 we only need 44.8 MB for each property, leading to a total memory usage of about 1.5 GB if we choose to create all possible indexes for S-O and O-S tables, in contrast to a memory requirement of 45.7 GB if we had used the simple layout.

Regarding modification of the join processing in case the ID-to-Position index is used, the only change that needs to be addressed is a different threshold resulted from calibration process. Specifically, since we anticipate that using the index will have better behavior in comparison with binary search, we need to estimate two different thresholds with regards as to when sequential search is preferable, with the threshold when ID-to-Position index is used being smaller than the threshold when binary search is used.

5.4 Query Execution Over Ontological Hierarchies

We begin this section by describing the system design and general architecture of our approach which is based on [82]. Essentially, it modifies the mentioned system by removing the external RDBMS that stores RDF data with the semantic index schema, and replacing it with an extension of PARJ, able to provide access to virtually complete ontological hierarchies in an efficient manner.

5.4.1 System Design

Three main reasons for the presence of long, and as a result highly inefficient, rewritings have been specified in [82]: i) subqueries with existentially quantified variables, ii) large ontological hierarchies and iii) multiple mappings for each ontology term. The last reason is usually relevant when arbitrary relational schemas are used as the target storage. In our case, where we have a specialized triple store as a back-end, a single trivial mapping has to be created for each predicate, with the exception of the rdf:type property, where a separate mapping has to be created for each distinct object, but this does not create problems as we will see later. Regarding the first reason, it is observed that exponential number of rewritings due to existential quantification seems to be rarely observed in real world ontologies and queries, leaving the second reason to be the most commonly encountered, as it is demonstrated in the following example. Consider the following OWL statements:

```
AssistantProfessor rdfs:subClassOf Professor
FullProfessor rdfs:subClassOf Professor
teaches rdfs:domain Professor
hasBScDegreeFrom rdfs:subPropertyOf degreeFrom
hasMScDegreeFrom rdfs:subPropertyOf degreeFrom
hasDoctoralDegreeFrom rdfs:subPropertyOf degreeFrom
FullProfessor rdf:subClassOf _R1
_R1 owl:onProperty hasDoctoralDegreeFrom
_R1 owl:someValuesFrom University
```

The first (resp. second) OWL statement declares that if an individual is an assistant professor (resp. full professor), then he is a professor. The third statement declares that if an individual teaches something, then he is a professor. The next three statements declare that if an individual has a Bsc, Msc or doctoral degree from a given entity, then this individual has (in general) a degree from this entity. Finally, the last three statements encode the knowledge that if an individual is a full professor, then there exists a university such that this individual has a doctoral degree from this university. In DL parlance this is denoted as: $FullProfessor \sqsubseteq \exists hasDoctoralDegree.University$. The first three statements correspond to three axioms defining a class hierarchy, the next three statements correspond to three axioms defining a property hierarchy, whereas the last three statements

correspond to an axiom involving existential quantifier. Following [82], the main observation upon which we base the architecture of our approach is that if the underlying data are (virtually or actually) complete with respect to class and property hierarchies, then during query rewriting all ontology axioms related to these hierarchies (in our example the first six statements) can be ignored, and we can use any rewriting method (tree-witness rewriting [52] is used in our case) for OWL2 QL ontologies to perform query rewriting with respect to the remaining axioms. Then, the produced rewriting is transformed to the query language of the underlying query execution engine using R2RML [1] mappings.

As Ontop is designed to work with R2RML mappings to an external relational database, we provide to it an abstraction of PARJ consisting of relational tables according to the vertical partitioning schema as presented in Section 5.2. In other words, we present each property as a relational table with two columns (subject and object) and provide SQL mappings over these tables, whereas in reality the underlying storage schema is the one described in Section 5.2. During system startup we automatically create the following mappings, without any manual user intervention, based on the data stored in PARJ:

- For each distinct property P defined in the ontology we add the mapping $P(x, y) \leftarrow$

  ```
  SELECT d1.value as x, d2.value as y
  FROM propI, dictionary d1, dictionary d2
  WHERE propI.subject=d1.id and
  propI.object=d2.id
  ```

  where $propI$ is the table corresponding to property P

- For each distinct named class C in the ontology we add the mapping $C(x) \leftarrow$

  ```
  SELECT d.value as x
  FROM propT, dictionary d
  WHERE propT.object=N and
  propT.subject=T.id
  ```

  where $propT$ is the table corresponding to the $rdf : type$ property and N is the integer value that corresponds to class C. During startup we prefetch these values for all the named classes of the ontology.

In the abstraction of the PARJ schema that it is provided to Ontop, there is no distinction between the S-O and O-S table replicas. The exact access methods will be decided internally by PARJ after the final rewriting has been produced. Also, as the id column of the $dictionary$ table is unique, it can be considered a primary key, with both columns $subject$

[1] https://www.w3.org/TR/r2rml/

```
Professor(x) ← SELECT subject as x
         FROM PropT where object=t1
/*t1 corresponds to Professor*/
Professor(x) ← SELECT subject as x
         FROM PropT where object=t2
/*t2 corresponds to FullProfessor*/
Professor(x) ← SELECT subject as x
         FROM PropT where object=t3
/*t3 corresponds to AssistantProfessor*/
Professor(x) ← SELECT DISTINCT subject as x
         FROM PropN
/*table PropN corresponds to property teaches*/
```

Figure 5.2: $\mathcal{T}$-Mappings entry for class Professor

and $object$ of all property tables to be foreign key referencing this primary key. As a result, when joining subject or object values obtained from different mappings in a query, joins can be performed directly on IDs instead of URIs, and only use the dictionary tables for lookups on the IDs that exist in the SELECT clause of the final query. For ease of presentation, in what follows we omit explicit references to the dictionary table. After these initial mappings have been declared, Ontop compiles knowledge about ontological hierarchies into the mappings in order to obtain the so-called $\mathcal{T}$-Mappings. Given the ontological axioms from the previous example, for the class $Professor$ three extra mappings will be added to the $\mathcal{T}$-Mappings, one containing as body the SQL query corresponding to the initial mapping for $AssistantProfessor$, one for the class $FullProfessor$ and one for the property $Teaches$, leading to four mappings in total, including the initial mapping for class $Professor$, as shown in Figure 5.2. Essentially, in order to obtain all professors from the data according to the property hierarchy, one should take the union of the four SQL queries in the body of the mappings (including duplicate elimination in the final result). In this case, for the first three mappings, two OR conditions could be introduced in the $WHERE$ clause, instead of union.

5.4.2 Union Wrappers for Ontology Hierarchies

As an end-to-end example regarding query rewriting, consider the following query from Example 6 over the ontology, asking for individuals x that work for an entity u and these individuals are professors and have a degree from the same entity u:

Example 6. *Consider the following query, that contains an extra filter:*

```
SELECT ?x ?u
WHERE {
?x worksFor ?u.
?x rdf:type Professor .
?x degreeFrom ?u.  }
```

First, the query will be rewritten with respect to axioms not involving hierarchies. In our case the result of this process will be the query unchanged, as the relevant axiom invovling existential quantifier is not applicable. Then, the query will be unfolded with respect to the $\mathcal{T}$-Mappings. One way to obtain the final query is to perform the union of each query atom first, and then perform the joins on these intermediate results. On the other hand, one could choose to flatten the queries by pushing the joins inside the unions. In our example, with four mappings for $Professor$, four mappings for $degreeFrom$ and one mapping for $worksFor$, this would result to a union of 16 queries. Finally one could follow a hybrid approach, based on some cost estimation. When a relational back-end stores the data, all these approaches can be highly inneficient. In the first case, a separate union with duplicate elimination should be performed for each atom with more than one mapping, and temporary results should be created and possibly indexed in order to perform subsequent joins efficiently. On the other hand, in the second approach the number of subqueries can be very large, with common tasks (for example a table scan) performed many times. Furthermore, duplicate answers coming from different subqueries lead to redundant processing and also a duplicate elimination must be performed to the final result. Having PARJ as the back-end, we chose to follow the first approach, having implemented a virtual union wrapper operator that eliminates the mentioned disadvantages, and at the same time it keeps the advantages of the standard PARJ design. Specifically out approach: i) performs pipelined union along with duplicate elimination ii) uses the original index (sort order) and adaptively decides for the access method iii) provides results in sorted order for subsequent operators as much as possible and iv) can be parallelized efficiently.

The main idea is to create a *union wrapper* for each class or property hierarchy encountered in a query. This wrapper acts as a virtual table that contains the complete answers for the given predicate. As an example, the union wrapper for the query atom $Professor$ is shown in Figure 5.3. This wrapper, as all wrappers that correspond to a class hierarchy, defines a virtual table with only one column and it provides two operations: i) scan the virtual table and return an iterator with the ordered distinct values that it contains and ii) search for a specific integer value and return it if it is contained in the virtual table, otherwise return an empty answer. The first operation is used when the union wrapper is the leftmost table in query execution according to the join order, whereas the second operation is used in all other cases. In our example the union wrapper contains four input vectors: the subject vectors of the O-S replica of the rdf:type property for the objects that correspond to values Professor, AssistantProfessor and FullProfessor and the subjects for the S-O replica of property $teaches$. We use the S-O replica of $teaches$ that contains the distinct subjects, as $Professor$ is defined to be the domain of the $teaches$ property. If it was the range of the property, the distinct objects array of the O-S replica would be used instead. During scan, all input vectors are scanned, and current value for each one is sent to a min heap implemented with a priority queue, that sorts them and outputs the minimum value. Only the first time a value is encountered it is sent to output, in order to have distinct results. When the minimum value of the min heap is being sent to output (or discarded) the input vector that provided that value sends the next one. In the case of search, all input vectors are searched using the adaptive search algorithm from the previous section. Search stops as soon as the value is found to at least one of the input vectors and the

given value is sent to output terminating the operation. In case of search, an alternative can be more efficient depending on the number of input vectors that correspond in the rdf:type property. For the specific example, the first three vectors can be replaced by the object vector of the S-O replica of the rdf:type property for the subject that corresponds to the value we are searching for and perform set intersection between this vector and a vector containing the values corresponding to Professor, AssistantProfessor and Full-Professor. This way we can avoid three different searches and only perform one search in the form of merge join (assuming that we first sort these three values) that terminates upon outputting the first result.

In case of the union wrappers for property hierarchies, such as the degreeFrom, the virtual table contains two columns and are three different operations: i)scan the whole table and provide an ouptut sorted on both columns, ii)search for a specific value in the first column and provide the output corresponding to this value sorted on the second column and iii) search for a specific pair of values. In the first case, we use a min heap with two input values sorting output to both, in the second case we search the input tables for the specific value in the first column and we use a min heap with one value, and in the last case we search the input tables for both values and stop the operation upon finding the first such result.

Regarding intra-query parallelism, incorporating the union wrappers does not require modifications to the approach described in Section 5.2, with the exception of scanning a union wrapper table in case it is the leftmost table according to join order. In this case, special effort is needed in order to ensure that the same values residing in different input tables will be produced from the same thread, so that duplicate elimination in the min heap will work properly, otherwise duplicate answers may be present in results of different threads. As a solution, we modify the way we assign a different portion of the first array of each input table to each thread. Instead of using positions in the array, we assign specific value intervals in each thread, based on the minimum and maximum values encountered in all input tables.

5.4.3 Join Ordering and Selectivity Estimation

As in RDF-3X and TriAD, we employ a bottom-up dynamic programming optimizer. As the level of parallelism during execution is determined by the number of threads, we assume that the benefit of each possible join order from parallelism will be a fixed proportion of its centralized cost, that is the execution cost if we consider that each property is consisting of a single shard. As a result of this assumption, we disregard parallelism during optimization. During cost estimation, we assume that a specific choice will be followed for all tuples of a join, either binary search or scanning. The latter will only take place when the join inputs are already fully sorted and it is estimated to be cheaper than binary search. Adaptivity during execution is expected to give a cost equal or lower to this estimation. For each property of a specific join order we choose to use the replica that leads to more efficient search. When we search for a specific subject or object, the choice is straightforward. When we are searching for both values, we use the replica that searches for ordered

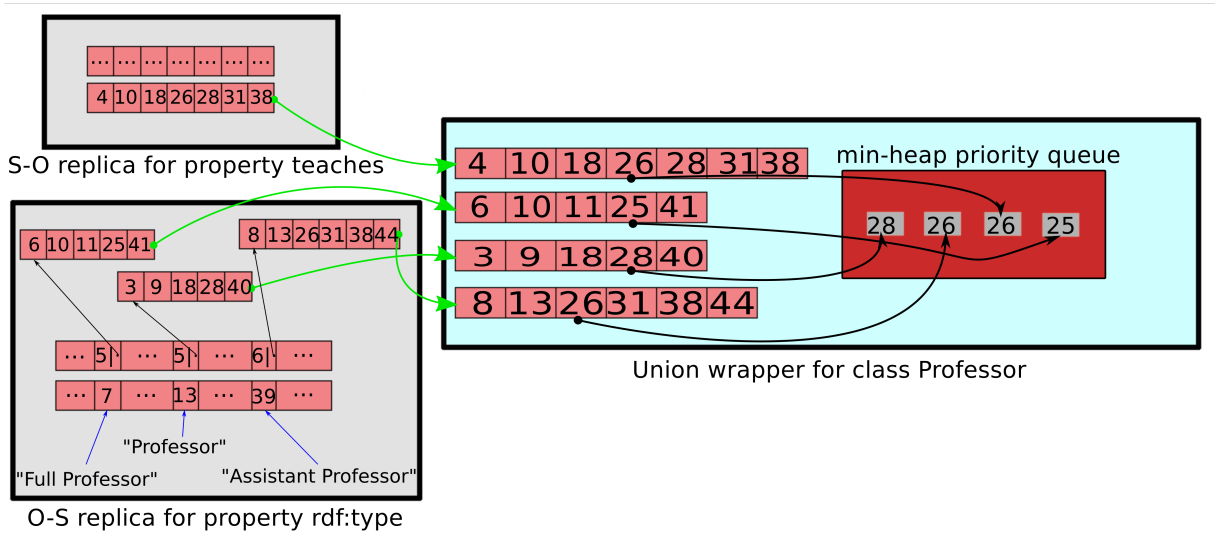


Figure 5.3: Union Wrapper for Class Professor

values according to the previous join. When we scan a table (leftmost table in the join order) we use the replica tha provides values sorted for the subsequent join. In union wrappers for property hierarchies, for some of the input tables we should use the S-O replicas and for others the O-S replicas. This decision is defined by the combination of two factors. First, whether we search for the subject or object of the given property and second, if the inverse of a subproperty is used in the definition of the hierarchy.

Regarding selectivity estimation, in order to estimate the sizes of intermediate results we use equi-depth histograms for each property. Such histograms are also built for the union wrappers, based on information about the hierarchies that can be found in the ontology. The reason for this is that the exact size of tuples in a union wrapper can be very difficult to estimate, as the common tuples in the different input tables that introduce duplicate results are discarded. Consequently, the final size can vary significantly and this can lead to very poor execution plans. In the two extremes, the final result size can be the sum of all input tables (no duplicates at all), or can be equal to the size of the larger input table. For this reason we take a sample and estimate the number of distinct tuples that will be produced from each union wrapper. These histograms are built after the initial data loading, and similar to the analyze command in RDBMSs they do not need to be recomputed, unless the underlying data are subject to significant change. As it is known that often estimates based on such histograms may not be accurate especially in the case of RDF data [70], we precompute some cardinalities between pairs of properties during data loading and use these as a corrective step. Specifically, for each pair of properties $prop_i$ and $prop_j$ we compute the cardinality of $prop_i \bowtie_{subject=subject} prop_j, prop_i \bowtie_{subject=object}$

$prop_j, prop_i \bowtie_{object=subject} prop_j$ and $prop_i \bowtie_{object=object} prop_j$. All these computations can be done in parallel and also, using our storage schema, we do not need to access the second array of storage at all, as we only need the number of objects for joins on subjects and the number of subjects for joins on objects. We plan to implement more elaborate techniques for cardinality estimation in the future, like for example estimations based on characteristic sets [69] or RDF data summaries [97].

5.5 Experiments

In this section we present an experimental evaluation of our approach. One first objective is to compare both versions our system (PARJ and PARJ-Ontop) with other centralized and distributed state of the art systems of similar functionality, in terms of query execution time. Furthermore, we aim to investigate the scalability of PARJ by performing experiments with varying number of datasets and threads, and also examine the effect of the ID-to-Position index during query execution. Finally, we want to empirically evaluate the effect of the adaptive query processing method in comparison with standard binary and sequential search

In-memory data storage and query processing for PARJ have been implemented in C as an extension of a SQLite, which is used as disk-based storage. Disk-based tables are created and saved during data import from RDF files. On application start-up the in-memory data structures are created reading from the tables. The dictionary can either be loaded in memory or kept in disk where for IRI-to-ID transformation (during query optimization) a clustered B+ tree on IRI is used and for ID-to-IRI transformation (during IRI construction of answer tuples) a clustered B+ tree on id. PARJ is called through a wrapper written in Java, where also query parsing and optimization is implemented.

All experiments were conducted on a 16-core server with Intel E5-4603 processors at 2.20 GHz and 128 GB RAM running Debian 8. We used the popular Lehigh University Benchmark (LUBM) [38] and Waterloo SPARQL Diversity Test Suite (WatDiv) [6] benchmarks, as well as the real-world YAGO dataset [40] which contains data from Wikipedia, WordNet and GeoNames. Our implementation of PARJ is publicly available and open-source, and all material required to reproduce the experiments is available online [2].

5.5.1 Setup

We have carried out experiments with both the stand-alone version of PARJ, which is not capable of reasoning, and also with our PARJ-Ontop implementation which is able to perform OWL2-QL query answering. Results for the latter version are presented in section 5.5.2.4. Regarding the stand-alone version, we compare PARJ to other systems that do not support OWL2-QL reasoning in two sets of experiments: in the first one we test the efficiency of our approach in the single-thread setting. In this setup we use as

[2] https://github.com/dbilid/experiments

competitors the in-memory RDF store RDFox (SVN version: 2776) and also RDF-3X [70] (version 0.3.8) for comparison with a state of the art disk-based system. The second setup is about multi-threaded execution. In the second setup we use as competitor the TriAD system which in [39] it is shown to outperform all competitors in the centralized parallel setting. We have used the optimized build for TriAD, as it is suggested in the installation manual.

Due to a hard-coded limit in the TriAD source code, we could not execute queries using more than 20 workers[3]. Note that in PARJ, each worker corresponds exactly to one thread, so given that hyper-threading is enabled, we found that the optimal performance was achieved when we used two threads for each processing core, resulting in 32 workers/threads in our testing machine. More details regarding the behavior of PARJ for different number of threads are given in Section 5.5.2.3. For TriAD it was not clear which number of workers should be the optimal, as this could be query depended. This is also the reason that we do not use TriAD in the single-thread setting. To have a better image and find the optimal setup, we executed TriAD with different number of workers, and we also modified the hard-coded limit and tried with up to 32 workers. For most queries, TriAD performance is degrading for more than 20 workers. From our testing we found that the overall best performance was achieved for 16 workers and this is the setup we used for TriAD in our experiments. Also, we present results for both TriAD settings: with summary mode enabled and disabled. For summary mode, we used the same number of partitions used in [39]: 200K for LUBM 10240 and 70K for WatDiv 1000.

Regarding result handling, as our intention is to concentrate in join processing, all systems were tested in the so called ”silent“ mode, that is we do not include the time for dictionary lookups and result tuple construction. In multi threaded execution this also means that we do not measure the time to aggregate the results together. Each query was executed 10 times and the average execution time is shown. We have deployed RDF-3X using an in-memory filesystem and as a result there is no need to report cold and warm cache times.

5.5.2 Results

We present results for scale 10240 of the LUBM benchmark in Table 5.2 (about 1.4 billion triples), for YAGO2 in Table 5.3 and for scale 1000 of the WatDiv benchmark (about 110 million triples). For WatDiv we used both basic test workload (Table 5.4) and incremental linear and mixed linear extensions of basic workload (Table 5.5). For WatDiv we generated all the queries proposed in the workloads. For LUBM we used the seven queries commonly used to test systems that do not perform reasoning tasks, which can be found in [107], and are labeled LUBM1-LUBM7, and we also used three extra queries from [78] (LUBM8-LUBM10). A timeout of 30 minutes was used for all queries. For YAGO2 we used the same raw data (about 285 million triples) and queries as in [2].

Regarding single thread execution, we first observe that RDFox is comparable to PARJ

[3]This was verified with the TriAD implementors

for some queries, but for other queries, especially for queries from the WatDiv incremental and mixed linear extensions, is highly inefficient. This confirms that this system is not optimized for query answering, but instead, it aims at efficient parallel materialization of RDF implications. Regarding RDF-3X, we can see that it performs more than one order of magnitude slower from PARJ for most queries. The reason is that despite the fact that it is deployed in an in-memory filesystem, its processing is oriented towards optimizing disk access, as it is not aware that it operates in memory. For example, it uses B+ trees to minimize the number of disk pages needed, it skips records with its sideways-information passing optimization only when it reads a new disk-page into memory, it uses compression on a per page basis and also its cost estimation is based on disk access. Nevertheless, there are some queries, for example queries in the ML-2 set or LUBM8, where RDF-3X outperforms the single-threaded PARJ execution. These are queries with large intermediate results, but only few final answers, where the record skipping using sideways information passing in RDF-3X results in substantial gains.

Regarding multi-thread execution we can see that for most queries the summary mode of TriAD is inferior to the simple mode, sometimes by a large margin. For example, for query LUBM 3 in Table 5.2 the execution time increases from 2 seconds to more than 15 seconds. For the specific query we saw that execution over the summary graph takes up most of the execution time. In any case, the results show that for parallel execution on a centralized environment the pruning from the graph summaries does not contribute to an important improvement which can justify the overhead of graph partitioning.

A comparison of PARJ with the best TriAD mode shows that we outperform TriAD by more than an order of magnitude for the average execution time of the LUBM 10240 queries: from 838 milliseconds for PARJ to 13263 for TriAD (Table 5.2). For basic WatDiv testing (Table 5.4), though TriAD performs slightly better for simple queries, PARJ performs better overall with a total average execution time of 11.27 ms (geomean: 7.76) whereas TriAD has a total average execution time of 13.95 (geomean: 6.8). For the more complex queries of WatDiv extended workloads (Table 5.5) PARJ clearly outperforms TriAD. For some queries the difference is more than two orders of magnitude. As an example, for query ML1-7 the time increases from 7 ms to 2154. The specific query contains a series of subject-object joins, which leads TriAD to perform blocking data transfers between workers and rehashing over large intermediate results, though the final result is relatively small.

Regarding the difference between the silent mode and the full result handling, we have executed all queries with full result handling (except from printing) in PARJ. That is we include answer tuple construction, dictionary lookups and sending all results to the coordinating thread. We do not include these results, as we saw that for most queries, usually with results up to a few thousand tuples, the difference is not important, but for queries with many million results the difference can be significant. This can be seen especially for query 2 from the LUBM benchmark (about 10M results) where execution time in multi threaded execution increases from 151 milliseconds in silent mode to 610 milliseconds in full result handling. The same holds for queries C3 (about 4.3M results) and IL-3-5 to IL-3-10 from WatDiv which have more than 50M results. Query IL-3-8 has by far the largest number of results (about 1.6 billion tuples with 9 columns). This is the reason that TriAD

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
LUBM1	15369	96677	1329510	800	4188	4467
LUBM2	2437	40368	21870	151	965	1101
LUBM3	5338	136554	23179	605	2004	15243
LUBM4	5	1	8	10	12	5
LUBM5	1	1	6	4	2	2
LUBM6	3	3	190	5	95	5
LUBM7	9213	31180	68769	473	13400	14125
LUBM8	9899	44144	6485	1336	2838	3906
LUBM9	58082	187192	208839	4014	42932	32982
LUBM10	14606	26690	51235	982	65925	41510
Avg	11495	56281	171009	838	13263	11334
Geomean	864	2536	5581	180	1071	881

Table 5.2: Results for LUBM 10240 (times in ms)

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
Y1	9	56	102	11	12	8
Y2	11	2390	380528	13	830	1381
Y3	165	1409	2915	20	280	137
Y4	5	20	110	10	9	3
Avg	48	969	95914	14	283	382
Geomean	17	248	1878	13	71	46

Table 5.3: Results for YAGO2 (times in ms)

runs out of memory for the specific query, since even in silent mode, each worker keeps in memory all the results instead of using an iterator to send the results to the master (or discard the results in silent mode) as they are produced, as it is the approach used by PARJ. Execution times for the full result handling mode of PARJ are included in the online material to reproduce experiments.

5.5.2.1 Effect of Runtime Join Optimization

In order to examine the effect of our adaptive join method, we have executed the queries of both datasets using four different strategies as shown in Table 5.6. For WatDiv benchmark we only report the average and geometric mean of all execution times. In the first (Binary) column we report the execution times when we always use binary search. In the second column (AdBinary) we use our adaptive join method in order to switch from binary to sequential search. In third column (Index) we always use the ID-to-Position index, whereas in the last column (AdIndex) we use the adaptive join method in order to switch from ID-to-position index to sequential search. One can observe that the impact of the adaptive join method is more important when binary search is employed (comparison of first and sec-

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
L1	5	5	40	10	3	5
L2	8	43	30	5	5	6
L3	2	244	13	4	2	3
L4	3	7	19	4	2	8
L5	9	57	40	6	3	46
Avg	5	71	28	6	3	14
Geomean	5	29	26	5	3	8
S1	49	1209	18	47	34	116
S2	3	284	27	3	4	17
S3	4	17	7	3	2	18
S4	4	153	10	5	5	29
S5	4	1*	14	4	4	20
S6	1	5	8	5	2	3
S7	1	695	7	5	2	3
Avg	9	338*	13	10	8	29
Geomean	4	61*	12	6	4	15
F1	5	24	15	6	5	19
F2	12	153	27	10	37	13
F3	3	59	73	9	29	74
F4	56	249	83	19	9	66
F5	3	10	108	7	40	58
Avg	16	99	61	10	24	46
Geomean	8	56	48	9	18	37
C1	21	50	140	12	39	598
C2	76	178	441	16	40	1574
C3	266	4810	127	45	43**	527**
Avg	121	1679	236	24	41**	900**
Geomean	75	350	199	21	41**	792**

* RDFox returns an empty result-set for query S5, whereas the correct answer is not empty.

** TriAD returns only distinct answers for query C3, even though modifier DISTINCT is not present in the SPARQL query. The number of results returned is only 8162 instead of 4335801.

Table 5.4: Results for WatDiv Basic Workload scale 1000 (times in ms)

	Single Thread			Multi-Thread		
	PARJ	RDFox	RDF-3X	PARJ-32	TriAD	TriAD-SG 200K
IL-1 5	3	27617	1339	5	584	5082
IL-1 6	4	204898	1832	4	1482	11814
IL-1 7	8	669099	1272	7	1862	14950
IL-1 8	3	700199	1633	5	1615	21238
IL-1 9	26	728518	1396	11	630	23844
IL-1 10	29	734363	1923	9	618	25752
Avg	12	510782	1566	7	1132	17113
Geomean	8	335194	1546	6	1002	15068
IL-2 5	2	6574	1525	6	476	5340
IL-2 6	5	62149	2046	4	952	11156
IL-2 7	2	78211	1794	3	344	58749
IL-2 8	4	80453	1865	16	1148	62448
IL-2 9	9	86995	1998	6	1062	67045
IL-2 10	4	87872	1867	5	1093	70658
Avg	4	67042	1849	7	846	45899
Geomean	4	51948	1841	6	770	31807
IL-3 5	13259	187101	542948	1494	11195	17093
IL-3 6	58379	397964	357310	7070	13603	25492
IL-3 7	23208	342533	Timeout	1192	1809	23492
IL-3 8	71918	1214564	Timeout	4903	Out Of Memory	Out Of Memory
IL-3 9	26437	966919	Timeout	2082	7182	39462
IL-3 10	41867	951513	175247	1882	8118	46593
Avg	39178	676766		3104		
Geomean	33565	552681		2496		
ML-1 5	2	11481	163	2	56	374
ML-1 6	2	2	83	2	33	1152
ML-1 7	1	1	728	7	2154	4646
ML-1 8	2	1	824	4	103	2018
ML-1 9	5	98058	994	4	198	11766
ML-1 10	4	14111	1482	3	930	9841
Avg	3	20609	712	4	579	4966
Geomean	2	178	478	3	206	2786
ML-2 5	3175	1136335	936	201	413	1849
ML-2 6	2	12182	166	5	92	1041
ML-2 7	121	27151	678	15	296	895
ML-2 8	69	818424	2863	19	1996	24500
ML-2 9	4335	919541	282	259	330	1587
ML-2 10	52	849283	1952	9	728	32449
Avg	1292	627153	1146	85	643	10387
Geomean	151	249327	741	30	419	3599

Table 5.5: Results for WatDiv Incremental and Mixed Linear Workloads scale 1000 (times in ms)

Query	Binary	AdBinary	Index	AdIndex
LUBM1	22186	15454	16557	15369
LUBM2	2877	2443	2535	2437
LUBM3	6562	5491	6415	5338
LUBM4	5	7	7	5
LUBM5	1	1	1	1
LUBM6	2	2	2	3
LUBM7	12246	11866	9197	9213
LUBM8	15725	9782	10420	9899
LUBM9	77468	63586	58171	58082
LUBM10	22359	14892	16217	14606
Avg	15943	12352	11952	11495
Geomean	1034	892	898	864
Watdiv1000 Avg	8439	8003	5013	4869
WatDiv 1000 Geomean	33	28	25	23

Table 5.6: Impact of Adaptive Processing for LUBM 10240 and WatDiv 1000 (times in ms) for 1 thread

ond column), whereas when the ID-to-Position index is used (comparison between third and fourth column) its contribution to better performance is smaller. This is in line with the result of our calibration method, where when binary search is used, the result threshold is about 200 positions, whereas when ID-to-Position index is used the threshold is about 20 positions. Also, it seems that the impact is more important for LUBM queries, where in case of binary search it leads to a decrease of 23% in average execution time. The reason for that is that the average execution time for WatDiv queries is heavily affected by the IL-3 queries, where the impact of the adaptive method is not important, as sequential search can rarely be used in these queries. This is also the reason for the great reduction in average execution time of WatDiv queries when the ID-to-Position index is used, as the aforementioned queries are greatly profit from the index.

5.5.2.2 Effect of ID-to-Position Index

We now proceed to describe the evaluation of our ID-to-Position Index compared to standard binary search using the LUBM 10240 dataset in the single-thread setting. Table 5.7 shows the number of binary searches and the number of sequential searches which were performed using the decision of our adaptive join method, using a threshold of about 200 computed with our calibration algorithm. The fact that sequential searches heavily outnumber binary searches provides a strong indication that ordering is present in the RDF dataset. In order to compare our index with binary search, we kept the threshold the same as computed in the case of binary search, and executed the queries by performing our index based lookup instead of binary search, measuring the exact number of total execution cycles used in the index lookup or binary search procedure each time, as well as the cache misses for each cache level. If we exclude queries no 1 and 3-6, as they nearly perform only sequential searches, we can see that our ID-to-Position index results in more

Query	#Binary	#Sequential	Binary Search				ID-to-Position Index			
			Cycles	L1 M	L2 M	L3 M	Cycles	L1 M	L2 M	L3 M
LUBM1	1	107525748	2236	130	49	9	3135	102	43	8
LUBM2	204795	10854018	502M	26.7M	10.8M	3.5M	355M	18.3M	4.4M	543K
LUBM3	1	33169741	2401	140	50	8	4175	139	42	3
LUBM4	4	68	38745	666	368	235	16862	469	182	34
LUBM5	1	10	2423	94	29	0	2395	162	83	5
LUBM6	1	570	2033	106	26	0	2003	130	48	0
LUBM7	2257238	28768005	2.95B	254M	80.1M	2.30M	2.12B	211M	58.9M	1.08M
LUBM8	8645	84755793	17.4M	1.20M	682K	84.1K	11.2M	841K	351K	21.7K
LUBM9	409590	351307982	1.06B	53.6M	19.7M	2.92M	655.7M	39.1M	11.18M	639.7K
LUBM10	558279	116015419	1.22B	66.7M	24.2M	2.98M	798.2M	50.76M	12.7M	634.3K

Table 5.7: Number of binary searches and sequential searches for LUBM10240 chosen by out adaptive join method and comparison of binary search with ID-to-Position index with respect to total execution cycles and L1, L2 and L3 cache misses

than 30% decrease in total execution cycles and similar or larger decrease in the number of cache misses for all levels of cache hierarchy.

5.5.2.3 Scalability

In this section we experimentally show the scalability of PARJ with regard to a varying number of threads and varying dataset size. As far as the first issue is concerned, we can already observe from Section 5.5.2 and specifically from Tables 5.2, 5.4 and 5.5, that running PARJ in multi threaded mode with 32 threads performs on average about 15 times better than the single thread version, but for the simple queries, when execution time is less than few tens milliseconds, multi-threaded execution does not seem to provide important gains. There are two reasons for that. The first one is the overhead of spawning multiple threads and the second is that query parsing and optimization take up a large fragment of the total execution time, which cannot be avoided in multi-threaded execution. The best example of this is query S1 from WatDiv benchmark which is a star join query with 9 triple patterns and more than 40 milliseconds of the reported time of 49 milliseconds is spent on producing the join order in the optimizer.

In order to better examine the behavior of PARJ for a varying number of threads we have executed the queries from LUBM benchmark for scale 10240 with 1, 2, 4, 8 and 16 threads as shown in Figure 5.5. We exclude from this presentation simple and very selective queries L4, L5 and L6 that do not appear to improve from parallelism, since already in the single-threaded execution their execution time is only a few milliseconds, much of which is due to query parsing and optimization. On the other hand, complex queries L1, L3, and L7-L10, and also the simple but not selective query L2 show large and nearly linear improvement. The reason that we do not show results beyond 16 threads in Figure 5.5 has to do with the capabilities of our testing machine, which has exactly 16 processing cores. As stated before, best results were obtained with 32 threads as hyper-threading

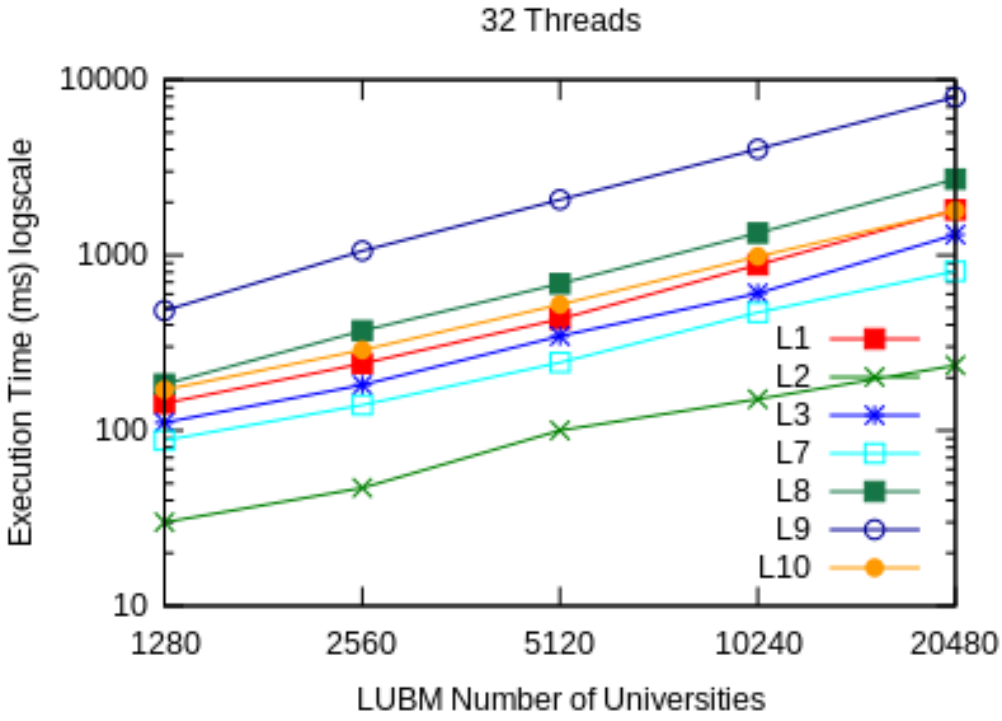


Figure 5.4: LUBM 32 threads execution times in ms for different dataset sizes

was enabled, but improvement from 16 to 32 threads cannot be evaluated and interpreted reliably for the specific scalability experiment, as here we aim to examine the behavior of PARJ for a varying number of threads given that the underlying hardware can provide full processing resources to each thread.

We have also examined the scalability of our system for a varying dataset size. Findings in Figure 5.4 show a similar situation for a varying number of universities up to 20480 (about 2.83 billion triples) in the execution with 32 threads, confirming the excellent scalability of PARJ.

5.5.2.4 Results for Query Execution over OWL2 QL Ontologies

In this section we provide experimental comparison of our approach for query execution over OWL2 QL ontologies with other state of the art methods. We use the name PARJ$_{Ontop}$ for our prototype as described in Section 5.4. As in the stand-alone version of PARJ, this implementation is publicly available [4]. As main competitors we have used the semantic index mode of Ontop with PostgreSQL 10 as backend and a commercial RDF store providing support for OWL2 QL query answering via query rewriting, which we will call system A, as due to its license we cannot reveal its real name when providing experimental results. System A is a disk-based system, but for the experiments we have deployed its database

[4] https://github.com/dbilid/PARJ-Ontop

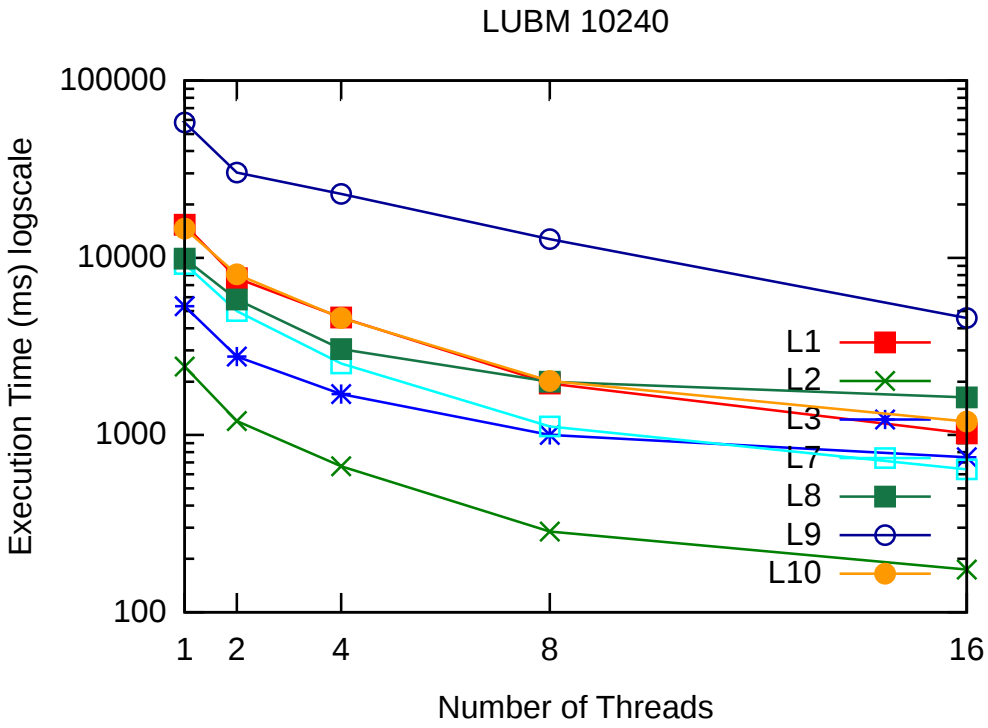


Figure 5.5: LUBM 10240 execution times in ms for different number of threads

in an in-memory filesystem. We have also tried to use a second commercial system which is based mainly on materialization, but for the dataset used in our experiments data loading was not completed even after 8 days. For the semantic index experiments we have used an older version of Ontop (1.12), as in latest versions the semantic index mode is not maintained. The dataset used in the experiments is from the $\text{LUBM}^{\exists}_{20}$ benchmark [63] for scale of 1000 universities and with incompleteness ratio 5%, which contains about 147 million triples and the raw data size in NTriples format is 25.5 GB. The queries are the same used in [82]. In contrast with the experiments described in previous sections, here, for all the systems, we include in the results the time needed to perform dictionary lookups and tuple construction. The reason is that we could not modify System A so as to exclude these features. Results are presented in Table 5.8. PARJ_{Ontop} was the only system that successfully executed all queries in a 30 minutes time limit per query, as q1 was timed out for semantic index, and q5 returned an error in System A. For the rest of the queries, even the single thread version of PARJ is on average about an order of magnitude faster than the other two systems. Regarding the overhead of union wrappers, we have executed all the queries with the hierachies materialized in PARJ, in order to estimate the impact they have on query execution. The average execution time for single thread execution decreased from 15554 milliseconds to 13790, with the overhead being less than 13%, even though all queries involved union wrappers of large hierarchies, some of them, like q7, with up to six different union wrappers involved in the same query.

	PARJ-1	PARJ-32	SI-PostgreSQL	SystemA
q1	197997	10619	TIMEOUT	673268
q2	3977	251	32477	2610
q3	2369	190	59530	15803
q4	3890	482	2749	47029
q5	4828	459	60490	ERROR
q6	5	11	25318	240
q7	467	70	19413	8730
q8	184	26	924	7853
q9	1	1	1	145
r1	1	1	1	151
r2	522	66	3049	525
r3	3170	390	26433	30745
r4	181	25	803	797
r5	170	62	1	1505

Table 5.8: Results for LUBM$^{\exists}_{20}$ 1000 (times in ms)

5.5.2.5 Comparison With Distributed RDF Stores

A comparison of a parallel centralized system with distributed systems is not straightforward, as many factors come into play in order to have a result that will be as fair and complete as possible. In this section we attempt some first comparison of PARJ with existing RDF stores based on a recently published survey [2] and we plan to further investigate this issue experimentally in the future. The aforementioned survey presents an experimental comparison of 12 distributed systems designed for shared-nothing clusters, chosen as the most competitive and innovative from a variety of approaches and characteristics. The experiments were performed on a cluster with 12 servers, each with 148GB of memory and 24 cores, using, among others, the LUBM 10240 (only queries LUBM1-LUBM7) and WatDiv 1000 (only basic workload) benchmarks. For both these benchmarks the single server results of PARJ (in the full result handling mode) are comparable with the faster of the reported systems which is the non-adaptive version of AdPart (the adaptive version is not included in the results of [2]). Specifically, the average and geometric mean of execution times for first seven queries of LUBM 10240 are 918 and 75 milliseconds respectively (compared with 419 and 103 for PARJ in full result handling mode) whereas the geometric means for the 4 query categories of the basic workload of WatDiv 1000 are 9, 7, 160 and 111 milliseconds (compared with 9, 10, 12 and 48 for PARJ in full result handling mode).

6. CONCLUSIONS AND FUTURE WORK

In this thesis we have studied the problem of efficient query answering in OBDA systems over external data sources. For the case where we have a single relational data source, we have identified redundant processing as a bottleneck in query processing and we have proposed solutions to overcome this problem, by providing a complete cost-based query translation method. We believe that using cost-based planning is a prominent direction towards OBDA query optimization, that has not been fully explored yet. In future work, we plan to incorporate decisions about physical database design by analyzing the mapping assertions. One more direction regarding future research has to do with duplicate elimination in case the OBDA system is equipped with query processing capabilities, in other words when it acts as a mediator as the Exareme OBDA mediator presented in Chapter 4. In this setting, along with decisions regarding which query fragments should be evaluated in external databases, one should decide when duplicate elimination should be "pushed" to *endpoints* or performed by the OBDA processing engine during data import.

Regarding query processing for federated execution, as presented in Chapter 4, an interesting extension has to do with the support of non-relational external sources. In the context of OBDA there have been some first results about access to other types of data sources, as for example NoSQL databases [19] and web data sources [10]. One useful functionality for Exareme mediator would be to support such data sources as endpoints and provide the user with unified access to both relational and non-relational endpoints. In this scenario we also plan to employ PARJ as an endpoint for storing RDF data.

Regarding Chapter 5, we have presented the in-memory system PARJ for parallelizing OBDA join queries on RDF graphs. We have shown that our design has excellent scaling capabilities and performance. For future work, we first plan to perform a more thorough experimental comparison with distributed RDF stores. As we mentioned, it is straightforward to extend PARJ to a "cluster" version through full replication, such that during query execution each worker start processing from different initial shard. We plan to implement and compare this version with the current state of the art distributed systems. We also want to further evaluate PARJ on a high-end server with larger available memory, in order to load and process larger RDF graphs. Based on the scaling capabilities presented during the experiments, we anticipate that our approach will be able to efficiently handle such datasets.

Another prominent research direction has to do with query optimization and support for geospatial queries. In this case, the initial queries over the ontology can be expressed in the query language GeoSPARQL[72], which is a geospatial extension of SPARQL and it is standardized by the Open Geospatial Consortium (OGC). The system Ontop-spatial [11] is a state of the art system that performs OBDA query translaton from GeoSPARQL to SQL enhanced with spatial operators for execution in spatially enabled relational systems like PostGIS. An interesting extension would be the support of geospatial operators in the Exareme mediator system, where spatially enabled databases could act as endpoints, and also spatial processing capabilities should be added to the mediator itself in order to

perform spatial joins between data coming from different endpoints. A similar extension could be added to PARJ, in order to support GeoSPARQL queries coming from Ontop-spatial.

ABBREVIATIONS - ACRONYMS

RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
OWL	Web Ontology Language
W3C	World Wide Web Consortium
OGC	Open Geospatial Consortium
OBDA	Ontology-based Data Access
CQ	Conjunctive Query
UCQ	Union of Conjunctives Queries
JUCQ	Join of Unions of Conjunctives Queries
NPD	Norwegian Petroleum Directorate
LUBM	Lehigh University Benchmark
RAM	Random-Access Memory
TGD	Tuple-Generating Dependency
UDF	User-Defined Function
YAGO 2	Yet Another Great Ontology 2
DAG	Directed Acyclic Graph
JDBC	Java Database Connectivity
XML	Extensible Markup Language
API	Application Programming Interface
PARJ	Parallel Adaptive RDF Joins
SLD-resolution	Selective Linear Definite clause resolution
IRI	Internationalized Resource Identifier
R2RML	RDB-to-RDF Mapping Language
RML	RDF Mapping Language
UoA	National and Kapodistrian University of Athens

NTUA	National Technical University of Athens

APPENDIX A. NPD QUERIES 31-34

```
SELECT DISTINCT ?q ?u
WHERE {
?q :inLithostratigraphicUnit ?u .
?u rdf:type :LithostratigraphicUnit .
}
```

Listing A.1: Query NPD 31

```
SELECT DISTINCT ?quadrant ?name
WHERE {
?quadrant rdf:type :Quadrant .
?quadrant :name ?name .
}
```

Listing A.2: Query NPD 32

```
SELECT DISTINCT ?unit ?era
WHERE {
?unit :geochronologicEra ?era .
?unit rdf:type :LithostratigraphicUnit .
}
```

Listing A.3: Query NPD 33

```
SELECT DISTINCT ?wellbore ?discovery ?year
WHERE {
?wellbore rdf:type :Wellbore .
?wellbore :wellboreForDiscovery ?discovery .
?discovery :discoveryYear ?year
}
```

Listing A.4: Query NPD 34

REFERENCES

[1] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 411–422, 2007.

[2] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. *PVLDB*, 10(13):2049–2060, 2017.

[3] Razen Al-Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB J.*, 25(3):355–380, 2016.

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

[5] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ICS-FORTH RDFSuite: Managing voluminous RDF description bases. In *SemWeb*, 2001.

[6] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of RDF data management systems. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*, pages 197–212, 2014.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD Conference*, pages 1383–1394. ACM, 2015.

[8] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. DBpedia: A nucleus for a web of open data. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007.*, pages 722–735, 2007.

[9] Franz Baader, Diego Calvanese, Deborah McGuinness, Peter Patel-Schneider, Daniele Nardi, et al. *The Description Logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[10] Konstantina Bereta, George Papadakis, and Manolis Koubarakis. Obda for the web: Creating virtual rdf graphs on top of web data sources. *arXiv preprint arXiv:2005.11264*, 2020.

[11] Konstantina Bereta, Guohui Xiao, and Manolis Koubarakis. Ontop-spatial: Ontop of geospatial databases. *Journal of Web Semantics*, 58:100514, 2019.

[12] Dimitris Bilidas and Manolis Koubarakis. Efficient duplicate elimination in SPARQL to SQL translation. In *Proceedings of the 31st International Workshop on Description Logics co-located with 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018), Tempe, Arizona, US, October 27th - to - 29th, 2018*, volume 2211 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[13] Dimitris Bilidas and Manolis Koubarakis. Scalable parallelization of RDF joins on multicore architectures. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*, pages 349–360, 2019.

[14] Dimitris Bilidas and Manolis Koubarakis. Handling redundant processing in OBDA query execution over relational sources (currently under review). 2020.

[15] Dimitris Bilidas and Manolis Koubarakis. In-memory parallelization of join queries over large ontological hierarchies. *Distributed and Parallel Databases*, pages 1–38, 2020.

[16] Dina Bitton and David J DeWitt. Duplicate record elimination in large data files. *ACM Transactions on database systems (TODS)*, 8(2):255–265, 1983.

[17] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. Smooth scan: Statistics-oblivious access paths. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 315–326. IEEE, 2015.

[18] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser.

Smooth scan: robust access path selection without cardinality estimation. *The VLDB Journal*, pages 1–25, 2018.

[19] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Julien Corman, and Guohui Xiao. A generalized framework for ontology-based data access. In *International Conference of the Italian Association for Artificial Intelligence*, pages 166–180. Springer, 2018.

[20] Damian Bursztyn, François Goasdoué, and Ioana Manolescu. Teaching an RDBMS about ontological constraints. *Proceedings of the VLDB Endowment*, 9(12):1161–1172, 2016.

[21] Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, 8(3):471–487, 2017.

[22] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Dl-lite: Tractable description logics for ontologies. In *AAAI*, volume 5, pages 602–607, 2005.

[23] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The dl-lite family. *Journal of Automated reasoning*, 39(3):385–429, 2007.

[24] Diego Calvanese, Martin Giese, Dag Hovland, and Martin Rezk. Ontology-based integration of cross-linked datasets. In *International Semantic Web Conference*, pages 199–216. Springer, 2015.

[25] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system R. *Commun. ACM*, 24(10):632–646, 1981.

[26] Surajit Chaudhuri and Moshe Y Vardi. Optimization of real conjunctive queries. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 59–70. ACM, 1993.

[27] Alexandros Chortaras, Despoina Trivela, and Giorgos Stamou. Optimized query rewriting for OWL 2 QL. In *International Conference on Automated Deduction*, pages 192–206. Springer, 2011.

[28] J. Crompton. Keynote talk at the W3C workshop on sem. web in oil & gas industry, 2008. Available from `http://www.w3.org/2008/12/ogws-slides/Crompton.pdf`.

[29] David DeHaan and Frank Wm Tompa. Optimal top-down join enumeration. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 785–796. ACM, 2007.

[30] David J. DeWitt. The wisconsin benchmark: Past, present, and future. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[31] Jin-Hang Du, Haofen Wang, Yuan Ni, and Yong Yu. HadoopRDF: A scalable semantic data analytical engine. In *Intelligent Computing Theories and Applications - 8th International Conference, ICIC 2012, Huangshan, China, July 25-29, 2012. Proceedings*, pages 633–641, 2012.

[32] Pit Fender, Guido Moerkotte, Thomas Neumann, and Viktor Leis. Effective and robust pruning for top-down join enumeration algorithms. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 414–425. IEEE, 2012.

[33] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Query rewriting and optimization for ontological databases. *ACM Transactions on Database Systems (TODS)*, 39(3):25, 2014.

[34] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[35] Goetz Graefe and William J McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 209–218. IEEE, 1993.

[36] Jinghua Groppe and Sven Groppe. Parallelizing join computations of SPARQL queries for large semantic web databases. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1681–1686. ACM, 2011.

[37] The W3C SPARQL Working Group. SPARQL 1.1 Overview. W3C Recommendation 21 March 2013. Retrieved November 17, 2020, from `https://www.w3.org/TR/sparql11-overview/`.

[38] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[39] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. Triad: a distributed shared-

nothing RDF engine based on asynchronous message passing. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 289–300, 2014.
[40] Johannes Hoffart, Fabian M Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61, 2013.
[41] Dag Hovland, Roman Kontchakov, Martin G Skjæveland, Arild Waaler, and Michael Zakharyaschev. Ontology-based data access to slegge. In *International Semantic Web Conference*, pages 120–129. Springer, 2017.
[42] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011.
[43] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.
[44] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 19–30. VLDB Endowment, 2003.
[45] Zachary G Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S Weld. An adaptive query execution system for data integration. *ACM SIGMOD Record*, 28(2):299–310, 1999.
[46] Jason St Jacques, David Toman, and Grant E Weddell. Object-relational queries over $CFD_{nc}^{\forall -}$ knowledge bases: OBDA for the SQL-literate. In *Description Logics*, 2016.
[47] Zoi Kaoudi and Ioana Manolescu. RDF in the clouds: a survey. *The VLDB Journal*, 24(1):67–91, 2015.
[48] Evgeny Kharlamov, Dag Hovland, Ernesto Jiménez-Ruiz, Davide Lanti, Hallstein Lie, Christoph Pinkel, Martin Rezk, Martin G Skjæveland, Evgenij Thorstensen, Guohui Xiao, Dmitriy Zheleznyakov, and Ian Horrocks. Ontology based access to exploration data at Statoil. In *International Semantic Web Conference*, pages 93–112. Springer, 2015.
[49] Evgeny Kharlamov, Dag Hovland, Martin G Skjæveland, Dimitris Bilidas, Ernesto Jiménez-Ruiz, Guohui Xiao, Ahmet Soylu, Davide Lanti, Martin Rezk, Dmitriy Zheleznyakov, et al. Ontology based data access in statoil. *Journal of Web Semantics*, 44:3–36, 2017.
[50] Evgeny Kharlamov, Ernesto Jiménez-Ruiz, Dmitriy Zheleznyakov, Dimitris Bilidas, Martin Giese, Peter Haase, Ian Horrocks, Herald Kllapi, Manolis Koubarakis, Özgür Özçep, et al. Optique: Towards OBDA systems for industry. In *Extended Semantic Web Conference*, pages 125–140. Springer, 2013.
[51] Evgeny Kharlamov, T Mailis, Konstantina Bereta, Dimitris Bilidas, Sebastian Brandt, Ernesto Jiménez-Ruiz, Steffen Lamparter, Christian Neuenstadt, O Özçep, Ahmet Soylu, et al. A semantic approach to polystores. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2565–2573. IEEE, 2016.
[52] Stanislav Kikot, Roman Kontchakov, and Michael Zakharyaschev. Conjunctive query answering with OWL 2 QL. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*, 2012.
[53] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.
[54] Herald Kllapi, Dimitris Bilidas, Ian Horrocks, Yannis Ioannidis, Ernesto Jiménez, Evgeny Kharlamov, Manolis Koubarakis, Dmitriy Zheleznyakov, et al. Distributed query processing on the cloud: the optique point of view (short paper). 2013.
[55] Herald Kllapi, Eva Sitaridi, Manolis M Tsangaris, and Yannis Ioannidis. Schedule optimization for data processing flows on the cloud. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 289–300, 2011.
[56] Roman Kontchakov, Carsten Lutz, David Toman, Frank Wolter, and Michael Zakharyaschev. The combined approach to ontology-based data access. In *Twenty-second international joint conference on artificial intelligence*, 2011.
[57] Manolis Koubarakis and Kostis Kyzirakos. Modeling and Querying Metadata in the Semantic Sensor Web: The Model stRDF and the Query Language stSPARQL. In *ESWC*, 2010.
[58] Davide Lanti, Martin Rezk, Guohui Xiao, and Diego Calvanese. The NPD benchmark: Reality check for OBDA systems. In *Proc. of the 18th Int. Conf. on Extending Database Technology (EDBT)*, 2015.
[59] Davide Lanti, Guohui Xiao, and Diego Calvanese. Cost-driven ontology-based data access. In *The*

Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I, volume 10587 of *Lecture Notes in Computer Science*, pages 452–470. Springer, 2017.

[60] John W Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 2012.

[61] John W. Lloyd and John C Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3-4):217–242, 1991.

[62] Yongming Luo, François Picalausa, George HL Fletcher, Jan Hidders, and Stijn Vansummeren. Storing and indexing massive RDF datasets. In *Semantic search over the web*, pages 31–60. Springer, 2012.

[63] Carsten Lutz, Inanç Seylan, David Toman, and Frank Wolter. The combined approach to OBDA: Taming role hierarchies using filters. In *International semantic web conference*, pages 314–330. Springer, 2013.

[64] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, 2002.

[65] Stefan Manegold, Peter Boncz, and Martin L Kersten. Generic database cost models for hierarchical memory systems. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 191–202. Elsevier, 2002.

[66] Jose Mora and Óscar Corcho. Engineering optimisations in query rewriting for OBDA. In *Proceedings of the 9th International Conference on Semantic Systems*, pages 41–48. ACM, 2013.

[67] Jaeseok Myung, Jongheum Yeon, and Sang-goo Lee. Sparql basic graph pattern processing with iterative mapreduce. In *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*. ACM, 2010.

[68] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. RDFox: A highly-scalable RDF store. In *International Semantic Web Conference*, pages 3–20. Springer, 2015.

[69] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, pages 984–994. IEEE Computer Society, 2011.

[70] Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD Conference*, pages 627–640. ACM, 2009.

[71] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110. ACM, 2008.

[72] Open Geospatial Consortium. OGC GeoSPARQL - A geographic query language for RDF data. OGC® Implementation Standard, 2012.

[73] Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras, and Nectarios Koziris. H_2RDF+: an efficient data management system for big RDF graphs. In *SIGMOD Conference*, pages 909–912. ACM, 2014.

[74] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *Data Engineering, 1988. Proceedings. Fourth International Conference on*, pages 311–319. IEEE, 1988.

[75] Arjan Pellenkoft, César A Galindo-Legaria, and Martin Kersten. The complexity of transformation-based join enumeration. In *VLDB*, pages 306–315, 1997.

[76] Héctor Pérez-Urbina, Ian Horrocks, and Boris Motik. Efficient query answering for owl 2. *The Semantic Web-ISWC 2009*, pages 489–504, 2009.

[77] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking data to ontologies. In *Journal on data semantics X*, pages 133–173. Springer, 2008.

[78] Anthony Potter, Boris Motik, Yavor Nenov, and Ian Horrocks. Distributed RDF query answering with dynamic data exchange. In *International Semantic Web Conference (1)*, volume 9981 of *Lecture Notes in Computer Science*, pages 480–497, 2016.

[79] Roshan Punnoose, Adina Crainiceanu, and David Rapp. SPARQL in the cloud using Rya. *Inf. Syst.*, 48:181–195, 2015.

[80] Wilson Qin and Stratos Idreos. Adaptive data skipping in main-memory systems. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2255–2256. ACM, 2016.

[81] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. An intermediate algebra for optimizing RDF graph pattern matching on mapreduce. In *ESWC (2)*, volume 6644 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2011.

[82] Mariano Rodriguez-Muro, Roman Kontchakov, and Michael Zakharyaschev. Ontology-based data access: Ontop of databases. In *International Semantic Web Conference*, pages 558–573. Springer, 2013.
[83] Mariano Rodríguez-Muro and Martin Rezk. Efficient SPARQL-to-SQL with R2RML mappings. *Web Semantics: Science, Services and Agents on the World Wide Web*, 33:141–169, 2015.
[84] Kurt Rohloff and Richard E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the SHARD triple-store. In *PSI EtA*, page 4. ACM, 2010.
[85] Kurt Rohloff and Richard E. Schantz. Clause-iteration with mapreduce to scalably query datagraphs in the SHARD graph-store. In *DICT@HPDC*, pages 35–44. ACM, 2011.
[86] Riccardo Rosati and Alessandro Almatelli. Improving query answering over DL-Lite ontologies. In *Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*, 2010.
[87] Prasan Roy, Sridhar Seshadri, S Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *ACM SIGMOD Record*, 29(2):249–260, 2000.
[88] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: mapping SPARQL to Pig Latin. In *SWIM*, page 4. ACM, 2011.
[89] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF querying with SPARQL on spark. *PVLDB*, 9(10):804–815, 2016.
[90] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. Adoption of the linked data best practices in different topical domains. In *Semantic Web Conference (1)*, volume 8796 of *Lecture Notes in Computer Science*, pages 245–260. Springer, 2014.
[91] Timos K Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.
[92] Juan F Sequeda, Marcelo Arenas, and Daniel P Miranker. OBDA: query rewriting or materialization? in practice, both! In *International Semantic Web Conference*, pages 535–551. Springer, 2014.
[93] Juan F Sequeda and Daniel P Miranker. Ultrawrap: SPARQL execution on relational data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 22:19–39, 2013.
[94] Leonard Shapiro, David Maier, Paul Benninghoff, Keith Billings, Yubo Fan, Kavita Hatwal, Quan Wang, Yu Zhang, H-M Wu, and Bennet Vance. Exploiting upper and lower bounds in top-down query optimization. In *Database Engineering and Applications, 2001 International Symposium on.*, pages 20–33. IEEE, 2001.
[95] Yasin N Silva, P-A Larson, and Jingren Zhou. Exploiting common subexpressions for cloud query processing. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1337–1348. IEEE, 2012.
[96] Dominik Ślezak, Jakub Wróblewski, Victoria Eastwood, and Piotr Synak. Brighthouse: an analytic data warehouse for ad-hoc queries. *Proceedings of the VLDB Endowment*, 1(2):1337–1345, 2008.
[97] Giorgio Stefanoni, Boris Motik, and Egor V Kostylev. Estimating the cardinality of conjunctive queries over RDF data using graph summarisation. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1043–1052. International World Wide Web Conferences Steering Committee, 2018.
[98] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564. ACM, 2005.
[99] Julien Subercaze, Christophe Gravier, Jules Chevalier, and Frederique Laforest. Inferray: fast in-memory RDF inference. *Proceedings of the VLDB Endowment*, 9(6):468–479, 2016.
[100] Arun Swami and K Bernhard Schiefer. On the estimation of join result sizes. In *International Conference on Extending Database Technology*, pages 287–300. Springer, 1994.
[101] Manolis M Tsangaris, George Kakaletris, Herald Kllapi, Giorgos Papanikos, Fragkiskos Pentaris, Paul Polydoras, Eva Sitaridi, Vassilis Stoumpos, and Yannis E Ioannidis. Dataflow processing and optimization on grid and cloud infrastructures. *IEEE Data Eng. Bull.*, 32(1):67–74, 2009.
[102] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
[103] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF storage and retrieval in jena2. In *SWDB*, pages 131–150, 2003.
[104] Guohui Xiao, Dag Hovland, Dimitris Bilidas, Martin Rezk, Martin Giese, and Diego Calvanese. Ef-

ficient ontology-based data integration with canonical IRIs. In *European Semantic Web Conference*, pages 697–713. Springer, 2018.

[105] Guohui Xiao, Davide Lanti, Roman Kontchakov, Sarah Komla-Ebri, Elem Güzel-Kalaycı, Linfang Ding, Julien Corman, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. The virtual knowledge graph system Ontop. ISWC 2020 - 19th International Semantic Web Conference, 2020.

[106] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. Triplebit: a fast and compact system for large scale RDF data. *Proceedings of the VLDB Endowment*, 6(7):517–528, 2013.

[107] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4):265–276, 2013.

[108] Jingren Zhou, P-A Larson, and Ronnie Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 1060–1071. IEEE, 2010.

[109] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 533–544. ACM, 2007.