University of Crete
Computer Science Department

Heuristic Optimization of SPARQL queries over Column-Store DBMS

Petros Anagnostopoulos-Tsialiamanis

Master's Thesis

Heraklion, September 2011

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Ευρετική Βελτιστοποίηση ερωτήσεων SPARQL σε ΣΔΒΔ βασισμένα σε κολόνες

Εργασία που υποβλήθηκε από τον
Πέτρο Αναγνωστόπουλο-Τσιαλιαμάνη
ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Πέτρος Αναγνωστόπουλος - Τσιαλιαμάνης, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Βασίλης Χριστοφίδης, Καθηγητής, Επόπτης

Δημήτρης Πλεξουσάκης, Καθηγητής

Ειρήνη Φουντουλάκη, Ερευνήτρια

Άγγελος Μπίλας, Αναπλ. Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Σεπτέμβριος 2011

Heuristic Optimization of SPARQL queries over Column-Store DBMS

Petros Anagnostopoulos-Tsialiamanis
Master's Thesis
Computer Science Department, University of Crete

Abstract

During the last decade we have witnessed a tremendous increase in the amount of semantic data available on the Web in almost every field of human activity. More and more corporate, governmental, or even user-generated datasets break the walls of "private" management within their production site, are published, and become available for potential data consumers, i.e., applications/services, individual users and communities. In this context, The Web of Data extends current Web to a global data space connecting data from diverse domains. This gives added value for decision support and business intelligence applications, and enables new types of services that operate on top of an unbound, global data space and not on a fixed set of data sources as in Web 2.0 mashups. A central issue in this respect is the manipulation and usage of data based on their meaning by using effective and efficient support for storing, querying, and manipulating semantic RDF data, the lingua franca of Linked Open Data and hence the default data model for the Web of Data.

In this thesis we are focusing on the problem of scalable processing and optimization of semantic queries expressed in SPARQL using modern relational engines. Existing native or SQL-based engines for processing SPARQL queries heavily rely on statistics regarding the stored RDF graphs as well as adequate cost based planning algorithms to optimize complex join queries. Extensive data statistics are quite expensive to compute and maintain for large scale evolving semantic data over the Web. The main challenge in this respect is to devise heuristics-based query optimization techniques generating near to optimal execution plans without any knowledge of the underlying datasets. For this reason we propose the first *heuristics-based SPARQL planner* (HSP) that is capable of exploring the syntactic variations of triple patterns in a query in order to choose a near to optimal execution plan without the use of a cost model. Furthermore, we have implemented HSP plans on top of the MonetDB column-based DBMS. We have paid particular attention to the efficient implementation of HSP logical plans to the underlying MonetDB query execution engine by translating them into MonetDB's physical algebra (MAL). We have finally, experimentally evaluated *the quality* and *execution time* of the plans produced by HSP with a state-of-the-art Cost-based Dynamic Programming (CDP) algorithm employed by RDF-3X using synthetically generated and real RDF datasets.

In all queries of our workload, HSP produce plans with the same number of merge and hash joins as CDP. Their differences lie on the employed ordered variables as well as the execution order of joins which essentially affect the size of intermediate results. With the exception of queries which are not substantially different in their syntax, HSP plans executed on MonetDB outperform those of CDP executed in RDF-3X up to three orders of magnitude. More precisely, HSP tries to produce plans that maximize the number of merge joins over the ordered variables which are shared among the triple patterns of a query and relies on various heurists to decide which ordered variables will be used in selections and joins as well as which underlying access paths will be exploited for evaluating the triple patterns (essentially sorted triple relations in MonetDB).

Supervisor: Vassilis Christophides
Professor

Περίληψη

Κατά την τελευταία δεκαετία παρατηρούμε μία τεράστια αύξηση του πλήθους σημασιολογικών δεδομένων τα οποία είναι διαθέσιμα στο διαδίκτυο και για ένα μεγάλο αριθμό δραστηριοτήτων. Δεδομένα από επιχειρήσεις, κυβερνήσεις ή ακόμα και από απλούς χρήστες παύουν να αποτελούν ;ιδιωτική; πληροφορία μέσα στον χώρο παραγωγής τους, δημοσιεύονται και βρίσκονται διαθέσιμα προς χρήση από ενδεχόμενους καταναλωτές, όπως εφαρμογές/υπηρεσίες, ανεξάρτητους χρήστες ή και κοινότητες χρηστών. Σε αυτό το πλαίσιο, το Ιστός των Δεδομένων (Web of Data) επεκτείνει τον τρέχον Παγκόσμιο Ιστό σε ένα παγκόσμιο χώρο δεδομένων που συνδέει πληροφορίες από διάφορους τομείς. Το γεγονός αυτό αυξάνει την αξία της υποστήριξης αποφάσεων και εφαρμογών επιχειρηματικής νοημοσύνης (business intelligence) και επιτρέπει τη δημιουργία νέου τύπου υπηρεσιών στη βάση ενός παγκόσμιου χώρου δεδομένων χωρίς όρια, και όχι απλά σε ένα αυστηρά καθορισμένο σύνολο πηγών δεδομένων, όπως στην περίπτωση των Web 2.0 mashups. Δεδομένου ότι η RDF είναι το lingua franca για τα Linked Open Data και ως εκ τούτου για το βασικό μοντέλο δεδομένων στον Παγκόσμιο Ιστό, ένα κεντρικό ζήτημα εδώ είναι η διαχείριση και η χρήση των RDF δεδομένων, και πιο συγκεκριμένα η αποτελεσματική και αποδοτική υποστήριξη για την αποθήκευση και αναζήτηση τους μέσω επερωτήσεων.

Σε αυτή την εργασία επικεντρωνόμαστε στο πρόβλημα της κλιμακώσιμης επεξεργασίας και βελτιστοποίησης SPARQL επερωτήσεων χρησιμοποιώντας σύγχρονες σχεσιακές μηχανές. Οι υπάρχουσες γηγενείς (native) μηχανές και οι μηχανές βασισμένες σε SQL για την επεξεργασία επερωτήσεων SPARQL, στηρίζονται σε μεγάλο βαθμό στη χρήση στατιστικών που αφορούν τους αποθηκευμένους RDF γράφους, καθώς επίσης και σε αλγόριθμους σχεδίασης οι οποίοι χρησιμοποιούν μοντέλα κόστους προκειμένου να βελτιστοποιήσουν σύνθετες επερωτήσεις σύζευξης. Τέτοιου τύπου στατιστικά είναι αρκετά ακριβά τόσο στον υπολογισμό όσο και στην διατήρηση τους για μεγάλης κλίμακας εξελισσόμενα σημασιολογικά δεδομένα του Παγκόσμιου Ιστού. Η βασική πρόκληση που τίθεται είναι η επινόηση τεχνικών βελτιστοποίησης για τη κατασκευή πλάνων εκτέλεσης επερωτήσεων βασισμένων σε ευρετικούς κανόνες, οι οποίοι δημιουργούν πλάνα όσο δυνατόν βέλτιστα πλάνα εκτέλεσης χωρίς την χρήση οποιασδήποτε γνώσης για τα αποθηκευμένα RDF δεδομένα. Για αυτό το λόγο προτείνουμε τον πρώτο κατασκευαστή πλάνων για SPARQL επερωτήσεις βασισμένο σε ευρετικούς κανόνες (heuristic-based SPARQL planning - HSP), ικανό να αναγνωρίζει τις συντακτικές παραλλαγές των προτύπων πρόσβασης τριάδας σε μία επερώτηση προκειμένου να επιλέξει το βέλτιστο δυνατό πλάνο εκτέλεσης χωρίς τη χρήση μοντέλου κόστους. Στην εργασία αυτή, τα HSP πλάνα έχουν υλοποιηθεί πάνω από τη MonetDB, ένα Σύστημα Διαχείρισης Βάσεων Δεδομένων που βασίζεται στην τεχνολογία κολόνων (column-based DBMS). Μεγάλη προσοχή δώθηκε στην αποδοτική υλοποίηση των λογικών πλάνων HSP στην μηχανή εκτέλεσης επερωτήσεων της MonetDB, με την μετάφραση των HSP πλάνων στην φυσική άλγεβρα της MonetDB (MAL). Τέλος, αποτιμήσαμε πειραματικά την ποιότητα και τον χρόνο εκτέλεσης των HSP πλάνων και συγκρίναμε τα μεγέθη αυτά με τα πλάνα που παρήγαγε ο αλγόριθμος Cost-based Dynamic Programming (CDP). Το γηγενές σύστημα επεξεργασίας SPARQL επερωτήσεων RDF-3X χρησιμοποιήθηκε για την εκτέλεση των CDP πλάνων. Για την πειραματική αυτή αποτίμηση χρησιμοποιήσαμε τόσο συνθετικά όσο και πραγματικά RDF δεδομένα.

Σε όλες τις επερωτήσεις που χρησιμοποιήσαμε, οι αλγόριθμοι HSP και CDP παρήγαγαν πλάνα με τον ίδιο αριθμό πράξεων σύζευξης με συγχώνευση (merge join) και κατακερματισμό (hash join). Η διαφορά των παραγόμενων πλάνων έγκειται στις μεταβλητές οι οποίες χρησιμοποιούνται στις πράξεις σύζευξης με συγχώνευση, καθώς και στην σειρά εκτέλεσης των πράξεων σύζευξης η οποία επηρεάζει το μέγεθος των ενδιάμεσων αποτελεσμάτων. Στην πλειοψηφία των επερωτήσεων,

ο χρόνος εκτέλεσης των HSP πλάνων στη MonetDB έχουν καλύτερος μέχρι και 3 τάξεις μεγέθους από τον χρόνο εκτέλεσης των CDP πλάνων τα οποία εκτελούνται στην μηχανή RDF-3X. Πιο συγκεκριμένα, ο αλγόριθμος HSP προσπαθεί να παράγει πλάνα τα οποία μεγιστοποιούν τον αριθμό πράξεων σύζευξης με συγχώνευση πάνω από ταξινομημένες μεταβλητές που είναι κοινές στα πρότυπα πρόσβασης τριάδων μιας επερώτησης. Βασίζεται σε ένα σύνολο ευρετικών κανόνων για να αποφασίσει ποιες ταξινομημένες μεταβλητές θα χρησιμοποιηθούν στις πράξεις επιλογής και σύζευξης. Οι ευρετικές αυτές μέθοδοι χρησιμοποιούνται επίσης για να αποφασίσουν τις σχέσεις (ταξινομημένες σχέσεις τριάδων στην MonetDB) πάνω στις οποίες θα αποτιμηθούν τα πρότυπα πρόσβασης τριάδων της επερώτησης.

Επόπτης Καθηγητής: Βασίλης Χριστοφίδης
Καθηγητής

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω θερμά τον επόπτη καθηγητή μου Βασίλη Χριστοφίδη για την ιδιαίτερα ενδιαφέρουσα συνεργασία μας, για την συνεχή προθυμία του για βοήθεια καθώς και στην κατανόηση που επέδειξε καθ όλη τη διάρκεια των μεταπτυχιακών μου σπουδών. Θα ηταν μεγάλη παράληψη μου εάν δεν ευχαριστούσα την συν-επιβλέπουσα της εργασίας μου, Ειρήνη Φουντουλάκη για τις άπειρες ώρες που αφιέρωσε και την ουσιαστική συμβολή της στην περάτωση αυτής της εργασίας. Θα ήθελα επίσης να ευχαριστήσω τον Λευτέρη Σιδηρουργό για τις πολύτιμες τεχνικές και όχι μόνο, πληροφορίες μου παρείχε σχετικά με τις βάσεις δεδομένων που βασίζονται σε κολόνες.

Παράλληλα, θα ήθελα να ευχαριστήσω στον Γιάννη Τζίτζικα για την ευκαιρία που μου πρόσφερε να εργαστώ στο εργαστήριο.

Ιδιαίτερα θα ήθελα να ευχαριστήσω τον Δημήτρη Ανδρέου για τις καίριες παρατηρήσεις του, καθώς και τους Νέλη Βουζουκίδου και Κίμωνα Μπούκα για την συμβολή τους κατά την πραγματοποίηση της εργασίας αυτής. Επίσης, θα ήθελα να ευχαριστήσω όλα τα μέλη της ομάδας των πληροφοριακών συστημάτων για την ευχάριστη συνεργασία μας αλλά και για το γεγονός ότι υπήρξαν όχι μόνο συνάδελφοι αλλά και φίλοι Ακόμα, ευχαριστώ τους καλούς μου φίλους για τις υπέροχες αναμνήσεις που μου χάρισαν από τα χρόνια που πέρασα στο Ηράκλειο.

Στο σημείο αυτό θα ήθελα να ευχαριστήσω την οικογένεια Πεδιαδίτη για την απλόχερη και εγκάρδια φιλοξενία που μου πρόσφεραν όποτε την χρειάστηκα.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου Βασίλη και Κατερίνα και τα αδέλφια μου Σωτήρη και Αφροδίτη, για την υποστήριξη και την αγάπη με την οποία με περιέβαλαν.

Contents

Table of Contents iii

List of Figures v

1 Introduction 1

2 Preliminaries 4
- 2.1 RDF . . . 4
- 2.2 SPARQL . . . 5
- 2.3 Column Store Databases: The Case of MonetDB . . . 6
  - 2.3.1 Row Store databases . . . 6
  - 2.3.2 Column Stores . . . 7
  - 2.3.3 The MonetDB System . . . 8

3 SPARQL MonetDB Interpreter 11
- 3.1 Storage Scheme . . . 11
- 3.2 SPARQL Join Queries . . . 12
- 3.3 Query Optimization . . . 13
  - 3.3.1 Finding Merge Joins in the Query Plan and Assigning Ordered Relations to Triple Patterns . . . 15
  - 3.3.2 Constructing Logical Plans . . . 22
  - 3.3.3 Constructing Physical Plan . . . 24

4 Experiments 29
- 4.1 Description of Datasets . . . 29
  - 4.1.1 SP^2Bench . . . 29
  - 4.1.2 Berlin SPARQL Benchmark . . . 30
  - 4.1.3 YAGO . . . 31
  - 4.1.4 Barton . . . 31
  - 4.1.5 Summary on Datasets . . . 31
- 4.2 Description of Query Workload . . . 33
  - 4.2.1 SP^2Bench Queries . . . 34
  - 4.2.2 YAGO Queries . . . 38
- 4.3 Query Planning and Execution . . . 42
  - 4.3.1 SP^2Bench Queries . . . 43
  - 4.3.2 YAGO Queries . . . 55
- 4.4 Cost of the plans . . . 63

4.5 Concluding Remarks 64

5 Related Work **65**

5.1 Storing, Indexing and Querying RDF data 65

5.1.1 Logical Storage Schema for RDF Data 65

5.1.2 Physical Storage Schema for RDF Data 67

5.1.3 Indexes 67

5.1.4 Query Processing 70

6 Conclusions and Future Work **71**

References **72**

List of Tables

2.1 A set of RDF triples from the SP^{2}Bench SPARQL performance benchmark dataset . . . 5

3.1 A set of RDF triples from the SP^{2}Bench SPARQL performance benchmark dataset . . . 11
3.2 Dictionary . . . 12
3.3 Triple set and permutation *pos* thereof . . . 12

4.1 Characteristics of datasets . . . 30
4.2 The most 3 frequently property values of each dataset . . . 33
4.3 Query characteristics for SP2B and YAGO datasets . . . 34
4.4 Plan characteristics for SP2B and YAGO datasets . . . 40
4.5 Query Execution Time (in ms) for SP2Bench and YAGO Queries (Warm Runs) 40
4.6 The size of aggregated indexes in entries for SP2B and YAGO datasets . . . 42
4.7 The Cost . . . 63

List of Figures

2.1 Join Processing in a column-store . . . 9

3.1 Variable Graph for Queries $Q1$, $Q2$, and $Q4$ . . . 16
3.2 Variable Graph for Queries Qa, Qb and Qc . . . 19
3.3 Logical Plan for Query Qc . . . 24
3.4 Merge Join on variable $?x$ . . . 26
3.5 Physical Plan for the *selection* s0 . . . 27
3.6 Physical Plan for the *selection* s1 . . . 27
3.7 Physical Plan for the *merge join* on variable $?x$. . . . 28

4.1 Distinct subjects, properties and objects in the four datasets . . . 32
4.2 Cumulative Frequency Distribution of subjects, properties and objects in SP2B, Berlin, Barton . . . 33
4.3 Query Execution time . . . 41
4.4 **SP1** Plan by **HSP** and **CDP** . . . 43
4.5 **SP2a** Plan by **HSP** . . . 45
4.6 **SP2a** Plan by **CDP** . . . 46
4.7 **SP2b** Plan by **HSP** . . . 48
4.8 **SP2b** Plan by **CDP** . . . 49
4.9 **SP3a_2** Plan by **HSP** . . . 50
4.10 **SP3a_2** Plan by **CDP** . . . 50
4.11 **SP3b_2** Plan by **HSP** . . . 50
4.12 **SP3b_2** Plan by **CDP** . . . 51
4.13 **SP3c_2** Plan by **HSP** . . . 51
4.14 **SP3c_2** Plan by **CDP** . . . 52
4.15 **SP4a_2** Plan by **HSP** and **CDP** . . . 52
4.16 **SP4b** Plan by **HSP** . . . 53
4.17 **SP4b** Plan by **CDP** . . . 53
4.18 **SP5** Plan (**HSP** & **CDP**) . . . 54
4.19 **SP6** Plan by **HSP** . . . 54
4.20 **SP6** Plan by **CDP** . . . 54
4.21 **Y1** Plan by **HSP** . . . 55
4.22 **Y1** Plan by **CDP** . . . 56
4.23 **Y2** Plan by **HSP** . . . 57
4.24 **Y2** Plan by **CDP** . . . 58
4.25 **Y3** Plan by **HSP** . . . 59
4.26 **Y3** Plan by **CDP** . . . 60

4.27 **Y4** Plan by **HSP** 61
4.28 **Y4** Plan by **CDP** 62

5.1 Logical Storage Schemas for RDF data 66
5.2 Logical Storage Schemas for RDF data 67
5.3 Property Tables 68
5.4 SPARQL Access Patterns 68

Chapter 1

Introduction

Beyond any doubt, during the last decade we are experiencing a paradigm shift on the World Wide Web (WWW). We have witnessed a tremendous increase in the amount of semantic data that is available on the Web in almost every field of human activity. Various knowledge bases with billions of RDF triples from Wikipedia, U.S. Census, CIA World Factbook, open government sites in the US and the UK, news and entertainment sources, have been created and published online. In addition, numerous vocabularies and conceptual schemas in e-science are published nowadays as Semantic Web (SW) ontologies (in RDFS or OWL), most notably in life sciences, environmental or earth sciences and astronomy, in order to facilitate community annotation and interlinkage of both scientific and scholarly data of interest. Data published on the Web in such a way that their meaning is explicitly defined while they can be freely interlinked with others, form essentially a global space of shared data which can be exploited by a variety of applications and tools. Linked Data and Web 2.0 technologies have essentially transformed the Web from a publishing-only environment into a vibrant place for information dissemination where data is exchanged, integrated, and materialized in distributed repositories: Web users are no longer plain data consumers but have become active data producers and data dissemination agents.

In this global data space, the real added-value we can get from the available semantic data relies on our capability to access and interpret them. Most of the applications require data reshaping and integration in order to adapt them to other contexts of use than the ones they were envisioned for. Semantic Data Management (SDM) refers to a range of techniques for the manipulation and usage of data based on their meaning. In particular, we are interested in management techniques for storing, querying, and manipulating semantic data expressed in RDF, the lingua franca of linked open data and hence the default data model for Web of Data. A practical reason for investigating this problem resides in the fact that much of the freely available data represents potential new business opportunities for the industry, while their processing and management is still in its infancy especially when larger scale and complex semantic datasets are involved. In this thesis we are focusing on a core SDM problem, namely scalable processing and optimization of semantic queries using modern relational engines.

One of the main challenges in this area is related to the fine-grained character of the RDF data model featuring triples (of the form *subject-predicated-object*) rather than entire records or entities, and thus queries over semantic data stored in relational databases entail a significantly larger number of joins over entity attributes which are not always known at query compile-time (e.g. SPARQL triple patterns featuring predicate variables). Existing

native or SQL-based engines for processing SPARQL queries heavily rely on statistics regarding the stored RDF graphs as well as adequate cost based planning algorithms to optimize complex join queries. Extensive data statistics are quite expensive to compute and maintain for large scale evolving semantic data over the Web. The main challenge in this respect is to devise heuristics-based query optimization techniques generating near to optimal execution plans without any knowledge of the underlying datasets. These heuristics-based optimization techniques can be of course applied in a centralized setting but also in distributed/parallel setting such as the cloud. In this context, the main contributions of this thesis are:

- We propose the first *heuristics-based SPARQL planner* (HSP) that is capable of exploring the syntactic variations of triple patterns in a query in order to choose a near to optimal execution plan without any cost model. In particular, HSP tries to produce plans that maximize the number of merge joins over the ordered variables that are shared among triple patterns and relies on various heurists to decide which ordered variables will be used in selections and joins as well as which underlying access paths will be exploited . In this respect, we propose an original reduction of query planning to a *maximum independent set* computation over a SPARQL graph representation where nodes are query variables and edges the triple patterns connecting them. Then the qualifying independent sets are translated to blocks of merge joins connected when is needed by hash joins.

- We have implemented HSP plans on top of the *MonetDB column-based DBMS*. We have paid particular attention to the efficient implementation of HSP logical plans to the underlying MonetDB query execution engine (using the MAL physical algebra). The main challenge in this respect stems from the decomposed nature of rows in columns which incur subtle tuple reconstruction operators after the execution of several MAL operators for every HSP logical operator. In addition, to respect the maximal number of merge joins determined by HSP, we have employed *bushy* rather than *left-deep* query plans (as produced by the standard SQL optimizer of MonetDB).

- We have experimentally evaluated HSP using synthetically generated RDF datasets according to SP2B[1] [42] benchmark, as well as real RDF datasets that are widely used such as YAGO[2]. In particular, we *compare the quality* and *execution time* of the plans produced by HSP with the Cost-based Dynamic Programming (CDP) algorithm of RDF-3X [35]. In all workload queries, HSP produces plans with the same number of merge and hash joins as CDP. Their differences lie on the employed ordered variables as well as the execution order of joins which essentially affect the size of intermediate results. With the exception of queries which are not substantially different in their syntax HSP plans executed on MonetDB outperform those of CDP executed in RDF-3X up to three orders of magnitude.

Compared to existing SPARQL-engines, we believe that HSP exhibits some original features: (a) Unlike most SQL-based SPARQL engines (SW-store [7], Oracle RDF [16], 3store [24], Sesame [15], Virtuoso RDF [19]), HSP is capable to rewrite SPARQL queries in order to exploit as much as possible the ordered triple relations as well as to impose selections and join ordering using RDF-specific heuristics (the sparseness of real RDF graphs

[1] dbis.informatik.uni-freiburg.de/index.php?project=SP2B
[2] Yet Another Great Ontology: www.mpi-inf.mpg.de/yago-naga/yago

affects selectivity of operations w.r.t. the involved triple positions); (b) Rather than relying on extensive statics as in the case of most cost-based native SPARQL engines (Hexastore [53], RDF-3X [35], Yars2 [25]), these HSP statistics are proved to yield near-to-optimal bushy plans which when executed over MonedDB come with significant performance gains.

The rest of this Thesis is organized as follows. In Chapter 2 we briefly discuss the RDF data model, the SPARQL language for querying RDF data and present the basic principles of the column-store based DBMS and in particular of MonetDB. Chapter 3 discusses the Heuristic-based SPARQL Planning (**HSP**) algorithm. In Chapter 4 we discuss the results of the performance evaluation of **HSP** plans More specifically, we discuss extensively the properties of the datasets, the query workload we used in our evaluation, and the results of the comparison of the produced **HSP** and **CDP** plans on the basis of their quality and execution time in MonetDB (for the former) and RDF-3X for the latter. In Chapter 5 we present state of the art works on the subject of Semantic Data Managment.

Chapter 2

Preliminaries

In this chapter we introduce the Resource Description Framework (RDF) [20] and present the SPARQL query language [41] for querying RDF data. Finally, we discuss *column-store* DBMS and more specifically we present the MonetDB system[1].

2.1 RDF

The Resource Description Framework (RDF) [20] is a framework for representing information about Web resources. It is comprised of W3C recommendations that enable the encoding, exchange and reuse of structured data, providing means for publishing both human-readable and machine-processable vocabularies. Nowadays current W3C recommendations for RDF are used in a variety of application areas. The Linked Data initiative, which aims at connecting data sources on the Web, has already become very popular and has exposed many data sets using RDF [20] and RDFS [13]. DBpedia [2], BBC music information [30] and government datasets are only few examples of the constantly increasing Linked Data cloud.

RDF is based on a simple data model that is easy for applications to process and manipulate Web data. We state here the basic assumptions underlying RDF.

- In RDF essentially anything we wish to describe is a resource. A resource may be anything from a person to an institution, the relation a person has with an institution, a Web page, part of a Web page, an entire collection of pages or a Web site. In the context of RDF, a resource is uniquely identified by a URI (Universal Resource Identifier) [20].
- The building block of the RDF data model is a *triple*. An RDF graph is a *set of triples*. A triple is of the form *(subject, predicate, object)* where the *predicate* (also called *property*) denotes the relationship between *subject* and *object*. In our work and without loss of generality we restrict that the subject and predicate of a triple are URIs and the object can be a URI or a literal. An RDF graph can be viewed as a node and edge labeled directed graph with subjects and objects of triples being the nodes of the graph and predicates the edges.

We consider two infinite and disjoint sets $\mathbb{U}$ and $\mathbb{L}$ denoting URIs and literals respectively.

[1] MonetDB: http://www.monetdb.org/Home

[2] http://dbpedia.org

	subject(s)	*predicate*(p)	*object*(o)
t_1:	sp2b:Journal1/1940	rdf:type	sp2b:Journal
t_2:	sp2b:Inproceeding17	rdf:type	sp2b:Inproceedings
t_3:	sp2b:Proceeding1/1954	dcterms:issued	``1954''
t_4:	sp2b:Journal1/1952	dc:title	``Journal 1 (1952)''
t_5:	sp2b:Journal1/1941	rdf:type	sp2b:Journal
t_6:	sp2b:Article9	rdf:type	sp2b:Article
t_7:	sp2b:Inproceeding40	dc:terms	``1950''
t_8:	sp2b:Inproceeding40	rdf:type	sp2b:Inproceedings
t_9:	sp2b:Journal1/1941	dc:title	``Journal 1 (1941)''
t_{10}:	sp2b:Journal1/1942	rdf:type	sp2b:Journal
t_{11}:	sp2b:Journal1/1940	dc:title	``Journal 1 (1940)''
t_{12}:	sp2b:Inproceeding40	foaf:homepage	http://www.dielectrics.tld/siecle/samplings.html
t_{13}:	sp2b:Journal1/1940	dcterms:issued	``1940''

Table 2.1: A set of RDF triples from the SP^{2}Bench SPARQL performance benchmark dataset

Definition 1. An RDF *triple* (*subject*, *predicate*, *object*) is any element of the set $\mathcal{T} = \mathbb{U} \times \mathbb{U} \times (\mathbb{U} \cup \mathbb{L})$, where $\mathbb{U}$ is the set of URIs and $\mathbb{L}$ the set of literals ($\mathbb{U}$ and $\mathbb{L}$ are disjoint). A set of RDF triples is called an RDF *graph*.

It should be stressed that in this thesis, we are interested only in ground triples and thus ignore non-universally identified resources, called unnamed or blank nodes [40]. From this point on, and w.l.g. we will be using the term triples to refer to ground triples. In this context, an RDF graph is formed by a set of RDF (ground) triples.

Example 1. Table 2.1 shows a set of triples from the SP^{2}Bench [42] SPARQL performance benchmark dataset.

2.2 SPARQL

During the past years, many query languages have been proposed for the RDF data model. Some of them include RQL [29], RDQL [43], SeRQL [14], and TRIPLE [45]. SPARQL [41] is the official W3C recommendation language for querying RDF graphs and has the ability to extract information about both data and schema. SPARQL is based on the concept of matching graph patterns. The simplest graph patterns are triple patterns, which are like an RDF triple but with the possibility of a variable in any of the subject, predicate or object positions. A query that contains a conjunction of triple patterns is called basic graph pattern. A basic graph pattern matches a subgraph of the RDF graph when variables of the graph pattern can be substituted with the RDF terms in the graph.

In addition to the sets $\mathbb{U}$ and $\mathbb{L}$ (of URIs, and literals respectively) we assume the existence of an infinite set $\mathbb{V}$ of variables (disjoint from the above sets).

Definition 2. An RDF *triple pattern* (*subject*, *predicate*, *object*) is any element of the set $\mathcal{TP} = (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{L} \cup \mathbb{V})$.

A SPARQL query consists of three parts. The pattern matching part, which includes several interesting features of pattern matching of graphs, like optional parts, union of patterns, nesting, filtering (or restricting) values of possible matchings, and the possibility of choosing the data source to be matched by a pattern. The solution modifiers, which once the output of

the pattern has been computed (in the form of a table of values of variables), allow to modify these values by applying classical operators like projection, distinct, order, limit, and offset. Finally, the output of a SPARQL query can be of different types: yes/no queries, selections of values of the variables which match the patterns, construction of new triples from these values, and descriptions of resources.

The syntax of SPARQL follows an SQL-like select-from-where paradigm. The select clause specifies the variables that should appear in the query results. Each variable in SPARQL is prefixed with character "?". The graph patterns of the query are set in the where clause. Finally, a filter expression specifies explicitly a condition on query variables.

The following SPARQL query asks for the year a journal with title `''Journal 1 (1940)''` has been issued and that was revised in `''1942''`.

Listing 2.1: Query 1

```
select   ?yr, ?journal
where {  ?journal rdf:type bench:Journal .                  (tp0)
         ?journal dc:title ''Journal 1 (1940)'' .           (tp1)
         ?journal dcterms:issued ?yr .                      (tp2)
         ?journal dcterms:revised ?rev .                    (tp3)
         filter (?rev=''1942'') }
```

A SPARQL graph pattern expression is defined recursively as follows:

1. A tuple from $(\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{V}) \times (\mathbb{U} \cup \mathbb{V} \cup \mathbb{L})$ is a triple pattern (graph pattern).

2. If P1 and P2 are graph patterns, then expressions (P1 '.' P2), (P1 OPTIONAL P2), and (P1 UNION P2) are graph patterns.

3. If P is a graph pattern and R is a SPARQL built-in condition, then the expression (P FILTER R) is a graph pattern.

Built-in conditions are constructed using elements of the set $\mathbb{V} \cup \mathbb{L} \cup \mathbb{U}$, logical connectives ($\neg$, $\wedge$, $\vee$), inequality symbols ($<$, $\leq$, $\geq$, $>$), the equality symbol ($=$), unary predicates like bound, isBlank, and isIRI, plus other features (see [41] for a complete list).

In this work, we restrict to the fragment of filters where the built-in condition is a Boolean combination (only through conjunction) of terms constructed by using equality ($=$).

The answer of a SPARQL query with a select clause is a set of *mappings* where a mapping (i.e. the SPARQL analog of the relational *valuation*) is a set of pairs of the form $(variable, value)$.

Example 2. Consider the query presented in Listing 2.1 and the set of RDF triples shown in Table 2.1. The result of the evaluation of this query on this set of RDF triples is the following set of mappings:

$$\{(?yr, 1940), (?journal, \textsf{sp2bench:Journal1/1940})\}$$

2.3 Column Store Databases: The Case of MonetDB

2.3.1 Row Store databases

Traditional database systems store and process data one tuple at a time, i.e., one row of a table at a time, thereby known by the general term *row-stores*. The storage layout in a row-store is typically based on pages and its processing model is typically based on the *volcano*

ideas, i.e., the query plan is given to process one tuple at a time. Each tuple is processed by every operator in the plan, before the next tuple is examined. For example, assume the following simple query.

$$\text{select } (\text{R.b}) \text{ from } \text{R} \text{ where } \text{R.a} \geq 5$$

A query processing plan for a row-store would go one by one through all tuples of table R. For each tuple it would first call the select operator to evaluate the predicate of the where clause. In the case that the given tuple satisfies the predicate, it would then return the requested attribute.

During query processing, in order to access the next attribute of a tuple (row) we need to know how many bytes must be read from the row and from which point exactly. We must also need to call the *next* operator in the plan and be aware of the data type which can of course be different than that of the previous attribute. Then, the same process must be repeated for the next row(s) of the table.

2.3.2 Column Stores

Over the years, application needs have changed: applications (such in the case of OLAP), do not always need to process full tuples of a table. Instead, they focus on analyzing a subset of a table's attributes, in order for instance to run various aggregations for data analysis. For this kind of applications a *column-store* architecture seems more natural which lead to the design of a number of novel systems, such as MonetDB [33], MonetDB/X100 [12] and C-Store [48].

Column-oriented DBMSs store data one column at a time as opposed to one tuple at a time in the case of row stores. This type of processing allows a system to benefit a lot in terms of I/O for queries that require only a subset of a table's attributes. For example, assume a table representing students in a university's database. This table will typically consist of numerous attributes, i.e., student's first and last name, address, student ID, department, enrollment date, average grade etc. Now imagine a series of queries with a goal of analyzing the data, e.g., find all students that are enrolled at the university for at least 5 years, all students with an average grade of more than 85/100 etc. This kind of queries need to see only a subset of the table's attributes. However, in a row-store the default action would be to load the whole table from disk to memory (assuming no indexes exist). A column-store, on the other hand, needs to load only the attributes (i.e., columns in their physical representation) relevant to the query.

Another strong point of column-stores is the increased opportunities for compression [2, 56]. Physically storing together columns, brings similar data closer, i.e., data of the same type with high chances of having the same or similar values. This way, significant compression levels can be achieved. For instance, dictionary compression in a row-store would typically happen at the page level. On the other hand, the obvious drawback of a column-store setting, is the on-the-fly tuple reconstruction needed to bring the necessary columns back in a tuple format. Tuple reconstruction eventually accounts to a *join* between two columns based on tuple IDs/positions and becomes a significant cost component in column-stores especially for multi-attribute queries [26, 3, 23, 31]. For example, for each relation R_i in a query plan Q, a column-store needs to perform at least $N_i - 1$ tuple reconstruction operations for R_i within Q, given that N_i attributes of R_i participate in Q. This is obviously a tradeoff depending on various parameters, such as the number of attributes from the table that are relevant to the given query, etc.

2.3.3 The MonetDB System

In this section, we will briefly describe the MonetDB system to introduce the necessary background for the rest of our presentation. MonetDB [33] is an open-source column-store database developed in the database group of CWI. MonetDB differs from the mainstream systems in its reliance on a decomposed storage scheme, a simple (closed) binary relational algebra, and hooks to easily extend the relational engine.

In MonetDB, every n-ary relational table is represented by a group of binary relations, called BATs [11]. A BAT holds an unlimited number of binary associations, called BUNs (**B**inary **UN**its). The two attributes of a BUN are called head (left) and tail (right) in the remainder of this document. A BAT represents a mapping from an oid-key to a single attribute *attr*. Its tuples are stored physically adjacent to speed up its traversal, i.e., there are no holes in the data structure. The keys represent the identity of the original n-ary tuples, linking their attribute values across the BATs that store an n-ary table. For base tables, they form a dense ascending sequence enabling highly efficient positional lookups. Thus, for base BATs, the key column is a virtual non-materialized column. For each relational tuple t of a relation R, all attributes of t are stored in the same position in their respective column representations. The position is determined by the insertion order of the tuples. This tuple-order alignment across all base columns allows the column-oriented system to perform tuple reconstructions efficiently in the presence of tuple order-preserving operators. Basically, the task boils down to a simple merge-like sequential scan over two BATs, resulting in low data access costs through all levels of modern hierarchical memory systems. SQL statements are translated by the compiler into a query execution plan composed of a sequence of simple binary relational algebra operations. MonetDB is a late tuple reconstruction column-store: when a query is issued, the relevant columns are loaded from disk to memory but are glued together in a tuple N -ary format only prior to producing the final result. This way, intermediate results are also in a column format. In MonetDB, each relational operator materializes the result as a temporary BAT or a view over an existing BAT. Intermediates can efficiently be reused [27].

For example, assume the following query:

$$\textsf{select } R.c \textsf{ from } R \textsf{ where } 5 \leq R.a \leq 10 \textsf{ and } 9 \leq R.b \leq 20$$

This query is translated into the following (partial) plan:

```
Ra1 := algebra.select(Ra, 5, 10);
Rb1 := algebra.select(Rb, 9, 20);
Ra2 := algebra.KEYintersect(Ra1, Rb1);
Rc1 := algebra.project(Rc, Ra2);
```

Operator `algebra.select(A, v1, v2)` searches all key-attr pairs in base BAT `A` for attribute values between `v1` and `v2`. For each qualifying attribute value, the respective key value (position) is included in the result. Since selections happen on base BATs, intermediate results are also ordered in the insertion sequence. In MonetDB, intermediate results of selections are simply the keys of the qualifying tuples, thus the positions of where these tuples are stored among the column representations of the relation. In this way, given a key/position we can fetch/project (positional lookup) different attributes of the same relation from their base BATs very

fast. Since both intermediate results and base BATs have the attributes ordered in the insertions sequence, MonetDB can very efficiently project attributes by having cache-conscious reads. Operator `algebra.project(A, r)` returns all key-attr pairs residing in base BAT `A` at the positions specified by `r`. This is a *tuple reconstruction operation*. Iterating over `r`, it uses cache-friendly in-order positional lookups into `A`. Operators `algebra.KEYintersect(r1, r2)` and `algebra.KEYunion(r1, r2)` are tuple reconstruction operators that perform the conjunction/disjunction of the selection results by returning the intersection/union of keys from `r1` and `r2` . Due to order-preserving selection, both `r1` and `r2` are ordered on key. Thus, both intersection and union can be evaluated using cache-, memory-, and I/O- friendly sequential data access. The results are ordered on key, too, ensuring effcient tuple reconstructions.

One of the most interesting features of MonetDB is that the actual algorithm used for each operator is decided at the very last minute as part of the operator call, i.e., a select operator, will on-the-fly decide whether it will do a simple scan select or a binary search if it finnds out that the data is sorted. Similarly a join will decide on the fly the proper algorithm depending on the input properties. To a large degree this is one more positive side-effect of the column-based query processing, i.e., processing one column at a time in a bulk mode. This way, MonetDB can delay for example the decision of which join algorithm to use up until the moment that the join operator will be called. By then, MonetDB has fully created the join inputs and thus can take better decisions.

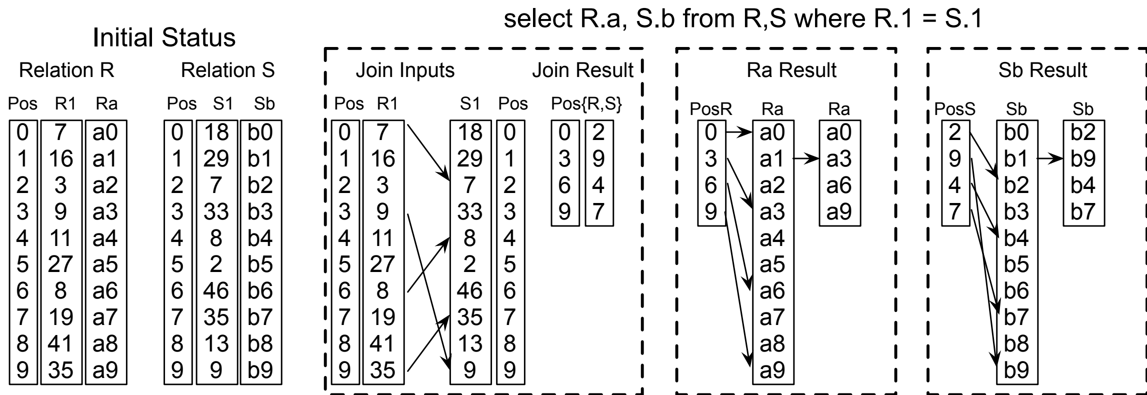


Figure 2.1: Join Processing in a column-store

The join operator in a column-store takes as input two BATs $\texttt{b1} = (\texttt{key1}, \texttt{attr1})$ and $\texttt{b2} = (\texttt{key2}, \texttt{attr2})$. The operator finds the joinable $\texttt{attr1} - \texttt{attr2}$ pairs and produces as output a new BAT that contains the qualifying $\texttt{key1} - \texttt{key2}$ pairs. These keys are then used in the remainder of the query plan to fetch the qualifying values of the necessary attributes from the base tables and in the correct order. An example of join processing can be seen in Figure 2.1 . To find the actual joinable tuples across the two columns, the operator decides on the fly the most appropriate algorithm, e.g., hashjoin, nested loops join, etc. In addition, the choice of which is considered the inner or outer input is a dynamic one taken at the operator level by exploiting the complete knowledge about the inputs, i.e., their sizes. In the execution plans that are produced by our algorithms (Section 3.3.3) we employ the operators `leftjoin`, `uselect` and `semijoin`. `leftjoin` is a special kind of join that operates on memory positions and whose right and left operands (i.e., columns) cannot be switched. `uselect` returns the

head values of all BUNs of a BAT with a certain value. Finally, the `semijoin` operator returns the intersection taken over only the head columns of the two input BATs with their tails being the tails of corresponding BUNs of the left operand.

Chapter 3

SPARQL MonetDB Interpreter

3.1 Storage Scheme

In our work we assume the RDF graph is stored in a ternary relation that contains triples of the form *(subject, property, object)*. As triples contain URIs and literals, we adopt a standard approach as in [5, 38] where each URI and literal is mapped to a unique identifier, and this mapping is stored in a relational table, referred to as a *dictionary*. Table 3.1 shows a set of triples from the SP2Bench [42] performance benchmark and Table 3.2 the *dictionary* for this set. Note that the extract cost of a join must be paid when the results for a query are returned because we need to obtain from the unique identifiers the URIs and literals.

To guarantee that we can anwer every possible triple pattern efficiently since variables may appear in every position of the triple pattern, we store in the database all possible permutations (as in the majority of the state of the art works on SPARQL query processing – see Chapter 5) of subject (s), object (o) and property (p) views of the triple table after having replaced the URIs and literals with their unique identifiers, hence obtaining the following *ordered relations*: *spo*, *sop*, *ops*, *osp*, *pos*, *pso*. Each one uses a different order for arranging its components: for instance for *spo*, the triples are sorted lexicographically first by subject (s), then by object (o) and then by property (p). We store ordered relations as regular tables

	$subject(s)$	$predicate(p)$	$object(o)$
t_1:	sp2b:Journal1/1940	rdf:type	sp2b:Journal
t_2:	sp2b:Inproceeding17	rdf:type	sp2b:Inproceedings
t_3:	sp2b:Proceeding1/1954	dcterms:issued	‘‘1954’’
t_4:	sp2b:Journal1/1952	dc:title	‘‘Journal 1 (1952)’’
t_5:	sp2b:Journal1/1941	rdf:type	sp2b:Journal
t_6:	sp2b:Article9	rdf:type	sp2b:Article
t_7:	sp2b:Inproceeding40	dc:terms	‘‘1950’’
t_8:	sp2b:Inproceeding40	rdf:type	sp2b:Inproceedings
t_9:	sp2b:Journal1/1941	dc:title	‘‘Journal 1 (1941)’’
t_{10}:	sp2b:Journal1/1942	rdf:type	sp2b:Journal
t_{11}:	sp2b:Journal1/1940	dc:title	‘‘Journal 1 (1940)’’
t_{12}:	sp2b:Inproceeding40	foaf:homepage	http://www.dielectrics.tld/siecle/samplings.html
t_{13}:	sp2b:Journal1/1940	dcterms:issued	‘‘1940’’

Table 3.1: A set of RDF triples from the SP^{2}Bench SPARQL performance benchmark dataset

Dictionary					
Oid	*Value*	*Oid*	*Value*	*Oid*	*Value*
001	sp2b:Journal1/1940	002	sp2b:Inproceeding17	003	sp2b:Proceeding1/1954
004	sp2b:Journal1/1952	005	sp2b:Journal1/1941	006	sp2b:Article9
007	sp2b:Inproceeding40	008	sp2b:Journal1/1942	009	sp2b:Journal
010	sp2b:Inproceedings	011	''1954''	012	''Journal 1 (1952)''
013	sp2b:Article	014	''1950''	015	''Journal 1 (1941)''
016	''Journal 1 (1940)''	017	www.dielectrics.tld/siecle/samplings.html	018	''1940''
019	rdf:type	020	dcterms:issued	021	dc:title
022	dc:terms	023	foaf:homepage		

Table 3.2: Dictionary

Triples							
	s	p	o		s	p	o
t_1	001	019	009	t_8	007	019	010
t_2	002	019	010	t_9	005	021	015
t_3	003	020	011	t_{10}	008	019	009
t_4	004	021	012	t_{11}	001	021	016
t_5	005	019	009	t_{12}	007	023	017
t_6	006	019	006	t_{13}	001	020	018
t_7	007	022	014				

Triples							
	p	s	o		p	s	o
t_1	019	001	009	t_2	019	002	010
t_5	019	005	009	t_6	019	006	006
t_8	019	007	010	t_{10}	019	008	009
t_{13}	020	001	018	t_3	020	003	011
t_{11}	021	001	016	t_4	021	004	012
t_9	021	005	015	t_7	022	007	014
t_{12}	023	007	017				

Table 3.3: Triple set and permutation *pos* thereof

in MonetDB, in contrast to the approach advocated in RDF-3X [38] uses clustered B+-tree indexes to store this kind information. In addition, the mapping of URIs/literals to unique identifiers is stored in two separate structures: one for URIs and one for literals. The second mapping is stored in a regular MonetDB table whereas URIs are compressed: for each token of a URI (where the delimeter used is one of ".", "/", ":") we assign and store a small number (up to 8 bits).

Table 3.3 shows the triple set after having substituted the literals and URIs with unique identifiers and the ordered relation *pos* of the considered set of triples.

3.2 SPARQL Join Queries

In this work we consider SPARQL *join* queries that are of the form:

$$\begin{array}{ll} \texttt{select} & ?v_1, ?v_2, \ldots \\ \texttt{where} & \{pattern_1 \,.\, pattern_2 \,.\, \ldots\} \end{array}$$

where $pattern_i$ is a *triple pattern* (Section 2.2), $?v_j$ denotes a variable and "." denotes the join opetation between triple patterns. For a triple pattern tp, we denote by $pos(x, tp)$ the position of x (URI, literal or variable), in triple pattern tp (one of s, o, p for *subject*, *object* and *predicate* resp.). We also define function $cond(tp)$ that returns the set of conditions represented as a pair of the form $(pos(?v, tp), c)$ where c is a constant or a variable. In a SPARQL join query, a variable that appears in multiple triple patterns implies a join between those triple patterns. To evaluate a SPARQL query, the query engine must find the variable bindings that satisfy the triple patterns, and return the bindings for the variables in the query's select clause.

Definition 3. A SPARQL join query is defined by a set of triple patterns $Q = \{ tp_0, \ldots, tp_k \}$.

We denote by $vars(Q)$ the set of variables and $ul(Q)$ the set of URIs, constants and literals in Q respectively. We overload function $vars()$ to denote the set of variables for a triple pattern tp, and function $ul()$ to denote the set of constants of a triple pattern tp. The set of variables that appear in the select clause of a SPARQL join query are called *projection variables*. We will write $pvars(Q)$ to refer to the set of projection variables of query Q. We also define weight function $\beta : V \to \mathbb{N}$ that represents the number of triple patterns a variable $?v$ appears in (minus 1) in a query Q. More specifically:

$$\beta(?v) = |\{tp \mid tp \in Q, v \in vars(tp)\}| - 1$$

We call *join (shared) variables* the variables with positive weight, whereas variables with zero-weight are called *unused variables*. Finally, we define the *number of distinct positions* of a variable $?v$ in a query Q, to be

$$dp(?u) = \left| \bigcup_{\{tp \in Q \mid ?v \in vars(tp)\}} \{pos(tp, ?v)\} \right|$$

Example 3.

Listing 3.1 shows query Q that consists of triple patterns $tp0$, $tp1$, $tp2$, $tp3$, $tp4$ and $tp5$. $?x$, $?u2$ are *projection* variables ($pvars(Q) = \{ ?x, ?u2 \}$). $?u1$, $?u2$ are *unused* variables (i.e., appear only in triple patterns $tp1$, $tp5$ resp.) with zero weight ($\beta(?u1), \beta(?u2) = 0$). $?x$ and $?y$ are *join* variables with $?x$ appearing in triple patterns $tp0$, $tp2$, $tp3$ and $?y$ in triple patterns $tp2$, $tp3$, $tp4$ and $tp5$. For ease of readability we denote constants (URIs and literals) as <>. The weight of variables $?x$, $?y$ is $\beta(?x) = 2$ and $\beta(?y) = 3$ resp. (the first appears in 3 whereas the second in 4 triple patterns). Variable $?x$ appears only at the subject position of the triple patterns $tp0$, $tp1$, $tp2$ and hence $dp(?x) = 0$ whereas variable $?y$ appears in both the *subject* and *object* positions for triple patterns $tp2$, $tp3$ so $dp(?y) = 1$.

Listing 3.1: Q

```
select  ?x, ?u2
where { ?x <> <> .      (tp0)
        ?x <> ?u1 .     (tp1)
        ?x <> ?y .      (tp2)
        ?y <> <> .      (tp3)
        ?y <> <> .      (tp4)
        ?y <> ?u2 }     (tp5)
```

3.3 Query Optimization

In this section we present our approach for producing the *query plans* for the SPARQL queries we consider in this thesis. As discussed in Chapter 1, we propose a framework for the efficient evaluation of SPARQL queries over a *column store database*. Due to the fine-grained nature of RDF data where information is stored in the form of triples, instead of records, queries over RDF data involve a large number of joins which form the largest part of the query workload. RDF data does not come with schema or integrity constraints, so that a query optimizer can take advantage of those to produce an efficient query plan. In addition to that,

join variables are not as predictable as in the case of relational data where joins are *explicitly* specified in the `where` clause of an SQL query. The above set different requirements for RDF query processing. In this thesis, we advocate a *heuristic-based* approach for the construction of query plans over *column-store* databases. These heuristics can be distinguished into (i) *relational-specific* (ii) *RDF-specific* and (iii) *column-store* specific ones.

H0: maximize the number of merge joins in the query plan. This heuristic is a relational specific one but can be also applied in our context since we use a relational schema (i.e., triple table) as the logical schema to store the RDF graph.

H1: Promote the triple patterns that have the *most* number of literals and URIs. This heuristic is similar to the *bound as easier heuristic* of relational and datalog query processing, according to which, the more bound components a triple pattern has, the more selective it is [50, 51]. This heuristic helps us in considering triple patterns for joins that have higher selectivity that when used as join operands reduce the *size of the join's result.* This heuristic is also related to the *selection* operator since it helps us to push the selections as early as possible in a query plan.

H2: Promote the triple patterns that have most number of literals. For this heuristic we consider the *selectivity* of constants, and more precisely we assume that *literals are more selective than URIs.* This heuristic, similar to the previous one, helps us in choosing selective triple patterns in the case of joins and push the selections as early as possible in a query plan.

H3: The different positions in which the same variable appears in a set of triple patterns captures the number of different joins this variable participates in. A variable that appears always in the same position in all triple patters, for example as subject, entails many self joins with low selectivity. On the other hand, if it appears both as object and property, chances are the join result will be smaller. The following precedence relation captures this preference:

$$p \bowtie o \prec s \bowtie p \prec s \bowtie o \prec o \bowtie o \prec s \bowtie s \prec p \bowtie p$$

where s, p, o refer to the subject, property, and object position of the variable in the triple pattern. This ordering stems from our observations while studying RDF data graphs as shown in Table 4.1 (Section 4.1.5). RDF data graphs tend to be sparse with a small diameter, while there are hub nodes, usually subjects. As a result, query graph patterns that form linear paths are more selective.

H4: Promote the set of triple patterns that have the the *least* number of *projection variables*. This heuristic allows us to consider as late as possible the triple patterns that contain projection variables so to promote *late tuple reconstruction.* In the case in which the compared sets of triple patterns have the same set of projection variables, we prefer the set with the maximum number of unused variables that are not projection variables.

H5:
Given the position and the number of variables in a triple pattern we derive the following order, starting from the most selective, i.e., the one that is likely to produce less intermediate results, to the least selective.

$$s\ p\ o \prec s\ ?p\ o \prec ?s\ p\ o \prec s\ p\ ?o \prec$$
$$?s\ ?p\ o \prec s\ ?p\ ?o \prec ?s\ p\ ?o \prec ?s\ ?p\ ?o$$

The above ordering is based on the observation that given a subject and an object there only very few, if not only one, properties that can satisfy the triple pattern. Similarly, it is very rare that a combination of a subject and property has more than one object value. In the same line of thinking we derive the rest of the orders. An exception to this rule is when the property has the value rdf:type, since that is a very common property and thus these triples should not be considered as selective.

3.3.1 Finding Merge Joins in the Query Plan and Assigning Ordered Relations to Triple Patterns

Our main objective is to produce query plans with the *maximum number of merge joins* (heuristic **H0**). In order to achieve this, we need to select (a) the appropriate join variable as well as the the appropriate ordered relation (access path) among the six available ones (*spo*, *sop*, *ops*, *osp*, *pos*, *pso*) so that the evaluation of a triple pattern returns RDF triples in an order that will enable the use of merge joins.

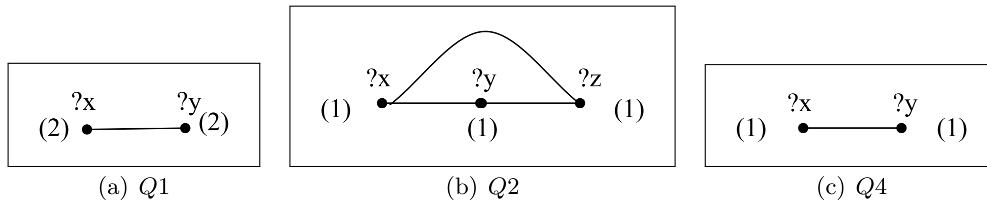
To implement heuristic **H0**, we reduce the problem of finding the maximum number of merge joins to the problem of finding the *maximum weight independent sets* in a graph. In graph theory, an *independent set* is a set of *vertices* in a graph, *no two of which share an edge* [22]. If each vertex of a graph G is assigned a positive integer (the weight of the vertex) the *maximum weight independent set problem* consists in finding *independent sets of maximum total weight* which is an NP-hard problem in general [21] and remains NP-hard even under restrictions in the forms of graphs. However, an RDF join graph is small enough (a couple to maybe few tens of vertices) that an independent set can be easily found in a few milliseconds in modern hardware.

The idea of reducing our problem to the problem of finding independent sets with maximum weight is the following: maximizing the number of merge joins can be translated into finding groups of maximum size of triple patterns that can be joined on the same variable. To do this, we represent the query as a *variable graph* where *(i)* nodes in the graph are the query variables, *(ii)* two nodes are connected if they belong to the same triple pattern and *(iii)* a node carries a weight which is the number of triple patterns it appears in minus 1. Consequently, the nodes in this graph that are returned as members of an independent set, are the variables for which we will define the merge joins. The time complexity of variable graph construction is $O(|n|)$, where n is the number of triple patterns.

In the following we will present the algorithms we use to decide *i)* the merge joins (Algorithm FindAccessPaths()), and *ii)* the ordered relations that will be used to evaluate the query's triple patterns (Algorithm AssignOrderedRelations()). First, we will discuss the notion of *variable graph* that we will use extensively in our algorithms.

Definition 4. Let Q be a set of RDF triple patterns. The variable graph $G(Q)$ is a weighted node labeled graph $G(Q) = (V, E, \beta)$ where *(i)* V is the set of nodes, *(ii)* E is the set of edges, $E \subseteq V \times V$, and *(iii)* β is the weight function for each node in V. More specifically, for each variable $?v$ in $vars(Q)$, we add a node $?v$ in V if $\beta(?v) \neq 0$, with weight $\beta(?v)$. An edge $(?v, ?u)$ is defined between nodes $?v$, $?u$ iff there exist some triple pattern tp_k in Q where, $\{v_i, v_j\} \in vars(tp_k)$.

Example 4. Figure 3.1(a) shows the variable graph obtained for query $Q1$ (Listing 3.2). We choose to variable's name to denote the corresponding node in the graph, and the node's weight is denoted in parentheses next to it. Note that for query $Q1$, variables $?u1$, $?u2$ do not appear in the query's graph since they are *unused* variables (have zero weight). The variable graph for query $Q2$ (Listing 3.3) is shown in Figure 3.1(b). Query $Q2$ contains a *cycle*: see that a path is formed (whose start and end node are the same variable – $?x$). Nevertheless, the cycle is not captured in the variable graph. Query $Q3$ (Listing 3.4) consists of a single triple pattern where the same variable is used in two different positions. This is a special case of a cycle with an *empty* variable graph. Note that the graph does not contain any nodes per definition since variable $?x$ belongs to a single triple pattern (in two different positions). Note that there are no self-loop edges in a variable graph. Finally, query $Q4$ (Listing 3.1(c)) contains two triple patterns that are joined by two variables and whose query graph is shown in 3.1(c). From the queries discussed here we can see that the variable graph represents only information about joins.



(a) $Q1$ (b) $Q2$ (c) $Q4$

Figure 3.1: Variable Graph for Queries $Q1$, $Q2$, and $Q4$

Listing 3.2: $Q1$

```
select  ?x ?y
where { <> <> ?x.   (tp0)
        ?x <> ?u1 .  (tp1)
        ?x <> ?y .   (tp2)
        ?y <> ?u2 .   (tp3)
        ?y <> <> }   (tp4)
```

Listing 3.3: $Q2$

```
select  ?x ?y
```

```
where { ?x <> ?y.    (tp0)
        ?y <> ?z .   (tp1)
        ?z <> ?x }   (tp2)
```

Listing 3.4: $Q3$

```
select  ?x
where { ?x <> ?x.}   (tp0)
```

Listing 3.5: $Q4$

```
select  ?x
where { ?x <> ?y .   (tp0)
        ?x ?y <> }   (tp1)
```

Algorithms FindAccessPaths() and AssignOrderedRelations() use function $AccPath$

$$AccPath : \mathcal{TP} \rightarrow \mathcal{P} \times \mathbb{V}$$

where $\mathcal{TP}$= ($\mathbb{U} \cup \mathbb{V}$) $\times$ ($\mathbb{U} \cup \mathbb{V}$) $\times$ ($\mathbb{U} \cup \mathbb{L} \cup \mathbb{V}$) denotes the set of triple patterns, $\mathcal{P}$= { *spo*, *sop*, *ops*, *osp*, *pos*, *pso*} the set of ordered relations and $\mathbb{V}$ the set of variables. Function $AccPath$ stores for a triple pattern tp the ordered relation it will be evaluated on, and tp's variable whose values will be returned sorted. Function $AccPath$ is represented as a Map with as key a triple pattern from the set $\mathcal{TP}$ and value an element in the set $\mathcal{P} \times V$. We write $AccPath.get(tp).rep$ to denote the ordered relation that triple pattern tp will be evaluated on, and $AccPath.get(tp).sort$ to denote the variable of tp whose values will be returned sorted by the evaluation.

Algorithm FindAccessPaths() takes as input a SPARQL join query Q and returns map $AccPath$ whose keys are the triple patterns in Q. It first applies Algorithm CMWIS() [39] (lines $7-8$) (implementing heuristic **H0**) that computes the maximum weight independent sets for query Q, that is the variables that will be used to perform the merge joins. It is the case that for some queries, there are multiple independent sets with the maximum weight. In this case, instead of choosing randomly one of the independent sets, heuristics **H1-H4** are applied in this order to eliminate candidates.

In the case in which more than one independent sets are returned, FindAccessPaths() applies first heuristic **H1** using function $argmax_{IS \in SV} \mid ul(IS) \mid$ to eliminate a subset of the obtained independent sets (lines $9-11$). Function $argmax$ stands for the argument of the maximum, that is the set of points of the given argument for which the value of the given expression attains its maximum value. In our case, function $ul(IS)$ denotes the set of constants of the triple patterns associated with the variables in independent set IS.

If at this step, more than one independent sets remain, Algorithm FindAccessPaths() applies heuristic **H2** (lines $12-14$) using function $argmax_{IS \in SV} \mid literals(IS) \mid$ where $literals(IS)$ denotes the sets of literals of the triple patterns associated with the variables in independent set IS. If, after the application of **H2**, more than one independent sets are left, Algorithm FindAccessPaths() applies heuristic **H3** (lines $15-17$) through function $argmax_{IS \in SV}\ varsdp(IS)$ where $varsdp(IS)$ is a function defined as follows: $varsdp(IS) = \mid \{\forall ?v \in IS, dp(?v)\} \mid$. In other words, it returns the number of distinct positions for each variable in an independent set. If multiple independent sets remain, then we employ the precedence relation between join patterns as discussed in heuristic **H3**. After the application

Algorithm 1: FindAccessPaths

```
   Input: Query Q
   Output: AccPath : TP × P × V
 1 bestCand ← 0 be a set of sets of variables;
 2 SV be a set of sets of variables;
 3 V be a set of variables ;
 4 T ← Q be a set of triple patterns ;
 5 while T ≠ ∅ do
 6     SV ← 0, V ← 0;
 7     Let G(T) be the variable graph constructed from the set of triple patterns in T ;
 8     SV ← CMWIS(G(T)) ;
 9     if (| SV |> 1) then
10         /* apply heuristic H1 */;
11         SV ← argmax_{IS∈SV} | ul(IS) | ;
12         if | SV |> 1 then
13             /* apply heuristic H2 */;
14             SV ← argmax_{IS∈SV} | literals(IS) | ;
15             if | SV |> 1 then
16                 /* apply heuristic H3 */;
17                 SV ← argmax_{IS ∈ SV} varsdp(IS);
18                 if | SV |> 1 then
19                     /* apply heuristic H4 */;
20                     SV ← argmin_{IS ∈ SV} | pvars(IS) | ;

21     /* select randomly one of the sets */;
22     V ← select one of the sets in SV ;
23     /* update the set of variables*/;
24     bestCand ← bestCand ∪ {V} ;
25     /* Update the set of triple patterns to contain only the triple patterns that do not
       contain some variable in the selected set of variables */;
26     T = T \ {tp | T, vars(tp) ∩ V ≠ ∅} ;
27 /* compute structure AccPath */;
28 forall v ∈ bestCand do
29     /* select the triple patterns that the variable appears in that have not been assigned
       an ordered relation */;
30     T = {tp | v ∈ vars(tp), tp ∉ AccPath.keys()} ;
31     forall tp ∈ T do
32         AssignOrderedRelations(tp, v) ;
33 /* assign to the remaining triple patterns an ordered relation according to the
   selectivity heuristic */ ;
34 forall tp ∈ Q, tp ∉ AccPath.keys() do
35     ρ(n) = AssignOrderedRelations(tp, nil) ;
```

of heuristic **H3** and if still multiple independent sets remain, we apply heuristic **H4** (lines 18 − 20) using function $argmin_{IS \in SV} \mid pvars(IS) \mid$ where function $pvars(IS)$ returns the set of the projection variables in the triple patterns the variables in IS belong to. Function $argmin$ stands for the argument of the minimum, that is the set of points of the given argument for which the value of the given expression attains its minimum value. In the case in which multiple independent sets still remain we choose randomly one of the resulting sets (lines 21 − 26) and we remove from the set of input triple patterns the ones that the selected variables participate in. The process is repeated until there are no triple patterns left for consideration.

To select the ordered relation that will be used to evaluate a triple pattern *tp*, we iterate over all variables in the selected independent set, and for each of the variables, we compute the triple patterns the variable belongs in (lines 28 − 30).

Then Algorithm AssignOrderedRelations() is called for each triple pattern in this set and the corresponding variable, to compute the ordered relation that will return the triples that match the triple pattern ordered on the input variable (lines 31 − 32). After all the variables in the independent sets found have been considered, Algorithm AssignOrderedRelations() is called for each triple pattern to which no ordered relation has been assigned (lines 34 − 35).

It takes as input a triple pattern *tp* of query Q, a variable u of *tp* and the $Map\ AccPath$ that contains the assignment of triple patterns to access paths.

The algorithm updates Map *AccPath* by taking under consideration heuristic **H5** and the position of variable u in the input triple pattern. In the case that no variable is provided, we choose the ordered relation that evaluate constants first (lines 3 − 13) otherwise, we promote the ordered relation that returns the triples ordered on the value of variable u (lines 14 − 25).

Example 5.

```
select  ?x, ?y
where { ?x <> <> .     (tp0)
        ?x <> ?u1 .    (tp1)
        ?x <> ?y .     (tp2)
        ?y <> <> .     (tp3)
        ?y <> <> }     (tp4)
```

Listing 3.6: Query Qa

Consider query Qa shown in Listing 3.6 whose variable graph is shown in Figure 3.2.

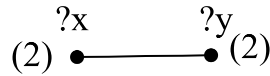


Figure 3.2: Variable Graph for Queries Qa, Qb and Qc

When applying the maximum weight independent sets algorithm CMWIS( ) on Qa's variable graph, we obtain the following independent sets

$$IS_1 \quad = \quad \{?x\} \quad and \quad IS_2 \quad = \quad \{?y\}$$

When applying heuristic **H1**, we select independent set IS_2 since $ul(IS_1) = 4$, whereas $ul(IS_2) = 5$: variable $?x$ belongs in triple patterns *tp*0, *tp*1 and *tp*2 with in total 4 constants,

Algorithm 2: AssignOrderedRelations

```
   Input: Triple Pattern tp of Query Q, Variable v of tp, Map AccPath
1  Let pos(x, tp) denote the position of variable or constant x in the triple pattern tp;
2  Let (value_1, value_2, value_3) denote an orderered relation from P;
3  if (v = nil) then
4      /* not interested in any variable */ ;
5      /* choose the ordered relations that evaluate the constants (URIs and literals) first
       */;
6      if (| ul(tp) |= 2) then
7          Let v be the variable of triple pattern tp;
8          AccPath.put(tp, ((value_1, value_2, pos(tp, v)), v));
9      end
10     if (| ul(tp) |= 1) then
11         Let v_1, v_2 be the variables of triple pattern tp
           AccPath.put(tp, ((value_1, pos(tp, v_1), pos(tp, v_2)), v_1));
12     end
13 else
14     /* if tp is of the form (?x, ?y, ?z) choose the ordered relation that returns the
       triples ordered for v */;
15     if (| vars(tp) |= 3) then
16         AccPath.put(tp, ((pos(v, tp), value_2, value_3), v));
17     end
18     if (| vars(n) |= 2) then
19         /*if tp is of the form (?x, ?y, c) or (c, ?x, ?y) or (?x, c, ?y) with c a URI or
           literal, choose the ordered relation where c is evaluated first */ ;
20         AccPath.put(tp, ((pos(c, tp), pos(v, tp), value_3), v));
21     end
22     if (| vars(n) |= 1) then
23         /*if tp is of the form (?x, c_1, c_2) or (c_1, ?x, c_2) or (c_1, c_2, ?x), with c_1, c_2 URIs
           or literals, choose the ordered relations where c_1, c_2 are evaluated first according
           to the heuristics */ ;
24         AccPath.put(tp, ((value_1, value_2, pos(v, tp)), v));
25     end
26 end
```

and variable $?y$ belongs in triple patterns $tp2$, $tp3$ and $tp4$ with total 5 constants. As a result, we obtain set IS_2.

Query Qb is similar to query Qa with the exception of triple pattern $tp3$ where the constant value for the object component is replaced with an unused variable ($?u2$). Qb's variable graph is the same as Qa's (depicted in Figure 3.2). Hence, we obtain again the same maximum weight independent sets IS_1 and IS_2 as before.

The application of heuristic **H1** returns both sets IS_1 and IS_2 since $ul(IS_1) = ul(IS_2) = 4$. Sets IS_1 and IS_2 are also returned after the application of heuristic **H2** since $literals(IS_1) = literals(IS_2) = 1$. When applying heuristic **H3** we obtain set IS_2 since the number of distinct positions for variable $?y$ is higher than the number of distinct positions for variable $?x$. More specifically, the sole appearance of a join pattern for $?x$ is $s = s$ ($tp0$, $tp1$, $tp2$), that is $dp(?x) = 0$, whereas the join patterns for variable $?y$ are $o = s$ ($tp2$, $tp3$ and $s = s$ ($tp3$, $tp4$) that is $dp(?y) = 1$. Hence independent set IS_2 is selected.

```
select  ?x, ?y
where { ?x url1 lit1 .      (tp0)
        ?x url2 ?u1 .       (tp1)
        ?x url3 ?y .        (tp2)
        ?y url4 ?u2 .       (tp3)
        ?y url5 lit2 }      (tp4)
```

Listing 3.7: Query Qb

Finally, query Qc shown in Listing 3.8 is a variation of query Qb where for triple pattern $tp0$, variable $?x$ is found at the object instead of the subject position. The structure of the queries is the same, and consequently they have the same variable graph (shown in Figure 3.2). Consequently, we obtain again the same maximum weight independent sets IS_1 and IS_2 as before.

The application of heuristics **H1**, **H2** and **H3** does not eliminate any of the obtained independent sets (same sets of constants (4), literals (0), distinct variable positions (1)). The application of heuristic **H4** returns independent set IS_2 since $pvars(IS_2) = 1$ (the projected variable is $?u2$ that belongs in triple pattern $tp3$ to which variable $?y$ belongs).

```
select  ?u2
where { url1 url2 ?x .     (tp0)
        ?x url3 ?u1 .      (tp1)
        ?x url4 ?y .       (tp2)
        ?y url5 ?u2 .      (tp3)
        ?y url6 url7 }     (tp4)
```

Listing 3.8: Query Qc

To see how the assignment of ordered relations to triple patterns is done, consider query Qc, and the independent set obtained $IS_2 = \{?y\}$. The triple patterns to which variable $?y$ belongs to are $T = \{tp2, tp3, tp4\}$. Algorithm AssignOrderedRelations() populates $AccPath$ structure as follows:

$$AccPath(tp2, (pos, ?y)) \quad AccPath(tp3, (pso, ?y)) \quad AccPath(tp4, (pos, ?y))$$

Consider the case of triple pattern $tp2$: Algorith AssignOrderedRelations() will select ordered relation *pos* since (a) $tp2$'s predicate is constant, and (b) the values for variable $?y$ (which is in the *object (o)* position ordered must be returned ordered. For triple pattern $tp3$,

AssignOrderedRelations() selects ordered relation pso because (a) $tp3$'s predicate is constant and (b) we must return the values for variable $?y$ (which is in the *subject (s)* position ordered. Finally, for triple pattern $tp4$, ordered relation pos is selected due to the application of heuristic **H5**.

The set of triple patterns to which variable $?x$ belongs to and to which no ordered relation has been assigned is $T' = \{tp0, tp1\}$ for which AssignOrderedRelations() updates structure $AccPath$as follows:

$$AccPath(tp0, (pso, ?x)) \quad AccPath(tp1, (pso, ?x))$$

3.3.2 Constructing Logical Plans

In this section we discuss Algorithm LPlanConstruction() that takes as input a query Q and the $AccPath$ structure obtained from Algorithm FindAccessPaths(), and returns the *root* node of the plan. It employs *multimap* $VN : \mathbb{V} \rightarrow N$ that stores for each variable in the query the node of the plan is associated with (through a *selection*, *scan* or *join* operation). First, the algorithm creates the *scan* and *selection* nodes in the plan (lines $3 - 11$). It iterates over all triple patterns that are keys in the $AccPath$ structure, and creates a *scan* node (line 5) that carries the following information *i)* the ordered relation to be used for the evaluation of triple pattern tp, *ii)* the variable whose values will be returned sorted from tp's evaluation, *iii)* a set of pairs of the form $(pos(u, tp), u)$ for each variable u in tp ($nvar$) If there are conditions associated with the triple pattern, then a *selection* node is created that carries *i)* the set of tp's conditions ($cond$) *ii)* a pointer to the child *scan node* and *iii)* all information stored in the scan node (lines $7 - 8$). Algorithm LPlanConstruction updates structure VN by iterating over all variables in triple pattern tp and adding for each variable the plan node (either selection or scan node) that has been created for this triple pattern (lines $10 - 11$).

In the following step, LPlanConstruction computes the *hash* and *merge* join nodes in the plan (lines $13 - 38$). It iterates over all variables that have an entry in multimap VN and constructs the join nodes. First, it finds all the plan nodes constructed so far whose *sort* is variable v and adds those nodes in set S (lines $16 - 19$). If there are more than one such plan nodes, a join node is created, whose condition is variable v, and children the nodes in set S. Multimap VN is then updated accordingly. In the following, the algorithm examines whether some of the children nodes of the join node have a common variable, different from the variable that is examined. In this case, the common variable is added to the join node's condition (lines $23 - 32$). After all the merge joins have been created, the Algorithm creates the hash joins, with the difference that all the nodes that contain each variable are selected, rather than just those which are ordered on the examined variable. In the case where the set of all the values of VN contains more than one value, we create cartesian products between the different nodes (lines $35 - 37$). Finally, a *projection node* is created using the projection variables of the query ($pvars(Q)$). The time complexity of LPlanConstruction algorithm is $O(|v^2|)$, where v is the number of shared variables of the query.

Example 6. Consider query Qc discussed earlier and the $AccPath$ structure obtained for this query:

$$AccPath(tp0, (pso, ?x)) \quad AccPath(tp1, (pso, ?x))$$
$$AccPath(tp2, (pos, ?y)) \quad AccPath(tp3, (pso, ?y)) \quad AccPath(tp4, (pos, ?y))$$

Figure 3.3 shows the obtained logical plan for query Qc. At the first step, *scan nodes* $scan0$,

Algorithm 3: LPlanConstruction

```
   Input: Query Q, AccPath : TP → P × V
   Output: root
1  Let VN be a multimap VN : 𝕍 → N with N a Node in the Logical Plan;
2  /* Calculate scan and selection nodes */;
3  foreach tp ∈ AccPath.keys() do
4      nvar ← {∀ v ∈ vars(tp), (pos(v, tp), v)} ;
5      P ← new Node_scan(AccPath.get(tp).rep, AccPath.get(tp).sort, nvar);
6      cond ← cond(tp);
7      if (| c |> 0) then P ← new Node_selection(cond, P.sort, nvar, P) root ← P;
8      foreach (v ∈ vars(tp)) do VN.put(v, P)
9  /* Calculate join nodes */;
10 hashjoinsFlag = false;
11 while true do
12     foreach v ∈ VN.keySet() do
13         S ← ∅;
14         foreach P ∈ VN.get(v) do
15             if (hashjoinsFlag ∨ P.sort == v) then S ← S ∪ P;
16         if (| S |> 1) then c ← v;
17         if (!hashjoinsFlag) then
18             joinNode ← new Node_mergeJoin(c, S);
19         else
20             joinNode ← new Node_hashJoin(c, S);
21         root ← joinNode;
22         VN.put(v, joinNode);
23         foreach P ∈ S do VN.remove(v, P);
24         foreach v_2 ∈ VN.keySet() do
25             if (v_2 ≠ v) then
26                 foreach P ∈ VN.get(v_2) do
27                     foreach P_2 ∈ P ∩ S do
28                         VN.remove(v_2, P_2);
29                         VN.put(v2, joinNode);
30                         if (∃ P_1, P_2 ..., P_k nodes with k > 1, P_i = P_j) then
31                             VN.remove(v2, P_i), i = 1, ..., k − 1;
32                             joinNode.cond.append(v_2);
33     if (hashjoinsFlag == true) then break;
34     hashjoinsFlag = true;
35 /* Calculate cartesian product node */;
36 CP = {P | P ∈ VN.values()};
37 if | CP |> 1 then root ← new Node_cartesian_product(CP);
38 /* Calculate projection node */;
39 root ← new Node_projection(pvars(Q), root);
```

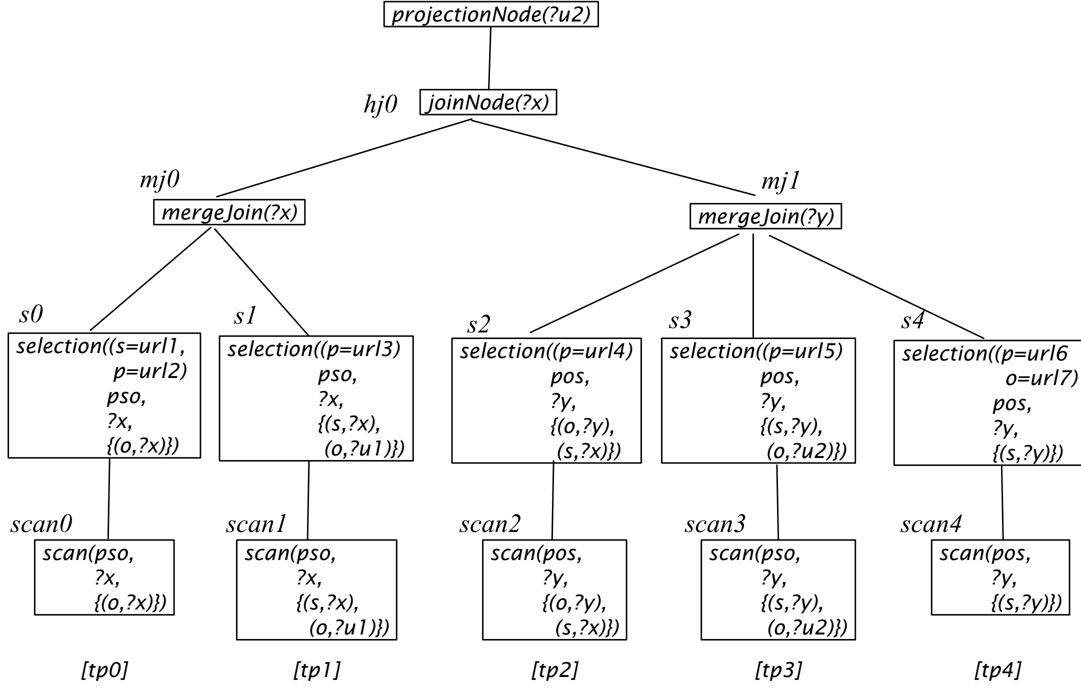


Figure 3.3: Logical Plan for Query Qc

$scan1$, $scan2$, $scan3$ and $scan4$ for triple patterns $tp0$, $tp1$, $tp2$, $tp3$ and $tp4$ respectively. Note that each scan node stores information related to the ordered relation the triple pattern will be evaluated on, the variable whose values will be returned sorted and the set of (pos, var) pairs for each variable in the triple pattern. Consider for instance triple pattern $tp3$: node $scan3$ is created for it that carries the following information: *i)* *pso* the ordered relation on which $tp3$ will be evaluated on ($AccPath.get(tp3).rep$), *ii)* variable $?y$ whose values will be returned sorted by the evaluation ($AccPath.get(tp3).sort$) and the set $\{(s, ?y), (o, ?u2)\}$ that designates the position in the triple pattern of each of its variables. Since all triple patterns of Qc contain constants, selection nodes $s0$, $s1$, $s2$, $s3$ and $s4$ are created. Note that each such node contains the same kind of information as the scan nodes, in addition to the *conditions* specified in the triple pattern. For example, in the case of $tp3$, we have added condition $(p = url5)$ since $tp3$ contains a single constant $url5$ at the predicate position. In the third step, the join nodes are created as follows: we first select the selection nodes whose sort variable is $?x$ which in our case are $s0$ and $s1$. The *merge join* node $mj0$ is created with condition variable $?x$ whose children nodes are nodes $s0$ and $s1$. In a similar manner, *merge join* node $mj1$ is created with condition variable $?y$ and whose children nodes are selection nodes $s2$, $s3$, $s4$. Then, the *hash join* node $hj0$ is created whose condition is variable $?x$. Finally, the *projection* node π is created whose condition is variable $?u2$.

3.3.3 Constructing Physical Plan

In this section we will present through an example the construction of the physical plan to be executed by MonetDB. It takes as input the logical plan created by Algorithm LPlanConstruction and converts the relational operators into operators of the MAL algebra.

Algorithm 4: algoMALsel

Input: A scan or selection node n
Output: Physical plan

1 **for** $i \leftarrow 0$ **to** $\|n.cond\| - 1$ **do**
2   **if** $i = 0$ **then**
3     $smal \leftarrow new\ Node_{bind}(n, n.cond[i], pos)$;
4     $smal \leftarrow new\ Node_{uselect}(smal, n.cond[i].value)$;
5     $tmpMAL \leftarrow smal$;
6   **else**
7     $smal \leftarrow new\ Node_{bind}(n, n.cond[i].pos)$;
8     $smal \leftarrow new\ Node_{semijoin}(smal, tmpN)$;
9     $smal \leftarrow new\ Node_{uselect}(smal, n.cond[i].value)$;
10     $tmpMAL \leftarrow smal$;
11 **foreach** $v \in n.nvar$ **do**
12   $smal \leftarrow new\ Node_{bind}(n, v.pos)$;
13   $smal \leftarrow new\ Node_{leftjoin}(smal, tmpN)$;
14   $n.Map.put(v.value, smal)$;

Algorithm 5: algoMALjoin

Input: A join node n
Output: MAL node

1 $jmal \leftarrow newNode_{join}(n.left.Map.get(n.cond.var), n.right.Map.get(n.cond.var))$;
2 **foreach** $v \in keys(n.left.Map)$ **do**
3   $tmpMAL \leftarrow new\ Node_{leftjoin}(n.left.Map.get(v), n)$;
4   $jmal.Map.put(v, tmpMAL)$;
5 **foreach** $v \in keys(n.right.Map)v$ **do**
6   $tmpMAL \leftarrow new\ Node_{leftjoin}(n.right.Map.get(v), n)$;
7   $jmal.Map.put(v, tmpMAL)$;

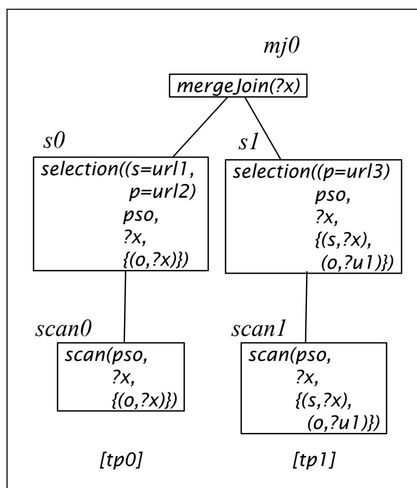


Figure 3.4: Merge Join on variable $?x$

bottom-up. Algorithms ScanSelectionToMAL() and JoinToMAL() construct the MAL commands which implement the physical selection and join operators. Due to the difficulty of the MAL language, we choose to present graphically the physical plan.

Consider the fragment of the logical plan for query Qc shown in Figure 3.4. This subplan corresponds to the merge join executed for variable $?x$. Using as input the selection node s0, bind node n1 is created, for the *subject* column of the ordered relation *spo*. Above this node, the uselect n2 is created which filters the returned values with value $url1$. The result of this filtering is used in a semijoin (node n4) with the bind node n3 which binds to *property* column. The semijoin gives as output one BAT that has as head the intersection of heads of the two entries and as tail the corresponding BUNs of the left operand. Then, the uselect node n5 filters the tail values of the semijoin output by applying the second condition (with value $url2$). Finally, the leftjoin node n7 is constructed that joins the results of this filtering with the bind node n6 that binds the *object* column from the ordered relation SPO using the values of variable $?x$, as we can see in Figure 3.5. In a similar manner, the values for variable $?x$ are obtained from the s1 selection node. In addition, projection variable $?u1$ is taken from the s1 node as shown in Figure 3.6. Finally, the resulting columns of the two selections on $?x$ are joined and in order to obtain the values of $?x$ an additional leftjoin node is constructed (see Figure 3.7). The same operations are performed for variable $?u1$.

o))'W

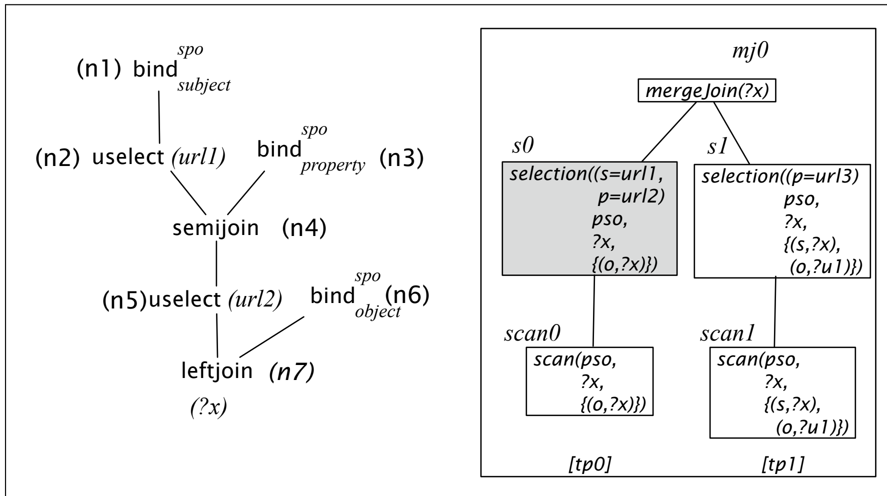


Figure 3.5: Physical Plan for the *selection* s0

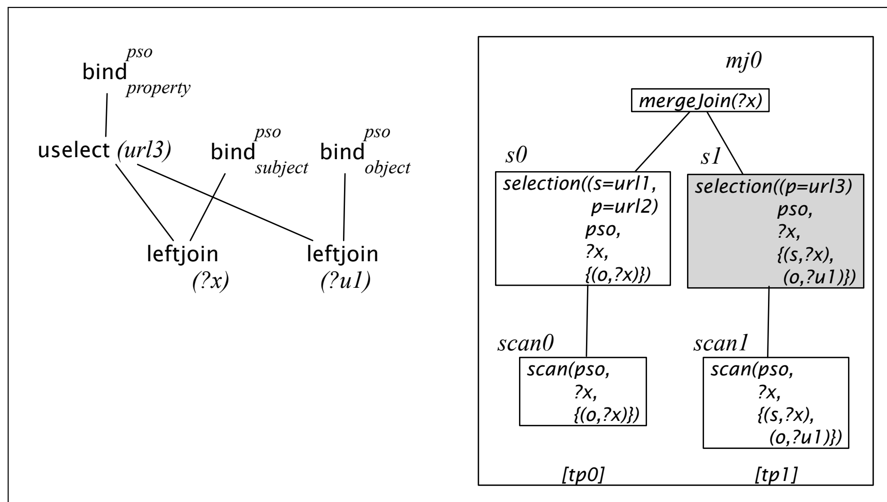


Figure 3.6: Physical Plan for the *selection* s1

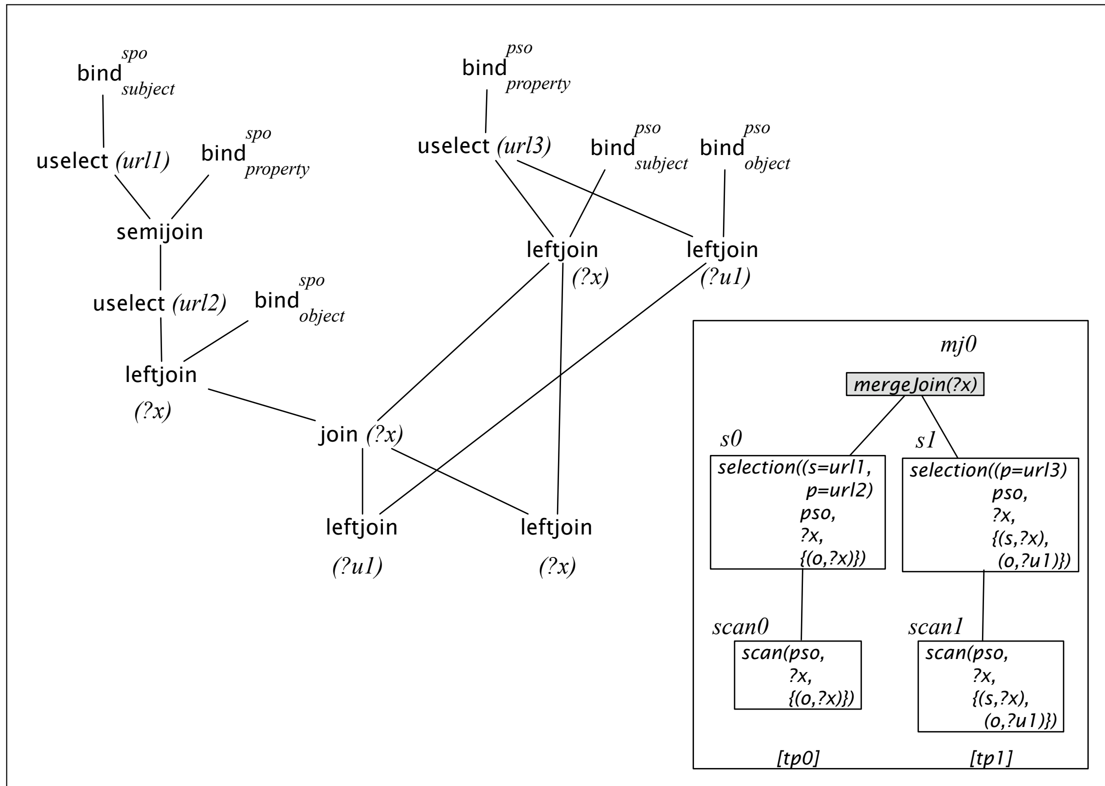


Figure 3.7: Physical Plan for the *merge join* on variable $?x$.

Chapter 4

Experiments

In this chapter we experimentally compare our Heuristic-based SPARQL Planning (HSP) algorithm with the Cost-based Dynamic Programming (CDP) algorithm of RDF-3X [35] and with the relational optimizer of MonetDB-SQL, using synthetically generated according to SP2B[1] [42] and Berlin[2] [10] benchmarks, as well as real RDF datasets that are widely used such as YAGO[3] and Barton[4] [6]). In order to better grasp the quality of the plans actually chosen for execution by the two planners, in the first part of this Chapter we analyze the main statistical properties of the four RDF datasets. The finding regarding the characteristics of the encoded RDF graphs confirm our underlying assumptions regarding the *selectivity* of *subject-property-object* components in a *single* SPARQL triple pattern as well as the selectivity of *join* between them. Then, we also detail the characteristics of the query workload we employ for the comparison of the quality and the execution time of the plans when using the above datasets.

All experiments were conducted in a Dell OptiPlex 755 desktop with CPU Intel Core 2 Quad Q6600 2.4GHz with 8MByte L2 cache, 8 GBytes of memory, and HDD 250 GByte, running Ubuntu 11.04 2.6.38-8-generic x86_64 Operating System. We used the MonetDB5 v11.2.0 (64-bit) database system to execute the HSP plans. This version of MonetDB was extended with the Redland Raptor v1.9.0[5] parser to parse the RDF triples and was compiled with gcc v4.5.2 C/C++ compiler. We also used the Redland RASQAL v0.9.20[6] system for parsing the SPARQL queries. CDP runs on the RDF-3X system version 0.3.5.

4.1 Description of Datasets

4.1.1 *SP^{2}Bench*

The first RDF dataset used in our measurements was produced using the *SP^{2}Benchmark* (SP2B) generator [42]. SP2B is motivated by the DBLP Digital Library containing bibliographic information in the area of Computer Science and in particular databases and programming languages. The generated RDF datasets mirror key characteristics and graph

[1] http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B
[2] http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark
[3] Yet Another Great Ontology: http://www.mpi-inf.mpg.de/yago-naga/yago
[4] http://simile.mit.edu/rdf-test-data/barton
[5] http://librdf.org/raptor/
[6] http://librdf.org/rasqal/

distributions encountered in the original DBLP catalog. The dataset has been generated using the command *"sp2b_gen -t 50000000"* and contains 50.000.869 distinct triples stored in file of 5.5G in the Turtle syntax. Because the Turtle parser of the Redland Raptor library employed by the MoneDB loader can only handle files up to 2GB, we have converted in the RDF-XML format the generated SP2B datasets. As we can see in Table 4.1 this dataset is composed of 27.452.995 distinct *strings* out of which 8.698.103 are URIs and 18.754.892 literals. The distinct *subjects*, *properties* and *objects* are 8.698.023, 77 and 23.439.187 respectively. *Subject-to-Subject* joins (row "$s = s$") yield six orders of magnitude less results than *Property-to-Property* (row "$p = p$") and five orders of magnitude less than *Object-to-Object* joins (row "$o = o$"). From the rest of the join patterns, *Subject-to-Object* (row "$s = o$") yields 62M results out of which 4M of them appears to be common distinct values (row "**#distinct** $s = o$") between subjects and objects. As expected property values are disjoint from subject and object values (rows "$p = o$" and "$s = p$").

We have additionally tried to load 100M of SP2B triples but the RDF-3X loader failed silently with the message *"data plus intermediate query results was larger than the virtual address space of our platform"*.

Dataset	**SP2B**	**YAGO**	**BSBM**	**Barton**
Type of data	Synthetic data	Real Data	Synthetic Data	Real Data
#Triples	50.000.869	16.348.563	25.000.244	78.497.317
#Strings	27.452.995	10.565.990	4.995.218	19.345.306
#URIs	8.698.103	10.565.990	2.260.404	13.505.849
#Literals	18.754.892	0	2.734.814	5.839.457
#Distinct subjects	8.698.023	4.339.591	2.258.129	12.326.602
#Distinct properties	77	91	40	285
#Distinct objects	23.439.187	8.396.118	4.287.021	16.939.942
Join Patterns				
# $s = s$	436.969.953	165.937.025	342.027.096	23.819.257.567.057
# $s = p$	0	9.396.314	0	0
# $s = o$	62.652.184	48.560.142	133.518.384	7.988.424.009.325
# $p = p$	303.442.470.589.817	65.992.300.043.771	36.574.104.603.766	761.331.104.427.921
# $p = o$	0	0	0	0
# $o = o$	31.433.696.872.183	30.079.265.139	3.974.837.898.676	64.565.928.543.619
#distinct $s = p$	0	82	0	0
#distinct $s = o$	4.684.292	2.169.728	1.549.972	9.921.523
#distinct $p = o$	0	0	0	0
# $o = o$ (*o is literal*)	563.726.087.151	0	1.432.020.045.552	15.282.104.707.856
# $o = o$ (*o is uri*)	30.869.970.785.032	30.079.265.139	2.542.817.853.124	49.283.823.835.763

Table 4.1: Characteristics of datasets

4.1.2 Berlin SPARQL Benchmark

The second synthetic dataset of our measurements was generated by the Berlin SPARQL Benchmark (BSBM) [10]. BSBM relies on an RDFS schema for e-commerce applications in which a set of products is offered by different vendors while consumers post reviews about products. As shown in Table 4.1, this dataset is composed of 25.000.244 distinct triples with 4.995.218 distinct string values out of which 2.260.404 are URIs and 2.734.814 literals. The underlying RDF-graph of this dataset is the densest among the others of our study. In particular, the distinct subjects are 2.258.129, a bit less than half of the distinct objects which

are 4.287.021. Like in other datasets, very few distinct properties are used. *Subject-to-Subject* joins yield five orders of magnitude less results than *Property-to-Property* and four orders of magnitude less than *Object-to-Object* joins. Subjects and objects share 1.5M distinct values while the join between them yields 133M results.

4.1.3 YAGO

YAGO (Yet Another Great Ontology) [1] was the third dataset containing facts extracted from Wikipedia and integrated with the WordNet thesaurus. In order to load this dataset into MonetDB some of the original YAGO triples were modified. First, given that Redland Raptor is stricter than the RDF parser employed by the RDF-3X loader, we had to manually clean up some of the invalid characters contained in the URIs of this dataset. Then, we eliminated the resulting duplicate triples. Last, since the RDF parser of the RDF-3X triple loader does not distinguish between URI $<A>$ and literal "A'', we converted all literals of the original YAGO dataset to URIs using as prefix the base URI of the corresponding RDF-XML file. This modification was necessary to guarantee that the same query yields the same results when evaluated in MonetDB and RDF-3X. After all these changes the size of the RDF-XML file was 1.5GB containing 16.348.563 distinct triples with 10.565.990 distinct strings (see Table 4.1). The underlying RDF-graph of this dataset is the sparsest among the others of our study. As we have already mentioned, all strings are URIs in the modified dataset. The distinct properties are 91, the distinct subjects are 4.339.591 while distinct objects are almost twice as much, 8.396.118. *Subject-to-Subject* joins yield five orders of magnitude less results than *Property-to-Property* and four orders of magnitude less than *Object-to-Object* joins. Subjects and objects share 2M distinct values while the join between them yields 49M results. Properties share common values only with subjects whose join yield 9,4M of results.

4.1.4 Barton

We finally included in our measurements the Barton Library RDF dataset [6] which is generated by converting to RDF the Machine Readable Catalog (MARC) catalog of the MIT Barton Libraries. The dataset is composed of 78.497.317 triples with 19.345.306 distinct strings out of which 13.505.849 are URIs and 5.839.457 literals (see Table 4.1). As a matter of fact, it is the dataset with more URIs than literals (with the exception of YAGO where we manually converted all literals to URIs). The distinct subjects, properties and objects are 12.326.602, 285 and 16.939.942, respectively. As a matter of fact, this dataset had the largest ratio of distinct values per subject than per object. *Subject-to-Subject* and *Object-to-Object* joins yield results of the same order of magnitude and only one order of magnitude less than the results of *Property-to-Property.* Properties do not share common values with subjects and objects. *Subject-to-Object* joins yields almost 8T of results with almost 10M matching distinct values.

4.1.5 Summary on Datasets

We can observe that in any of the four datasets triples share common values for properties and objects. Properties and subjects appear to share common values only in YAGO. The join between the subjects and objects of triples always return (significant) fewer results that the joins made on the same triple component. More precisely, joins on properties yield results that are 1 to 2 orders of magnitude larger than joins only on subjects or objects. The former

usually produce one order of magnitude smaller results than the latter. Yago data was the sparsest RDF-graph while BSBM data was the densest one. Literals were the majority of strings in both SP2Bench and BSBM. These findings confirm our heuristic related to the *ordering of join patterns based on their selectivity*

$$p = o \prec s = p \prec s = o \prec s = s \prec o = o \prec p = p. \text{ (Heuristic } \mathbf{H3}\text{)}$$

where $x \prec y$ denotes that the join pattern x is prefered to join pattern y when the selectivity thereof is concerned.

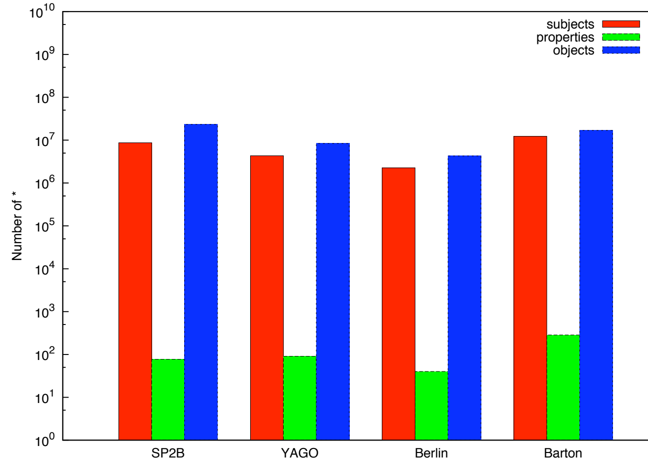


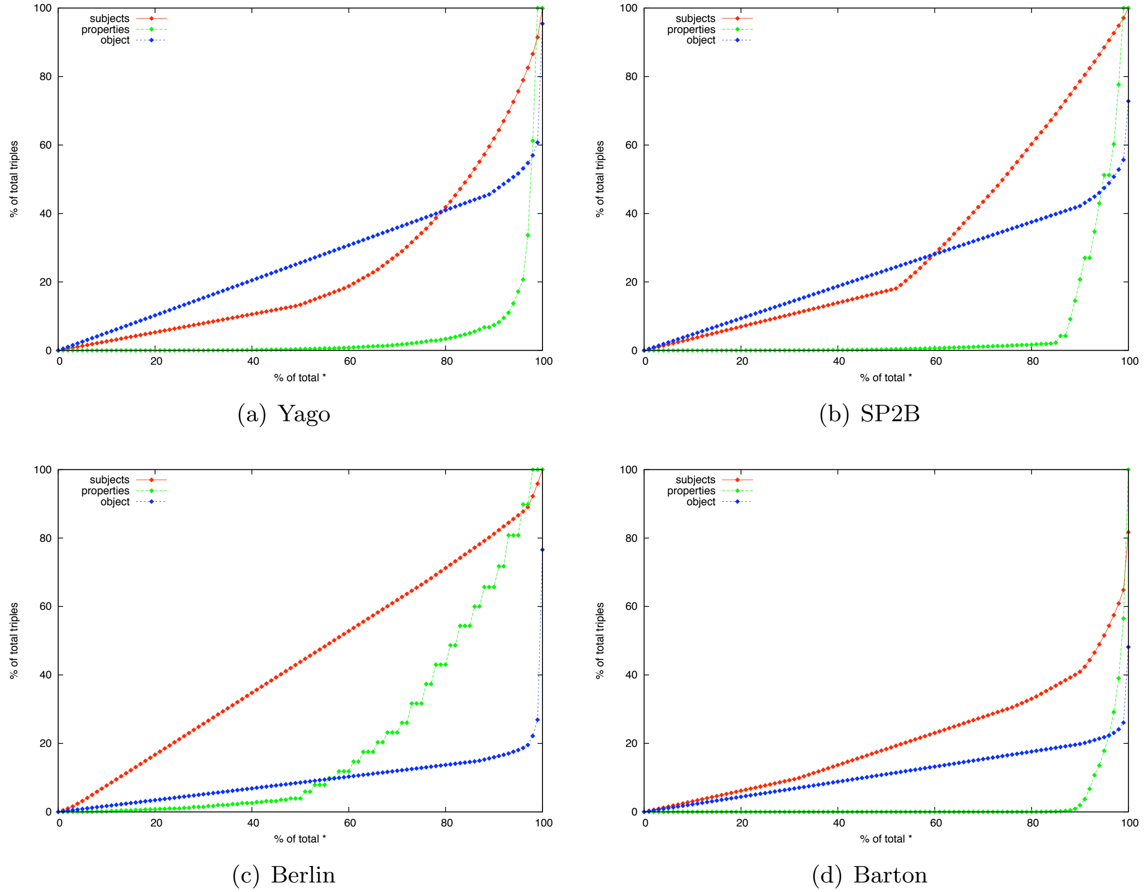
Figure 4.1: Distinct subjects, properties and objects in the four datasets

As can be seen in Figure 4.1, in all datasets, the number of distinct property values is very low. In three out of four datasets this number does not exceed 100. Only in the Barton dataset we have encountered 285 distinct properties. In all datasets, the number of distinct subject values is smaller than the number of distinct object values. We can therefore confirm our heuristic related to the selectivity of triples using conditions on subjects, properties and objects:

$$selectivity(property) < selectivity(subject) < selectivity(object) \; (Heuristic\ \mathbf{H5})$$

Figure 4.2 shows the cumulative frequency distribution of the Yago, Berlin, Barton and SP2B datasets. As we can observe from the cummulative frequency distribution of strings in the Yago dataset depicted in Figure 4.2(a), 50% of the property values return less than 5% of the triples in total. This percentage go up roughly to 90% for the other three datasets (see Figure 4.2(b), 4.2(c), 4.2(d)).

Table 4.2 illustrates the three most frequent property values appearing in the four datasets of our experiment. We can observe that `rdf:type` value is either the most or the second most frequently occurring property value. This confirms our heuristic that considers `rdf:type` as the least selective property among those appearing in SPARQL triple patterns.



(a) Yago (b) SP2B

(c) Berlin (d) Barton

Figure 4.2: Cumulative Frequency Distribution of subjects, properties and objects in SP2B, Berlin, Barton

Rank	SP2B			YAGO		
1st	dc:creator	11.0M	22.3%	rdfs:label	6.3M	38.8%
2nd	**rdf:type**	8.6M	17.4%	**rdf:type**	4.5M	27.6%
3rd	foaf:name	4.5M	9.1%	yago:describes	2.1M	13.0%

Rank	BSBM			Barton		
1st	**rdf:type**	2.5M	10.2%	**rdf:type**	22.9M	29.2%
2nd	dc:publisher	2.2M	9.0%	modsrdf:value	11.2M	14.3%
3rd	dc:date	2.2M	9.0%	modsrdf:name	5.1M	6.5%

Table 4.2: The most 3 frequently property values of each dataset

4.2 Description of Query Workload

To benchmark the plans produced HSP and CDP, we have chosen to evaluate six conjunctive queries (and variations thereof) from SP^{2}Bench [42] and four queries from YAGO [1] benchmarks. As can be seen in Table 4.3 these queries involve a different number of triple patterns,

variables and constants featuring selections as well as different kinds of joins among them (i.e., *star-* and *chain-shaped*) on different columnar positions (i.e. *s, p, o*). Variables which are not shared among triple patterns (i.e., join variables), or appear in SPARQL projections and filters are *unused*. We consider join queries that have different *structural characteristics* (i.e., kind of joins) and queries whose triple patterns have different *syntactic* characteristics (i.e, number of constants and shared variables and their positions). We observed that the majority of the queries for both datasets considered $s \bowtie s$ joins (suggesting star-shaped joins on the *subject* component of the triple pattern), followed by $s \bowtie o$ joins. The smaller the ratio of shared variables over triple patterns, the heaviest are the star-shaped joins defined on the corresponding position of the triple pattern. This is the case of queries SP2a and SP2b, followed by query Y1.

Query	SP1	SP2a	SP2b	SP3(abc)_2	SP4a	SP4b	SP5	SP6	Y1	Y2	Y3	Y4
# Triple Patterns	3	10	8	2	6	5	1	1	8	6	6	5
# Variables	2	10	8	2	5	5	2	1	6	4	7	7
# Projection Variables	2	1	1	1	2	2	2	1	2	1	1	3
# Shared vars	1	1	1	1	5	4	0	0	4	3	3	4
# TPs with 0 const	0	0	0	0	0	0	0	0	0	0	2	3
# TPs with 1 const	1	9	7	1	4	3	1	0	6	3	2	0
# TPs with 2 const	2	1	1	1	2	2	0	1	2	3	2	2
# TPs with 3 const	0	0	0	0	0	0	0	0	0	0	0	0
# Joins	2	9	7	1	5	4	0	0	7	5	5	4
Maximum star join[7]	2	9	7	1	1	1	0	0	4	3	2	1
Join Patterns												
# $s = s$	2	9	7	1	2	2	0	0	4	3	3	1
# $p = p$	0	0	0	0	0	0	0	0	0	0	0	0
# $o = o$	0	0	0	0	1	0	0	0	0	0	0	0
# $s = p$	0	0	0	0	0	0	0	0	0	0	0	0
# $s = o$	0	0	0	0	2	2	0	0	3	2	2	3
# $p = o$	0	0	0	0	0	0	0	0	0	0	0	0

Table 4.3: Query characteristics for SP2B and YAGO datasets

4.2.1 *SP^2Bench* Queries

Listing 4.1: **Query SP1**

```
select  ?yr, ?journal
where { ?journal rdf:type bench:Journal .           (tp0)
        ?journal dc:title ``Journal 1 (1940)'' .    (tp1)
        ?journal dcterms:issued ?yr }               (tp2)
```

SP1 query contains three triple patterns (tps) which share one variable on the subject position. It essentially involves three selections[8] on the constants of these tps and two joins. Evaluated over the SP2B dataset of 50M triples, the selection on tp0 evaluates to 16.701 triples, on tp1 to 1 triple and on tp2 to 3.139.378 triples. Finally, the result of the join over the results of selections on tp0 and tp1 returns a single triple triple.

[7]Signifies the number of triple patterns that participate in the star join with the largest number of triples.

[8]We consider that *one* selection operation is defined per triple pattern independently of the number of URIs and literals (i.e., constants) it contains.

Listing 4.2: **Query SP2a**

```
select  ?inproc
where { ?inproc rdf:type bench:Inproceedings .  (tp0)
        ?inproc dc:creator ?author .            (tp1)
        ?inproc bench:booktitle ?booktitle .    (tp2)
        ?inproc dc:title ?title .               (tp3)
        ?inproc dcterms:partOf ?proc .          (tp4)
        ?inproc rdfs:seeAlso ?ee .              (tp5)
        ?inproc swrc:pages ?page .              (tp6)
        ?inproc foaf:homepage ?url .            (tp7)
        ?inproc dcterms:issued ?yr .            (tp8)
        ?inproc bench:abstract ?abstract . }    (tp9)
```

SP2a query contains ten triple patterns which share a common variable in the subject position and it is the largest query of our workload (w.r.t. number of involved triple patterns). It involves ten selections on the corresponding triple patterns and nine joins forming a *star-shaped* query. The main characteristic of this query is related to the size of intermediate join results: with the exception of the selection operation on tp9 (resulting to 40.564 triples) and tp1 (returning 11.166.057 triples) the selections on the other patterns yield intermediate results of the same order of magnitude (between 3M to 4M triples). So the join ordering in the query plan will be decisive for the execution time of this query.

Listing 4.3: **Query SP2b**

```
select  ?inproc
where { ?inproc rdf:type bench:Inproceedings . (tp0)
        ?inproc bench:booktitle ?booktitle .   (tp1)
        ?inproc dc:title ?title .              (tp2)
        ?inproc dcterms:partOf ?proc .         (tp3)
        ?inproc rdfs:seeAlso ?ee .             (tp4)
        ?inproc swrc:pages ?page .             (tp5)
        ?inproc foaf:homepage ?url .           (tp6)
        ?inproc dcterms:issued ?yr . }         (tp7)
```

SP2b is essentially the same as the previous query (i.e., SP2a) from which the second (tp1) and last (tp9) triple patterns have been removed. Recall that tp1 has the lowest (returning 11,166,057 triples) while tp9 the highest selectivity (returning only 40,564 triples) among the query triple patterns and significantly reduces the final join results. Query **SP2b** instead contains eight triple patterns (tps) which yield intermediate results of the same order of magnitude (between 3M to 4M triples). Consequently, we expect that the ordering of joins in the produced query plans will not play an important role in query execution times.

Listing 4.4: **Query SP3a_1**

```
select  ?article
where { ?article rdf:type bench:Article .  (tp0)
        ?article ?property ?value          (tp1)
filter  (?property=swrc:pages)}
```

Listing 4.5: **Query SP3a_2**

```
select  ?article
where { ?article rdf:type bench:Article .  (tp0)
        ?article swrc:pages ?value }       (tp1)
```

Listing 4.6: **Query SP3b_1**

```
select  ?article
where { ?article rdf:type bench:Article .   (tp0)
        ?article ?property ?value           (tp1)
filter  (?property=swrc:month)}
```

Listing 4.7: **Query SP3b_2**

```
select  ?article
where { ?article rdf:type bench:Article .   (tp0)
        ?article swrc:month ?value }        (tp1)
```

Listing 4.8: **Query SP3c_1**

```
select ?article
where { ?article rdf:type bench:Article .   (tp0)
        ?article ?property ?value           (tp1)
filter (?property=swrc:isbn)}
```

Listing 4.9: **Query SP3c_2**

```
select ?article
where { ?article rdf:type bench:Article .   (tp0)
        ?article swrc:isbn ?value}          (tp1)
```

Queries **SP3a_1, SP3b_1, SP3c_1** contain two triple patterns sharing one variable on their subjects (i.e. one join), and one filter expression that affects the selectivity of triple pattern tp1. Queries **SP3a_2, SP3b_2, SP3c_2** are equivalent to the previous ones, with the exception that the constraint specified in the filter expression has been "pushed" in triple pattern tp1. This group of queries is useful to evaluate selections defined inside and outside SPARQL triple patterns (through filter expressions). Note that the selection on tp0 results to 1,001,081 triples while if we ignore the filters we need to scan for tp1 the entire triple relation (comprised of 50M triples).

Listing 4.10: **Query SP4a_1**

```
select  ?person ?name
where { ?article rdf:type bench:Article .         (tp0)
        ?article dc:creator ?person .             (tp1)
        ?inproc rdf:type bench:Inproceedings .    (tp2)
        ?inproc dc:creator ?person2 .             (tp3)
        ?person foaf:name ?name .                 (tp4)
        ?person2 foaf:name ?name2                 (tp5)
filter  (?name=?name2)}
```

Listing 4.11: **Query SP4a_2**

```
select  ?person ?name
where { ?article rdf:type bench:Article .        (tp0)
        ?article dc:creator ?person .            (tp1)
        ?inproc rdf:type bench:Inproceedings .   (tp2)
        ?inproc dc:creator ?person2 .            (tp3)
        ?person foaf:name ?name .                (tp4)
        ?person2 foaf:name ?name }               (tp5)
```

Query **SP4a_1** contains six triple patterns sharing six variables (i.e. five joins), and one SPARQL FILTER expression specifying an additional join. The query will allow us to benchmark the benefit of an early filter evaluation (i.e. along with the processing of the triple patterns) by allowing us to avoid a costly cartesian product computation (which is the case in the plan executed by RDF-3X). The equivalent syntactic form without the join condition specified by the SPARQL filter is given by query **SP4a_2**.

Listing 4.12: **Query SP4b**

```
select  ?person ?name
where { ?article rdf:type bench:Article .         (tp0)
        ?article dc:creator ?person .             (tp1)
        ?inproc rdf:type bench:Inproceedings .    (tp2)
        ?inproc dc:creator ?person .              (tp3)
        ?person foaf:name ?name }                 (tp4)
```

SP4b is an equivalent query to **SP4a** considering that the value of property `foaf:name` is a key for persons. It consists of five triples patterns sharing three out of four variables. Hence, it involves five selections on the constants of corresponding triple patterns and five joins which form a complex star and chain-shaped query. Note that the query involves both *Subject-to-Subject*, *Subject-to-Object* and *Object-to-Object* join patterns. It returns a final result of 4,033,222 triples.

Listing 4.13: **Query SP5**

```
select  ?s ?p
where { ?s ?p person:Paul\_Erdoes }      (tp0)
```

Listing 4.14: **Query SP6**

```
select  ?ee
where { ?publication rdfs:seeAlso ?ee } (tp0)
```

Queries **SP5, SP6** allow us to benchmark simple selections over property and object values of a unique triple pattern. **SP5** has high selectivity returning only 565 triples while **SP6** low selectivity returning 2,679,321 triples.

4.2.2 YAGO Queries

Listing 4.15: **Query Y1**

```
select ?GivenName ?FamilyName
where { ?p yago:hasGivenName ?GivenName .                   (tp0)
        ?p yago:hasFamilyName ?FamilyName .                 (tp1)
        ?p rdf:type yago:wordnet_scientist_110560637 .      (tp2)
        ?p y:bornIn ?city .                                 (tp3)
        ?city yago:locatedIn yago:Switzerland .             (tp4)
        ?p yago:hasAcademicAdvisor ?a .                     (tp5)
        ?a y:bornIn ?city2 .                                (tp6)
        ?city2 yago:locatedIn yago:Germany .}               (tp7)
```

Query **Y1** consists of eight triple patterns featuring four shared and two unused (`?GivenName`, `?FamilyName`) variables. It involves both *Subject-to-Subject* and *Subject-to-object* join patterns. The main characteristic of this query is that the selections on the constants of the tps yield intermediate results which vary by four orders of magnitude (range of 44 to 569K of triples). The joins of this query form a star and chain-shaped constellation. The star-shaped join consists of five joins on variable `?p`.

Listing 4.16: **Query Y2**

```
select  ?a
where { ?a rdf:type y:wordnet_actor_109765278 .      (tp0)
        ?a y:livesIn ?city .                         (tp1)
        ?a y:actedIn ?m1 .                           (tp2)
        ?m1 rdf:type y:wordnet_movie_106613686 .     (tp3)
        ?a y:directed ?m2 .                          (tp4)
        ?m2 rdf:type y:wordnet_movie_106613686 . }   (tp5)
```

Y2 consists of six triple patterns featuring three shared and one unused (`?city`) variable. The query involves both *Subject-to-Subject* and *Subject-to-Object* join patterns. Similar to query **Y1**, **Y2** forms star and chain-shaped joins. The star-shaped join consists of three joins on variable `?a`. Unlike the previous query, the triple patterns now yield intermediate results whose size is of the same order of magnitude (between 14K to 30K) and returns 14,705 triples.

Listing 4.17: **Query Y3**

```
select ?p
where { ?p ?ss ?c1 .                                    (tp0)
        ?p ?dd ?c2 .                                    (tp1)
        ?c1 rdf:type y:wordnet_village_108672738 .      (tp2)
        ?c1 y:locatedIn ?X .                            (tp3)
        ?c2 rdf:type y:wordnet_site_108651247  .        (tp4)
        ?c2 y:locatedIn ?Y . }                          (tp5)
```

Query **Y3** consists of six triple patterns two of which (tp0, tp1) do not include any literal or URI and thus require scanning the entire triple relation (16,348,563 triples). The size of intermediate results of the other three triple patterns is of the same order of magnitude (ranging from 17K to 70K). The query forms two star-shaped joins on variables `?c1` and `?c2`, consisting of three and two triple patterns resp., and one chain query on variable `?p`. The query result consists of 432 triples.

Listing 4.18: **Query Y4**

```
select  ?p1,?predicate,?p2
where { ?p1 ?u1 ?c1                    (tp0)
        ?c1 rdfs:label y:Paris         (tp1)
        ?p1 ?predicate ?p2             (tp2)
        ?p2 ?u2        ?c2             (tp3)
        ?c2 rdfs:label y:Hong Kong     (tp4)
```

Query **Y4** consists of five triple patterns three of which do not include any literal or URI. The query involves two selections on variables (triple patterns tp1 and tp4) and three scans over the entire triple relation (triple patterns tp0, tp2 and tp3) as well as one chain-shaped join that involves all the query's triple patterns. It is the lengthiest chain-shaped query used in our experiments.

Query	SP1	SP2a	SP2b	SP3(abc)_2	SP4a	SP4b	SP5	SP6	Y1	Y2	Y3	Y4
HSP												
Merge Joins	2	9	7	1	3	2	0	0	5	3	4	2
Hash Joins	0	0	0	0	2	2	0	0	2	2	1	2
Type of Plan	LD	LD	LD	LD	B	B	LD	LD	B	LD	B	B
CDP												
Mergejoin	2	9	7	1	3	2	0	0	5	3	4	2
Hashjoin	0	0	0	0	2	2	0	0	2	2	1	2
Type of Plan	LD	LD	LD	LD	B	B	LD	LD	B	B	B	B
Similar Plans	√	×	×	√	√	×	√	√	×	×	√	×

LD : Left Deep Tree, B : Bushy Tree

Table 4.4: Plan characteristics for SP2B and YAGO datasets

	MonetDB-HSP	RDF-3X	MonetDB-SQL
SP1	19.52	0.25	11.92
SP2a	3,267.01	355.50	3,561.07
SP2b	1,035.12	1,000.75	1,103.64
SP3a	80.92	85.14	82.91
SP3b	8.74	11.95	9.61
SP3c	12.55	13.97	14.81
SP4a	3,602.09	3,634.60	—
SP4b	1,766.29	2,781.75	1,909.13
SP5	0.06	0.10	0.09
SP6	0.43	22.85	0.48
Y1	6.04	15.75	7.69
Y2	8.65	9.95	9.07
Y3	25.69	81.20	538.65
Y4	2.32	90.45	1,113.24

Table 4.5: Query Execution Time (in ms) for SP2Bench and YAGO Queries (Warm Runs)

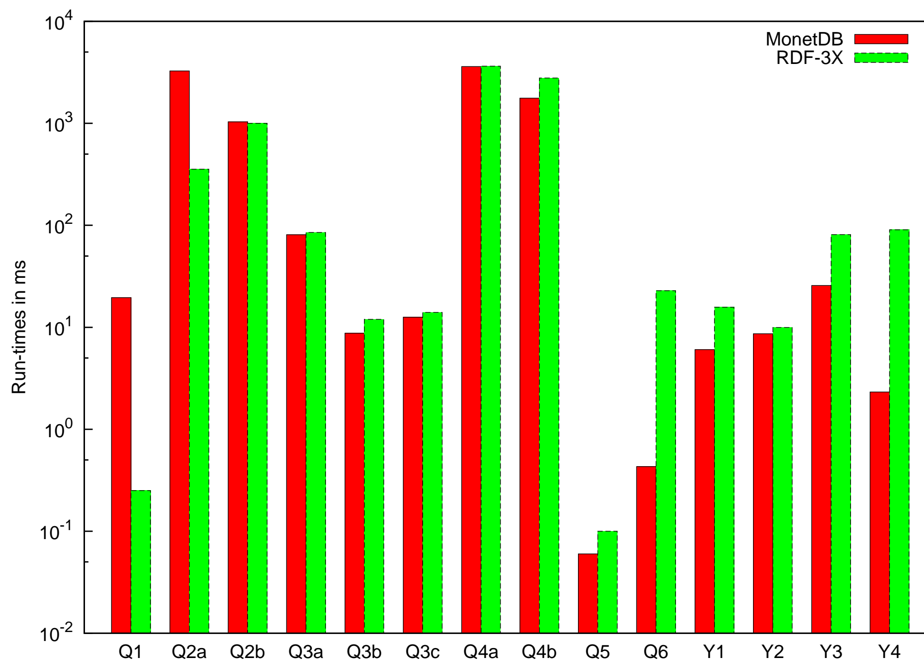


Figure 4.3: Query Execution time

4.3 Query Planning and Execution

In this section we present the plans chosen by the **HSP** and **CDP** planners for the two sets of queries described previously and report the time required to execute them in MonetDB and RDF-3X. The reported query execution times (see Tables 4.5 do not include the planning time, the time to transform the constants of every triple pattern to ids as well as the conversion of these ids back to strings in the final query result. To speed up the resolution of the ids to URIs/literals and decompression thereof, RDF-3X sorts and groups the query results to decompress only one element per group of duplicates. This time is not included in our measurements. We should point out that the planning time in both **HSP** and **CDP** is minimal compared to the execution time of each query. The percentage of planning time is about 4% of the total execution time of a query. To obtain the execution time we run each query 21 times. We ignored the time of the first execution (cold) and calculated the mean of the other 20 (warm) executions.

In the query plans we use the symbols $\bowtie^{mj}_{var}$ and $\bowtie^{hj}_{var}$ to denote *merge* and *hash* joins respectively between relations that store the values for variable var. We write $\sigma_{cond}(R)$ to denote a *selection* operation with condition *cond* on relation R. In the case of **HSP** R is one of the six access paths *spo*, *sop*, *ops*, *osp*, *pso*, *pos* and in the case of **CDP** can be either one of the aforementioned access paths or one of the aggregated indexes defined on either the *subject* (S), *object* (O) and predicate (P) position of a triple pattern or a binary combination thereof (SO, OS, SP, PS, PO, OP). When a triple pattern contains one or more unsued (unshared) and unprojected variables, CDP chooses to evaluate it on one of the aggregated indexes. The aggregated indexes store only the two out of the three columns of a triple. More precisely, they store two entries and an aggregated count, the number of occurrences of this pair in the full set of triples (value1,value2,count). RDF-3X builds each of the three possible pairs out of a triple and in each collation order (SP,PS,SO,OS,PO,OP). The aggregated indexes are much smaller than the full-triple indexes. They are organized in B+-trees just like the full triple compressed indexes. In addition to these indexes RDF-3X build all three one-value indexes contaning just (value1,count) entries which are very small. On aggregated indexes, the join operators multiply the multiplicities to get the number of duplicates of each output tuple. The size of the aggregated indexes is shown on Table 4.6. Finally, we write π_{vars} to denote a *projection* on the set of variables $vars$ of the input relation. For ease of readability we include below each operation the number of triples obtained by the evaluation thereof and when applicable the triple pattern concerned by the operation.

	SP^2B		$YAGO$	
	Aggr. Index Size	% of full-triple index size	Aggr. Index Size	% of full-triple index size
SP,PS	42,841,260	85.7%	9,441,513	57.7%
SO,OS	49,803,020	99.6%	16,300,486	99.7%
PO,OP	23,662,667	47.3%	9,246,884	56.6%

Table 4.6: The size of aggregated indexes in entries for SP2B and YAGO datasets

Then we describe the SQL translation of SPARQL queries that will serve in the following as the baseline experiment for the plans produced by the standard MonetDB/SQL optimizer. Since unlike HSP and CDP, the MonetDB/SQL optimizer produces only left deep plans, we could not translate the SPARQL queries into SQL ones using exactly the same access paths as those employed by HSP. We simply choose to evaluate each triple pattern of the SPARQL

query on the ordered relation that promotes the use of binary search for selections and returns the variable with the most number of appearances in the query sorted, to maximize (if possible) the number of merge joins. In the case in which a triple pattern contains constants, we chose the ordered relation according to HEURISTIC 5. Thus, the MonetDB/SQL optimizer will undertake the task of join ordering using runtime optimization techniques (e.g., sampling)

4.3.1 SP^2Bench Queries

SP1 (Listing 4.1): As can be seen in Table 4.4 both planners produce and execute exactly the same plan where all joins are evaluated as *merge joins*. Figure 4.4 shows the query plan for query **SP1**. Since the selection on triple pattern tp1 results to only one triple, the MonetDB run-time optimizer transforms the leftmost join (that returns a single triple) into an additional selection over the results of triple pattern tp0. In a similar manner, the last join of the plan is transformed into a selection. However, despite these additional physical optimizations, the query execution time of the plan in RDF-3X is two orders of magnitude faster than in MonetDB. This is due to the cost of *left join* operators employed by MonetDB to get the subject values of a triple from the obtained object and property values. Given that the query result consists of a single triple, we cannot justify the required execution time (most probably a MonetDB bug). HSP and MonetDB-SQL produce the same logical plans but different physical plans. MonetDB-SQL uses the leftjoinpath operator. This is the reason for which MonetDB takes half time than HSP's plan to execute the MonetDB-SQL plan.

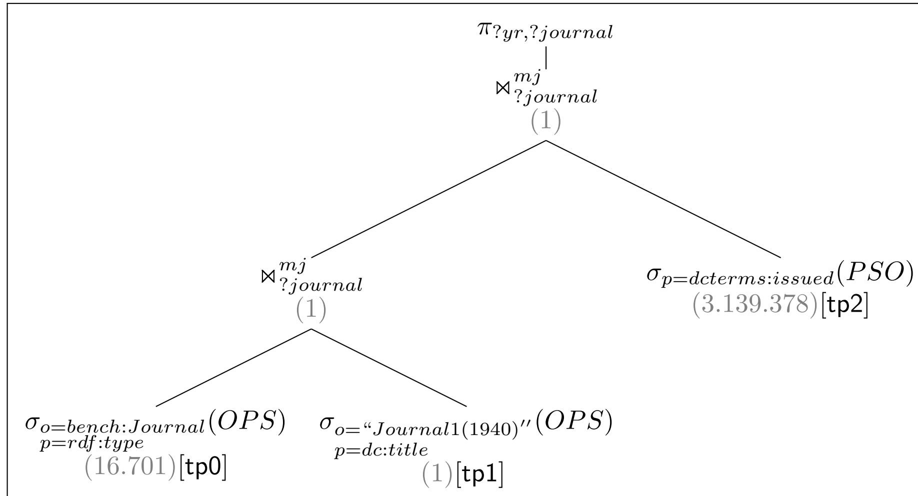


Figure 4.4: **SP1** Plan by **HSP** and **CDP**

SP2a: (Listing 4.2): The plans generated by both **HSP** and **CDP** planners involve a left deep tree of joins all evaluated as merge joins. The large difference in their execution times (Table 4.5) is due to the different join orderings produced by each planner. **HSP** chooses to start joining the results of the selection operator on pattern tp0 since this it has two constants compared to the others with only one (on a property field) (see Figure 4.5) Since **HSP** does not take into account the selectivity of the remaining patterns it randomly picks one (in the order they appear in the query). As we can see in Figure 4.5, the first join between the results of the selections on tp0 and tp1 retains 8.521.551 triples, whereas **CDP** correctly chooses to

join first the results of selections on tp4 and tp9 yielding only 24.441 triples (Figure 4.6) since it relies on a cost-based ordering of joins that uses the size of the intermediate results. This difference in the size of the results can explain the one order of magnitude difference in query execution time between the two systems. The same also happens for the other 8 joins.Also, CDP chooses to evaluate triple patterns tp1-tp9 on the aggregated index PS, the size of which is 85% that of the full-triple index, as shown in table 4.6. The execution time of the same plan in MonetDB is 200ms (instead of 3,267.01 ms required with the random ordering). MonetDB-SQL produces the same plan as HSP.

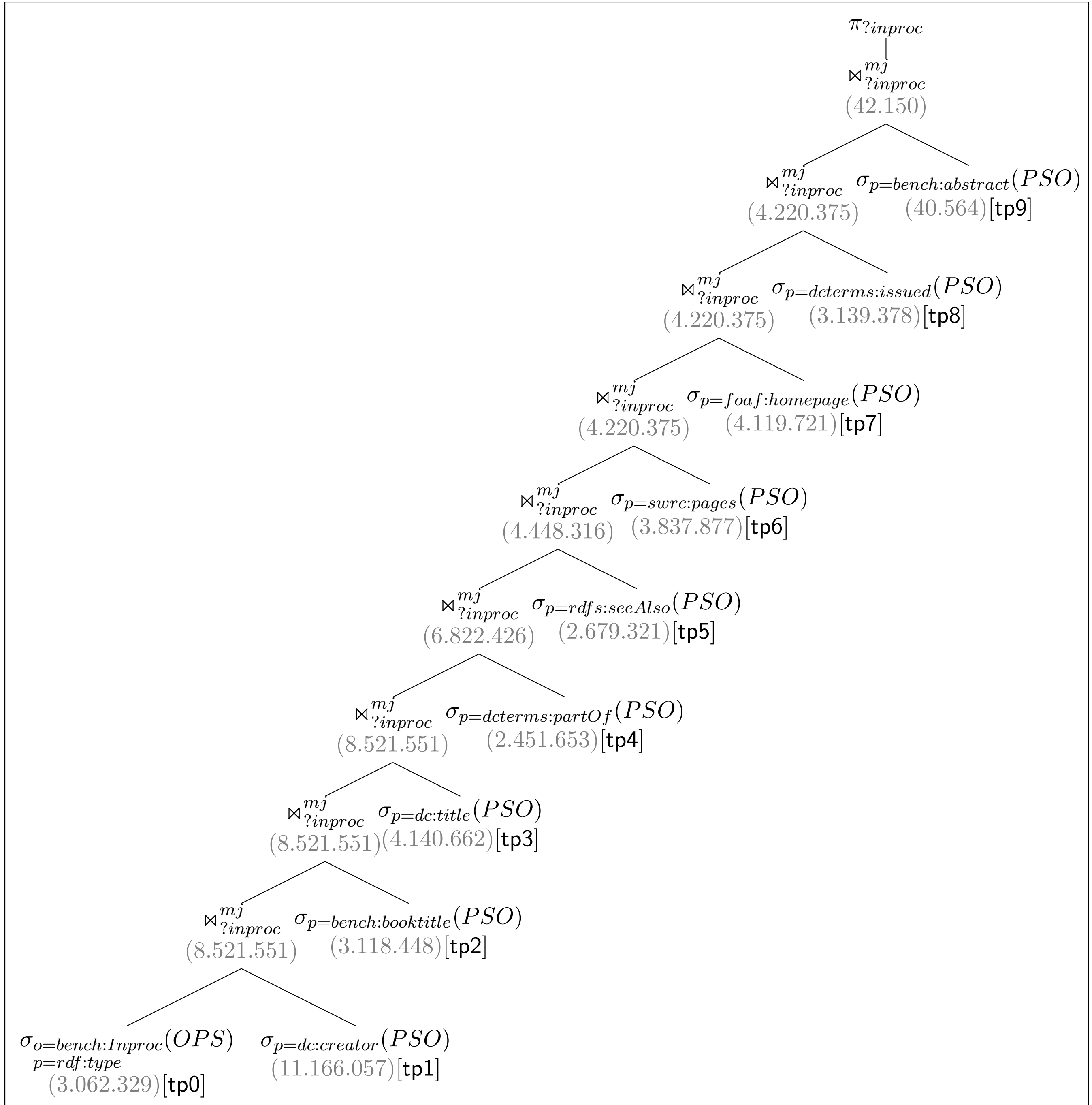


Figure 4.5: **SP2a** Plan by **HSP**

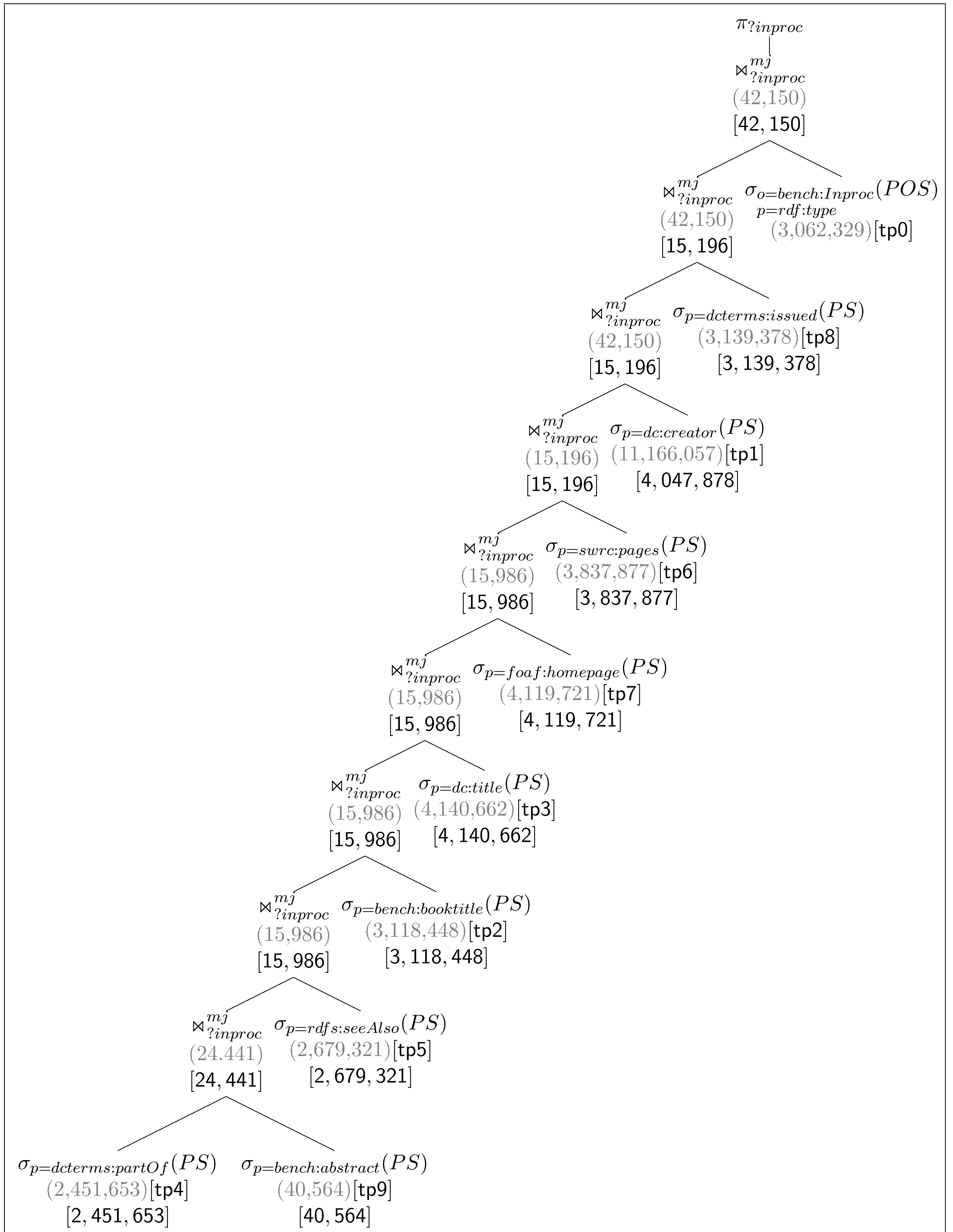


Figure 4.6: **SP2a** Plan by **CDP**

SP2b (Listing 4.3): As in the previous query, the two planners produce left deep plans where all joins are executed as merge joins. CDP chooses to evaluate triple patterns tp1-tp7 on the aggregated index PS, the size of which is 85% that of a full-triple index. As we can see in Figure 4.8 **CDP** chooses a join ordering which quickly reduces the size of intermediate results, but since there does not exist a triple pattern of high selectivity (recall that triple pattern tp9 with the highest selectivity has been removed from query **SP2a**), the query execution time is significantly increased. With the exception of tp0 which has two constants, the rest of the triple patterns feature only one constant on the property position and essentially form a star-shaped join on variable `?inproc` in the subject position. Also all of them also include the projection variable `?inproc`. Consequently, **CDP** starts joining the results of the selection on tp0 but since it cannot estimate the most selective among the remaining triple patterns (recall that it does not decide on the join ordering using selectity estimates) it randomly picks up one pattern for the join. The same also happens for the other six joins. Since the size of intermediate results is of the same order of magnitude, the execution time of this query in both systems is almost the same (see Table 4.5). MonetDB-SQL produces the same plan as HSP.

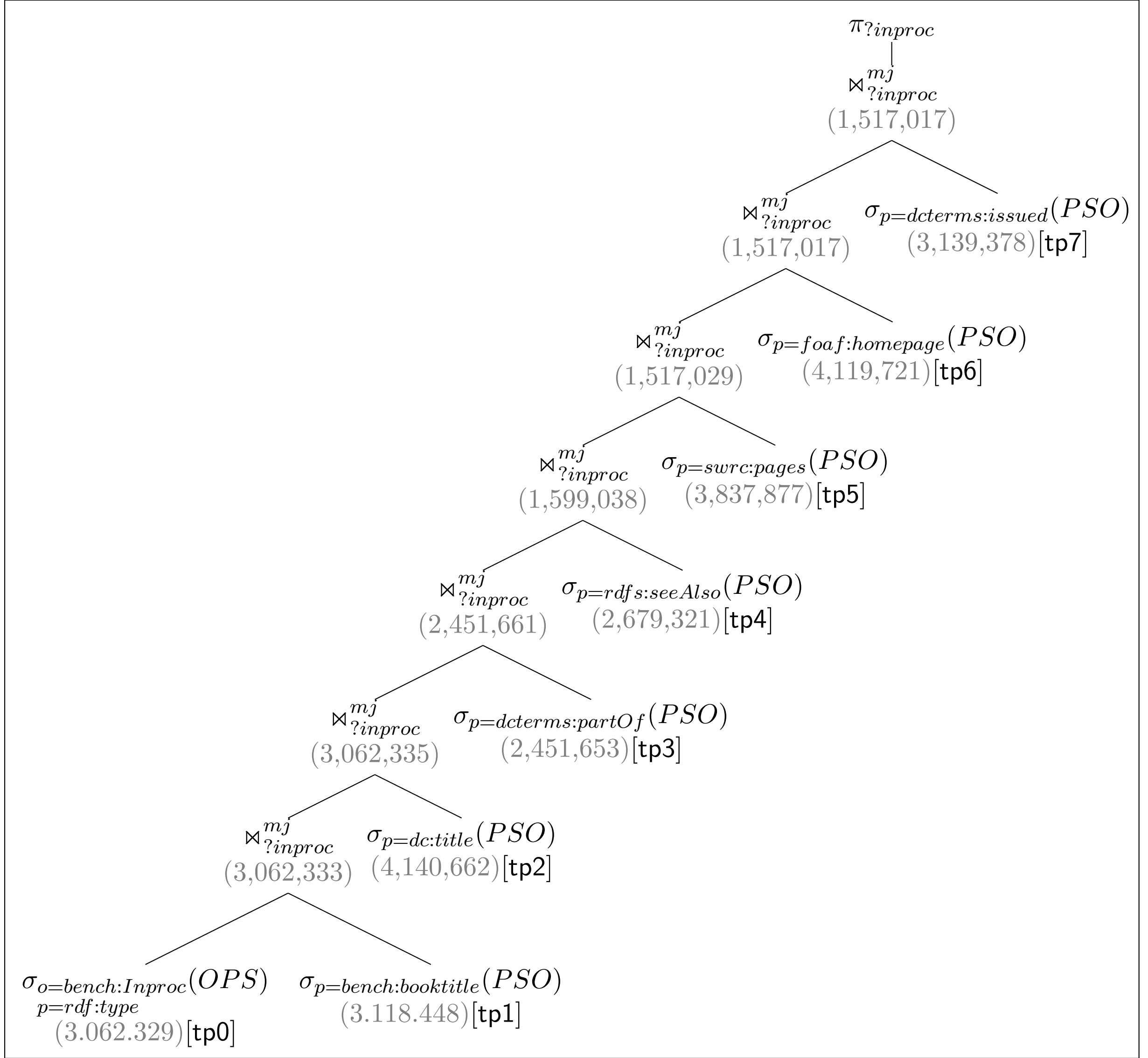


Figure 4.7: **SP2b** Plan by **HSP**

SP3a,SP3b,SP3c: Unlike **CDP**, **HSP** systematically rewrites filtering queries (queries **SP3a_1**, **SP3b_1** and **SP3c_1**, Listings 4.4, 4.6, 4.8) into an equivalent form involving only triple patterns (queries **SP3a_2**, **SP3b_2** and **SP3c_2**, Listings 4.5, 4.7, 4.9). This logical optimization significantly impacts the performance of RDF-3X which applies selections after the evaluation of joins (sometimes resulting to cartesian products). In this case, the execution time of queries in RDF-3X is one order of magnitude slower than in MonetDB. In order to provide a common comparison basis for both systems we manually transform these queries into their equivalent form (queries **SP3a_2**, **SP3b_2** and **SP3c_2**) produce the plans using **CDP** and evaluate them in RDF-3X. Thus, both systems will execute the same plan comprising two selections and one join which is evaluated as a merge join. Note that **CDP** evaluates the selection on triple pattern tp1 using the aggregated index *PS* and not index *pso*. Since

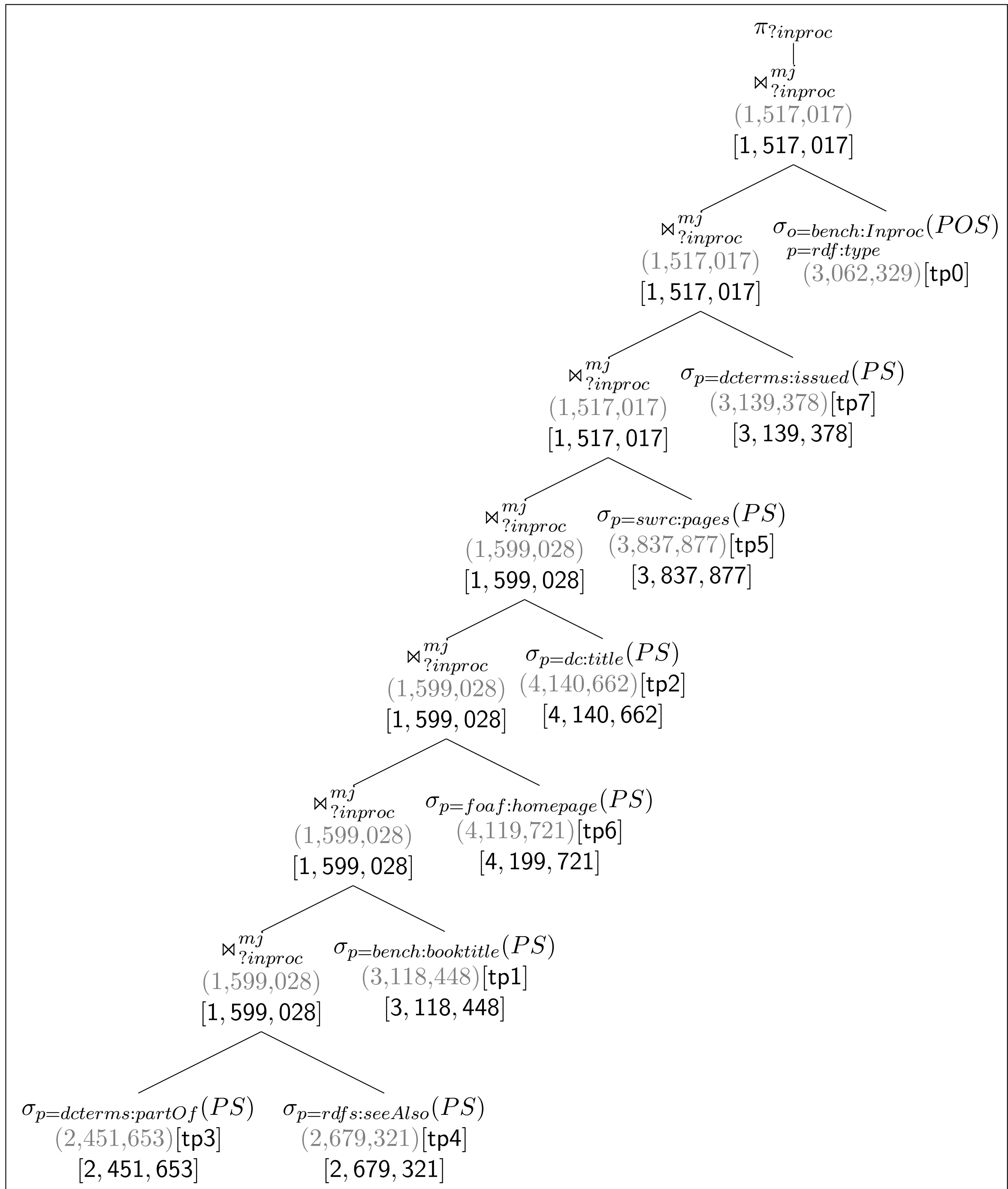


Figure 4.8: **SP2b** Plan by **CDP**

the first operation in all queries is the selection on tp0, their execution time is affected by the actual size of the second selection on tp1 (returning 3.837.877 triples in **SP3a_2**, 6.401 triples in **SP3b_2**, and 44.165 triples in **SP3c_2**). **HSP** plans evaluated in MonetDB outperform those of **CDP** evaluated in RDF-3X by a factor ranging from 6% to 37%. MonetDB-SQL

produces the same plan as HSP.

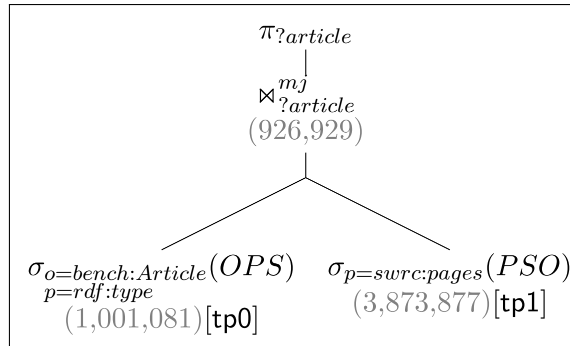


Figure 4.9: **SP3a_2** Plan by **HSP**

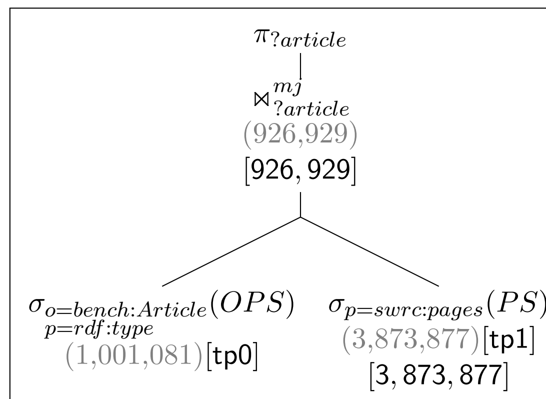


Figure 4.10: **SP3a_2** Plan by **CDP**

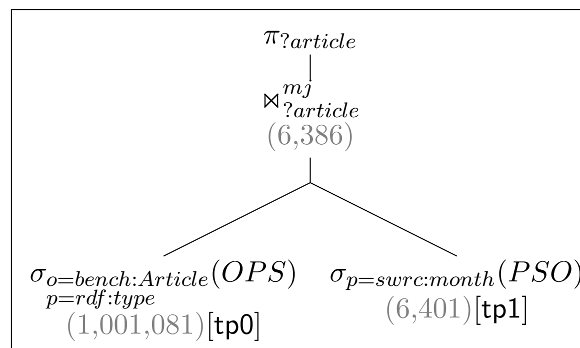


Figure 4.11: **SP3b_2** Plan by **HSP**

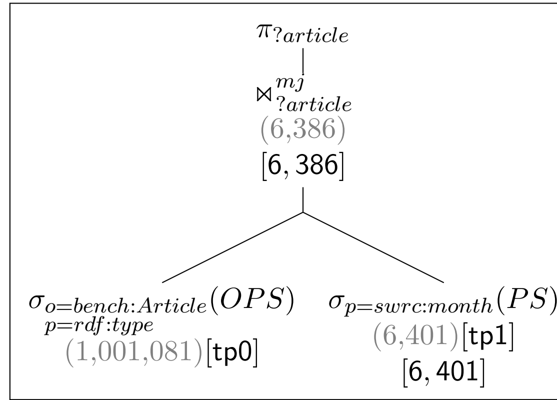


Figure 4.12: **SP3b_2** Plan by **CDP**

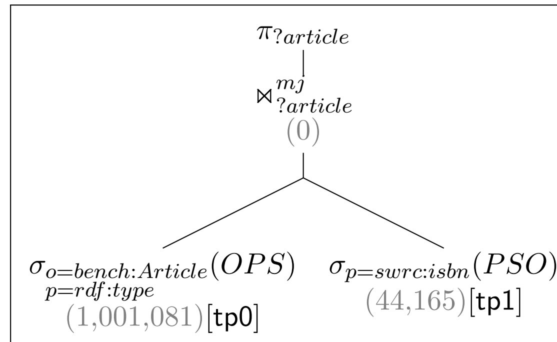


Figure 4.13: **SP3c_2** Plan by **HSP**

SP4a: As for queries **SP3a, SP3b, SP3c**, **HSP** rewrites query **SP4a_1** (Listing 4.10) into its equivalent form by removing the filter expression (query **SP4a_2**, Listing 4.11). On the other hand, **CDP** does not perform this logical optimization, and for this query it generates a plan with a cross product. Execution plans that include cartesian products cannot be evaluated in RDF-3X and so query execution fails. To benchmark RDF-3X for this query, we have manually created its equivalent form by eliminating the FILTER expression. For query **SP4a_2**, both planners choose to execute joins on variables `?name`, `?article` and `?inproc` as merge joins. In the following, two alternative plans can be considered. The first is to execute a hash join on variable `?person` and then on variable `?person2`. The second alternative is to perform the hash join first on variable `?person2` and then on variable `?person`). Since **HSP** does not decide on join ordering using information related to the size of the intermediate results, it chooses randomly one of the two plans. **HSP** and **CDP** finally produce the same plan with almost the same execution times for MonetDB and RDF-3X. MonetDB-SQL produces a plan that involves a cartesian product, a plan which the MonetDB server is unable to execute because of the huge intermediate results.

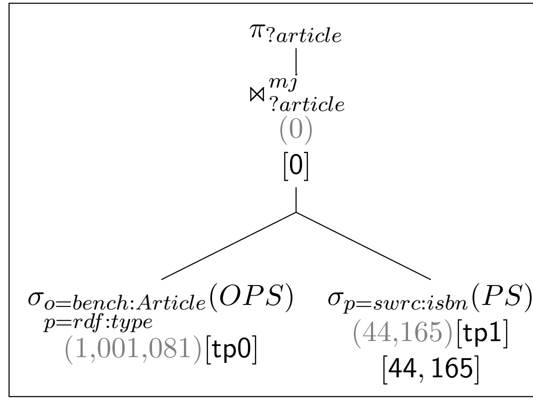


Figure 4.14: **SP3c_2** Plan by **CDP**

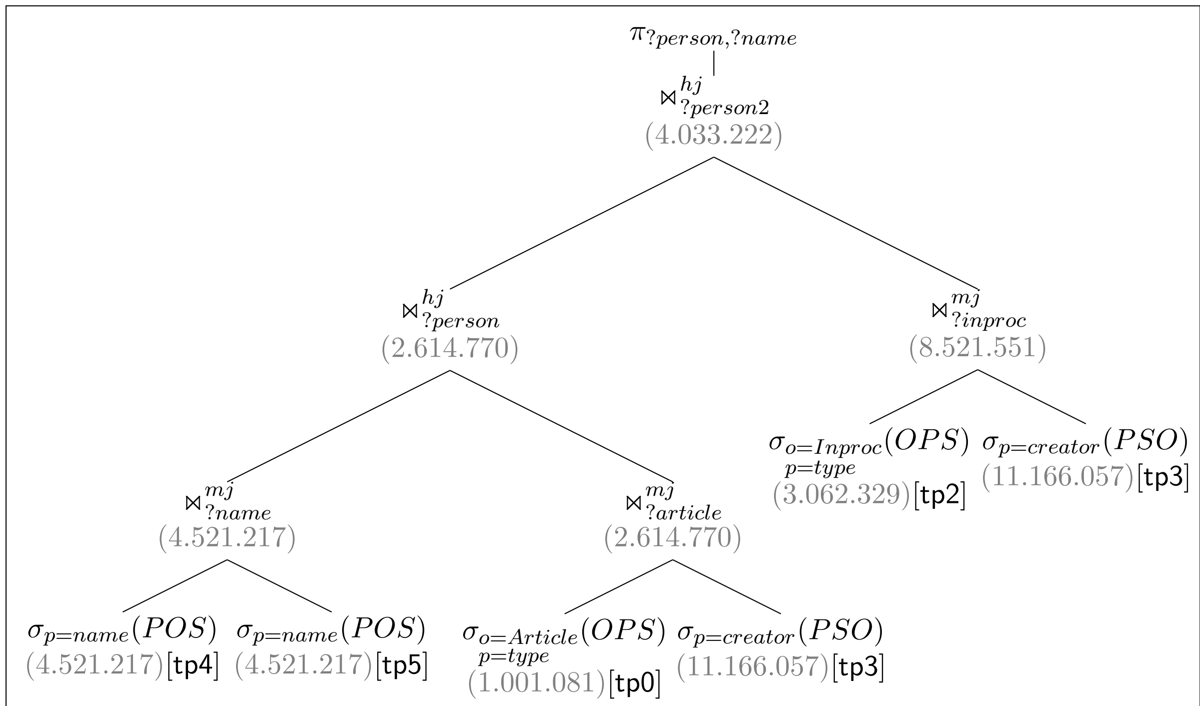


Figure 4.15: **SP4a_2** Plan by **HSP** and **CDP**

SP4b (Listing 4.12): The two planners produce different execution plans for this query. The two plans have the same number of merge joins but they are defined on different variables. **HSP** chooses to perform merge joins on variables `?article` and `?inproc`, while **CDP** on variables `?person` and `?inproc`. **HSP** privileges variable `?article` because the triple patterns tp0 and tp1 featuring this variable have three constants instead of two as in the case of tp1 and tp4 featuring the *?person* variable. Like for the previous query, all the triple patterns have low selectivity returning more than 1M of triples each, so this query involves a very large size of intermediate results. In this query, MonetDB outperforms RDF-3X by a factor of 57%.

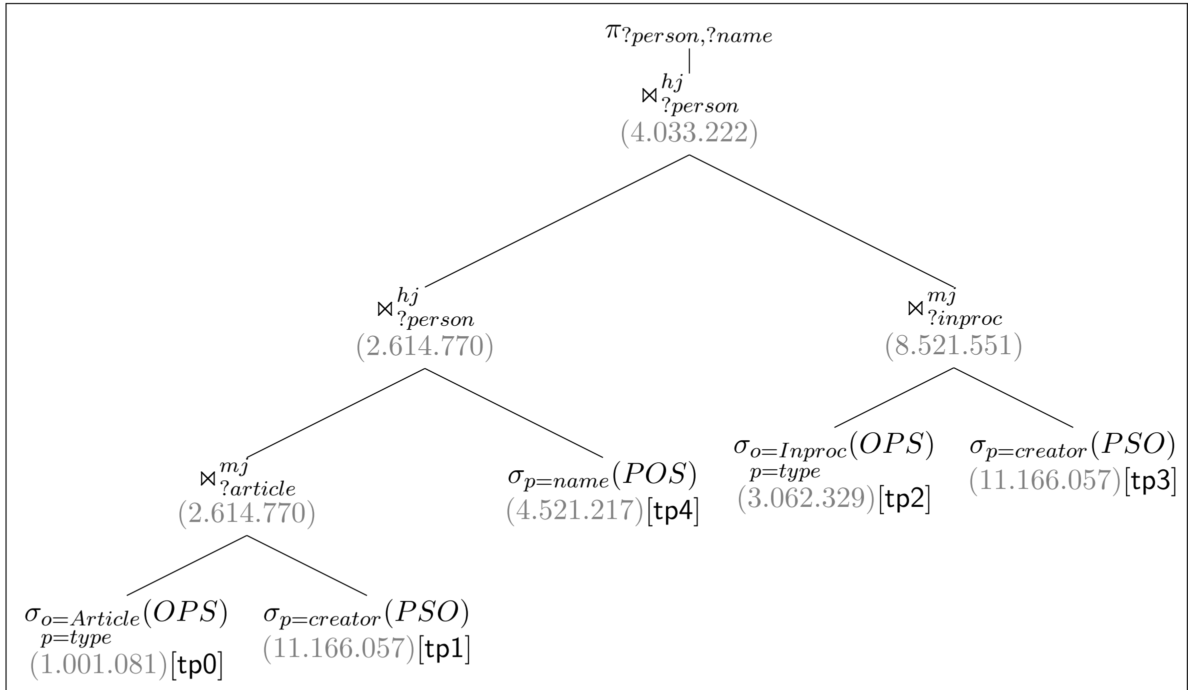


Figure 4.16: **SP4b** Plan by **HSP**

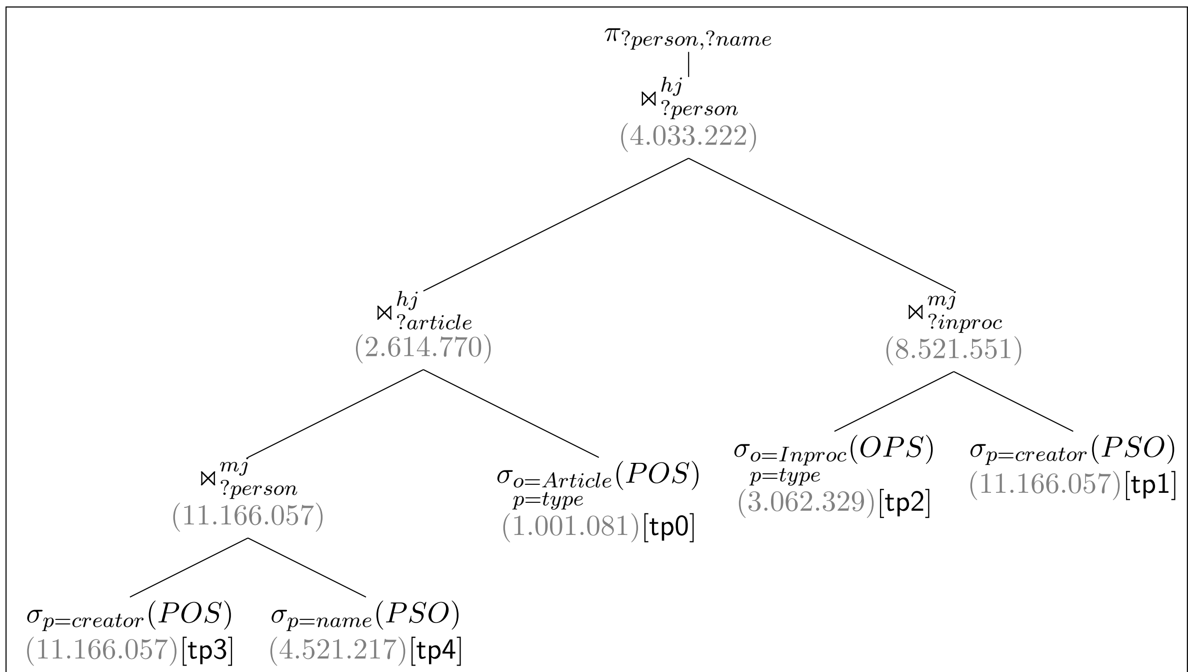


Figure 4.17: **SP4b** Plan by **CDP**

SP5 (Listing 4.13): Both **HSP** and **CDP** produce the same plan with only one selection. The query has high selectivity returning only 565 triples (out of 50M). Although, both systems require a short execution time for this query, MonetDB outperform RDF-3X by a factor of 67% (Table 4.5).

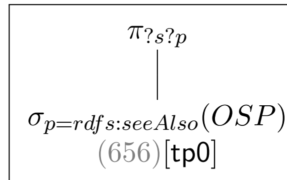


Figure 4.18: **SP5** Plan (**HSP** & **CDP**)

SP6 (Listing 4.14): As for the previous query, HSP and CDP produce the same plan with only one selection. MonetDB takes one order of magnitude less time than RDF-3X to execute this plan. Note that RDF-3X uses to execute the selection on the aggregated index PS, which has 47% of the size of full-triple index PSO. This behavior can be attributed to the decompression RDF-3X performs on the compressed tuples to their three component id's (for subject, property, object).

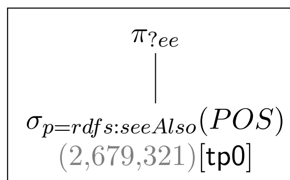


Figure 4.19: **SP6** Plan by **HSP**

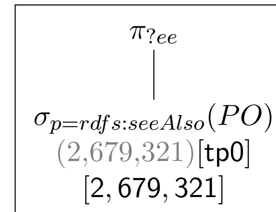


Figure 4.20: **SP6** Plan by **CDP**

4.3.2 YAGO Queries

Y1 (Listing 4.15): The two planners produce different plans for this query which have the same number of merge joins. As shown in Figure 4.21 **HSP** chooses to execute four merge joins all on variable `?p` and one on `?city2`. As a first step, **HSP** sorts the five triple patterns featuring variable `?p` based on their number of constants.Apart from tp2 featuring two constants, the remaining ones have only one constant (in property position) so its selectivity could not be estimated based on heuristics **H1** and **H2**. In addition, no further selection on the triple patterns can be performed by HSP based on heuristic **H3**, since tps involve a star-shaped join on a variable in the subject position. According to heuristic **H4**, tp0 and tp1 are pushed up, on higher level than tp3 and tp6 because each of them has one projection variable. On the other hand, **CDP** chooses to execute three merge joins on variable `?p` and one on each of the variables `?city` and `?city2`. MonetDB evaluates the **HSP** plan in only 6.04ms which is approximately 2.5 times faster than the evaluation time of the **CDP** plan in RDF-3X (15.75ms) (see Table 4.5). The MonetDB-SQL plan involves four merge-joins which is the maximum number of merge=joins which a left deep tree can have in this query, as it is the number of joins in which the most common variable of the query exists.

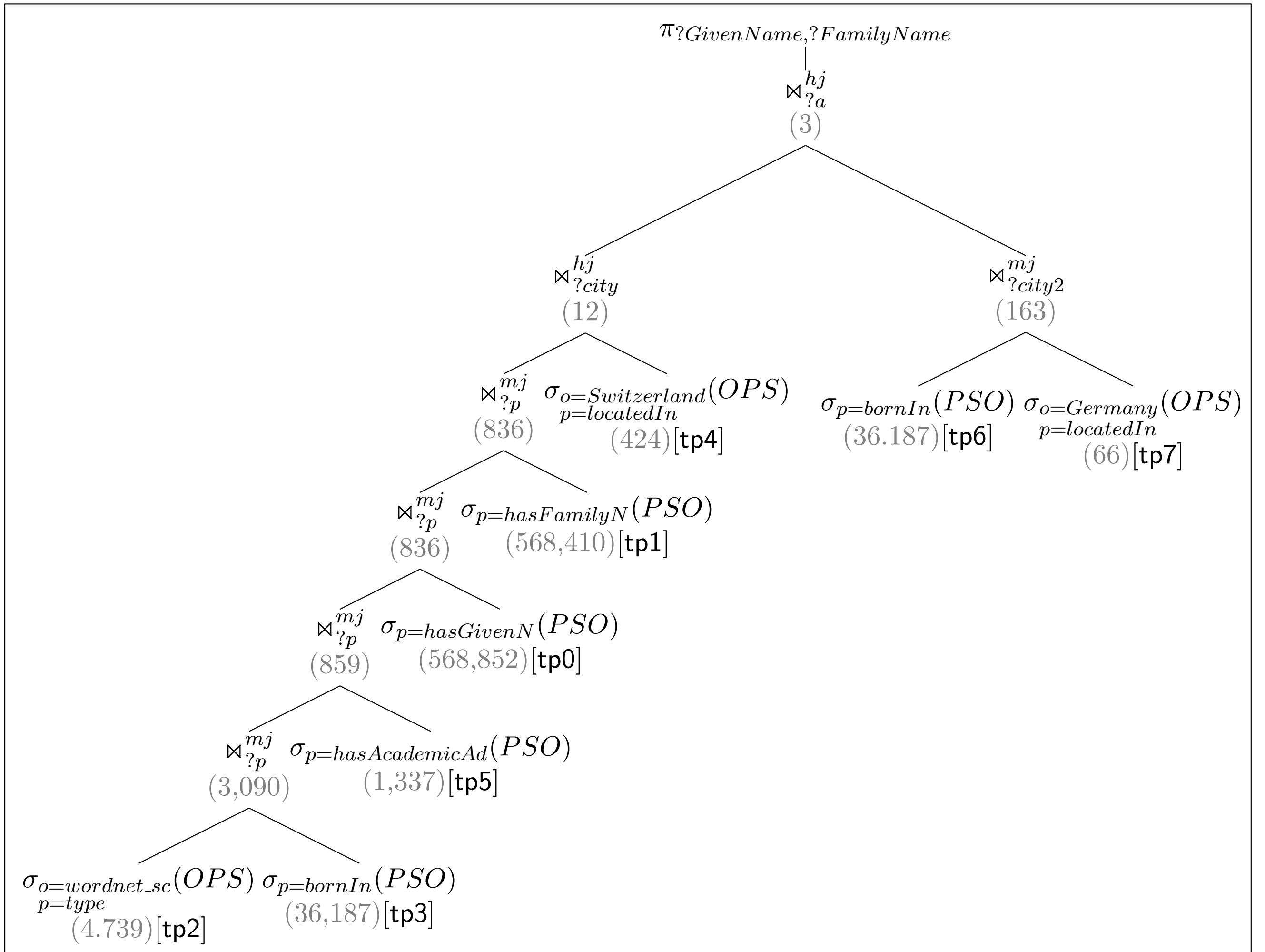


Figure 4.21: **Y1** Plan by **HSP**

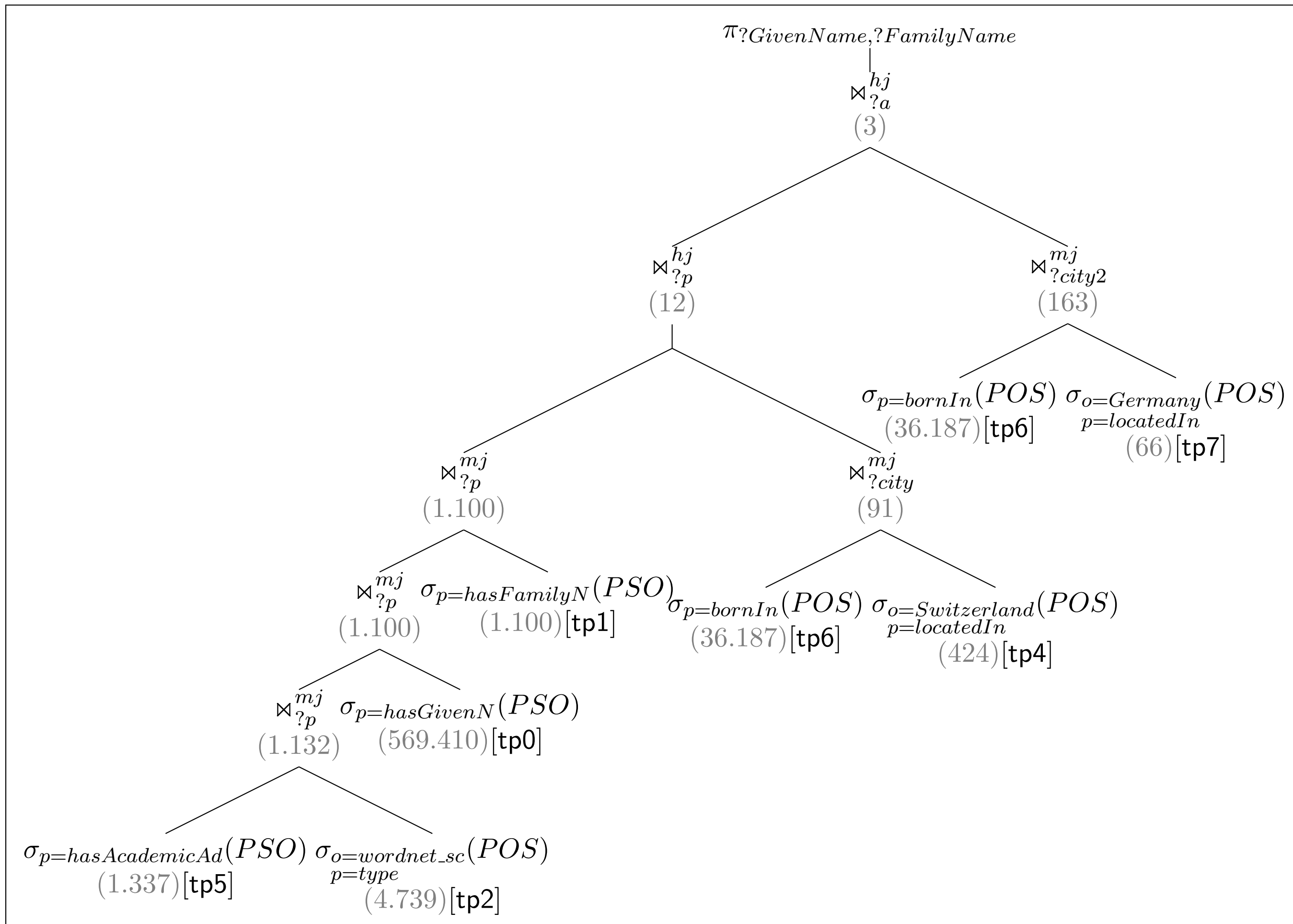


Figure 4.22: **Y1** Plan by **CDP**

Y2 (Listing 4.16): The two planners produce different plans. **HSP** produces a left deep plan while **CDP** produces bushy plan. HSP chooses to execute three merge joins on variable `?a`, as shown in Figure 4.23. CDP chooses to execute one merge join on each of the variables `?a`, `?m1` and `?m2` reducing the size of intermediate results early in the plan as can be seen in Figure 4.24. All triple patterns are highly selective returning results ranging from 15K to 30K of triples. HSP plans evaluated in MonetDB outperform those of CDP in RDF-3X by a factor of 15% (see Table 4.5). The MonetDB-SQL plan involves the same number of merge-joins and in the same variables as HSP but with different join ordering.

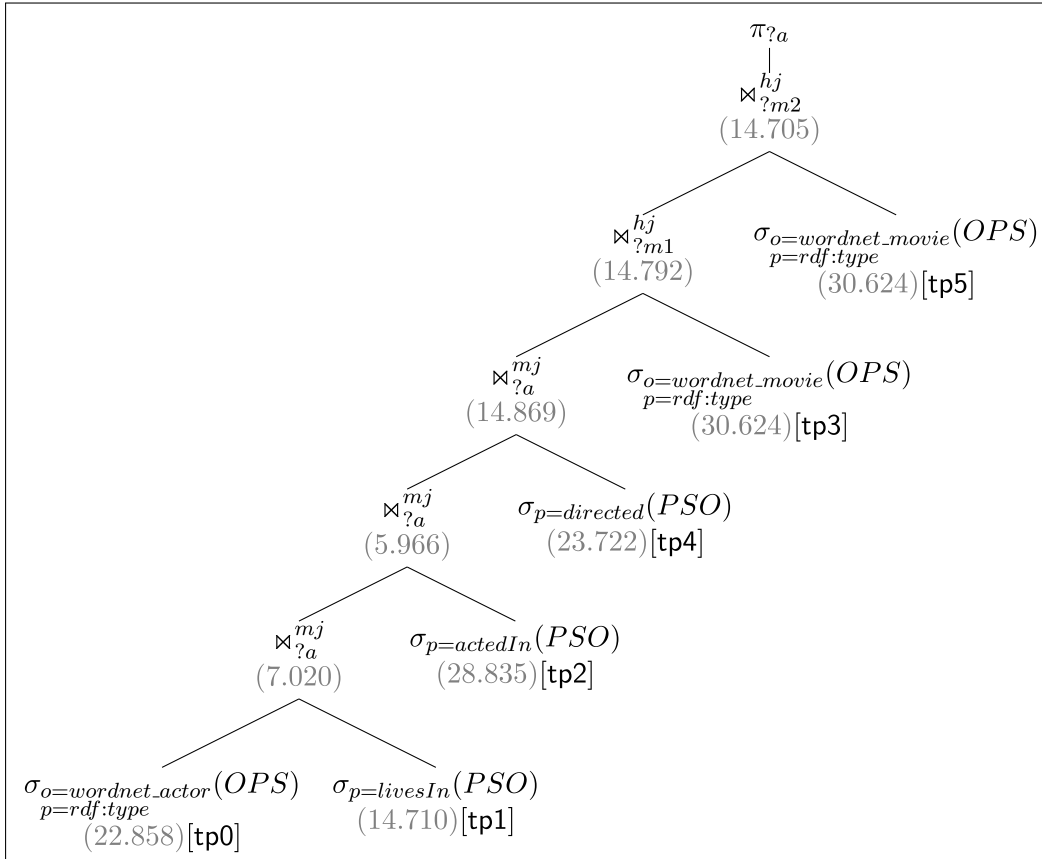


Figure 4.23: **Y2** Plan by **HSP**

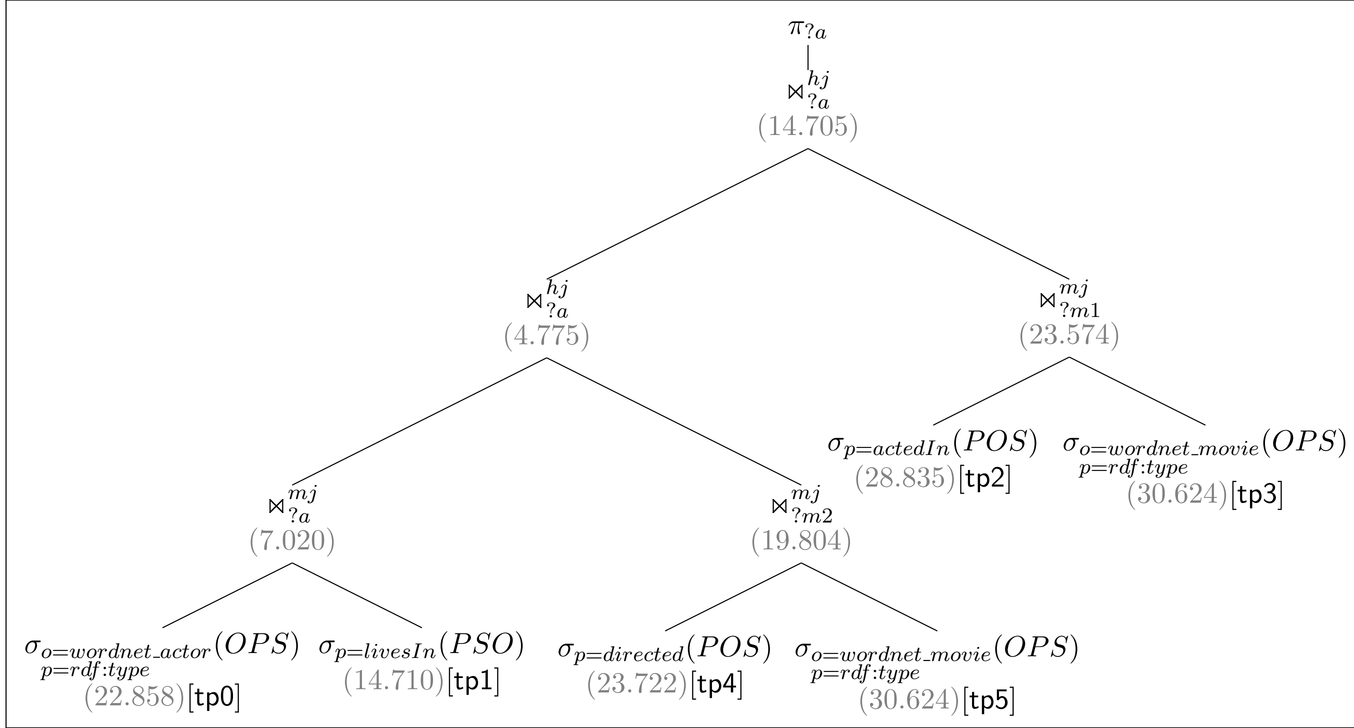


Figure 4.24: **Y2** Plan by **CDP**

Y3 (Listing 4.17): Both planners generate the same bushy plan, and both planners choose to evaluate merge joins on variables `?c1` and `?c2` because this is the only way to *maximize the number of mergejoins*. There are three triple patterns featuring variable `?c1`. Using heuristic **H1**, **HSP** chooses to start joining the results of selections on tp2 and tp3 because the triple pattern tp0 does not have any constant, so it returns the entire triple relation. The same heuristic is employed to decide on the first join on variable `?c2` since triple pattern tp1, like tp0, do not have any constant. CDP chooses to execute the selections on the corresponding triple patterns tp3 and tp5 on aggregated index PS which has 57% of the size of the full-triple index. It also chooses to execute the two scan operators on tp0 and tp1 on aggregated index OS so it has to decompress 16,300,486 triples instead of 16,348,563 triples that it should if it had chosen one of full-triple indexes. MonetDB evaluates the HSP plan just in 25.69ms which is almost three times faster than the execution time 81.20ms of CDP plan by RDF-3X (see Table 4.5). The left deep tree plan of MonetDB-SQL has only two merge-joins in contrast to four merge-joins which the HSP and CDP plans involve. The HSP plan executed in MonetDB outperforms by one order of magnitude the MonetDB-SQL plan.

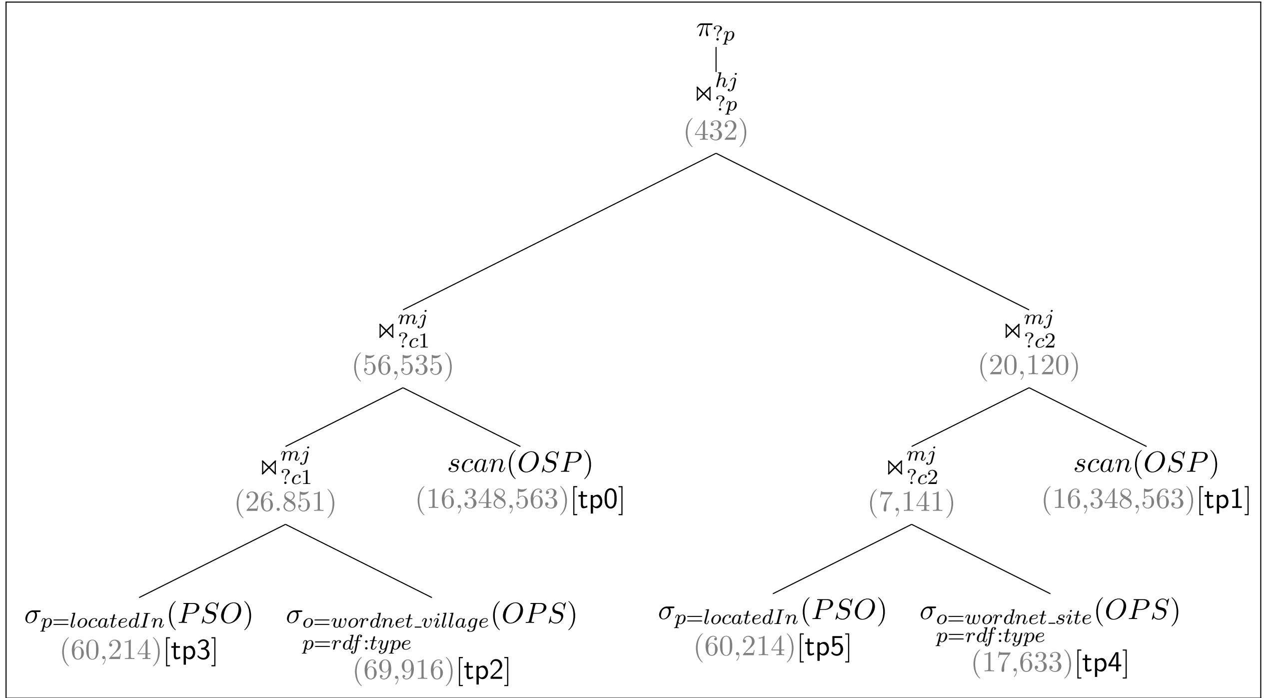


Figure 4.25: **Y3** Plan by **HSP**

Y4 (Listing 4.18): **HSP** and **CDP** produce query plans with merge joins on the same variables namely, `?c1` and `?c2`. The only difference lies on the order of the two hash joins involved in the plans. **HSP** considers three combination of variables which could be executed as merge joins: `?c1,p2`, `?c1,?c2` and `?p1,?c2`. Based on heuristic **H1**, **HSP** chooses set `?c1,?c2` in order to avoid a join in which both inputs consider the entire triple relation. The **HSP** plan executed in MonetDB outperforms by one order of magnitude the **CDP** plan executed in RDF-3X (see Table 4.5). MonetDB-SQL's plan has only one merge-join and the join ordering which the sql optimizer chooses, produces huge intermediate results. The HSP executed in MonetDB outperforms by three orders of magnitude the MonetDB-SQL plan.

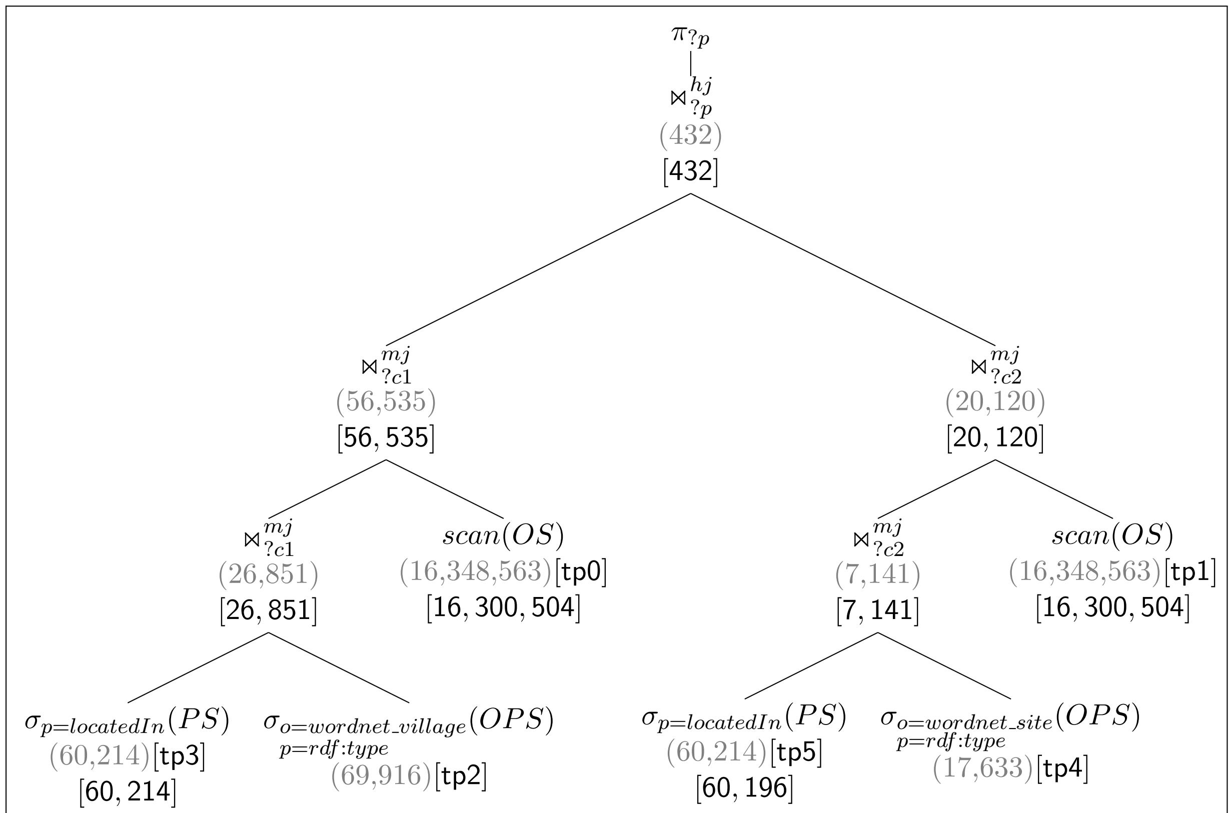


Figure 4.26: **Y3** Plan by **CDP**

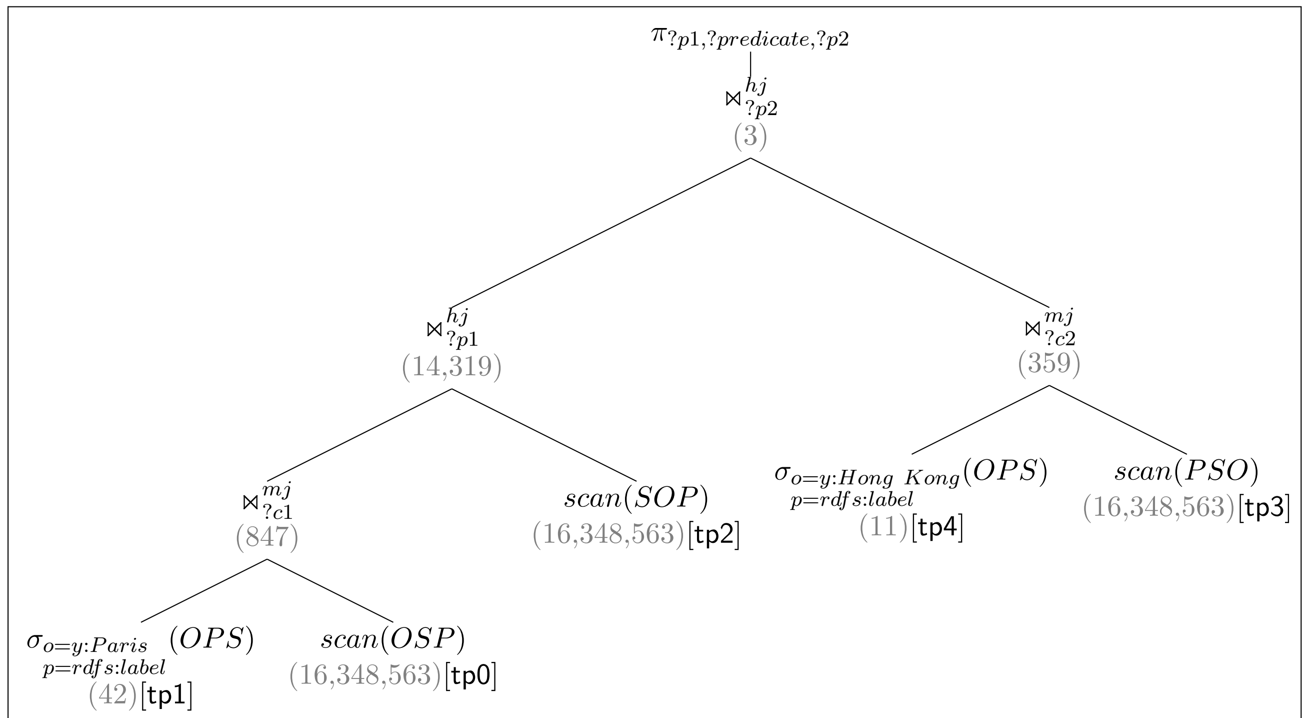


Figure 4.27: **Y4** Plan by **HSP**

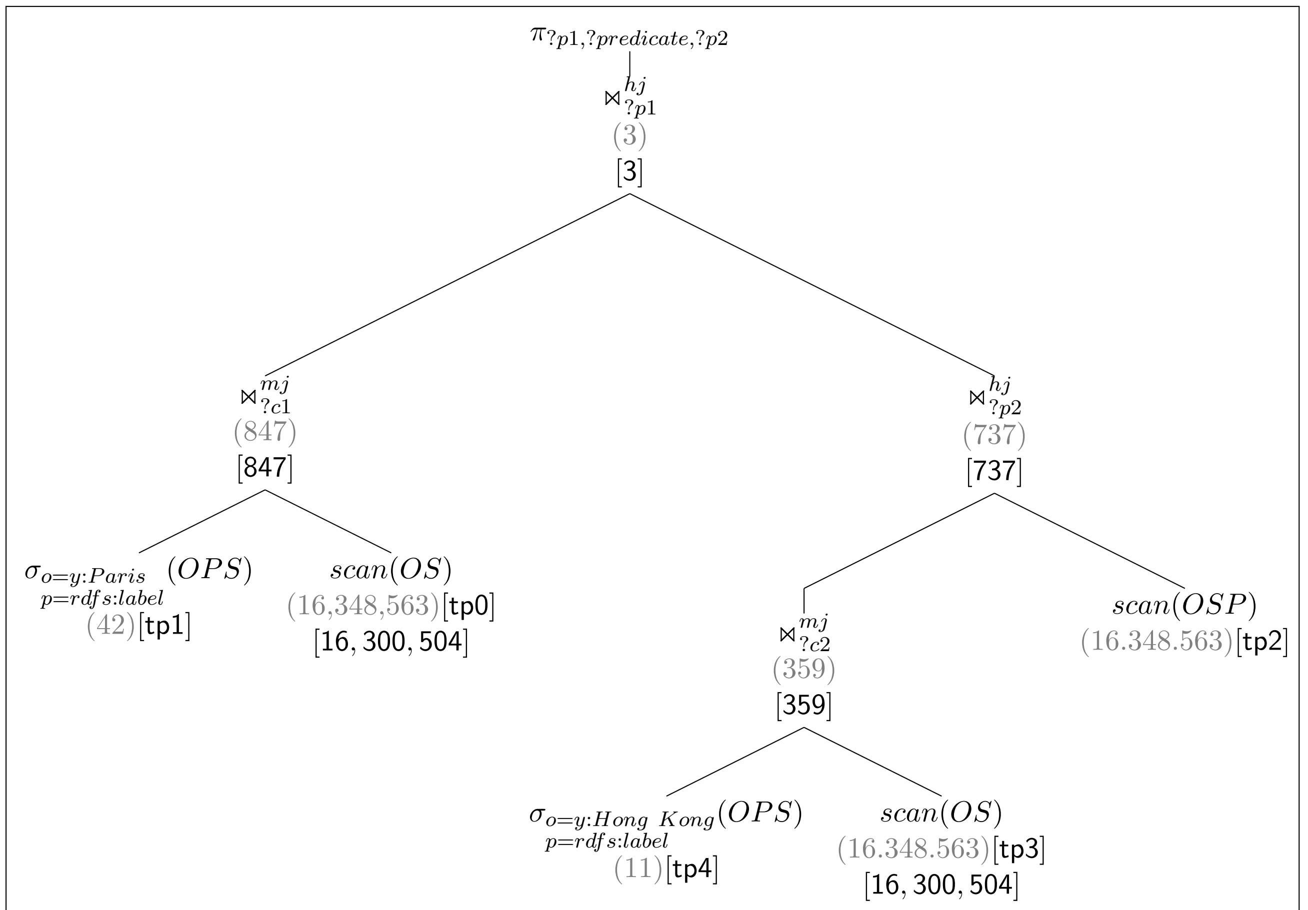


Figure 4.28: **Y4** Plan by **CDP**

4.4 Cost of the plans

The last experiment we conducted aims to measure the quality of the plans HPS actually produces compared to CDP. The most effective plan quality measure in this respect is the plan execution cost. In this experiment we rely on the cost model employed by CDP to rank the query plans produced by the two planners. In particular, we focus on the join cost estimation (concerning the size of intermediate results) since the selection cost is asymptotically the same in both systems (logarithmic for binary search in MonetDB and for B+tree traversal in RDF-3X).

The cost of a merge join is estimated as follows:

$$cost_mergejoin(lc, lr) = \frac{lc + rc}{100,000}$$

where lc and rc are the cardinalities of two join input relations.

Hashjoin's cost is estimated as follows:

$$cost_hashjoin(lc, rc) = 300,000 + \frac{lc}{100} + \frac{rc}{10}$$

where lc and rc are the cardinalities of two join input relations. In both cases we take as a left join input the smallest input relation.

	SP1	SP2a	SP2b	SP3a	SP3b	SP3c	SP4a	SP4b
HSP	32	873	830	487	100	105	354+953,381	264+953,381
CDP	32	31	54	487	100	105	354+953,381	299+858,461

	Y1	Y2	Y3	Y4
HSP	12+300,054	1+303,579	329+302,577	327+763,749
CDP	7+300,023	1.5+301,614	328+302,577	326+763,603

Green color: The cost of mergejoins
Red color: The cost of hashjoins

Table 4.7: The Cost

4.5 Concluding Remarks

In all queries of our workload, **HSP** produces plans without statistics with the same number of *merge* and *hash* joins as **CDP** plans. Their differences lie on the *join* variables and *join ordering*. These factors essentially affect the size of the intermediate results. **HSP** heuristics (**H1 - H4**) prove to be quite effective in choosing a *near to optimal join ordering* and set of *join variables* when queries exhibit quite different syntactical forms.

(a) For queries with the same produced plan, MonetDB outperforms RDF-3X (up to three orders of magnitude) with the exception of query **SP1**.

(b) For queries with different plans, we observe that when their triple patterns do not differ substantially in their syntax, then **HSP** heuristics fail to decide the triple pattern to select based on its selectivity, and thus RDF-3X outperforms MonetDB up to one order of magnitude. When selectivity of triple patterns yields intermediate results of the same order of magnitude, the join ordering has little influence on the query execution time. On the other hand, for sufficiently dissimilar triple patterns, MonetDB outperforms RDF-3X up to 2 orders of magnitude. More specifically, **HSP** fails to produce near optimal plans when the queries involve *heavy star joins* where the ration of shared veriables over the number of triple patterns is small, and triple patterns are not sufficiently distinguishable w.r.t. their syntax (i.e., same number of constants and where the shared variables appear at the same position).

(c) Despite the fact MonetDB systematically materialize the intermediate results, the execution of all query operators in MonetDB appears to be more efficient than in RDF-3X (exhibiting also limitation of the size of the data sets that can be processewd in main memory. Finally, query execution in RDF-3X is additionally penalized by the decompression of object identifiers performed systematically in scan and selection operations. This extra cost can become important when the selectivity of triple patterns is small.

Chapter 5

Related Work

5.1 Storing, Indexing and Querying RDF data

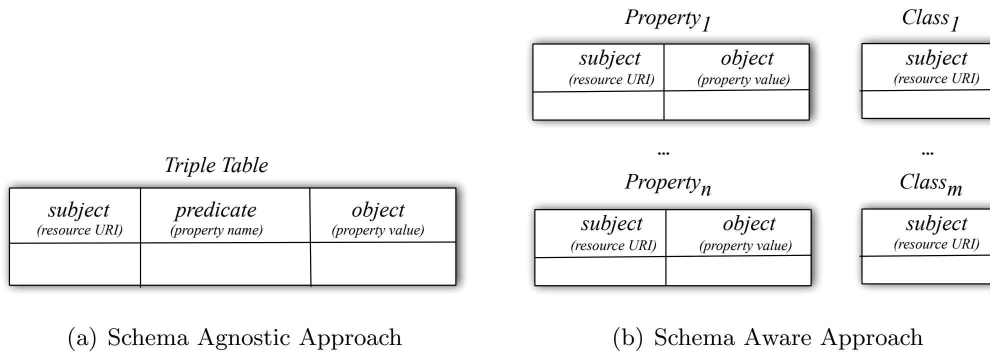
Despite their advantages in managing large data collections, relational database management systems (RDBMS) cannot yet fully support querying large volumes of RDF data using the SPARQL query language [41]. Whereas relational technologies deal well with complex and large schemas, storing RDF data typically boils down to using a single *triple table* that stores triples of the form (*subject*, *predicate/property*, *object*) where each of the table's attributes refers to one component of the RDF triple. Queries are translated into self joins over the large triple table, and relational database management systems are not adequately tuned to perform efficiently this kind of queries.

In this chapter we will discuss the different approaches for storing, indexing and retrieving semantic web data.

5.1.1 Logical Storage Schema for RDF Data

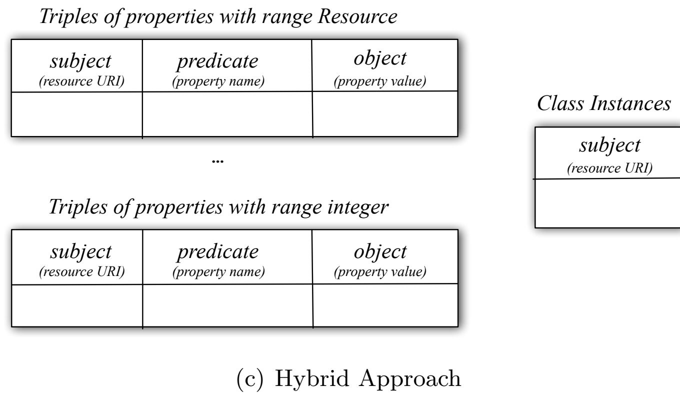
The three widely used as logical storage schemes for shredding RDF/S resource descriptions into relational tables are:

- *schema agnostic* in which RDF triples are stored in a large *triple table*. The triple table's attributes are *subject*, *predicate* and *object* and refer to the three components of an RDF triple. attribute This approach has been followed in the majority of works that deal with the storage and processing RDF data [49, 35, 37, 36, 44, 16, 8, 53] (see Figure 5.1(a)).
- *schema aware* [5, 18] in which for each property $Property_i$in a set of RDF triples, a binary table is created that stores the *subject* and *object* components of the triple with predicate $property_i$. This approach has been used in [5, 18] and is known as *vertical partitioning* (see Fig. 5.1(b)).
- *hybrid* that combines elements of the previous approaches. In the hybrid approach, one relational table per RDF class $Class_i$ is created, that stores the instances of the class. In a similar manner, dependending on the type of the property value (resource, string, integer), a table that stores the instances of the corresponding value types is created (see Figure 5.1(c)).



(a) Schema Agnostic Approach

(b) Schema Aware Approach



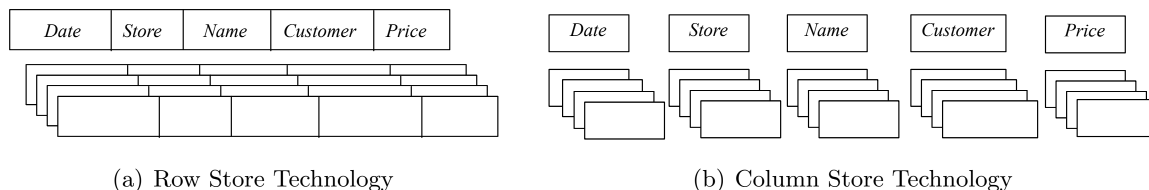
(c) Hybrid Approach

Figure 5.1: Logical Storage Schemas for RDF data

In the case of the schema agnostic approach, one triple table is created for any RDFS schema. On the other hand, in the case of schema aware approach, the properties and classes defined at the RDFS schema are considered to define the property and class tables of the logical schema where the data will be stored. Finally, the hybrid approach uses the RDF meta-schema (classes, attributes, value types) to define the logical data storage schema.

The advantages of the schema agnostic approach is the ease of RDF data representation and the decoupling of the logical schema from the existence of an RDFS schema. This decoupling makes schema evolution a trivial process, since the addition/deletion of a class or schema property corresponds to the addition/deletion of a set of triples. In a similar manner, in the case of the hybrid approach, this corresponds to deletion of triples from the property tables and the deletion of tuples from the table that stores the class instances. On the other hand, in the case of the schema aware approach, a change in the schema corresponds to the addition/deletion of a class or property table. The disadvantage of the schema agnostic approach is the loss of information related to the type of property values (i.e., the object of a triple), since all values of the object component of a triple are stored as a string. The hybrid and schema aware approaches record this kind of information which can be used for query processing.

As far as SPARQL query evaluation is concerned, the main disadvantage of the schema agnostic approach is that the algebraic plan obtained for a query involves a large number of *self joins* over a large triple table, whereas in the other two cases, the plan involves joins between the appropriate class and property tables. The schema agnostic approach is advantageous in



(a) Row Store Technology

(b) Column Store Technology

Figure 5.2: Logical Storage Schemas for RDF data

the cases in which a property is unbound (i.e., use of a variable and not a URI). On the other hand, in the case of schema aware and hybrid approaches, the initial user query should be translated into as many queries as the number of predicates in the dataset [49].

5.1.2 Physical Storage Schema for RDF Data

The majority of works that study the storage of RDF data use relational databases as the storage platform. Relational databases are based on *row store* (PostgreSQL[1], MySQL[2], Oracle[3]) or *column store* (MonetDB [4] and C-store[5]) technologies [4].

For databases that rely on column store technology, each attribute of a relational table is stored in a column (see Figure 5.2(b)). Consequently, the physical representation based on column storage is a fully compliant solution in the case in which the logical storage schema follows the vertical partitioning approach [5, 18] (*schema aware*). According to [44], systems based on the row-store technology that follow the schema agnostic approach outperform those that follow the vertical partitioning approach (i.e., schema aware).

Authors in [55, 54] follow a different approach based on the use of *property tables*. Property tables are distinguished into *clustered property table* and *property-class table* used to improve query performance in the case of very large RDF datasets where the schema agnostic approach is used.

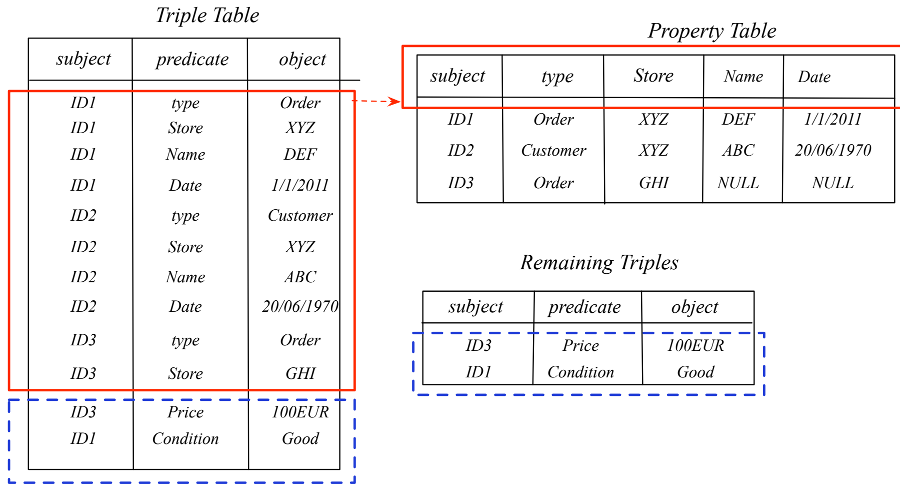
Clustered property tables contain clusters of properties that tend to be defined together (see Fig. 5.3(a)). The Property-Class table approach exploits the type property of subjects to cluster similar sets of subjects together at the same table and unlike the clustered property table approach, a property may exist in multiple property-class tables (see Figure 5.3(b).

The common denominator in all the state of the art works [35, 37, 36, 5, 18, 53] which support the processing of queries over RDF data is the use of a *mapping dictionary* in which the literals and URIs are replaced by unique identifiers (integers). The processing of queries is more efficient (given that most optimizers handle efficiently integers and not large strings – the case of URIs). In addition, the use of unique identifiers allows for better data compression as advocated in [35].

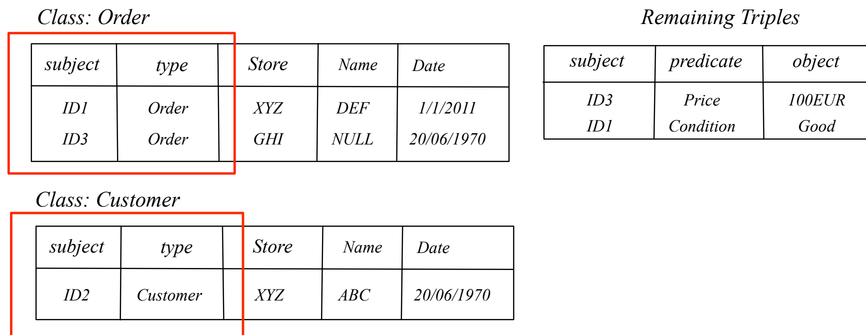
5.1.3 Indexes

The join access patterns supported by state of the art on SPARQL query evaluation engines are: *point*, *graph* and *hierarchical queries*. SPARQL point queries consist of a *single triple*

[1]www.postgresql.org
[2]www.mysql.org
[3]www.oracle.com
[4]http://monetdb.cwi.nl/
[5](http://db.csail.mit.edu/projects/cstore/



(a) Clustered Property Tables



(b) Property-Class Table

Figure 5.3: Property Tables

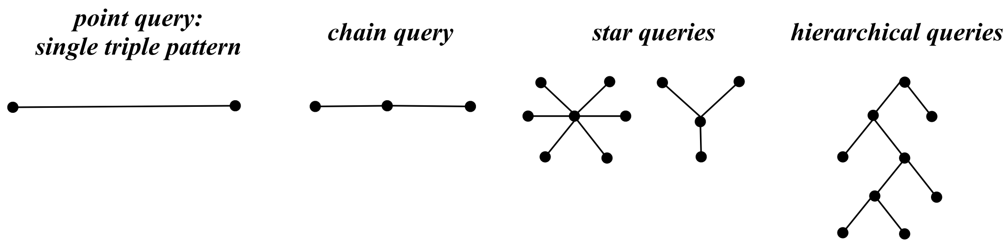


Figure 5.4: SPARQL Access Patterns

pattern whereas *graph queries* involve multiple joins between triple patterns and are distinguished to *path (chain)* and *star queries*. Hierarchical queries can be considered as a special class of path queries that consider the traversal of the RDFS *subclassOf* and *subpropertyOf* relationships (Figure 5.4).

The indexes proposed by the majority of state of the art work mostly support point and graph queries whereas there are few works that discuss indexes to suppport the efficient pro-

cessing of hierarchical queries. Authors in [17] propose indexes implemented as B+-trees on RDFS *class* and *property hierarchies* where classes and properties are assigned labels using a prefix-interval based scheme. The proposed approach achieves a logarithmic complexity for the evaluation of recursive queries that involve the traversal of *subClassOf* and *subPropertyOf* hierarchies as those are defined in an RDFS schema. Authors in [8] create indexes for subgraphs defined on the *subPropertyOf* and *subClassOf* RDFS relations for the efficient processing of queries that involve the recursive traversal of the aforementioned relations.

To process SPARQL point queries, state of the art work proposes a set of indexes that support the efficient retrieval of RDF triples. Authors in [35, 37, 36, 53] that use the schema agnostic approach for the logical storage schema (i.e., triples are stored in a large triple table) define indexes on *all possible permutations* of the triple table. That is, for the triple table *spo* that stores triples of the form (*subject*, *predicate*, *object*) the permutations *ops* (*object*, *predicate*, *subject*), *osp*, *pos*, *pso* are also specified. The benefit of having indexes on all possible permutations of the triple table is that any triple pattern can be answered with a single index scan. In the aforementioned works, the triples in an index are sorted lexicographically by (subject, object, property) for index *spo*, (object, subject, property) for index *osp* etc to facilitate the use of *sort-merge join* operator for evaluating joins.

In [36, 37, 35] the triples in all indexes are sorted lexicographically by the appropriate collation order and are directly stored in the leaf pages of the *clustered B+-tree*. [5, 18] and [53] follow the vertical partitioning approach for the logical storage schema. In [5, 18] the property tables are stored in the column store database C-Store [48]. In this work, the authors construct an index on the *subject* and *object* columns of property tables using *clustered B+-trees* and *non-clustered* B+-trees respectively. On the other hand, in Hexastore [53] a sextuple indexing scheme is used where the RDF triples are not assumed to be stored in a property-headed index (as dictated by the logical storage schema), but also in subject- and object-headed indexes. Authors in [52] discuss that the implementation of the aforementioned indexes using *clustered* B+-trees is more efficient than the implementation that uses B+-trees. [8] follows a more conservative indexing approach where the main objective is to *minimize* the number of indexes built. In this work, the authors propose the use of only a *subset* of the indexes discussed previously that support the aforementioned access patterns. The YARS2 [25] system uses an *inverted text* index built on the triple table that associates the RDF terms (literals, URIs and blank nodes) that appear as the object of a triple to the triple's subject. In this work, to support point queries, the authors store all possible permutations of the triple table on disk and build sparse indexes on the first two components of a triple.

To support graph queries the majority of the aforementioned works propose the use of indexes to compute the *selectivity* of a set of different access patterns. The information about the selectivity is used to calculate the cost of a join in a graph query. Towards this objective, authors in [35, 36] propose the use of *aggregated indexes* that store for *every combination* of values for the subject, predicate and object of a triple the number of triples that match this combination of values. To improve the performance of joins, the authors propose indexes that store the number of triples that can be joined with triples that contain a specific combination of RDF terms. This information is useful in approximating the cardinality of the result of a join. Authors in [35, 37] compute the frequent paths in the RDF dataset to estimate the cost of a join in the case of path and chain queries and use the size of the intermediate results to calculate the cost of a join. In [36] the previous work is extended to compute more accurately the size of a join. The idea is to translate a join between two triple patterns to an equivalent join where the variables are considered independent where essentially the

join happens between the triple table and one triple pattern. Consequently, they manage to compute accurately the size of the result of a join using the previously mentioned aggregated indexes. Finally, the YARS2 system [25] supports join sparse indexes that are implemented in a similar manner as the indexes over the triple table permutations. In [5, 18] the authors store the result of joins in the form of a standard relational table. Authors in [16] compute and store *subject-property matrices* where for each subject of a triple they store property paths that start from a specific subject.

5.1.4 Query Processing

The majority of the state of the art works on SPARQL query processing, consider only the *selection*, *projection* and *join* operators without addressing the *union* and *optional* operators who are mainly responsible for the complexity of SPARQL query evaluation. In the case of the join operation, the key problem that must be addressed lies on the very nature of the RDF data model: due to the fragmentation of information in the form of (*subject*, *property*, *object*) triples, a large number of self joins over a large triple table is required (in the case of the schema agnostic approach), or multiple joins in the case of vertical partitioning. The indexes that were previously presented, aim at supporting *(a)* the efficient retrieval of triples during query evaluation and *(b)* the computation of the cost of joins. The majority of works that follow the schema agnostic approach store the triple table sorted on its components therefore supporting the use of *sort merge joins* for the join evaluation. In [5, 18] the C-Store column store database [48] is used as the query evaluation platform and thus in order to solve the problems encountered in SPARQL query processing that were previously discussed, one must rely on possible optimizations of C-store. A similar approach is followed in [16] where the Oracle[6] database engine is used. To decide on the execution plans and more specifically on join ordering, authors in [35] follow a *dynamic programming* based approach [32]. The optimizer decides which of the available indexes to use by taking under consideration the constants in the triple patterns and choosing from all the possible indexes those that will return the triples in an order that will be useful for a subsequent join operation. To prune an execution plan, the optimizer computes the cost of the plan using the indexes that were previously discussed. In some cases, costly plans (according to the cost model used) are more interesting than cheaper plans because they return the results with an interesting order for a subsequent operation. The authors also follow a pipeline approach for query evaluation allowing the parallel execution of joins. Authors in [36] extend the approach of [35] by proposing an optimization known as *sideways information passing* for reducing the intermediate results for joins [28, 46]. The idea of this approach is that the operators pass information that is related to the join variables that can be used to reduce the size of the result obtained from a *selection* or from a *join* operation. The information carried by the operators is associated with the "gaps" in the values obtained from a scan for a triple pattern. In this case, using the sideways information passing approach an optimizer can stop the scan of a table for a triple pattern given the fact that there do not exist values for the join variable that has been considered. This specific approach is comparable to the semi-join programs [9, 47] and magic sets [34].

[6] www.oracle.com

Chapter 6

Conclusions and Future Work

In this work we propose the first *heuristics-based SPARQL planner (HSP)* that uses a set of heuristics based on syntactic and structural characteristics of SPARQL queries. In particular, *HSP* tries to produce plans that maximise the number of merge joins and reduce intermediate results by choosing triples patterns most likely to have high selectivity. The selection of join ordering is also based on the structural characteristics of the queries. We propose a set of heuristics for deciding which triple patterns of a SPARQL query are more selective, thus it is in the benefit of the planner to evaluate them first in order to reduce the memory footprint during query execution. These heuristics are generic and can be used separately or complementary to each other.

In our work we propose the reduction of the query planning problem to the problem of *maximum weight independent set*. Towards this end, we model a SPARQL query as a *variable graph* where nodes are query variables and edges denote the co-occurence of variables in a triple pattern. The qualifying independent sets are translated to blocks of merge joins connected when needed by other types of more costly joins supported by the underlying engine.

HSP was implemented on top of the MonetDB [33] columnar database engine. Our main focus was on the efficient implementation of *HSP* logical plans to the physical algebra of MonetDB. The main challenge stems from the decomposed model of rows in a columnar database. A main difference between HSP plans and the plans produced by the cost-based standard SQL optimiser of MonetDB is that we produce *bushy* rather than *left-deep* query plans in order to be able to incorporate the maximum number of *merge joins*.

We analyzed the main statistical properties of two synthetic and two real RDF datasets. The synthetic datasets were generated according to the SP2Bench and Berlin SPARQL benchmarks. We considered the YAGO and Barton Library widely used real RDF datasets. Our findings confirmed our underlying assumptions regarding the selectivity of subject-property-object components in a triple pattern as well as the selectivity of the join patterns.

We have experimentally evaluated *the quality* and *execution time* of the plans produced by *HSP* with the state-of-the-art cost-based Dynamic Programming algorithm (CDP) employed by RDF-3X using the aforementioned datasets. In all queries of our workload, HSP produces plans with the *same number* of *merge* and *hash* joins as CDP. Their differences lie on the employed ordered variables as well as the execution order of joins which essentially affect the size of intermediate results.

Our future work includes the study of possible additional heuristics. We wish to investigate the effects of applying our heuristics in a distributed environment such as MapReduce

and Hadoop. We also intend to extend our work to cope with different relational storage schemas (i.e. schema aware, hybrid), instead of only the traditional approach of a triple table (schema agnostic). Moreover, we are working to integrate our solution with MonetDB runtime optimization techniques. These techniques could be used when the heuristics fail to choose the join order evaluation. Finally, we wish to extend our optimizer to include all features of the SPARQL language.

Bibliography

[1] Yet Another Great Ontology. http://www.mpi-inf.mpg.de/yago-naga/yago.

[2] D. Abadi, S. R. Madden, and M. C. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. of the ACM SIGMOD Intb••l. Conf. on Management of Data*, 2006.

[3] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization Strategies in a Column-Oriented DBMS. In *Proc. of the Intb••l. Conf. on Database Engineering (ICDE)*, 2007.

[4] D. J. Abadi, S. R. Madden, and N. Hachem. ColumnStores vs. RowStores: How Different Are They Really? In *SIGMOD*, 2008.

[5] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.

[6] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Using the barton libraries dataset as an rdf benchmark. Technical Report MIT-CSAIL-TR-2007-036, MIT, 2007.

[7] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Sw-store: A vertically partitioned dbms for semantic web data management. VLDB Journal, 18(2), April 2009.

[8] L. Baolin and H. Bo. HPRD: A High Performance RDF Database. In *Network and Parallel Computing*, pages 364–374, 2007.

[9] P. A. Bernstein and D.-M. W. Chiu. Using Semi-Joins to Solve Relational Queries. *JACM*, 28(1):25–40, 1981.

[10] Christian Bizer and Andreas Schultz. The Berlin Sparql Benchmark. *International Journal On Semantic Web and Information Systems*, 2009.

[11] P. Boncz, A. Wilschut, and M. Kersten. Flattening an Object Algebra to Provide Performance. In *In Proc. of the Intb••l. Conf. on Database Engineering (ICDE)*, page 568b•“577, Orlando, FL, USA, 1998.

[12] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. of the Int'l Conf. on Innovative Database Systems Research (CIDR)*, 2005.

[13] D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. www.w3.org/TR/2004/REC-rdf-schema-20040210, 2004.

[14] J. Broekstra and A. Kampman. SeRQL: An RDF Query and Transformation Language. http://www.cs.vu.nl/ jbroeks/papers/SeRQL.pdf, 2004.

[15] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: An architecture for storin gand querying rdf data and schema information. In *Spinning the Semantic Web*, pages 197–222, 2003.

[16] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB*, pages 1216–1227, 2005.

[17] V. Christophides, D. Plexousakis, M. Scholl, and S. Tourtounis. On labeling schemes for the Semantic Web. In *ISWC*, pages 544–555, 2003.

[18] S. Madden D. J. Abadi, A. Marcus and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *VLDB Journal*, 18(2):385–406, 2009.

[19] Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. In *CSSW*, pages 59–68, 2007.

[20] B. McBride F. Manola, E. Miller. RDF Primer. www.w3.org/TR/rdf-primer, February 2004.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory qf NP-Completeness.* Freeman, 1979.

[22] Alan Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[23] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *Proc. of the Intb••l. Conf. on Very Large Data Bases (VLDB)*, 2006.

[24] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk rdf storage. In *PSSS*, 2003.

[25] A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In *ISWC*, pages 211–224, 2007.

[26] S. Idreos, M. Kersten, and S. Manegold. Self-organizing Tuple-reconstruction in Column-stores. In *Proc. of the ACM SIGMOD Intb••l. Conf. on Management of Data*, 2009.

[27] M. Ivanova, M. Kersten, and R. Goncalves. An architecture for recycling intermediates in a column-store. In *In Proc. of the ACM SIGMOD Intb••l. Conf. on Management of Data*, 2009.

[28] Z. G. Ives and N. E. Taylor. Sideways Information Passing for Push-Style Query Processing. In *ICDE*, pages 774–783, 2008.

[29] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *Proc. 11th World Wide Web Conference (WWW)*, Honolulu, Hawaii, USA, 2002.

[30] G. Kobilarov, T. Scott, Y. Raimond, S. Oliver, C. Sizemore, M. Smethurst, C. Bizer, and Robert Lee. Media Meets Semantic Web - How the BBC Uses DBpedia and Linked Data to Make Connections. In *Proc. of the 6th European Semantic Web Conference (ESWC)*, Heraklion, Crete, Greece, 2009.

[31] S. Manegold, P. A. Boncz, N. Nes, and M. L. Kersten. Cache-conscious radix-decluster projections. In *Proc. of the Intb••l. Conf. on Very Large Data Bases (VLDB)*, 2004.

[32] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, 2006.

[33] Monetdb. http://www.monetdb.org/Home.

[34] I. S. Mumick and H. Pirahesh. Implementation of Magic-sets in a Relational Database System. In *SIGMOD*, pages 103–114, 1994.

[35] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.

[36] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD Conference*, pages 627–640, June 2009.

[37] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB Journal*, August 2009.

[38] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB Journal*, 19(1), 2010.

[39] P. R. J. Ostergard. A new algorithm for the maximum-weight clique problem. *Nordic Journal of Computing*, 8:424–436, 2001.

[40] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM TODS*, 34(3), 2009.

[41] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. www.w3.org/TR/rdf-sparql-query, January 2008.

[42] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. In *ICDE*, 2009.

[43] A. Seaborne. RDQL - A query Language for RDF. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/, January 2004. W3C Member Submission 9.

[44] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2), 2008.

[45] M. Sintek and S. Decker. Triple: A Query, Inference and Transformation Language for the Semantic Web. In *Proc. of Deductive Databases and Knowledge Management (DDLP)*, 2001.

[46] M. Steinbrunn, K. Peithner, G. Moerkotte, and Alfons Kemper. Bypassing joins in disjunctive queries. In *VLDB*, pages 228–238, 1995.

[47] K. Stocker, D. Kossmann, R. Braumandl, and A. Kemper. Integrating Semi-Join-Reducers into State-of-the-Art Query Processors. In *Intl. Workshop on Research Issues in Data Engineering*, 2001.

[48] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniak, M. Ferreira, E. Lau, A. Lin, S. Madden, P. E.O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column Oriented DBMS. In *VLDB*, 2005.

[49] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *ISWC*, pages 685–701, 2005.

[50] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.

[51] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, 1988.

[52] C. Weiss and A. Bernstein. On-disk storage techniques for Semantic Web data - Are B-Trees always the optimal solution. In *Int'l Workshop on Scalable Semantic Web Knowledge Base Systems*, 2009.

[53] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[54] K. Wilkinson. Jena Property Table Implementation. Technical Report HPL-2006-140, HP Laboratories Palo Alto, 2006.

[55] K. Wilkinson, C. Sayers, H.A. Kuno, and D. Raynolds. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*, 2003.

[56] M. Zukowski, N. Nes, and P.A. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proc. of the Intb••l. Workshop on Data Management On New Hardware (DaMoN)*, 2008.