Emergent Relational Schemas for RDF

Minh Duc Pham

Committee	prof.dr. Frank van Harmelen prof.dr. Martin Kersten prof.dr. Josep Lluis Larriba Pey prof.dr. Thomas Neumann dr. Jacopo Urbani

The research reported in this thesis has been partially carried out at CWI, the Dutch National Research Laboratory for Mathematics and Computer Science, within the theme Database Architectures.

The research reported in this thesis has been partially carried out as part of the continuous research and development of the MonetDB open-source database management system.

SIKS Dissertation Series No. 2018-19 The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

The cover was designed by the author. Photo by Leo Rivas on Unsplash.

The printing and binding of this dissertation was carried out by Ipskamp Printing.

ISBN 978-94-028-1110-0

VRIJE UNIVERSITEIT

Emergent Relational Schemas for RDF

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor
aan de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op donderdag 6 september 2018 om 15.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Minh Duc Pham

geboren te Bac Ninh, Vietnam

promotor: prof.dr. P.A. Boncz
copromotor: prof.dr. S. Manegold

Tặng bố mẹ của con vì tình yêu thương vô bờ bến,
Tặng em yêu vì bao gian khổ, ngọt ngào
và con trai - nguồn vui vô tận ...

Abstract

The main semantic web data model, RDF, has been gaining significant traction in various domains such as the life sciences and publishing, and has become the unrivaled standard behind the vision of global data standardization and interoperability over the web. This data model provides the necessary flexibility for users to represent and evolve data without prior need of a schema, so that the global RDF graph (the semantic web) can be extended by everyone in a grass-roots and pay-as-you-go way. However, as identified in this thesis, this flexibility which de-emphasizes the need for a schema and the notion of structure in the RDF data poses a number of data management issues in systems that manage large amounts of RDF data. Specifically, it leads to (i) query plans with excessive join complexity which are difficult to optimize, (ii) low data locality which blocks the use of advanced relational physical storage optimizations such as clustered indexing, data partitioning, and (iii) a lack of schema insight which makes it harder for end-users to write SPARQL queries with non-empty-results.

This thesis addresses all three problems. We uncover and exploit the fact that real RDF data, while not as regularly structured as relational data, still has the great majority of triples conforming to regular patterns. Recognizing this structure information allows RDF stores to become both more efficient *and* easier to use. An important take-away from this thesis is that the notion of "schema" is understood differently in semantic web than in databases. In semantic web "schema" refers to ontologies and vocabularies which are used to describe entities in terms of their properties and relationships in a generic manner, that is valuable across many different application contexts and datasets. In databases, "schema" means the properties of data stored in a single database. We argue both different notions of schema are valuable. Semantic schemas could be a valuable addition to relational databases, such that the semantics of a table (the entity it may represent) and of its columns and relationships is made explicit. This can facilitate data integration. Relational schemas are valuable for semantic web data, such that RDF stores can better organize data on disk and in memory, SPARQL engines can do better optimizations, and SPARQL users can better understand the nature of an RDF dataset. This thesis concentrates on these latter points. Concretely, we propose novel techniques to automatically derive a so-called emergent relational schema from an RDF dataset that recovers a compact and precise relational schema with high triple coverage and short human-readable labels. Beyond the use of the derived emergent relational schema for conveying the structure information of RDF dataset to users and allowing humans to understand RDF dataset better, we have exploited this emergent

schema internally inside the RDF system (in storage, optimization, and execution) to make RDF stores more efficient. In particular, using emergent relational schema allows to make RDF storages more compact and faster-to-access, and helps reducing the number of joins (i.e., self-joins) needed in SPARQL query execution as well as the complexity of query optimization, showing significant performance improvement in RDF systems. This approach opens a promising direction in developing efficient RDF stores which can bring RDF-based systems on par with relational-based systems in terms of performance without losing any of the flexibility offered by the RDF model.

Besides the contributions on developing high performance RDF stores using the automatically derived emergent relational schema, in this thesis, we also provided insights and materials for evaluating the performance and technical challenges of RDF/graph systems. Particularly, we developed a scalable graph data generator which can generate synthetic RDF/graph data having skewed data distributions and plausible structural correlations of a real social network. This data generator, by leveraging parallelism though the Hadoop/MapReduce paradigm, can generate a social network structure with billions of user profiles, enriched with interests/tags, posts, and comments using a cluster of commodity hardwares. The generated data also exhibited interesting realistic value correlations (e.g., names vs countries), structural correlations (e.g., friendships vs location), and statistical distributions (e.g., power-law distribution) akin to a real social network such as Facebook. Furthermore, the data generator has been extended and become a core ingredient of an RDF/graph benchmark, LDBC Social Network Benchmark (SNB), which is designed to evaluate technical challenges and solutions in RDF/graph systems.

Samenvatting

Het semantische web-datamodel, RDF, wordt in toenemende gebruikt in verschillende domeinen zoals de life sciences en publishing, en is uitgegroeid tot standaard voor wereldwijde gegevensstandaardisatie en interoperabiliteit. RDF biedt flexibiliteit voor gebruikers om gegevens weer te geven en te ontwikkelen zonder dat daar een schema voor nodig is, zodat de wereldwijde RDF-graaf (het "semantische web") door iedereen kan worden uitgebreid op eigen initiatief. Deze flexibiliteit brengt een aantal problemen met zich mee in systemen die grote hoeveelheden RDF-gegevens beheren, omdat het de behoefte aan een schema en het begrip van structuur in de RDF-gegevens minder benadrukt. In de eerste plaats verhoogt het ontbreken van schema-informatie de complexiteit van query-optimalisatie, zodat in de praktijk RDF database-systemen een veel kleiner gedeelte van de zoekruimte kunnen bekijken, en er slechtere en dus veel langzamere query-plannen gevonden worden. Daarnaast zorgt de lage gegevenslokaliteit ervoor dat het gebruik van geavanceerde fysieke opslagoptimalisaties voor relationele databases, zoals geclusterde indexering en gegevenspartitionering, niet mogelijk is. Tot slot is het door een gebrek aan schema-inzicht moeilijk voor eindgebruikers om goede SPARQL queries te schrijven. Dit proefschrift gaat in op elk van deze drie problemen. We ontdekken en exploiteren het feit dat echte RDF datasets in vrij hoge mate tabulair gestructureerd zijn. Het automatisch herkennen van zulke structuur maakt het mogelijk RDF-opslag efficiënter en gebruiksvriendelijker te maken.

Een belangrijke constatering van dit proefschrift is dat aan het begrip "schema" een verschillende betekenis toegekend wordt in het semantisch web dan in databases. Binnen het semantisch web verwijst "schema" naar ontologieën en vocabulaires die worden gebruikt om concepten op een generieke manier te beschrijven, zodat die concepten in vele situaties en toepassingen (her-)bruikbaar zijn. In databases verwijst "schema" naar iets heel anders, namelijk naar de specifieke structuur van gegevens in een enkele dataset. Wij betogen dat beide betekenissen van een schema waardevol zijn. Semantische schema's zouden een waardevolle toevoeging kunnen zijn aan relationele databases: de semantiek van een tabel (de entiteit die het kan vertegenwoordigen) en van zijn kolommen en relaties wordt expliciet gemaakt. Dit kan de integratie van gegevens uit verschillende databases vergemakkelijken. Relationele schema's zijn ook waardevol voor semantische webgegevens: de opslag van RDF-gegevens op een schijf of in geheugen kan er beter mee georganiseerd worden zodat RDF-databases betere optimalisaties kunnen uitvoeren, en gebruikers kunnen beter begrijpen welke attributen werkelijk in een RDF-dataset aanwezig zijn.

Dit proefschrift stelt nieuwe technieken voor om automatisch een zogenaamd

"emergent" relationeel schema af te leiden van een RDF-dataset. Het resultaat is een compact en nauwkeurig relationeel schema waarin tabellen, kolommen en relaties korte namen krijgen die makkelijk voor mensen leesbaar zijn. Dit emergente relationele schema is niet alleen nuttig om mensen de structuur RDF-gegevens beter te laten begrijpen; het kan ook de computer helpen om een RDF database-systeem efficiënter te maken. In concreto, het gebruik van een emergent, relationeel schema maakt het mogelijk om RDF-opslag compacter en sneller toegankelijk te maken. Daarnaast helpt het bij het verminderen van het aantal joins (met name zelf-joins) dat nodig is voor SPARQL-queries en het verlagen van de complexiteit van de query-optimalisatie. Dit leidt tot een significante prestatieverbetering in RDF-systemen. Onze methode biedt een veelbelovend perspectief op het ontwikkelen van een efficiënte RDF-opslag die zich kan meten met relationele systemen qua prestatie zonder in te leveren op de flexibiliteit die het RDF-model biedt.

Naast de bijdragen aan het ontwikkelen van hoogwaardige RDF-opslag die gebruik maakt van het automatisch afgeleide, emergente, relationele schema geven we in dit proefschrift ook inzichten en methodes voor het evalueren van de prestaties van RDF-systemen. We hebben een schaalbare datagenerator ontwikkeld die synthetische RDF-graph gegevens kan genereren met scheve datadistributies en plausibele structurele correlaties. Deze gegevensgenerator kan dankzij parallellisatie via Hadoop / MapReduce, een sociale netwerkstructuur genereren met miljarden gebruikersprofielen, verrijkt met interesses, labels, berichten en opmerkingen met behulp van een cluster van alledaagse hardware. De gegenereerde data vertonen ook interessante, realistische waardecorrelaties (bijv. namen vs. landen), structurele correlaties (bijv. vriendschappen versus locatie) en statistische verdelingen ("power laws") die vergelijkbaar zijn met een echt sociaal netwerk zoals Facebook. Deze gegevensgenerator vormt nu de kern van een industriële benchmark, de LDBC Social Network Benchmark (SNB), die is ontworpen om RDF-graph systemen te evalueren.

Acknowledgements

I would dedicate this dissertation to my mother, mẹ Tâm, who planted the seed that I base my life on, but does not have opportunity to witness her beloved son complete his PhD. Her unconditional love, her everlasting belief and encouragement, however, gave me the strength to not give up in this epic journey.

A long journey would not be completed without the supports of many people. When it is about to end, I would like to take this unique opportunity to greatly thank all people whom I deeply owed for their great help.

First and foremost, I would like express my deep gratitude to my supervisor, prof. Peter Boncz, who picked me up from Schiphol Airport when I first arrived to Amsterdam and patiently guided me throughout my PhD journey. Peter is an inspiring and passionate scientist whom I would never stop learning from until the last day working with him. I honestly admitted that I owed him a lot. Without him, I would never obtain what I have now.

Being a member of MonetDB team, the Database Architecture group at CWI, is a honor. I was so glad and proud to work and share my PhD time with those great minds and supportive members. In particular, I would like to thank Prof. Martin Kersten and Prof. Stefan Mangegold for not only sharing me their innovative ideas and thoughtful insights, but also willingly supporting me with administrative paperwork. I would like to sincerely acknowledge Sjoerd Mullender whose the office door was always open for my questions on MonetDB coding, and Dr. Niels Nes for his eager support with my implementation of RDF/SPARQL in MonetDB SQL engine. My academic life would be a lot less interesting without other colleagues in the team: Thank you Mrunal, the 3D Hologram-mate, Holger, my 4-year officemate, Jenny, Thibault, Eleni, Yagiz, Stratos, Erietta, Sandor, Romulo, Lefteris, Hannes, Bart, Kostis, Fabian, Arjen, Eyal, Pedro, Mark, Panagiotis, Tim and Benno.

Further more, I would like to thank all committee members, Frank van Harmelen, Martin Kersten, Josep Lluis Larriba Pey, Thomas Neumann, and Jacopo Urbani for their time and efforts in being the committee members of my thesis.

Outside of my academic life and CWI office, lots of friend have made my life in the Netherlands much more meaningful. I would want to thank my badminton teammates Mathieu, Joyce, Koen and Mara for many great times together. I want to thank lots of Vietnamese friends in Amsterdam and Eindhoven, especially the Catan buddies Vân, Cảnh, Tú and Dũng, who always challenged me on the board game, and the football team whom I shared lots of joyful weekends.

My kisses and hugs go to all my family members for their endless support dur-

ing this lengthy journey. Their encouragement was worth more than I can express on paper. I am so grateful to my father, bố Hưởng, my mother-in-law, mẹ Nga, and my brother-in-law, em Đức, who have come to the Netherlands to give me a hand during many hard times. I deeply owed my brother, anh Việt Anh and his wife, chị Hiền, for taking great care of our family when I was busy working on my research. I would want to thank my brother, em Thịnh and his wife, em Hường, for their instant help when we are both living in a foreign country.

Finally, I would send all my heartfelt thanks to my beloved wife, Minh Vân, and my son, Minh. I am forever indebted to my wife for endlessly supporting and loving me with her compassion and understanding through the toughest moments of my life. Without her beside, I would never accomplish this degree. Thanks to my dear son for every single inspiring moments and being the endless driving force of my life. My every day with them has been a true gift.

Contents

Chapter 1

Introduction

1.1 The Semantic Web

Conceptually outlined by Berners-Lee et al. in 2001, the Semantic Web was proposed as an extension of the Web with semantic meta-data annotations. Specifically, via globally agreed-upon identifiers (e.g., the Uniform Resource Identifier (URI)), a well-defined data model (e.g., The Resource Description Framework (RDF)), and a schema language (RDF Schema) and many other standards, it provides a common framework for accessing, sharing and reusing data across Web applications and platforms [50]. In the RDF data model, a data set is represented as a collection of $<subject, predicate, object>$ triples, in which the object can be considered as the value for the property (i.e., predicate) of the described resource (i.e., subject). Using *vocabularies* and *ontologies* built from the so-called Web Ontology Language, the Semantic Web can enrich data with additional meaning, allowing not only people but also machines to automatically derive meaningful content from Web data.

Semantic Web technologies got early attention from within the AI research community who identified it as the emerging standard for knowledge representation. Subsequently, with the help of this community, standards for defining ontologies and reasoning were devised (OWL, SKOS, RDFS) [140, 142, 137]. Through the combination of data model, schema language, ontology definitions and reasoning from simple sub/super-class relationships (RDFS) to more complex (and less tractable) OWL profiles, the Semantic Web and its ontology language become important artifacts to represent human knowledge.

However, actual usage of Semantic Web data remained low, until researchers shifted attention to data management aspects, specifically the Linked Open Data (LOD) initiative that started around the first releases of DBpedia[1] (a semantic version of Wikipedia[2]). The drive to create more and more RDF datasets, preferably in open source, is seen there as an enabling factor that will contribute to the growing popularity of Semantic Web standards. The large volumes of RDF data available make the case for a database-style interaction, i.e. using a query language, for

[1] dbpedia.org
[2] wikipedia.org

which the SPARQL language was adopted [161]. The movement to "liberate" data in RDF should also be seen as part of the more general drive towards *open* data, also promoted among governments to increase democratic accountability but also as a driver for economic innovation based on public data. Berners Lee introduced the 5-star deployment scheme for open data[3] on the Web, where the highest standard (5-star data) is given to data that uses URIs to denote things and links to other data to provide context (i.e., RDF). As a result, the Semantic Web data volume has significantly increased with tens of thousands datasets publicly shared on the Web. In particular, the Linked Open Data initiative has been building a so-called LOD-cloud[4] of more than 50 billion RDF triples[5] from hundreds Semantic Web datasets in which each dataset may contain millions of entities (e.g., DBpedia[6]). Nevertheless, there remains a long way to go as the majority of open data has lower star ratings, such as 1-star data in closed formats (e.g. PDF), or three-star data in open formats (CSV files). Even much of the LOD cloud should actually be considered 4-star data, since the 5-star denomination is given if the data is actually interlinked, something which is only sparsely present in the current LOD Cloud.

While Linked Open Data is still struggling in capturing the awareness from the broad field of ICT outside the semantic web community, the day-to-day internet has silently been "flooded" with RDF through the emergence of semantic web annotations in web pages' metadata[7]. The flagship semantic web technologies in annotating the web metadata are "schema.org" and RDFa (the Resource Description Framework in Attributes). Launched by Google, Bing, Yahoo, and lately Yandex in 2011 "to create and support common ways to represent web page metadata", the schema.org vocabulary together with such machine-readable formats as Microdata and RDFa facilitate search engines in retrieving the meaning of each web page via semantic web annotations embedded in its metadata. This allows better rendering of commercial offerings (e.g., hotels, restaurants, products in search results) at a relatively little effort for the data publisher. The success of internet search annotations shows that the Semantic Web has potential for growth and can become commercially viable, though idealistic Linked Open Data proponents may be a bit off-put that the largest success of RDF so far has been in relatively shallow commercial product advertising.

1.2 RDF data management

The collection of triples in an RDF datasets forms a labeled directed graph. SPARQL is the W3C recommendation query language for RDF graphs, essentially allowing subgraph search. Most RDF storage systems that implement SPARQL internally store their data in SQL-based database systems using so-called triple tables (e.g., Sesame [68], Jena SDB [73, 185], ICS-Forth RDFSuite [38], Virtuoso [87], Oracle [23]). As such, each triple table contains three columns corresponding to the

[3] http://5stardata.info
[4] http://lod-cloud.net
[5] State of the LOD cloud http://lod-cloud.net/state
[6] dbpedia.org
[7] The availability of microdata, RDFa in web data http://webdatacommons.org/structureddata

$subject$, $predicate$ and $object$ (S,P,O) of the RDF triple. Even "native" SPARQL systems (e.g., 4store [105], RDF-3X [150], Hexastore [183]) that do not built directly on SQL technology still often adopt this representation, e.g. storing the triple tables as B-tree indexes (typically replicated, using different orders of S,P,O as index key).

The proponents of RDF often highlight two advantages of the model: *(i)* it is based on URIs such that not only meta-data but also data *instances* (e.g., "keys") can be standardized for interoperability over the web and *(ii)* it is extremely flexible and imposes few schema constraints [54], so the global RDF graph (the semantic web) can be extended by everyone in a grass-roots and pay-as-you-go way. However, some database researchers have taken a critical stance towards RDF [109, 39, 99, 149] because *(ii)*: RDF de-emphasizes the need for a schema and the notion of structure in the data, and this leads to performance issues in systems that manage large amounts of RDF data. Specifically, the reliance of RDF stores on triple tables leads to query plans with many self-joins. Also, the lack of a multi-attribute object structure in triple storage blocks the use of advanced relational physical storage optimizations, such as clustered indexing, hash/range data partitioning, etc., which are the cornerstone of mature data warehousing solutions. Our research in Section 3 reveals that despite the fact that most RDF data does not have a (RDFS) schema, the great majority of RDF triples in actual datasets do conform to regular structural patterns. Additionally, the lack of a schema also makes it harder for users to formulate queries on RDF graphs as it may not be obvious to the user which combination of triple predicates actually occurs in the data. To tackle this latter problem, the semantic web community has recently been studying graph structure analysis techniques to construct visual graph summaries to help users comprehend RDF graphs [72].

Despite these issues, RDF is the unrivaled standard behind the vision of global data standardization (e.g., LOD, see point *(i)*), and simply because RDF has been gaining significant traction in certain domains, such as the life sciences. With quickly growing RDF data volumes, there is a true need to better support it in database systems.

1.2.1 The RDF data management challenges

Even though there have been significant efforts in building efficient RDF stores, we identify here three main problems in RDF data management, namely (i) excessive join complexity, (ii) low storage locality and (iii) lack of user schema insight.
Excessive join complexity. Consider a simple SPARQL query:

```
SELECT ?a ?n   WHERE {
   ?b <has_author>     ?a.
   ?b < in_year> ''1996''.
   ?b <isbn_no>  ?n   }
```

This SPARQL query looks for the author and the ISBN number of a book published in 1996. Despite the fact that a book entity almost always has both the

isbn_no and has_author properties, the query plan typically used for this query by triple store systems still needs two separate joins for these properties to construct the answer (as shown in Figure 1.1). Note that a relational database system storing Book information would have a Book table and this would be a simple Scan-Select query without a join. The problem of having unnecessary joins is serious in most SPARQL queries as they commonly ask for many properties from a common subject i.e., containing so-called "star" pattern [96, 99]. However, relational query processors that know about the structure of data waste no effort here as they would store all data for each entity in a table (e.g., Book table). The only joins they process are "real" joins between different entities. The superfluous joins in SPARQL queries are not only costly at query execution time but also explode the query optimization complexity since the optimization search space (e.g., the number of bushy join trees) generated by widely-used dynamic programming algorithms for finding a good join order is $\mathcal{O}(3^N)$ where N is the number of joins [146]. In other words, if SPARQL queries contain star patterns of average size k, then the SPARQL query optimizer search space is $\mathcal{O}(3^k)$ times larger than necessary.

To make matters even worse, being unaware of structural correlations (e.g., the presence of a isbn_no triple makes the occurrence of a has_author triple with the same subject almost a certainty) also makes it difficult to estimate the join hit ratio between triple patterns. Capturing all correlated predicates in query plan cardinality estimation is unfeasible in the general case, resulting in the situation that even state-of-the-art query optimizers use the "independence assumption" (i.e., calculating the selectivity of conjunctive predicates using the simple product of individual predicate selectivities). In other words, if 1:100 of subjects have a has_author triple as well as isbn_no triple, the query optimizer will estimate the probability to have both as $\frac{1}{100} \cdot \frac{1}{100} = 0.0001$ while the real value is 0.01. Therefore, due to the independence assumption of that query optimizer, the cost model and the result sizes will be badly estimated, causing the choosing of a wrong query plan.

Concluding, SPARQL queries have more joins than necessary. This is not only a problem during runtime execution because of the extra join work. The other problem is that query optimization takes exponentially more effort, and given that search algorithms cannot cover the full space in queries with many joins and cut short the search, this often results in missing the best plan. Finally, the third problem is that in determining what is the best plan, a query optimizer depends on cost estimation, of which cardinality estimation is the most important component [125]. And due to the predicate correlation typically associated with these extra joins (star patterns) the estimates are often very wrong. The result of these three problems often is a performance disaster on complex SPARQL queries in RDF systems based on triple storage.

Low storage locality. A crucial aspect of efficient analytical query processing is data locality, as provided by a clustered index or partitioning schemes [163]. However, without the notion of classes/tables with regular columns/attributes, it is impossible to formulate a clustered index or partitioning schemes, which RDF stores therefore do not offer.

Current state-of-the-art RDF stores such as RDF-3X[150], Hexastore[183] cre-

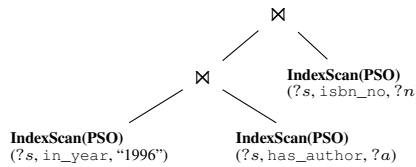


Figure 1.1: Example query plan

ate exhaustive indexes for all permutations of subject(S), predicate(P), object(O) triples as well as their binary and unary projections. This abundance of access paths does not create any of the access locality that a relational clustered index or partitioning scheme offers. As shown in the Figure 1.2a, the above example SPARQL query may use a OPS index (or POS index) to execute a range selection on the `in_year` attribute quickly, resulting in an ordered list of S values, however, for retrieving the other attributes (i.e., `isbn_no` and `author`) it needs a CPU intensive nested-loop index join into a PSO index; one for each attribute. This nested-loop join will hit the index all over the place: no locality despite so-called exhaustive indexing.

This is similar to relational query processing without index locality, i.e. unclustered index access: while the first access to the index is fast, the subsequent RID-list needs to be fetched from the main table leading to random lookups, resulting in random disk-IO or CPU cache misses. Even if the lookups can use index structures (e.g., B-tree), we get large amounts of random fetches, which on current hardware does not scale. Due to the growing imbalance between latency and bandwidth, both in the disk and the RAM level of the memory hierarchy, the cut-off point where such index-lookup queries are better than sequential column (range) scans has been exponentially decreasing. Where previously a 5% or smaller selection predicate could be run faster with an index than with a full scan, in 2012 this is 0.0000005% (and decreasing) [33, 64].

While RDF systems with exhaustive indexing still fall into the trap of non-locality of access, relational systems with their common optimization techniques such as clustered indexing and table partitioning can fully preserver the data locality for such star pattern SPARQL queries as the above example query. Specifically, as shown in Figure 1.2b, using clustered index, all the data records of the `Book` table are physically stored on the disk in the exact same order with respect to the clustering key `year`. The qualifying records can be retrieved by following a binary search on the clustering key `year` in order to locate the start and end RIDs of the qualifying data records ($\mathcal{O}(logn)$ time complexity), and then sequentially fetching a contiguous collection of records (from start to the end qualifying RIDs) which typically touches only a few data pages. Alternatively, using the table partitioning technique, the relational systems subdivides the table into small chunks accruing to the ranges (range partitioning) or the hash-based value (hash partitioning) of an attribute (e.g., `year` for the example data as shown in the Figure 1.2c) so that data records can be accessed at a finer level of granularity. In order to retrieve the qualifying data for a selection predicate (e.g., `year` = 1996), table partition pruning

O	P	S
. . .		
1995	year	4
1996	year	0
1996	year	6
1997	year	3
1998	year	2
. . .		

⋈

P	S	O
. . .		
isbn	0	i1996
isbn	2	i1998
isbn	3	i1997
isbn	4	i1995
isbn	6	i1996
. . .		

⋈

P	S	O
. . .		
auth.	0	a1996
auth.	2	a1998
auth.	3	a1997
auth.	4	a1995
auth.	5	foo
auth.	6	a1996
auth.		
. . .		

(a) Triple tables: while the first OPS access has locality (green), the subsequent PSO joins hit the indexes without locality (red)

year	author	isbn
. . .		
1995	a1995	i1995
1996	a1996	i1996
1996	a1996	i1996
1997	a1997	i1997
1998	a1998	i1998
. . .		

(b) Relational clustered index

1995

author	isbn
a1995	i1995

1996

author	isbn
a1996	i1996
a1996	i1996

1997

author	isbn
a1997	i1997

1998

author	isbn
a1997	i1997

(c) Relational partitioned tables

Figure 1.2: Access locality on the example Book query: Triple tables (a) vs relational clustered index (b) and partitioned tables (c). Both (b) and (c) achieve access locality (green)

will be performed to prune all the non-matching partitions whose their partitioning attribute `year` does not match the selection predicate. Then, the results can be retrieved by sequentially fetching qualifying records from a typically very small remaining chunk of data.

Essentially, the problem of non-locality in RDF query plan boils down to the numbering scheme for object identifiers (OIDs) in RDF systems. While loading RDF triples, current RDF stores typically assign OIDs to Subject (S), Predicate (P), Object (O) in order of appearance. This data-import friendly order might be quite random and uncorrelated with the access paths of interest to the database users. Given the fact that the OID order (whatever it happens to be) is heavily exploited in RDF systems, this is in fact the direct cause of non-locality in RDF query plans. Thus, one of the things that could be done in order to gain locality in RDF systems is to re-order the OIDs in a meaningful way such as ordering the OIDs of S,P,O with respect to important properties (e.g., `year`), and grouping triples by the entity they describe. However, this is not trivial as it may not be clear which entity a triple belongs to, and subjects of different entities may share the same property. Obviously, in order to do this properly, one needs to understand the

structure of RDF graph which is still missing in current RDF systems and will be addressed in our work.

Lack of user schema insight and empty query results. SPARQL query writers who are not familiar with the data they are querying, face the problem of having to guess which properties may actually occur in the data. Even if they would be informed by ontology classes, that define entities and their properties, queries that use these properties in star patterns will come back with empty answers if one or more of these properties does not occur in the data. What is missing in short is a relational database schema.

In this thesis, which walks the boundary of database techniques and semantic data management, we observe that the notion of "schema" is differently understood in the Semantic Web and database research communities. In the Semantic Web, "schemas" are ontologies and vocabularies which aim at modeling a knowledge universe in order to allow diverse organizations to consistently denote certain concepts in a variety of contexts, and are not required to be defined upfront ("schema-last"). Whereas, in the database world, a "schema" describes the structure of one particular dataset (i.e., database) without the intention for reuse or data integration, and must be declared before using the data ("schema-first"). We argue that both of the schema notions are valuable and should be available to data stores and their users. Relational database applications could benefit from the data integration power of the Semantic Web schema if tables and column names would have a meaning conveyed by a URI defined by an ontology. Semantic Web applications could profit from a relational database schema in order to help users better understand the dataset [160] and make RDF systems more efficient [158].

In this thesis, we are interested in deriving a relational schema for RDF data automatically. This schema could help SPARQL users write meaningful queries but it would also allow potentially to consider using SQL as a query language for RDF data, which would enable a huge amount of installed base of software tools to leverage Semantic Web data.

1.2.2 Self-organizing structured RDF

In this thesis, we propose self-organizing structured RDF data management in order to tackle the afore-mentioned three RDF data management problems. As the causes of these problems come from the fact that RDF model does not pay attention to the structure present in RDF graph, the key idea in this thesis research is to fully automatically discover the structure of RDF data sets without losing the flexibility of RDF, and leverage this structure both internally inside the database system (in storage, optimization, and execution), and externally towards the users who pose queries. This idea has been realized and experimentally evaluated inside the open-source MonetDB column-store system[8], known for its adaptive storage structures (such as Recycling [116] and Cracking [114]).

Our approach is to first provide an efficient technique for automatically discovering a compact and precise "emergent" relational schema in RDF datasets which

[8] www.monetdb.org

covers most of the input RDF triples (e.g., 90% of the dataset) and has useful labels. This schema not only helps the user to have better understanding of the RDF dataset, but also can be used for making RDF store much more efficient. Specifically, by exploiting this schema, we physically store the majority of the data ("regular" triples) in relational tables under the hood, and use a reduced triple table (e.g., PSO table) for the remaining "exception" data. With our relational table-based storage scheme, columnar decomposition offers much more compact storage as well as faster access than a normal triple table. Figure 1.3 shows the architecture of the proposed RDF store. As shown in this figure, the proposed store supports both SQL and SPARQL. This architecture is specifically applicable for RDF stores building on top of a relational technology such as MonetDB [61], Virtuoso [87], IBM-DB2 [63], Oracle [23] to adopt. As a by-product, all existing SQL tools such as data visualization can also be used for RDF data.

In our proposal self-organization is performed at bulk-load time. Subsequent modifications to the data are handled by inserting new triples into the exception table and possibly deleting rows from tables. As the exception table grows, periodically tuples are moved from there into the relational tables in addition to periodical self-reorganization that will add columns to tables or add new tables.

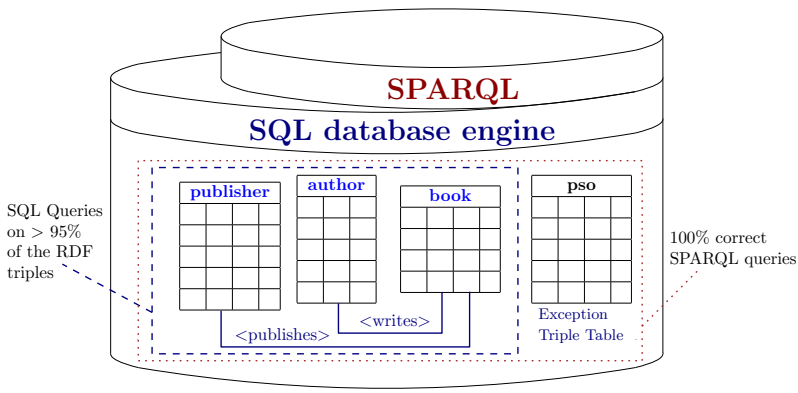


Figure 1.3: Proposed RDF store's architecture

1.2.3 Research questions

In order to realize the idea of building efficient self-organizing RDF store as well as providing user schema insight from RDF data, the following are the research questions which need to be addressed in the thesis:

Question 1: What is an "emergent" relational schema exactly and to which extent do actual RDF datasets conform to it?

Question 2: How to efficiently and scalably discover an emergent relational schema representation including foreign key relationships from RDF datasets?

Question 3: How to derive human-friendly names for the tables and columns in the emergent relational schema?

Question 4: How to exploit the emergent schema in order to make RDF stores efficient in terms of storage?

Question 5: How to exploit the emergent schema in order to make RDF stores efficient in terms of query execution?

Question 6: How to exploit the emergent schema in order to make RDF stores efficient in terms of query optimization?

Question 7: How do we exploit the emergent schema with minimum impact to RDBMS kernel?

1.3 RDF and graph benchmarks

During the course of this PhD research, addition work was performed on the topic of RDF database benchmarking. In order to evaluate the performance of RDF stores, a number of RDF/SPARQL benchmarks have been proposed such as BSBM [55], LUBM [100], SP2Bench [169], and LDBC SNB [88]. As our work on emergent schemas aims at developing a high performance RDF store, RDF/SPARQL benchmarks play an important role in evaluating and analyzing technical challenges in our research. During this PhD, a lot of time was invested in the LDBC project[9] in order to develop and standardize the LDBC SNB benchmark (Social Network Benchmark) and specifically on developing its novel data generator. A problem with existing benchmarks is that they are either limited in representing real datasets or are mostly relational-like [86]. While real-life data is highly correlated with skewed data distributions, these previous benchmarks commonly assumed data independence in their generated datasets, with often uniform data distributions. Besides, as the data models and the logical schemas of these benchmarks can be easily represented in the relational model as well, they hardly highlight the advantages of using RDF/SPARQL in modeling and processing generated data. This also de-motivates RDF vendors in leveraging RDF support for their database engines since the relational model with SQL can often perform even better than RDF engines over such regularly-shaped homogeneous datasets. Therefore, in order to test the performance of RDF stores over real datasets, building a benchmark that can generate a synthetic dataset simulating the real knowledge bases with highly correlated data and foster the advantages of RDF/SPARQL model is truly needed.

As RDF data can also be viewed as graph data, the emerging class of "property graph" databases can also be used for efficiently storing and querying RDF datasets. In recent years there has been a flurry of activity around graph databases, especially considering start-up companies providing new graph database systems (e.g., AllegroGraph[1], Bigdata[7], Neo4j[22], Sparksee[28], Virtuoso[31])[10]. However, by the time of our research on the RDF and graph benchmarks, no real property graph database benchmarks were available. Therefore, the LDBC SNB benchmark is not only aimed at evaluating RDF stores, but also designed for evaluating graph database systems. Specifically, our work aims at creating a RDF and graph benchmark for challenging query processing over scalable highly connected graphs in

[9] LDBC council, http://ldbcouncil.org/

[10] Graph database projects (http://en.wikipedia.org/wiki/Graph _database)

which the generated graph has specific characteristics of a social network and real data correlations. To do that, these following research questions will be addressed in the thesis:

Question 8: How to scalably generate realistic RDF/graph data that simulates the skewed data distributions and plausible structural correlations in a real social network graph?

Question 9: How to design an RDF/graph benchmark over the realistic dataset so that important technical challenges for RDF/graph database systems will be evaluated?

1.4 Thesis Outline and Contributions

The thesis studies RDF data management systems and RDF database benchmarks and is structured as follows.

Background and Related work. In Chapter 2, we first present the background and related concepts on Semantic Web technologies, focusing on RDF data model. We then discuss related works on RDF data management and RDF/graph benchmarks which inspired and motivated the research in this thesis.

Deriving an Emergent Relational Schema. Chapter 3 addresses the first three research questions on efficiently and scalably discovering a compact and precise emergent relational schema from RDF datasets. The research in this chapter is based on the following published paper:

- Minh-Duc Pham, Linnea Passing, Orri Erling and Peter Boncz. Deriving an Emergent Relational Schema from RDF Data. *Proc. WWW Conference*, Florence, May 2015.

Exploiting Emergent Schemas to make RDF systems more efficient. Chapter 4 presents our effort in taking advantage of derived emergent schema to more compactly store RDF data and more efficiently optimize and execute SPARQL queries. In this chapter, we also propose to extend the database kernel with a new query processing operator called RDFscan for handling the exception data. This addresses the research questions four to seven. The published papers for this chapter are:

- Minh-Duc Pham, Peter Boncz. Exploiting Emergent Schemas to make RDF systems more efficient. *Proc. ISWC*, October 2016
- Minh-Duc Pham , Peter Boncz. Self-organizing Structured RDF in MonetDB. PhD Symposium, *ICDE*, 2013.

Benchmarking RDF stores In Chapter 5, we describe research work on RDF benchmarking performed while participating in the LDBC Social Network Benchmark task force. We shortly introduce and describe this benchmark focusing specifically on my contribution in designing and developing its scalable correlated graph generator. The work on this chapter gives the answer for the research questions 8 and 9, and is based on the following published papers:

- Orri Erling, Averbuch,A., Larriba-Pey, J., Hassan Chafi, Gubichev, A., Prat, A., Minh-Duc Pham, Boncz, P. The LDBC Social Network Benchmark: Interactive Workload. *Proc. SIGMOD*, Melbourne, 2015.
- Minh-Duc Pham, Peter Boncz, Orri Erling. S3G2: a Scalable Structure-correlated Social Graph Generator. *Proc. TPCTC*, Istanbul, 2012.

Conclusion Chapter 6 concludes the thesis and discusses future directions on RDF data management.

Chapter 2

Background and Related Work

2.1 Semantic Web Technologies

This section is intended to provide basic knowledge on Semantic Web technologies for readers outside of the Semantic Web community such as database researchers. Thus, Semantic Web readers may skip this section and move directly to the Section 2.2.

> The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. - W3C -

Semantic Web technologies enable people to create formal description of concepts, build vocabularies and rules for given knowledge domains, and create data that can be shared and reused across applications. Most of these technologies such as RDF, SPARQL, OWL are represented in the Semantic Web Stack which illustrates the architecture of the Semantic Web as shown in Figure 2.1.

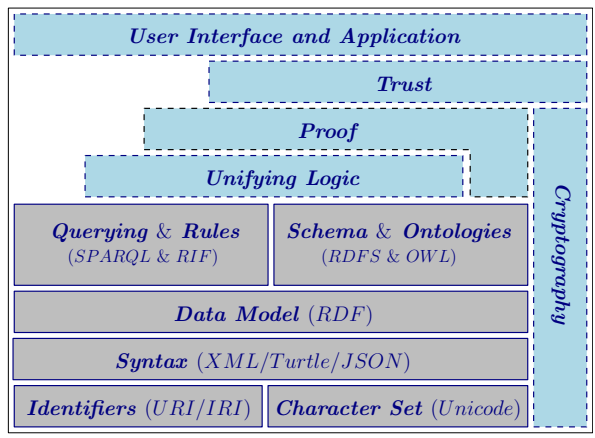


Figure 2.1: Semantic Web Stack

Each layer of this stack represents a technical component needed to realize the vision of the Semantic Web. However, while the components indicated at the bottom layers of the stack (in gray color) have been standardized, the upper-layer parts necessarily need novel technique to be fully implemented. Following is a summarization of the standardized components in the Semantic Web Stack.

- **Character Set (Unicode)**: Like the current Web, the Semantic Web relies on the well known Unicode character set as a standardized form for encoding, representing, and manipulating text on the Web.

- **Identifiers (URI/IRI)**: Semantic Web uses globally agreed-upon identifiers in form of URI/IRI specification for describing thing on the Web. Uniform Resource Identifiers (URI), which are already used on the World Wide Web, is a string of characters used to uniquely identify a resource (e.g., web document). URIs have been extended to IRIs (Internationalized Resource Identifier) so as to support the usage of Unicode characters in the identifier. In the rest of our thesis, whenever URI is used, IRI can be used as well as a more general concept.

- **Syntax (XML/Turtle/JSON)**: Formally defined syntaxes are required in Semantic Web in order to encode the Semantic Web data in machine readable form. For that, existing syntaxes which have been dominant in all the web contents such as the Extensible Markup Language (XML) and JavaScript Object Notation (JSON) can be used. Together with these generic syntaxes, the Semantic Web also introduced novel syntaxes which have terse grammars and are generally recognized as being more readable than XML such as Terse RDF Triple Language (Turtle) and its subset N-triples.

- **Data Model (RDF)**: Semantic Web requires an agreed-upon common data model in order to share and exchange data across different Semantic Web applications. This model necessarily needs to be generic and simple enough so that it can express any data and fact in different knowledge domains, and yet structured enough for a machine to understand it. Therefore, the Resource Description Framework (RDF) with the innate flexibility of a schema-less data model and the simplicity of its "triple-based" representation, is selected as the core data model for representing Semantic Web data. This data model can be serialized by any syntaxes described in the lower layer of the stack.

- **Schemas and Ontololies (RDFS and OWL)**: In order to bring semantics and meaning to the Semantic Web content, formal languages that define schemata and ontologies using semantically well-defined vocabularies were created. In the Semantic Web standards, these languages include RDF Schema (RDFS) and Web Ontology Language (OWL), in which RDFS is created within RDF to describe taxonomies of classes and properties and use them to create lightweight ontologies, and OWL describes more detailed ontologies by offering more constructs over RDFS.

- **Querying and Rules (SPARQL and RIF)**: SPARQL Protocol and RDF Query Language (SPARQL) is the standardized query language for retrieving and manipulating RDF data. As ontologies and knowledge bases defined with RDFS and OWL languages can be used for reasoning, along with SPARQL, rule-based languages that provide rules beyond the constructs available from these languages, are being standardized in Semantic Web in order to infer novel data from existing content. The Semantic Web standard rule language is Rule Interchange Format (RIF).

A more detailed description of the Semantic Web Stack as well as its variants can be found in the study of Gerber et al.[97].

2.1.1 Resource Description Framework

The Resource Description Framework (RDF) is the W3C recommendation model for representing information about resources in the Web data [131, 74]. Using the RDF data model, each Web resource, which is identified by an Uniform Resource Identifier (URI) or an Internationalized Resource Identifier (IRI), can be simply described in term of its properties and property values. For example, a person identified by the URI `<http://dbpedia.org/resource/MT>`, whose name is Mark Twain, whose birth place is Floria, whose birth date is "1835-11-30", and who is the author of the book `<http://dbpedia.org/resource/The_Adventures_of_Tom_Sawyer>` can be described in four RDF statements as follows.

```
<http://dbpedia.org/resource/MT> <name> ''Mark Twain''

<http://dbpedia.org/resource/MT> <birth_place> ''Floria''@en

<http://dbpedia.org/resource/MT> <birth_date> ''1835-11-30''
^^xsd:date

<http://dbpedia.org/resource/MT> <author_of> <http://dbpedia.
org/resource/The_Adventures_of_Tom_Sawyer>
```

Each RDF statement is thus basically a triple of <**s**ubject, **p**roperty, **o**bject> (or <**s**ubject, **p**redicate, **o**bject>), in which the subject is the identifier of a resource, the object is the value for the property (i.e., predicate) of the described resource.

2.1.1.1 Basic RDF Terms

Each element (i.e., subject, property or object) of an RDF statement belongs to one of the three disjoint sets of RDF terms: URIs (or IRIs), Literals, and blank nodes.

URIs. URIs (and their internationalized version IRIs) are used as global identifiers in the RDF data model for identifying any Web resource. The generic syntax of URI is formally defined in the Internet Standard 66 [133]. A particular kind of URI is the Uniform Resource Locator (URL) which is typically used to identify a Web page. For example, the URL `http://dbpedia.org/resource/Mark_Twain` is an URI to identify the resource about the writer "Mark Twain" on DBpedia.

In the RDF serialization syntax such as Turtle, URIs are written enclosed in angle brackets $<$ and $>$ (e.g.,`<http://dbpedia.org/resource/Mark_Twain>`). In order to avoid having repeatedly long prefix strings, URIs may also be abbreviated by using Turtle's `@prefix` directive and the CURIE Syntax for expressing compact URIs [51]. Specifically, if there is a URI prefix defined for the above example URI such as `@prefix dbp: <http://dbpedia.org/resource/>`, the original example URI then can be abbreviated as `dbp:Mark_Twain`.

Literals. Literals are a set of lexical values such as numbers, dates, and strings. Anything represented by a literal may also be represented by a URI, but using literals is often more convenient or intuitive. Literals can either be plain or typed. A "plain literal" is a string with an optionally addition language tag. In the above RDF statements, "Mark Twain" and "Floria"@en are plain literals. A "typed literal" is a string combined with a datatype URI such as "1835-11-30"^^`xsd:date`. For the datatype URIs (e.g., `xsd:date`), the RDF uses many simple types from XML Schema [52] such as numerics, date time, booleans.

Blank Nodes. A blank node (or *bnode*) is an indication used for representing anonymous resource (i.e., a resource for which an explicit URI or literal is not given). It can be denoted through blank node identifiers using RDF serialization format such as RDF/XML, Turtle, N3 and N-Triples. Following is an example of blank node in RDF statements using N-Triples format.

```
<http://dbpedia.org/resource/MT> <author_of> _b:Node01 .

_b:Node01 <rdfs:label> ''The Adventures of Tom Sawyer'' .

_b:Node01 <rdf:type> dbo:Book
```

A blank node is only limited in its local scope (e.g., in an particular RDF document), and thus, cannot be referenced from the outside of its originating scope.

Hereafter, we will use following formal notations for the three disjoin subsets of RDF terms:

- **U**: Set of all URIs
- **B**: Set of all blank nodes
- **L**: Set of all literals

Definition 1 *The set of RDF terms* $RDF_t = \mathbf{U} \cup \mathbf{L} \cup \mathbf{B}$

2.1.1.2 RDF triple

In an RDF triple, the subject can either be a URI or a blank node, while the predicate must be a URI. The object in the RDF triple can be anything, either a URI or a blank node or a literal. An RDF triple can be formally defined as following.

Definition 2 *An RDF triple is defined as a triple* $t = (s, p, o)$ *in which* $s \in \mathbf{U} \cup \mathbf{B}$*,* $p \in \mathbf{U}$*, and* $o \in \mathbf{U} \cup \mathbf{L} \cup \mathbf{B}$*.*

2.1.1.3 RDF graph

A finite set of RDF <s,p,o> triples forms a labeled directed RDF graph, in which the subjects and the objects are the nodes of the graph and the predicates are the edges connecting these nodes.

Definition 3 *An RDF graph is a finite set of RDF triples* $G \subset (\mathbf{U} \cup \mathbf{B}) \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{L} \cup \mathbf{B})$*.*

Following is an example RDF dataset in the Turtle syntax and its graph representation.

```
#Prefix declarations
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix dbp: <http://dbpedia.org/property/> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

#RDF triples
dbr:Mark_Twain foaf:name ''Mark Twain'' .
dbr:Mark_Twain dbp:birthPlace ''Floria''@en .
dbr:Mark_Twain dbp:birthDate ''1835-11-30''^^xsd:date .
dbr:Mark_Twain dbp:authorOf _b:Node001 .
_b:Node001 rdfs:label ''The Adventures of Tom Sawyer'' .
_b:Node001 rdf:type dbo:Book .
```

Figure 2.2: RDF triples

2.1.1.4 RDF dataset and named graphs

An RDF store may hold multiple RDF graphs and “name" each graph so as to allow an application to query either from the whole RDF dataset or from specific RDF graphs. Each “named graph" is identified by an IRI and formally defined as following.

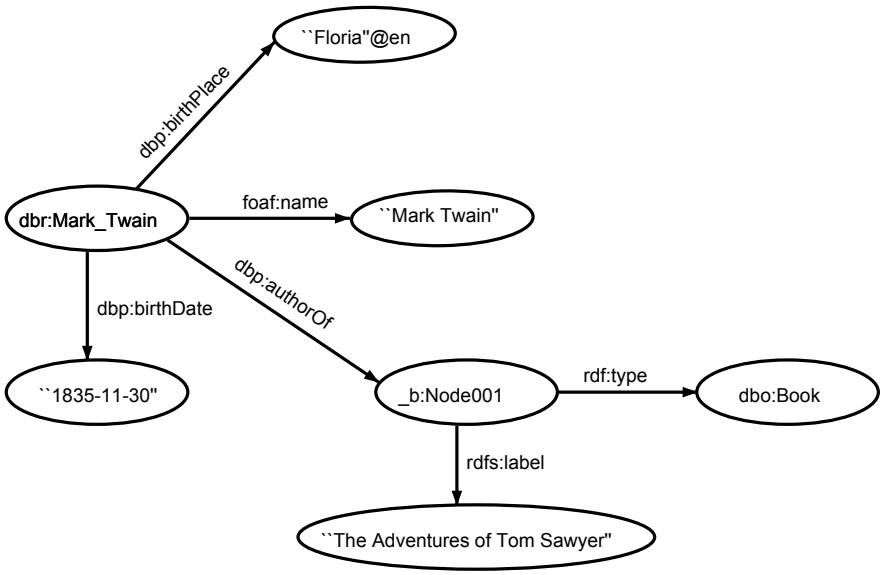


Figure 2.3: RDF graph

Definition 4 *A named graph is a pair of (u, G) where* $u \in \mathbf{U}$ *is the name of the graph and G is an RDF graph.*

An RDF dataset is then composed of one graph, the *default graph*, which does not have a name, and zero or more named graphs.

Definition 5 *An RDF dataset D is a set {G, (*u_1*,* G_1*), (*u_2*,* G_2*),...,(*u_n*,* G_n*)} where G is the default graph,* u_1*,* u_2*, ...,* u_n *are distinct URIs, and each pair (*u_i*,* G_i*) is a named graph.*

2.1.2 RDF Semantics and Web Ontology Language (OWL)

In addition to the core RDF data model described in the previous section, in this section, we will cover the semantics aspects of RDF and outline related standards that are used for extending RDF with richer semantics such as RDFS and OWL.

2.1.2.1 RDF schema (RDFS)

RDF's vocabulary description language, RDF Schema, is a semantic extension of RDF that provides the mechanism to describe groups of related resources and the relationships between these resources. In early 2004, the RDFS specification became W3C Recommendation data-modeling vocabulary for the RDF data [65].

RDF schema language is written in RDF using a set of "built-in" vocabulary terms provided by RDF standard. These terms are identified under the core RDF namespace, `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, or the

core RDF schema namespace `http://www.w3.org/2000/01/rdf-schema#`. Conventionally, the prefixes `rdf:` and `rdfs:` are respectively used for associating with these namespaces. In the following, we will discuss the most prevalent terms in the core RDF vocabularies.

`rdf:type`. The most frequently-used term in the core RDF vocabulary is `rdf:type` which is used for assigning resources of certain commonalities to classes. In Turtle syntax, the abbreviation "a" is allowed to be used for the `rdf:type` property. Note that a resource can be assigned to multiple classes, e.g., as shown in the Figure 2.4, `ex:MarkTwain` is an instance of classes `dbo:Person` and `dbo:Writer`. In this example, `rdf:Property`, another term in the core RDF vocabulary, is also used together with `rdf:type` in order to specify that both `dbo:birthPlace` and `dbo:birthPlace` belong to the class of properties.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix dbo: <http://dbpedia.org/ontology/> .

ex:MarkTwain   rdf:type dbo:Person .
ex:MarkTwain   rdf:type dbo:Writer .
dbo:birthPlace a rdf:Property .
dbo:birthName  a rdf:Property .
```

Figure 2.4: Example of using rdf:type

Vocabularies for *RDF lists*. RDF standardizes an agreed-upon vocabulary for describing collections, i.e. "list structure" using a linked-list pattern. As the set-based RDF triples do not have any inherent ordering, this vocabulary provides "ordering" semantics for the RDF data model. The basic terms of this vocabulary are `rdf:first`, `rdf:rest`, and `rdf:nil`, in which `rdf:first` indicates the first element in the (sub-)list, `rdf:rest` connects to the subsequent (sub-)list, and `rdf:nil` indicates an empty list and is usually used to close the list. Figure 2.5 shows an example of a closed RDF list which contains ordered elements {*ex:Elem1*, *ex:Elem2*, *ex:Elem3*}.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

_:list1        rdf:first ex:Elem1 .
_:list1 rdf:rest  _:list2 .
_:list2 rdf:first ex:Elem2 .
_:list2 rdf:rest  _:list3 .
_:list3 rdf:first ex:Elem3 .
_:list3 rdf:rest rdf:nil .
```

Figure 2.5: Example of RDF list

Vocabularies for the classes and properties' relationships. In order to specify

well-defined relationships between classes and properties, RDFS extends the original core RDF vocabulary with the key terms: `rdfs:Class`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`.

- `rdfs:Class` is a class of resources that are RDF classes. This term is recursive defined as `rdfs:Class` is also an instance of `rdfs:Class` itself.
- `rdfs:subClassOf` indicates that all the instances of one class are instances of another. For example, {`foaf:Person rdfs:subClassOf foaf:Agent`} states that every person is an agent. In particular, a triple {`c1 rdfs:subClassOf c2` } indicates that `c1` is an instance of `rdfs:Class`, `c2` is an instance of `rdfs:Class`, and `c1` is a subclass of `c2`. This term allows to declare the hierarchies of classes among RDF classes.
- `rdfs:subPropertyOf` indicates that all resources related by one property are also related by another. For example, given the statement {`ex:mother rdfs:subPropertyOf ex:parent`}, any RDF statement with property `ex:mother` (e.g., {`Tom ex:mother Maria`}) also infers another statement with property `ex:parent` (e.g., {`Tom ex:parent Maria`}).
- `rdfs:domain` and `rdfs:range` are used to indicate the domain and the range of a property, respectively. In particular, `rdfs:domain` states that the resource which has a given property is an instance of one or more classes, and `rdfs:range` states that the values of a property are instances of a class. Specifically, a triple {`p1 rdfs:domain c1`} indicates that all resources that has property `p1` belong to the class `c1`. A triple {`p1 rdfs:range c2`} indicates that the values of the property `p1` (e.g., the object in a triple with property `p1`) are instances of class `c2`. If there are multiple classes for the `rdfs:domain` and `rdfs:range` of a property, then the intersection of these classes will be used. Figure 2.6 shows an example of using `rdfs:domain` and `rdfs:range`. As specified in this example, only instances of class `ex:UsedProduct` have property `ex:price`, and the value for this property is an integer number (e.g., `xsd:int`).

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>. .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

ex:price rdfs:domain ex:UsedProduct .
ex:price rdfs:range xsd:int .
```

Figure 2.6: Example of rdfs:domain and rdfs:range

2.1.2.2 Web Ontology Language (OWL)

The Web Ontology Language (OWL) is a Semantic Web language for representing rich and complex knowledge about things and the relationships between them. It

evolved from the earlier proposal for Web ontology language DAML+OIL [78], and soon became a W3C Recommendation language in 2004 [140]. The first version of OWL was subsequently revisited and extended to the second OWL (i.e., OWL 2) W3C Recommendation language in 2009 [79].

Like RDF Schema (RDFS), OWL provides a vocabulary for representing semantics in RDF data, however with more facilities for expressing meaning and semantics, OWL goes far beyond the basic semantics of RDFS. Specifically, while RDFS vocabulary merely describes generalization-hierarchies of properties and classes, OWL adds a wealth of new vocabularies allowing to specify more complex relationships among classes and properties including: "among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes". Some novel and frequently used terms from OWL are briefly described in the following.

- `owl:equivalentClass` is an built-in OWL property to specify that two classes have the same class extension. For example, classes `US_Presidents` and `PrincipalResidentOfWhiteHouse` are stated to have the same members.
- `owl:disjointWith` is an built-in OWL property to specify that two classes have no individuals in common or the intersection of their extensions is empty. For example, classes `Person` and `Tree` have no common member.
- `owl:equivalentProperty` is used to states that two properties have the same property extension (e.g., properties `hasParent` and `childOf`).
- `owl:disjointPropertyWith` is used to states that two properties cannot be used for relating two things (e.g., properties `isMotherOf` and `isFatherOf` cannot be used for describing the relationship of the same two people in a same direction).
- `owl:inverseOf` to specifies the inverse relation between properties. For example, properties `hasChild` and `hasParent` having `owl:inverseOf` relationship means that if {P1 `hasChild` P2} then {P2 `hasParent` P1}.
- `owl:sameAs` is used to state that two resources (e.g., identified by two URIs) actually refer to the same thing. This property is often used for defining the mapping between different ontologies. For example, { `dbr:Citrus` `owl:sameAs` `ex:Citrus` }.
- `owl:differentFrom` is used to state that two resources (e.g., two URI references) refer to different individuals (e.g., { `dbr:A_Dogs_Tale` `owl:differentFrom` `dbr:Eves_Diary` }).

OWL has three increasingly-expressive sub-languages designed toward different user communities, namely OWL Lite, OWL DL, and OWL Full. These sublanguages were updated and extended in OWL 2, resulting in an OWL 2 sublanguage (OWL 2 DL) and OWL 2 profiles OWL 2 EL, OWL 2 QL, and OWL

2 RL. Each of the three sub-languages (i.e., OWL Lite, OWL DL, OWL Full) is a syntactic extension of its predecessor. In particular, every legal OWL Lite ontology is a legal OWL DL ontology, and every legal OWL DL ontology is a legal OWL Full ontology. The following are the short descriptions on these three sub-languages of OWL:

- OWL Full is designed to provide maximum expressiveness and the syntactic freedom of RDF with no computational guarantees. It uses all the OWL language primitives and allows the combination of these primitives in arbitrary ways with RDF and RDF Schema. OWL Full is fully upward-compatible with RDF, both syntactically and semantically, and can be viewed as an extension of RDF. However, as it is so powerful, OWL Full is undecidable [79] and it is unlikely that a reasoning software can efficiently perform complete reasoning for it.
- OWL DL is a sub-language of OWL Full that provides maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time) [79]. It restricts application of the constructors from OWL and RDF. OWL DL permits efficient reasoning support, however, it loses the full compatibility with RDF. Accordingly, not every RDF document is a legal OWL DL document while every legal OWL DL document is a legal RDF document.
- OWL Lite uses further restriction to limit OWL DL to a subset of the language constructors (e.g., OWL Lite excludes enumerated classes, disjointness statements, and arbitrary cardinality. It supports cardinality constraints, but only permits cardinality values of 0 or 1). OWL Lite targets at users who primarily need classification hierarchy and simple constraints.

Further detail on OWL can be found in the reference document of OWL [140] and OWL 2 [79].

2.1.3 SPARQL - RDF query language

SPARQL (aka SPARQL protocol and RDF query language) is a semantic query language for retrieving and manipulating data from RDF stores. The original SPARQL specification (i.e., SPARQL 1.0) and its extension (i.e., SPARQL 1.1) became official W3C Recommendation in 2008 [110] and in 2013 [106], respectively.

2.1.3.1 SPARQL syntax

SPARQL is directly built on top of the RDF data model, and its syntax is closely tied with RDF-specific syntax such as Turtle. On a high level, a SPARQL query can be decomposed into five main basic parts: *Prefix Declarations*, *Dataset Clause*, *Result Clause*, *Query Clause*, *Solution Modifiers*.
Prefix Declarations defines URI prefixes (similar to Turtle's `@prefix` directive) in order to use shortcuts instead of repeatedly long URIs in the query.

Dataset Clause specifies the particular part of the dataset over which the query will be executed.
Result Clause specifies the SPARQL query type (i.e., SELECT, ASK, CONSTRUCT, or DESCRIBE) so as to indicate what results should be returned by the query. In our research, we only focus on the SELECT query type which extracts matched (RDF) graph patterns specified by the input query from a SPARQL endpoint, and returns the list of bindings for the variables in the SPARQL query as the result in a table format.
Query Clause consists of the query patterns (i.e., SPARQL triple patterns), conjunctions, disjunctions, and optional patterns that will be used for generating variable bindings from RDF data. More details on SPARQL query clause will be described in Section 2.1.3.2.
Solution Modifiers allow to modify the result by applying standard classical operators such as ORDER BY (sorts the result set), LIMIT (sets the maximum number of results returned), DISTINCT (removes all duplications in the result set), REDUCED (allows to eliminate some duplicate results from the result set), OFFSET (specifies the position in the overall sequence of results from which the results will be returned), PROJECT (chooses certain variables to return in the results).

The following is an example SPARQL query and its main basic parts. In this query, lines with prefix '#' are comments, the shortcuts of URIs' prefixes are defined by using `@prefix` directive at the beginning of the query. This query asks for the names of authors and the books which they wrote (i.e., specified by the Result Clause and the Query Clause) from an RDF document *"book_author.xml"* (i.e., specified by the Dataset Clause). The number of returned results is limited to 2 (i.e., specified by the Solution Modifier "LIMIT 2").

If the matching patterns for the Query Clause are found from the RDF document "book_author.xml", two matching patterns will be returned as the result of the query like the following.

name	book
"Mark Twain"	http://dbpedia.org/resource/Adventures_of_Huckleberry_Finn
"Mark Twain"	http://dbpedia.org/resource/The_Adventures_of_Tom_Sawyer

Figure 2.8 shows the basic grammar of a SPARQL query. The full SPARQL grammar can be found from the official W3C recommendation of SPARQL query language [1].

2.1.3.2 SPARQL query clause

In SPARQL query, the query clause is (almost always) indicated by the WHERE keyword and surrounded by the opening and closing braces ({ }). It thus can be simply considered as the WHERE clause of SPARQL query.

The typical forms of the query clause contain one or more set of triple patterns. Each conjunctive set of triple patterns is called a basic graph pattern (BGP). Like the RDF triple, a triple pattern (*tp*) contains three elements subject, predicate, and

[1] SPARQL grammar: https://www.w3.org/TR/rdf-sparql-query/#grammar

```
#Prefix Declarations
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>

# Dataset Clause
FROM <http://dbpedia.org/data/book_author.xml>

# Result Clause
SELECT ?name ?book

# Query Clause
WHERE {
    ?person foaf:name ?name .
    ?person dbp:birthPlace ``Floria''@en .
    ?person dbp:authorOf ?book .
    ?book rdf:type dbo:Book
}

# Solution Modifiers
LIMIT 2
```

Figure 2.7: Example SPARQL query

Query	:=	[Header*] Form [Dataset] WHERE Pattern Modifiers
Header	:=	PREFIX pname_rs iri_ref | BASE iri_ref
Form	:=	SELECT [DISTINCT|REDUCED] ('*'|var*) | ASK | CONSTRUCT var* | DESCRIBE
Dataset	:=	FROM graph_clause | FROM NAMED graph_clause
Modifiers	:=	ORDER BY [ASK|DESC] var* | LIMIT value | OFFSET value
Pattern	:=	Pattern . Pattern | {Pattern} UNION {Pattern} | Pattern OPTIONAL {Pattern} | Triple_Pattern | FILTER Constraint
var	:=	('?'|'$')value

(value, pname_rs, iri_ref, graph_clause $\in$ String)

Figure 2.8: Basic SPARQL grammar

object. However, in a triple pattern, each of these element can be a variable. In the example SPARQL query shown in Figure 2.7, the query clause contains one basic graph pattern of four triple patterns. In the triple pattern such as (`?person`, `dbp:authorOf`, `?book`), the subject and the object are variables.

The triple pattern and basic graph pattern are formally defined as the following.

Definition 6 *A triple pattern is defined as tp = (s,p,o) in which* $s \in \mathbf{U} \cup \mathbf{B} \cup \mathbf{V}$*, p*

$\in \mathbf{U} \cup \mathbf{V}$, *and* $o \in \mathbf{U} \cup \mathbf{L} \cup \mathbf{B} \cup \mathbf{V}$ *where* $\mathbf{U}$, $\mathbf{B}$, $\mathbf{L}$, $\mathbf{V}$ *are the sets of URIs, blank nodes, literals, and variables respectively.*

Definition 7 *A basic graph pattern is a set of conjunctive triple patterns: bgp = {tp} where tp is a triple pattern.*

Generally, in SPARQL query, a basic graph pattern is identified by a conjunctive set of triple patterns surrounded by braces { }. From basic graph patterns, more complex graph patterns can be formed in the SPARQL query clause in various ways by either conjunctively grouping BGP's (i.e., group graph pattern) or by using these four SPARQL keywords: GRAPH, UNION, OPTIONAL, FILTER.
Group Graph Pattern: is a set of graph patterns delimited with { }. When a group graph pattern consists only of triple patterns or only of BGP's, the group graph pattern is equivalent to the corresponding set of triple patterns. Figures 2.9 and 2.10 show examples of group graph patterns which contain one and two basic graph patterns, respectively. These two group graph patterns are equivalent to the same set of triple patterns and thus will return the same matchings from RDF dataset. We note that the {} is the empty group graph pattern.

```
SELECT ?person
WHERE {  ?person foaf:name ``Mark Twain'' .
         ?person dbp:birthPlace ``Texas'' }
```

Figure 2.9: Query clause with one basic graph pattern

```
SELECT ?person
WHERE {  { ?person foaf:name ``Mark Twain'' } .
         { ?person dbp:birthPlace ``Texas'' } }
```

Figure 2.10: Query clause with two basic graph patterns

GRAPH: specifies the named graph (identified by a URI or the binding values of a variable) against which a basic graph pattern should be matched. Figure 2.11 shows an example of using GRAPH keyword in the query clause. In this example, the GRAPH keyword specifies that the query can only access the named graph `<http://example.org/foaf/bob>` in order to retrieve the matchings for the basic graph pattern of its query clause.
UNION: allows the matching on one of several alternative graph patterns. The result of the query is the union of all the matchings for each of the alternative graph pattern. Given the RDF dataset and the SPARQL query in Figure 2.12, using UNION keyword, the query will return the result as the combination of matchings for each of the graph patterns *{?book dc10:creator ?person}* and *{?book dc11:creator ?person}*. Therefore, the result for this query will be {"Alice", "Bob"}.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX data: <http://example.org/foaf/>

SELECT ?name
FROM NAMED <http://example.org/foaf/alice>
FROM NAMED <http://example.org/foaf/bob>
WHERE {
        GRAPH data:bob {
                ?x foaf:mbox <mailto:b@work.example> .
                ?x foaf:name ?name }
}
```

Figure 2.11: Example of using GRAPH in query pattern

OPTIONAL: allows the use of optional patterns in the entire query pattern so that the query result will be extended with the bindings from an optional pattern if matching of the pattern exists in the data, and ignores the variable binding if nothing is matched. From the view of the SQL world, the OPTIONAL operator is equivalent to the Left Outer Join where the results always include the left part of the query (e.g., non-optional part), even if there exists no match for the right part of the query (e.g., optional part). Figure 2.13 shows an example dataset, a SPARQL query using OPTIONAL keyword, and the query results. As we can see from the example, even though there is no matching for the pattern *{?person foaf:mbox ?mail}* with a person named *"Bob"*, the result without the binding of *?mail* variable is still returned for *"Bob"*.

FILTER: specifies further constraints and conditions that the query solutions should match. Given the RDF dataset as in Figure 2.13, Figure 2.14 shows an example SPARQL query using FILTER keyword and the query results. In this example, the FILTER constraint is set for the *?mail* variable so that the returned results must contain the "@cwi.nl" in the bindings of this variable. Therefore, only one matching is returned as the query result.

Concluding, a query pattern can be formally and recursively formed based on these following rules.

- Any basic graph pattern is a query pattern.
- If p, q are query patterns then { p . q } (conjunction), {p UNION q} (union), or {p OPTIONAL q} (alternative) is also a query pattern.
- If p is a query pattern, x is a URI or a variable then {p GRAPH x} is also a query pattern.
- If p is a query pattern, c is a filter condition then {p FILTER c} is also a query pattern.

```
# Example dataset
@prefix dc10:  <http://purl.org/dc/elements/1.0/> .
@prefix dc11:  <http://purl.org/dc/elements/1.1/> .
@prefix foaf:  <http://xmlns.com/foaf/0.1/>

_:a  dc10:label     "Book 1" .
_:a  dc10:creator   "Alice" .
_:b  dc11:label     "Book 2" .
_:b  dc11:creator   "Bob" .
_:b  foaf:title     "Book title 2" .
```

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX dc10:  <http://purl.org/dc/elements/1.1/>

SELECT ?person
WHERE {
        { ?book dc10:creator ?person }
        UNION
        { ?book dc11:creator ?person }
}
```

person
"Alice"
"Bob"

Figure 2.12: Example of using UNION in query clause and the result

2.1.3.3 SPARQL query graph

Similar to the RDF dataset and RDF graph, SPARQL query also forms a directed graph (i.e. *SPARQL query graph*) where nodes are formed by the subjects and objects of the query's triple patterns and edges are the properties of these patterns. We note that, different from the nodes and edges in an RDF graph, a node or an edge of a SPARQL graph can also be a variable. Figure 2.15 shows the SPARQL graph representation for the example query in the Figure 2.7.

Based on the shape of its graph representation, the SPARQL query can further be classified into such shape-specific categories as SPARQL star query or path query as shown in the examples in Figures 2.16 and 2.17. The star shape graphs actually appear frequently in the basic graph patterns of real SPARQL queries [42, 151] and will be discussed more details in the next chapters of this thesis.

2.1.3.4 SPARQL 1.1

SPARQL 1.1 extends the W3C 2008 Recommendation for SPARQL 1.0 by adding new features to the query language such as aggregates, sub-queries, negation, complex filtering, property paths, and an expanded set of more than 70 new keywords, built-in functions and operators. It helps fixing many shortcomings and limitations

```
# Example dataset
@prefix dc10:  <http://purl.org/dc/elements/1.0/> .
@prefix foaf:  <http://xmlns.com/foaf/0.1/>

_:a  foaf:name        "Alice" .
_:a  foaf:mbox        "alice@cwi.nl" .
_:a  foaf:mbox        "alice@example.com" .
_:a  dc10:description "Data scientist" .
_:b  foaf:name        "Bob" .
_:b  dc10:description "A novel writer" .
```

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>

SELECT ?name ?mail
WHERE {
        ?person foaf:name ?name .
        OPTIONAL { ?person foaf:mbox ?mail }
}
```

name	mail
“Alice”	“alice@cwi.nl”
“Alice”	“alice@example.com”
“Bob”	

Figure 2.13: Example of using OPTIONAL in query clause

of the SPARQL 1.0 (e.g., no aggregates, no subqueries, limited graph operations), and brings the feature set of SPARQL closer to other classical query languages. Furthermore, while no update operator was introduced in SPARQL 1.0, SPARQL 1.1 Update has been created with the intension to be the standard language for executing updates to RDF graphs. It allows to perform various update operations to an RDF store such as insert/delete triples into/from an RDF graph, load/drop RDF graph into/from the graph store. Beyond the query language, SPARQL 1.1 also adds other features that were widely requested, including service description, a JSON results format, and support for entailment reasoning.

Even though, SPARQL 1.1 has been introduced for quite many years, there are still many ongoing efforts in fully supporting SPARQL 1.1 from RDF database vendors. Several RDF/Graph database systems have claimed to support SPARQL 1.1 and SPARQL 1.1 update such as Oracle Spatial and Graph [24], StarDog [29], MarkLogic [18], GraphDB (or formerly, BigOwlim [8]), AllegroGraph [32], Jena TDB [15].

```
PREFIX dc10:  <http://purl.org/dc/elements/1.0/>
PREFIX foaf:  <http://xmlns.com/foaf/0.1/>

SELECT ?name ?mail
WHERE {
?person foaf:name ?name .
?person foaf:mbox ?mail .
FILTER regex(str(?mail), "@cwi.nl")
}
```

name	mail
"Alice"	"alice@cwi.nl"

Figure 2.14: Example of using FILTER in query clause

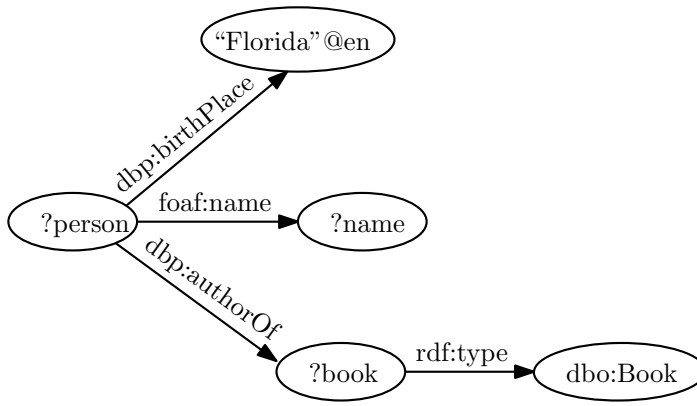


Figure 2.15: SPARQL query graph

2.2 RDF storage

In this section, we discuss different approaches for storing data in RDF systems (i.e., RDF stores). Generally, existing studies categorize RDF stores into two different approaches: *Non-native RDF stores* and *native RDF stores* [92, 145], in which non-native RDF stores are the storage solutions that make use of existing database systems (typically the Relational database systems) for storing RDF data, while native RDF stores are not based on existing database systems but implement their own storages (mostly focusing on indexing techniques) specific to the RDF model. However, we argue that this classification of RDF storage approaches does not convey exactly the implementation of RDF systems. This is because the literature and principles that have mostly come out of relational database research (in development and experiments) were also applied to the Semantic Web technology as well as to the RDF/SPARQL systems. Thus, we argue there is no big dividing wall between non-native RDF systems (or SQL-based systems) and native RDF systems. An ex-

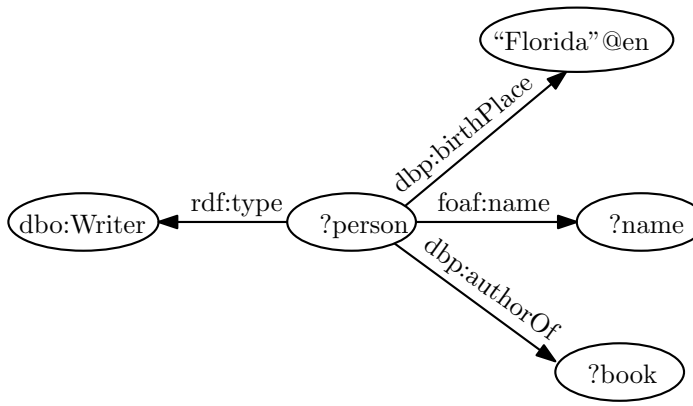


Figure 2.16: SPARQL star query

Figure 2.17: SPARQL path query

ample RDF store that can illustrate our argument is Virtuoso which is, by certain criteria, one of the best implementations of RDF/SPARQL [55, 147, 60]. Virtuoso is originally SQL-based and the so-called Virtuoso native RDF store [92] is indeed implemented and stored entirely within Virtuoso's SQL database system [91, 87]. In other chapters of this thesis we show that understanding the structure of RDF data is actually the crucial point that effects RDF data storage as well as SPARQL query optimization. Thus, in our review of current literature on RDF systems, we divide these into two classes: structure-aware RDF storage and non-structure-aware RDF storage.

In the following sections, we will review these two classes of RDF storages on a large number of centralized-based RDF systems, and further discuss storage approaches of distributed RDF data management systems.

2.2.1 Non-structure aware RDF storage

Non-structure aware RDF storages include all RDF stores that do not exploit any structure information on the input RDF data in order to design the storage solutions. This respects the inherent schema-less nature of the RDF data model. The representative RDF storage solutions of this approach are *triple table* [183, 151, 108, 45] and *vertical partitioning* [34]. We note that, as RDF triples can be viewed as graph data (i.e., RDF graph), a growing number of pure graph database systems, such as Apache Titan [6], Neo4j [22], Sparksee [28], OrientDB [25], InfiniteGraph [14], to name a few, can also be used for storing and processing RDF data without understanding the structure of the data. These systems reflect the object-oriented view of graphs, and elevate graphs to first class citizens in their data model ("property

graphs"), query languages, and APIs. In this thesis, we focus on systems which are designed to manage semantic web data conforming to the RDF data model.

2.2.1.1 Triple tables

As a RDF dataset is a collection of (s,p,o) triples, using a single large table of three columns (subject, property, object) – *triple table* – is the most straight forward approach to store RDF data. Basically, each column in the triple table corresponds to an element (subject, property, or object). We note that the literal values and the IRIs are typically not stored directly as strings in the triple table, but instead, stored as numeric object identifiers (OIDs) uniquely associated with these values. This allows the use of fixed-length records in the triple-table as well as significantly reduces the storage space as the RDF data commonly has many frequently repeated IRIs and literal values. The OID's are typically generated using a hash-based approach (i.e., using a hash-function) [103, 102, 167] or a counter-based approach (i.e., maintaining a counter and increasing the counter for each new resource) [68, 107, 108, 150, 151]. A dictionary (in form of a table or a certain data structure such as BTree+) is usually created to maintain the mapping between an OID and its corresponding IRI or literal value. Further optimizations can also be applied for improving the storage space of the dictionary such as separately storing and encoding the common namespace prefix of many IRIs [130, 90]. More discussions on the variations of OID dictionary implementation can be found at [132]. We also note that for storing the "triple table" (or "quad table" in case the named graph is taken into account in addition to the existing $\{s, p, o\}$ triples), an RDF store may use any data structure (e.g., B-tree, Hash map, Bitmap, etc). Early RDF systems which follow the triple-table approach are Redland [48], 3store [102], Oracle [76], and RDFStore [164]

To improve the performance of the query processing (e.g., faster look-up, less expensive joins), indexes are additionally added for each of the columns of the triple table. A typical indexing approach is to store the RDF triples in a collation order of (s, p, o). For example, a "SPO table" is a triple table which is first ordered by the subject (S), then for each S, sub-ordered by the property (P), and finally ordered by the object (O) for the same (S,P). To efficiently answer queries on different SPARQL patterns, many RDF systems store the input RDF triples in sorted orders on all the various permutation of (s, p, o). This is an aggressive indexing technique as it is a maximal approach and confronts the RDF store with the generic downsides of unclustered indexes: namely, increased storage space (a factor 6 if all triple orders are indexed or even 24 for all quad orders), as well as additional maintenance cost when the RDF data is updated. Examples include Mulgara [187], HexaStore [183], RDF-3X [151], YARS [108], HPRD [45], Virtuoso [90], BitMat [43], TripleT [93], BRAHMS [117], RDFJoin [138], RDFKB [139], iStore [175], Parliament [121], Owlim [120], and BlazeGraph [9].

In particular, **Mulgara** (formerly, Kowari), a native transactional RDF store, builds a so-called "Perfect Indexing" based on six different order of "quads" $\{s, p, o, m\}$ (where m describes which model, or more correctly graph, the RDF statements appear in) to store RDF data: $spom$, $posm$, $ospm$, $mspo$, $mpos$, $mosp$.

Each of these six indexes is implemented as a multi-version blocked AVLTree. If the meta data is ignored for simplicity, the required indexes in Mulgara are reduced to 3 different orderings of (s, p, o): spo, pos, osp.

Yars [107] adapts information retrieval and database techniques in order to build two sets of indexes called *lexicon* and *quad indexes* for efficient RDF storage and indexing. In particular, the former covers the string representations of RDF graph and includes an inverted index for fast full text searches. The latter is used to efficiently store all RDF quads of {s, p, o, c}, in which c (context) refers to various kind of application-specific metadata for a given set of RDF statements. The quad indexes is the combination of six B+Tree indexes building on different orders of {s, p, o, c} (i.e., $spoc$, poc, ocs, csp, cp, os), which can supposedly cover all 16 possible access patterns of the quad {s, p, o, c}. In a more recent version of the system, **YARS2** [108], instead of using B-Tree indexes, Harth et al. consider two alternative index structures: extensive hash-tables and an in-memory sparse index. Using hash tables can significantly decrease the disk I/Os as it enables search operation in constant time, however, this requires maintaining 16 hash tables for a complete index on quads. On the other hand, the in-memory sparse index which refers to an on-disk sorted and blocked data file can achieve constant lookup times similar to the hash-tables, however, there is a trade-off between the performance and the occupied memory size as larger blocks requires less sparse index entries, but more disk I/Os for reading, while smaller blocks requires more less sparse index entries, but less disk I/Os.

HexaStore [183] engine stores the RDF data in six collation orders spo, sop, pso, pos, ops, osp using clustered BTree indexes. This helps providing efficient single triple pattern lookup and fast merge-join of two triple patterns. However, even though HexaStore uses typical dictionary encoding to limit the storage space needed for the URIs and the literals, the space requirement of Hexastore is still five time larger than the space required for storing RDF triples in a single triple table.

In addition to the six indexes of {s, p, o} orders (like in HexaStore), Neumann et al. add so-called *projection indexes* for each strict subset of {s, p, o} to their **RDF-3X** engine [151]. Specially, they build clustered B+tree indexes with composite keys over 9 collation orders s, p, o, sp, so, ps, po, os, op. However, thanks to their delta compression scheme (e.g., storing only the difference between sorted triples), the size of indexes in RDF-3X does not exceed the dataset size.

Taking the context information (e.g., named graph) into consideration, **Virtuoso** [90], a widely used RDF system, builds a quad store on different permutations of {s, p, o, g} (where g is the named graph). Its current index scheme consists of two full indices over RDF quads (i.e., PSOG and POGS) and three partial indices (SP, OP, and GS). Being rooted in a row-wise RDBMS, Virtuoso has recently been turned into a compressed column store and its performance has been significantly improved by incorporating advanced relational database techniques (e.g., vectorized execution) from state-of-the-art column stores such as MonetDB [20] and Vectorwise [192]. This conversion of Virtuoso to column store was performed during the course of my PhD as one of our task for the LOD2 project [59]. We note that the use of row-wise or column-wise storage is effectively orthogonal to the discussion of structure-aware vs non-structure aware store. Even though Virtu-

oso still maintains both the row/column storage schemes, Virtuoso 7 (the currently latest version of Virtuoso), by default, uses column-wise storage for its indices because of several compelling advantages. First of all, column store leads to better compression. In particular, the space consumption for column-wise Virtuoso index structures is usually about one-third of the space consumed by the equivalent row-wise structures. Secondly, column store interfaces better with vectorized execution (i.e., a bulk processing of "cache-sized" block, typically 100-10000, of tuples between operators in a query execution pipeline [193]) which makes the query interpreter more CPU efficient. It finally also exposes opportunities for better parallel I/O in index access. Moreover, the performance difference between column store and row store can be viewed more clearly in considering typical database workloads, namely, OLTP (On-line Transaction Processing) workload which contains a large number of short on-line transactions and OLAP (On-line Analytical Processing) workload which contains complex queries. In the RDF world, OLTP workload typically contains RDF lookup workloads (e.g., simple web requests) which have low query complexity and are index intensive. For these workloads, column store has less locality than row store, but the workloads have small queries with no locality anyways. Thus, locality is not the dominant performance factor and hence the performance difference between these storage schemes is small (column store is a bit slower than row store). For the OLAP-like RDF workloads such as BSBM-BI [59] and RDF-ized TPC-H [89], the query evaluation is the dominant cost, and massive data volumes are involved. Consequently, the smaller footprint of compressed columnar storage and the more efficient vectorized execution clearly make columnar storage better than row storage. As real-world use is a mix of such workloads, Virtuoso 7 decided to move to column-storage by default. Last but not least, even though Virtuoso is one of the best non-structure aware RDF store, it is being on the way to becoming a structure-aware system as Orri has created structure aware prototype and showed further improvement on the query processing (see more details in Chapter 4).

BitMat [43] is a main-memory based bit-matrix structure that is used for making RDF triples representation compact and allowing efficient basic join query processing by employing logical bitwise AND/OR operators on the structure. In particular, as each RDF triple can be viewed as a 3-dimensional entity, Bitmap in essence is built as a 3-dimensional bit-cube, in which each cell is a bit (of value 1 or 0) representing a unique triple and denoting the presence or absence of that triple. Moreover, as a typical RDF dataset covers a very small set of the 3-dimensional space formed by distinct subjects $\times$ distict predicates $\times$ distinct objects, Bitmat tends to be very sparse. It exploits this sparsity in order to achieve compactness by applying the D-gap compression scheme [10] for compressing each bit-row, showing small memory footprint in comparing to the size of raw RDF data. However, doing compression on a sparse bit vector is getting to the delta compression on the columnar store. Thus, the BitMat representation of RDF triples is in fact not so different comparing to the columnar representation of e.g., SPO index, using delta compression at physical level.

Fletcher et al. propose a so-called *Three-way Triple Tree* (**TripleT**) [93] secondary memory index to improve the data locality of existing indexing techniques.

Specifically, they observe that existing multi-indexes approaches suffer from weak data locality since a piece of data can appear in multiple locations and in several different data structures. Thus, in contrast to these approaches, they build a single B-Tree index over all the "atoms" occurring in RDF graph regardless of the role of each atom in the RDF triples (i.e., an atom can be either a subject, a predicate, or an object). Each key atom in the index points to actual data stored in one of three different buckets namely S-bucket, P-bucket, and O-bucket, corresponding to the atoms which appear as subjects, predicates, or objects in RDF triples, respectively. Moreover, to facilitate query processing (e.g., merge join), the actual data in these buckets are sorted in different orders, in which S-bucket contains pairs of (p, o) sorted in OP order, P-bucket contains pairs of (s,o) sorted in SO order, and O-bucket contains pairs of (s,p) sorted in SP order. However, given the fact that an appropriate implementation of S, P, O indexes should use the dictionary for storing and mapping the real S, P, or O literals/URIs to 32-bits OID's, all S's (or P's, or O's) in different S, P, O indexes are physically stored as pointers to S's (or P's or O's) in the common dictionary. Thus, the common dictionary plays a similar role as the TripleT atoms index, and there is no clear advantages of TripleT-buckets in comparing to the SOP, PSO and OSP indexes.

OWLIM (and currently **GraphDB**) [120, 53, 13] is a family of semantic repository components implemented in Java and packaged as a Storage and Inference Layer (SAIL) for the Sesame openRDF framework [68]. It is comprised of a native RDF store, a reasoner and a query answering engine that supports the SeRQL [67] and SPARQL languages. The OWLIM reasoner is implemented on top of the Triple Reasoning and Rule Entailment Engine (TRREE) [30]. There are two variants of OWLIM for different environments: SwiftOWLIM (free-for-use) and BigOWLIM (commercial), that share the same rule-language and are identical in terms of reasoning expressivity and integration. However, whereas SwiftOWLIM is an entirely in-memory system, has no query processing engine of its own but relies on the Sesame framework, BigOWLIM uses a file-based storage layer and implements a number of its own query and reasoning optimizations. Besides, while the indices of SwiftOWLIM are essentially hash-tables, BigOwlim consists of two main sorted indices on RDF statements POS and PSO, two context indices $PSCO$ and $POCS$, and a literal index allowing faster look-ups of numeric and date/time object values [119].

BlazeGraph (formerly **BigData**) [9] is a high performance RDF graph database platform that provides supports RDF/SPARQL Sesame APIs, the Apache TinkerPop™ stack, and graph mining API with scalable solutions. The product has been written entirely in Java and available via either a GPL or a commercial license, and can be deployed either on a standalone server or on a cluster-based architecture. Recently, Blazegraph has extended its implementation with the use of graphical processing units (GPU) as accelerator for graph analytics, leveraging its high computational power and high memory bandwidth (for graphs that fits inside the GPU memory). Its stand-alone implementation can support up to fifty billion triples or quads. Even though it is hard to find the implementation details of BlazeGraph storage engine in published literatures, its presentations and white papers [9] mention that RDF data is stored in two set of indexes (lexicon indexes and statement indexes) built by

using sharded B+Tree. The lexicon indexes map RDF terms to unique IDs, while the statement indexes contains three indexes i.e., spo, pos, osp, (or six indexes) to provides perfect access path for RDF triples (or quads). Besides, the system also implements certain advanced query optimization technique such as runtime query optimization (inspired by ROX [35]) and vectorized query engine.

2.2.1.2 Vertical partitioning

The vertical partitioning approach was suggested by Abadi et al. in **SwStore** [34] to leverage the use of fully decomposed storage model [80] in storing RDF data. Specifically, in this approach, RDF triples are decomposed into multiple binary tables of (subject, object) columns where each table is corresponding to a unique property (the number of tables is thus equal to the number of unique properties in the input RDF data). Naturally, this approach can be implemented using column-oriented database system. As such, SwStore is built on top of the C-Store engine [174]. In this RDF store, an input SPARQL query is mapped into its equivalent SQL and then handled by the C-Store engine. According to the studies of [34] and [170] – an extensive experiment of this approach by Sidirourgos et al – the vertically-partitioned RDF store performs the best when all binary tables are lexicographically sorted according to the SO order. Each table thus can also be viewed as a slice of PSO triple table on the same P value. However, even though materialized views are created for frequent joins, SwStore does not show better performance than triple-table-based RDF stores such as HexaStore [183] and RDF-3X [151].

2.2.1.3 Summary

All the triple table-based systems even with the use of the exhaustive multiple indexing schemes will suffer high query execution cost due to a large number of self-joins that typical SPARQL processors must perform (i.e., one join for each query triple pattern). Accordingly, it is hard to get a good quality SPARQL query optimization as the query optimization complexity is exponential in the amount of joins [179]. And without analyzing the full optimizer search space (i.e., when there is a large number e.g., more than 10, of joins needed), the best optimization plan will potentially be missing. The indexing approaches taken all fall in the "unclustered index" category, which means that they lack physical locality and lead to random access, which hardware does not support efficiently. Further, all this indexing makes updating RDF stores expensive, something which is not tested by the current generation of benchmarks.

2.2.2 Structure-aware RDF stores

Structure-aware RDF storage leverages structure information derived from the input RDF data in order to improve the storage solutions as well as the efficiency of their SPARQL execution engine. Most of the existing RDF stores of this approach build so-called *property tables* to store the RDF dataset, or automatically discover the set of correlated properties for optimizing their query execution.

Property tables were introduced to reorganize RDF data in multiple tables so that subjects of the same fixed set of properties are stored in a single table (similar to a traditional relational table with a fixed set of columns). This allows to retrieve multiple triple patterns of a same subject without a single join. However, most of the early RDF systems [184, 68, 69, 76] do not provide automatic methods for recognizing the schema. They rely on the database administrator doing the table modeling manually, but given that RDF graphs often contain many different structures, this limits the applicability (and observed popularity) of this approach.

Sesame [68] is an open source framework for storing, querying RDF data, and reasoning with RDF schema. Using its Storage And Inference Layer (SAIL) which abstracts the actual storage from querying and inferencing interfaces, Sesame allows the use of different storage engines including relational databases (e.g., PostgreSQL [26], MySQL [21]), RDF triple stores (e.g., Ontotext GraphDB [13], Mulgara [187], Allegograph [32]), and even remote storage services for its backends without modifying any other Sesame components. *Sesame native* is the native triple store offering from Sesame as a reference implementation of the Sesame API. It uses dedicated on-disk data structures to store RDF data. We note that from May 2016, Sesame officially forked into an Eclipse project called RDF4J [11]. While most storage backends place Sesame in the non-structure-aware camp, the Sesame framework also supports the use of property table storage. However, as mentioned before, actual schema and physical design is left to the application tuning expert. It also allows to derive the table definitions automatically from the ontology of a dataset, which we discuss in the sequel.

Jena is a Java open-source framework which provides similar features as Sesame including a connection API to different storage engines and inferencing support [135, 136, 185]. Moreover, this framework not only provides reasoning support for RDFS (like Sesame) but also for OWL (Web Ontology Language) semantics. Jena implements two components for RDF storage and query namely Jena SDB [2] and Jena TDB [3], in which SDB uses conventional SQL databases (e.g., PostgreSQL, MySQL, Oracle) for the storage and query of RDF data, and TDB is a Jena native store supporting full range of Jena APIs. As of June 2013, Jena SDB has not been actively developed while still being maintained, and it is recommended to use Jena TDB for better performance and scalability. While most configurations of TDB and SDB fall into the non-structure aware camp, the Jena framework also supports a property table implementation [184] in which a new table is created for each `rdf:type` property, however the actual layout (i.e., grouping of predicates) of each property table needs to be defined by the application. In particular, the database developer also needs to specify multi-valued predicates so as to store them separately in new tables.

As ontologies and semantics web vocabularies also provide descriptions on certain concepts (or "ontology classes") and relationships between them, a number of RDF systems such as DLDB [155], DBOWL [148], Rstar [130], RDFSuite [38] use the ontology class structure as relational schemas for structurally storing their

[2] Jena SDB, https://jena.apache.org/documentation/sdb/

[3] Jena TDB, https://jena.apache.org/documentation/tdb/

RDF data. In particular, **DLDB** [155] creates tables corresponding to the definition of classes or properties in ontology. This can be considered as the hybrid of the property table and the vertical partitioning approaches. For naming the tables in the schema, the ontology classes' names or the properties ID are used. **DBOWL** [148] maps axioms in a given ontology to a set of relational database views in order to facilitate the execution of instance retrieval queries (e.g., get all instances of a concept defined in the ontology). **Rstar** maintains structural information of RDF data by storing both the ontology classes and instance data separately in different tables in the database and using as so-called "InstanceOfClass" table to bridge the ontology information and the instance data. ICS-Forth **RDFSuite** [38], a suite of tools for RDF validation, storage and querying, explores the available knowledge on RDF schema (RDFS) in designing a persistent RDF storage for loading resource descriptions in an object-relational DBMS (e.g., PostgreSQL [26]).

Using ontologies can be a quick approach to get certain knowledge on the structure in RDF data, however, according to our analysis [160] (and Chapter 3), each single ontology class is actually a poor descriptor for the actual structure (i.e., "relational" schema) of the data. This is because in all prominent RDF datasets (e.g., the LOD cloud) typically only a small percentage (e.g., 30%) of ontology class properties are used in the actual RDF triples, and a subject usually contains attributes from multiple ontologies. In other words, RDF triples in the wild tend not to conform 1:1 to a single ontology class, and assuming so goes counter to the grass-roots decentralized pay-as-you-go philosophy of the Semantic Web.

Automated methods for detecting property tables as well as structure information from RDF triples have been proposed in some newer systems [171, 128, 181, 134, 63]. One of the first implementation that automatically explores the structure from RDF data was proposed by Sintek et al. [171]. In their **RDFBroker** system, they identify for each subject in the RDF graph a so-called "signature" which is actually the set of properties of that subject, then create a signature table for storing the subjects of the same signature and its property values. In order to reduce the large number of signature tables, a greedy algorithm is proposed to merge smaller tables into a larger one while trying to minimum the number of NULL values added because of the merging operation.

Levandoski et al. [128] later built a so-called data-centric storage by leveraging previous work on association rule mining to automatically identify predicates that often occur together. They propose two-phase algorithm involving *clustering* and *partitioning* which aims to maximize the size of each group of predicates while trying to keep NULL values under certain threshold.

Matono and Kojima [134] construct so-called paragraph tables which are similar to property tables from adjacent RDF triples that are physically correlated. However, this method relies on well-structured input RDF documents and the parse order of RDF triples.

Recently, Bornea et al. [62] built an RDF store, **DB2RDF**, on top of IBM's relational system DB2 using hash functions to shred RDF data into multiple multi-column tables. In particular, each predicate is hashed to a specific column pair of a relational table where one column stores the predicate label and the other stores the predicate value. All predicates of a given entity are aimed to (ideally) fit on a

single row, in order to avoid self-joins when retrieving multiple predicate values of the entity. For that, DB2RDF tries to assign predicates that do co-occur together to different columns, and overload columns with predicates that do not co-occur together. The promising initial research prototype regrettably was never realized into a competent commercial variant. This proposal may capture the co-occurring predicates of each entity, however as all the entities with different set of predicates are stored together in a single relational table, it does not provide a human-readable entity-based schema to the users. This lack of human-readable representation is also a disadvantage of the other mentioned automatic structure-aware systems.

In addition to the use of structure information in the storage engine, there are also a number of studies on exploiting structure information from RDF data in order to improve SPARQL query optimization. Neumann et al. [149] extract the so-called "characteristic sets" – a set of properties that occurs frequently with the same subject – from RDF triples but merely use them for estimating join cardinalities to facilitate query processing. Gubichev et al. further exploits characteristics sets in exploring the structure of SPARQL query in order to improve the cost model of a query plan. Both of these works are implemented in the RDF-3X engine and do not make any effort in optimizing the storage based on the explored structure information. Brodt et al. [66] recognize that all relations and attributes of a resource are stored continuously in such index as SPO which is similar to a record-based RDBMS where all attributes of a resource are physically and continuously stored in the same record. They thus proposed a new operator, called *Pivot Index Scan*, to efficient deliver attribute values for a resource (i.e., subject) with less joins using something similar to a SPO index. However, as such, it does not recognize structure in RDF to leverage it on the physical level.

None of the existing structure-aware RDF stores attempt to find (foreign key) relationships between property tables, nor do they attempt to make the schemas human-readable, nor do they aim to allow both SQL and SPARQL access to the same data, nor do these efforts focus on leveraging such storage inside database kernels with new algorithms. Our research tries to achieve all of the aims.

Table 2.1 summaries the storage layouts and supported features (e.g., updating, inferencing) of centralized RDF stores.

2.2.3 Distributed RDF Stores

Most of the existing distributed RDF stores either relies on novel cloud-based platforms such as NoSQL key-value stores [75] and MapReduce implementation [85] (e.g., Hadoop), or exploits a set of centralized RDF systems distributed among many nodes for parallel processing.

The MapReduce-based approach has led to RDF systems in which RDF triples are stored in distributed file system (e.g., HDFS), and query processing is done by leveraging the MapReduce paradigm [83]. Representative works in this approach are SHARD [166], HadoopRDF [113], PredicateJoin [190], EAGRE [189]. Generally, in this approach, HDFS files are first scanned to find matching bindings of each triple pattern, then MapReduce joins [85] are executed in order to retrieve the matching for all triple patterns of the input SPARQL query. Obviously, the

detailed storage model in HDFS files significantly influences the performance of these systems as it determines the access of RDF triple and the number of MapReduce joins [153]. In particular, SHARD directly stores RDF triples of the same subject in one line in HDFS file. HadoopRDF and PredicateJoin use the property-based partitioning approach (similar to the vertical partitioning in the centralized RDF systems) in order to group triples having the same property in a single file. As the `rdf:type` property file may have many triples, HadoopRDF further splits this HDFS file of this property into multiple files based on the object values of the triples. While these above mentioned systems are non-structure-aware RDF stores, EAGRE propose a structure-aware approach for storing RDF graphs in HDFS files. In this system, by grouping subjects of similar properties into an entity class, the input RDF graph is first transformed into a compressed entity-based RDF graph which contains entity classes and connections between them. Then, the global compressed entity graph is partitioned using the METIS algorithm [118, 19] so that entities will be stored in HDFS according to the partition they belong to. This storage layout together with carefully scheduling Map tasks can reduce the I/O cost of the query processing by determining and only scanning the data blocks that contain query answer in the Map phase, and exploiting the use of multidimensional indexing techniques i.e., space filling curves [124] for efficient data indexing. These approaches benefit from the high scalability and fault-tolerance offered by MapReduce, but also suffer a non-negligible overhead due to the iterative, synchronous communication protocols of this framework. Even the system that try to minimize the I/O costs like EAGRE by cannot completely avoid costly Hadoop-based joins. [101].

Instead of using HDFS for storing data, many distributed RDF systems use NoSQL Key-value stores such as Apache Accumulo [3], Apache Cassandra [4], Amazon SimpleDB [2], HBase [5], Amazon DynamoDB [84] as their underlying storage facility. Representative key-value store-based RDF systems are Rya [162], CumulusRDF [123], Stratustore [172], H2RDF [156], and Amada [40]. As the key-value databases naturally offer an index built on the key itself, most of the existing key-value-based RDF systems store the data in multiple indexes built on different permutation of s, p, o like in the centralized RDF systems. However, due to the storage overhead of having exhaustive indexing, these distributed RDF stores uses much less indexes than the centralized RDF systems. In particular, most of them only use three indexes SPO, POS, and OSP for efficiently providing different access path for RDF s, p, o triples. Moreover, based on the particular capabilities of underlying key-value stores, these RDF systems propose different design for mapping the indexes to the key and values. For example, considering the SPO index, Rya maps sorted index on combination of S, P, O as the key, and leaves the value empty, while H2RDF builds sorted index on the combination of S,P as key and maps O to the values. The key-value RDF stores can provide very fast lookups and efficient for selective queries, however, as most of the NoSQL databases do not even support joins, joins need to be performed out of the key-value store using fullscans, or alternatively lead to avalanches (potentially billions) of key lookups using the NoSQL APIs. For example, Rya implements an index nested loops join algorithm which may only be efficient for selective queries. H2RDF, instead of hav-

ing only one centralized index nested loop algorithm for joins, alternatively uses Map-Reduce for handling non-selective queries.

Besides using the available cloud-based platforms and frameworks, many distributed RDF systems exploit existing centralized RDF stores for their storage and query processing. The input RDF graph is then divided into multiple partitions of which each will be stored in a node by using a particular centralized RDF store (e.g., RDF-3X). For query processing, the input SPARQL query is decomposed into sub-queries such that each sub-query can be processed locally at a node. The final result is then formed by aggregating over all the answers for each each subquery. The detailed query processing algorithm of each system generally depends on its partitioning strategy. The representative distributed RDF systems of this approach are GraphPartition [112], WARP [111], Partout [95], TriAD [101], [188], 4store [104], Virtuoso Cluster [87], and BlazeGraph (BigData) [9].

In particular, GraphPartition — one of the first systems with this approach — uses RDF-3X for storing and indexing triples in each node. In this system, the input RDF graph is partitioned on vertices using METIS [19] such that the number of edge cuts is minimum. Here, an edge is cut if its source vertex and its destination vertex (i.e., subject and object of an RDF triple) are partitioned into two separated partitions. For query processing, it first check whether the decomposed query is PWOC (parallelizable without communication) so that the final result can be obtained as the union of answers from each RDF-3X engine. Otherwise, Hadoop joins need to be performed to join answers of subqueries from RDF-3X engines. WARP [111] further extends the partitioning and replication strategies of GraphPartition in order to reduce the storage overhead by taking into account frequent structures in query workload. Specially, in this system, the rarely-used RDF data is not replicated. Partout [95] proposes an optimized data partitioning and allocation algorithm such that queries can be executed over a minimum number of nodes by exploiting frequent access pattern in representative query workload (or query log). Similar to GraphPartition, both WARP and Partout use RDF-3X as the centralized RDF store at each node. TriAD (Triple-Asynchronous-Distributeds), instead of using a particular centralized RDF store, maintains six local indexes over all permutations of SPO in each node (e.g., each local indexes are similar to the exhaustive indexes in centralized RDF store such as HexaStore, RDF-3X). Its partitioning algorithm is again done by using METIS software. By using a custom Message Passing Interface (MPI) protocol that allows slave nodes operate largely automatically and execute multiple join operators in parallel, this main-memory shared-nothing system can be considered as the first RDF system that employs asynchronous join execution.

Virtuoso Cluster [87, 59] and 4store [104] use their own centralized RDF storage engine for storing and processing data at each node. In particular, 4store is a clustered system designed to run on relatively small cluster. It divides the nodes into *Storage* nodes and *Processing* nodes in which data is divided among a number of segments (non-overlapping slices of data) and stored in storage nodes, while the SPARQL engine and RDF parser locate in the processing node. In this system, the partition strategy is simply based on the assigned OID of each resource and the number of segments (i.e., resource ID mod segments). Virtuoso Cluster uses a

hash-based elastic partitioning strategy in which the data partitions divided among a number of database server processes can migrate between each others, and partitions may split when growing a cluster. It also maintains statistics per partition for detecting hot spots and allows the replication of hash tables in every processes for efficient parallel hash joins (e.g., in case hash join build side is small).

By optimizing the data partition algorithm and the join operators between nodes, these systems may avoid the overhead of iterative Map-Reduce paradigm and Hadoop-based joins, however, as these system rely on centralized RDF stores for their storage and query processing, they still have all data management problems mentioned for centralized RDF stores.

Table 2.2 summaries the storage backend and storage layouts of distributed RDF systems.

Store	Structure-aware	Storage layout	Update support	Inference support
Redland		TT	✓	
3store		TT		✓
Oracle		TT		✓
RDFStore		TT		✓
Mulgara		TT/MI		✓
HexaStore		TT/MI		
RDF-3X		TT/MI	✓	
YARS/YARS2		TT/MI	✓	
HPRD		TT/MI		
Virtuoso		TT/MI	✓	✓
BitMat		TT/MI		
TripleT		TT/MI		
OWLIM		TT/MI	✓	✓
BlazeGraph		TT/MI	✓	✓
BRAHMS		TT/MI		
RDFJoin		TT/MI		
RDFKB		TT/MI	✓	✓
iStore		TT/MI		
Parliament		TT/MI	✓	✓
StarDog		TT/MI	✓	✓
RDFCube		TT/MI		
SwStore		VP		
Sesame	✓	PT	✓	✓
Jena	✓	PT	✓	✓
DLDB	✓	PT/O		✓
DBOWL	✓	PT/O		✓
Rstar	✓	PT/O		✓
RDFSuite	✓	PT/O	✓	✓
RDFBroker	✓	PT/A		
DataCentric	✓	PT/A		
Paragraph	✓	PT/A		
DB2RDF	✓	PT/A		

Table 2.1: Centralized RDF stores' storage layout and feature support. (TT: *Triple Table*, MI: *Multiple Indexing*, VP: *Vertical Partitioning*, PT: *Property Table*, PT/O: *Ontology and vocabulary-based Property Table*, PT/A: *Auto-detected Property Table*)

Systems	Storage backend	Storage layout/Partitioning	Structure-aware
SHARD	HDFS	Triple-based files	
HadoopRDF	HDFS	Property-based files	
PredicateJoin	HDFS	Property-based files	
EAGRE	HDFS	Entity-based graph partition	✓
Rya	KV/Accumulo	SPO, POS, OSP Indexes	
CumulusRDF	KV/Cassandra	SPO, POS, OSP Indexes	
Stratustore	KV/SimpleDB	SPO Index	
H2RDF	KV/HBase	SPO, POS, OSP Indexes	
Amada	KV/DynamoDB	SPO, POS, OSP Indexes	
GraphPartition	Hadoop/RDF-3X	Graph Partition (METIS)	
WARP	CS/RDF-3X	Graph Partition	
Partout	CS/RDF-3X	Graph Partition	
TriAD	SPO indexes	Graph Partition (METIS)	
4store	CS	Modular partitioning	
Virtuoso Cluster	CS	Hash-based elastic partitioning	
BlazeGraph	CS		

Table 2.2: Distributed RDF systems' storage scheme. (HDFS: Hadoop Distributed File System, KV: Key-value store, CS: Centralized RDF Store)

Chapter 3

Deriving an Emergent Relational Schema from RDF Data

In this chapter, we motivate and describe techniques that allow to detect an "emergent" *relational schema* from RDF data. We show that on a wide variety of datasets, the found structure explains well over 90% of the RDF triples. Further, we also describe technical solutions to the semantic challenge to give short names that humans find logical to these emergent tables, columns and relationships between tables. Our techniques can be exploited in many ways, e.g., to improve the efficiency of SPARQL systems, or to use existing SQL-based applications on top of any RDF dataset using a RDBMS.

3.1 Introduction

By providing flexibility for users to represent and evolve data without the need for a prior schema – sometimes called the "schema last" approach – and identifying properties and (references to) subjects uniformly using URIs, RDF has been gaining ground as the standard for global data exchange and interoperability, recently through the popularization of micro-formats such as RDFa, which are increasingly used embedded in web pages. This creates a need for database technologies that can query large amounts of RDF efficiently with SPARQL *or SQL*.

SQL-speaking relational database systems (RDBMS's) require to declare a schema upfront ("schema first") and can only store and query data that conforms to this schema. RDF systems typically rely on a "triple-store" architecture, which store all data in a single table containing `S`, `P` and `O` (subject, property, object) columns[1]. SQL systems tend to be more efficient than triple stores, because the latter need query plans with many self-joins – one per SPARQL triple pattern. Not only are these extra joins expensive, but because the complexity of query optimization is exponential in the amount of joins, SPARQL query optimization is much more complex than SQL query optimization. As a result, large SPARQL queries often exe-

[1] With "triple-store" we mean RDF or graph stores that use any data structure, be it a graph edge-list, B-tree, hash map, etc. that stores individual triples (or quads), or graph edges without exploiting their connection structure.

cute with a suboptimal plan, to much performance detriment. RDBMS's can further store data efficiently e.g. using advanced techniques such as column-wise compression, table partitioning, materialized views and multi-dimensional data clustering. These techniques require insight in the (tabular) structure of the dataset and have so far not been applicable to RDF stores.

Semantic Web technology has its roots in Artificial Intelligence and knowledge representation, and we think it is seldom realized that its notion of "schema" in the term "schema last" differs from the corresponding "schema" notion in "schema first" for relational technology. Semantic Web schemas – *ontologies* and *vocabularies* – are intended to allow diverse organizations to consistently denote certain concepts in a variety of contexts. In contrast, *relational schemas* describe the structure of one database (=dataset), designed without regard for reuse in other databases.

Our work shows that actual RDF datasets exhibit (i) a very *partial* use of ontology classes and (ii) subjects share triples with properties from classes defined in *multiple* ontologies. To illustrate, (i) in the crawled WebDataCommons data there is information on less than a third of the ontology class properties in the actual triples, and (ii) we find in DBpedia that each subject combines information from more than *eight* different ontology classes on average. As such, when analyzing the actual structure of RDF datasets by observing which combinations of properties typically occur together with a common subject (called "**C**haracteristic Sets" of properties [149]), any single ontology class tends to be a poor descriptor. Knowledge of the actual structure of a dataset is essential for RDBMS's to be able to store and query data efficiently. Our work allows RDF stores to automatically discover this actual structure, which we call the **e**mergent relational schema. The emergent relational schema enables to internally store RDF data more like relational tables, allowing SPARQL query execution to use less self-joins, which also reduces the complexity of query optimization [99]. Note that not all triples in a RDF dataset need to conform to this relational schema for these techniques to be effective, as long as the great majority does. Hence, RDF remains as flexible as ever in emergent relational schema aware systems.

There is also a usability advantage if the actual structure of an RDF dataset would be conveyed to a human user. A common problem when posing SPARQL queries is that queries come back empty if properties that one expects to occur given ontology knowledge, turn out not to be present in the data. Or, one may lack any ontology knowledge and thus have little to go by when querying. However, automatically deriving a human-friendly relational schema from a RDF dataset introduces additional challenges to recognizing its structure, since all the schema elements (tables, columns) should get correct and short *labels*, and the emergent relational schema must be *compact* to be understandable.

Our work presents a self-tuning algorithm that surmounts this challenge, which we tested on a wide variety of RDF datasets. We integrated our techniques in two open-source state-of-the art data management systems: the well-known RDF store Virtuoso and the MonetDB DBMS. The RDF bulkload in MonetDB now offers efficient SQL access to any RDF dataset via its emergent relational schema, allowing the wealth of SQL-based applications over ODBC and JDBC (e.g. Business Intelligence tools like Tableau) to be used. By doing so we are enriching RDBMS's

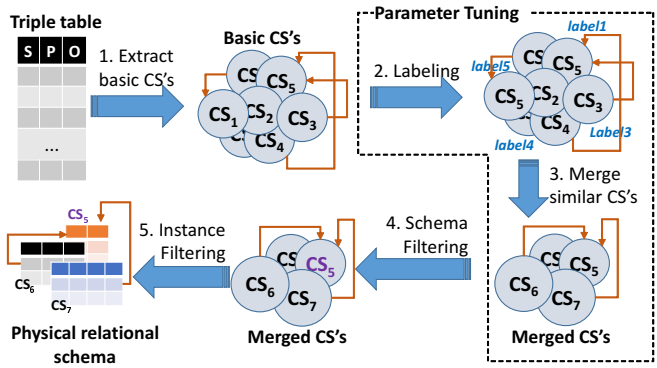


Figure 3.1: Overall structural exploration steps

with web standards, because these relational tables, columns and foreign key (FK) constrains are identifiable using ontology-based URIs, and even the primary key values and foreign key values themselves are URIs (RDF subjects resp. non-literal objects). As such, our work is a bridge between the Semantic Web and RDBMS's, enriching both worlds.

Contributions of our work are the following:

1) We identify an important difference between Semantic Web schema information (describing a knowledge universe) and relational schemas (describing one dataset), and argue that *both* should be available to data stores and their users.

2) We present methods for detecting the basic table structure and the relationships between them from an RDF dataset, and propose several approaches to semantically and structurally optimize the relational schema to make it compact.

3) We present techniques to assign human-friendly names to tables/columns and their FK relationships.

4) Our experiments on a wide variety of RDF datasets show that (i) over 90% of the triples in these conform to a compact emergent relational schema, (ii) our algorithms are efficient and can be executed during RDF bulk load with little overhead, (iii) RDF stores can improve both query optimization and execution by exploiting the emergent relational schema, and (iv) we illustrate with a user survey that the short human-readable labels we find have good quality.

3.2 Emerging A Relational Schema

The five steps of our emergent relational schema algorithm detect something akin to a UML class diagram by analyzing Characteristic Sets (CS's) [149] in an RDF input dataset:

1. Basic CS Discovery. We discover all occurring CS's from a bulk-loaded SPO table and count their frequencies. Then, we analyze properties in each CS that are not literals, i.e. refer to URIs (and hence to other CS's) in order to explore the relationships between CS's.

2. CS Labeling. We assign class, attribute and relationship labels (human-understandable names) to the recognized CS's using multiple methods.
3. CS Merging. We merge CS's that are semantically or structurally similar to each other, with the purpose of making the schema more *compact*. We re-run Steps 2 and 3 iteratively in order to *automatically tune* the similarity threshold parameter τ_{sim} to the nature of the dataset.
4. Schema Filtering. We filter low frequency CS's, but make sure to conserve highly referenced CS's (akin to relational "dimension tables"). As reference relationships can be indirect (via via) we use a PageRank-like algorithm to count how often referenced each CS is. We also filter out CS properties that are too sparsely populated.
5. Instance Filtering. We filter out instances (rows) to increase literal type homogeneity, and filter out individual triples to eliminate erroneously multi-valued attributes, and to improve foreign-key cardinality homogeneity.

The "class diagram" where each merged-CS that survived filtering is a class, is represented as a relational schema consisting of tables and foreign key relationships. Each class becomes one table, and its properties its columns, but relationships and multi-valued attributes lead to additional tables. Properties for which multiple literal types occur frequently, are represented by multiple table columns. The $<10\%$ triples that do not fit this schema remain stored in a separate SPO table. We now discuss the five steps in detail.

3.2.1 Basic CS Discovery

Given an RDF dataset R, the Characteristic Set of a subject s is defined as $cs(s) = \{p|\exists o : (s, p, o) \in R\}$ [149].

We first identify the basic set of CS's by analyzing all triples stored in an RDF table in SPO order. Such table is produced by a standard bulk load employed by triple stores. While loading the triples into this representation, the URIs get encoded in a dictionary, such that columns `S`, `P` and `O` are not URI strings, but integers called *object identifiers* (OIDs) pointing into this dictionary. This is a standard technique. These integer OIDs form a dense domain starting at 0.

We now make a single pass through the SPO table and fill a hash map where the key is the set of OIDs of properties that co-occur for each subject. Note that due to the SPO ordering these are easily found as the `P`'s of consecutive SPO triples with equal `S`. The key of the hash map is the offset in the SPO table where the CS first occurs. Its hash is computed by XOR-ing the hashes of all `P`'s (which are OIDs). The insert-order in the hash table (starting at 1) provides us with a dense numeric OID for each CS. Further, we remember in an array indexed by `S` which stores such CS-OIDs, to which CS each subject belongs (this array is part of the URI dictionary). Note that not all URIs in the dictionary may occur as a subject in the SPO table, for which case this array is initialized with zeros. After making the single pass over the SPO table, we will have all occurring *basic CS*-s in the hash map, and we also keep an occurrence count there.

Further we make a second pass over the SPO table, where we look at type information. For each triple with a literal object, we maintain a histogram of literal-

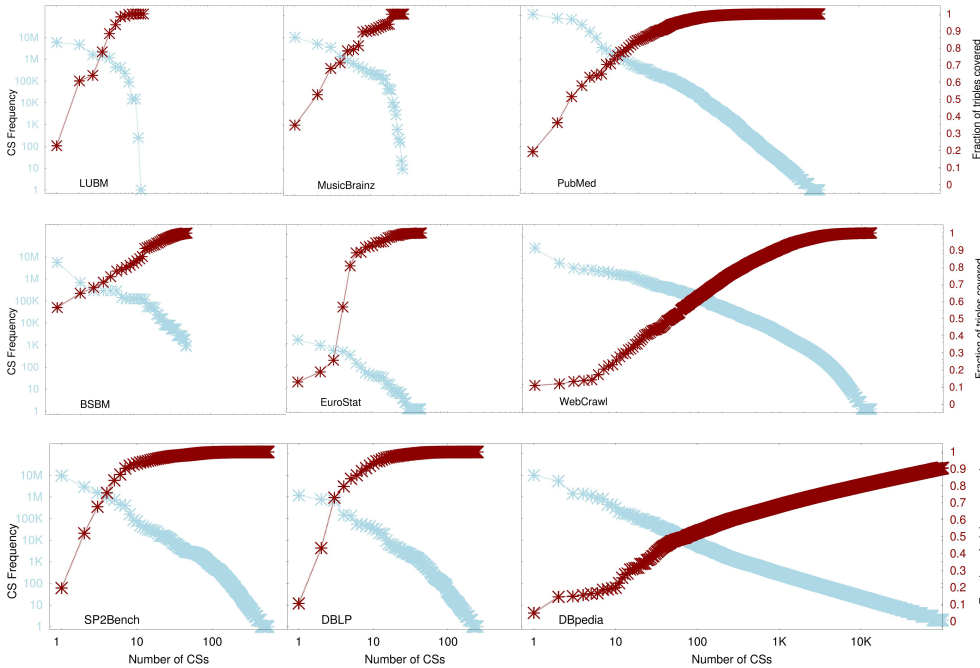


Figure 3.2: CS Frequency (light blue) vs. Cumulative number of covered triples (dark red)

type occurrences per property in a second hash map with key `[P,type]` and a count value. For each triple that is a non-literal, on the other hand, we look up to which CS its subject belongs (`srcCS`) and to which its object (`dstCS`) – this can be done efficiently using the array mentioned before. If there is a `dstCS`, we maintain another histogram stored in a third hash map with as key `[srcCS,P,dstCS]` and a count value. This histogram records how often basic-CS's refer to each other and over which property (relationship statistics).

These algorithms are all simple and obviously linear in average-case complexity, therefore we omit a listing or further analysis. Figure 3.3 shows an example of the found basic CS and their relationships after the exploration process.

Diversity of the basic CS's. Table 3.1 shows statistics on the basic CS's and their properties for the **s**ynthetic RDF benchmark datasets LUBM[2], SP2B[3],and BSBM[4], the originally **r**elational datasets converted to RDF MusicBrainz[5], EuroStat[6], and

[2] swat.cse.lehigh.edu/projects/lubm/
[3] dbis.informatik.uni-freiburg.de/forschung/projekte/SP2B/
[4] wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/
[5] linkedbrainz.c4dmpresents.org/data/musicbrainz_ngs_dump.rdf.ttl.gz
[6] eurostat.linked-statistics.org

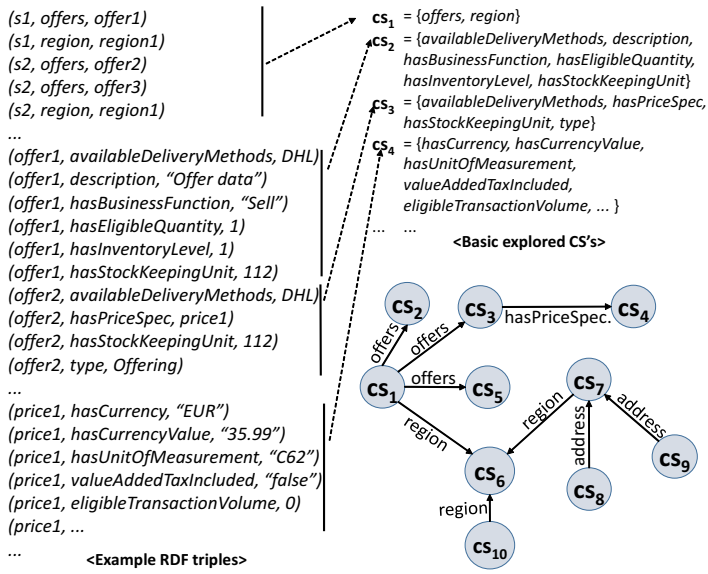


Figure 3.3: Example of basic CS's and their relationships

DBLP[7], PubMed[8] and the **n**ative RDF datasets WebDataCommons[9] ("WebData.") and DBpedia[10]. The number of basic CS's can vary significantly regardless the number of input triples. If one would naively propose to store RDF data using a separate relational table for each basic CS, we now see that a complex RDF dataset like DBpedia would lead to an unacceptable number of small tables. As we can also see that while most of the datasets have a single literal type for each CS property, DBpedia and WebDataCommons have many properties with more than one literal type in its object values (i.e., multi-type properties), so native datasets appear to be both complex and "dirty".

Data coverage by basic CS's. Figure 3.2 shows the frequencies and the cumulative number of triples covered by the basic CS's sorted by their frequencies, for one of each kind of dataset (synthetic, relational, native). In this figure, the number of CS's needed for covering a large portion of the triples (e.g., 90%) can be significantly different between the datasets. We show for reference in Table 3.1, that 90% of the synthetic benchmark datasets can be covered by using a small number of CS's (e.g., 7 for SP2Bench). Many Linked Open Data datasets originate from existing sources whose data is kept in relational databases. We see that in such datasets a few CS's can cover almost all data. However, for complex datasets originally created as RDF (native), in order to cover 90% of the triples, many CS's are needed, in case of DBpedia more than 85,000.

[7] gaia.infor.uva.es/hdt/dblp-2012-11-28.hdt.gz
[8] www.ncbi.nlm.nih.gov/pubmed
[9] A 100M triple file of webdatacommons.org
[10] dbpedia.org - we used v3.9

Datasets	#triples*	#CS's	#CS's to cover 90%	Avg. #prop.	#multi-type properties
LUBM	100M	17	7	5.71	0
BSBM	100M	49	14	12.61	0
SP2Bench	100M	554	7	9.8	0
synthetic	*data created by benchmark data generator*				
MusicBrainz	179M	27	10	4.7	0
EuroStat	70K	44	8	7.77	0
DBLP	56M	249	8	13.70	0
PubMed	1.82B	3340	35	19.27	0
relational	*RDF data from a relational database dump*				
WebData.	90M	13354	930	7.94	551
DBpedia	404M	439629	85922	24.36	1507
native	*real data originating as RDF*				

Table 3.1: Statistics on basic CS's.
(*: Number of triples after removing all duplicates)

	mixed	**p**artial
dataset	number of ontology classes used per CS	%ontology class properties used per CS
LUBM	1.94	37%
BSBM	3.96	3%
SP2Bench	4.94	4%
MusicBrainz	3.93	1%
EuroStat	3.14	84%
DBLP	6.58	8%
PubMed	4.94	-
WebData.	2.27	33%
DBpedia	8.35	5%

Table 3.2: Partial & mixed ontology class usage in CS's

3.2.2 CS Labeling

When presenting humans with a relational schema, short labels should be used as aliases for machine-readable and unique URIs for naming tables, columns and relationships between tables. For assigning labels to CS's, we exploit semantic information (ontologies) as well as structural information. Because not all ontologies follow the same structure, we developed a simple vocabulary to standardize minimal aspects of an ontology, namely classes and their properties, relationships between classes, their labels, as well as the subclass hierarchy. We expressed a large

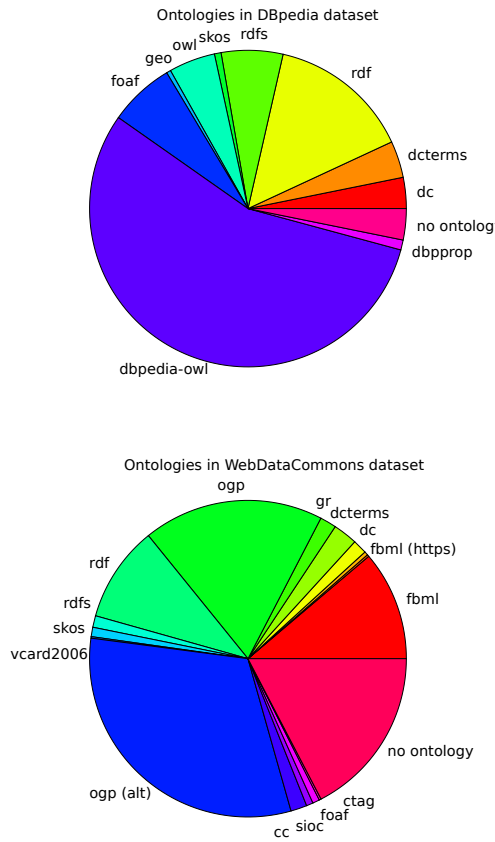


Figure 3.4: Ontologies used in native RDF datasets

set of common ontologies in this vocabulary. Our proposed system is extensible, as new ontology information can easily be added.

Figure 3.4 shows ontology class usage in the CS's and the percentage of triples corresponding to each ontology in several datasets. As shown in the graphs, each dataset contains a mix of multiple ontologies where even the most popular ontology covers less than 56% of the data. The first column in the table shows that properties from within *a single CS* typically stem from a number of different ontologies, e.g., the average number of ontologies used in each CS in DBpedia is 8.35. We also looked at the percentage of properties of each ontology class when used in a CS. Since an ontology class may be used in multiple CS's, we compute a weighted average (where the number of subjects in a CS is the weight). The second column in the table shows this percentage to be less than 10% in most of the datasets. In other words, the datasets make only very partial usage of the properties of each ontology class. The partial usage and mixing together mean that any individual on-

label of `rdf:type`	subjects CS %	all %
Thing	100	83
Organization	100	7
RadioStation	**9**7	**0**.2
Company	1	4

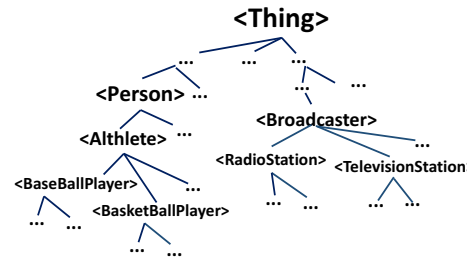


Figure 3.5: Choosing a CS label from explicit RDF type annotations that refer to ontology classes in a hierarchy.

tology class is a poor descriptor of the structure of the data. Our emergent relational schema, aims to provide a better description.

Type properties. Certain specific properties explicitly specify the *concept* a subject belongs to. The most common RDF property with this role is `rdf:type`, where the `O` of triples with this property may be the URI identifying an ontology class. Recall that our first step is to find a good UML-like class diagram for the RDF dataset, where a CS roughly corresponds to an UML class, and specifically here we are trying to find a human-friendly short name (label) for each CS. Even though we stated above that any individual ontology class is a poor descriptor for the *structure* of a CS, ontologies do provide valuable clues for choosing a *label* (name) for the CS. The subjects that are member of a CS may have different `rdf:type` object values, this number is also variable (there can be zero such type annotations, but also multiple). To choose one, we look at the frequency of that type annotation. First, we use the global infrequent threshold τ_{inf} (e.g., 5%) to exclude infrequent type annotations to be used for finding the CS class label. For the rest, we count (i) how many subjects in the CS have it, and (ii) how many subjects in the whole dataset have it. Similar to TF/IDF [168], dividing (i) by (ii) provides a reasonable ranking to choose an appropriate ontology class. Finally, if the ontology class has label information (and this information is typically available), we then use it as the label for the CS.

We should, however, in this ranking also take into account the class hierarchy information that an ontology provides. Thus, we account for missing superclass annotations by inferring them for the purpose of this ranking. In Figure 3.5, if a triple in some CS has `rdf:type Company`, but not `Organization` or `Thing` explicitly, we still include these annotations in the ranking calculation.

In this example, "RadioStation" is chosen as its coverage of the subjects in the CS is above τ_{inf} ($97 > 5$) and its ranking score (97/0.2=485) is the highest.

Discriminative Properties. Even if no type property is present in the CS, we can still try to match a CS to an ontology class. We compare the property set of the CS with the property sets of ontology classes using the TF/IDF similarity score [168].

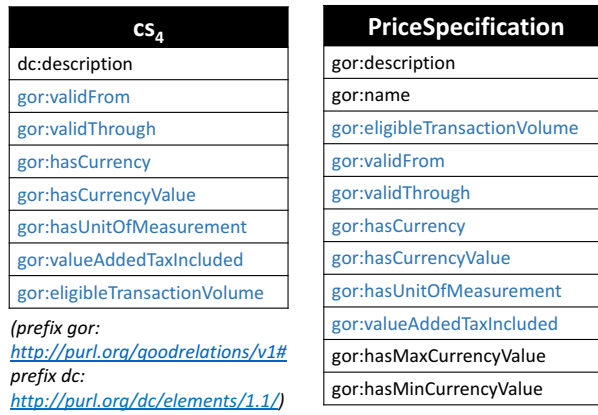


Figure 3.6: Example CS vs. Ontology class

This method relies on identifying "discriminative" properties, that appear in few ontology classes only, and whose occurrence in triple data thus gives a strong hint for the membership of a specific class. An example is shown in Figure 3.6. In this example, as cs_4 and the class `PriceSpecification` of the GoodRelations ontology[11] share discriminative properties like `gor:hasUnitOfMeasurement` and `gor:valueAddedTaxIncluded`, `PriceSpecification` can be used as the label of cs_4. Detailed computation of the TF/IDF-based similarity score between a CS and an ontology class can be found in [157]. An ontology class is considered to be matching with a CS if their similarity score exceeds the similarity threshold τ_{sim}. The ontology class correspondence of a CS, if found, is also used to find labels for properties of the CS (both for relationships and literal properties).

Relationships between CS's. If the previous approaches do not apply, we can look at which other CS's refer to a CS, and then use the URI of the referring property to derive a label. For example, a CS that is referred as `Address` indicates that this CS represents instances of an `Address` class. We use the most frequent relationship to provide a CS label. For instance, in WebDataCommons 93532 instances refer to a CS via property `address` and only 3 via property `locatedAt`. Thus, `Address` is chosen as the label.

URI shortening. If the above solutions cannot provide us a link to ontology information for providing attribute and relationship labels, we resort to a practical fall-back, based on the observation that often property URI values do convey a hint of the semantics. That is, for finding labels of CS properties we shorten URIs (e.g., `http://purl.org/goodrelations/v1#offers` becomes `offers`), by removing the ontology prefix (e.g., `http://purl.org/goodrelations/v1#`), as suggested by [152].

Note that for CS's without any ontology match or relationships with other CS's, we may find no class label candidates, in which case a synthetic default label is

[11] purl.org/goodrelations/

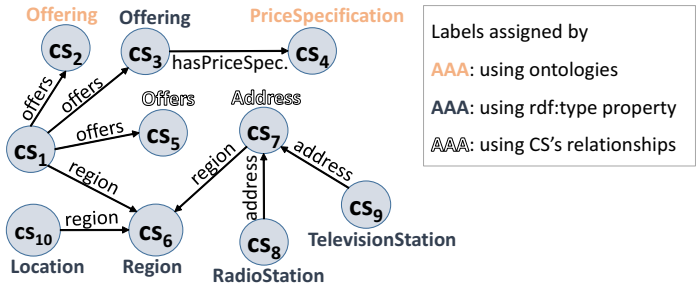


Figure 3.7: CS's with assigned labels

used. Labels are intended to help users comprehend the data, but in any case should be overridable by manual labeling. A future approach might be to look for sources on the web, such as search engines; but for the moment we prefer to keep our techniques stand-alone, as these are part of RDF bulk-load.

Figure 3.7 shows the labels assigned to each CS in the example dataset by using different labeling methods (e.g., the label of cs_4 is assigned based on the matching between its property set and that of ontology classes, the label of cs_7 is derived from the CS's relationships, ...). In this example, cs_1 does not have any specific label as there is no sufficient information for assigning a good label to it.

3.2.3 CS Merging

After basic exploration, there may be thousands of CS's, in case of DBpedia even 100,000. This means the individual CS's have only a few subjects (=rows, in relational terms) in them, so that storing them in a relational table would incur overheads (e.g. tables not filling a disk page, large database catalog, expensive metadata lookup). Further, many of these basic CS's are very similar to each other (differing only in a few properties) and denote the same concept. When querying for that concept, one would have to formulate a UNION of many tables, which is cumbersome and also slows down queries. Finally, a relational schema with thousands of tables is just very hard to understand for humans. Therefore, the next step is to reduce the number of tables in the emergent relational schema by *merging* CS's, using either *semantic* or *structural* information.

Figure 3.8 shows an example of merging cs_i and cs_j. We note that all subjects that fall in a basic CS do so because there exist triples for all properties in that CS, such that a relational table representing the CS would have no NULL cells. In this example, cs_i and cs_j represent already the results of merging other CS's (the merging process is iterative) . As shown in the figure, the number of NOT-NULL cells in table t_{ij} is equal to the total number of NOT-NULL cells of the tables t_i and t_j, however, the number of NULL cells increases due to properties not in the intersection of the two CS's becoming padded with NULLs in the merged CS.

Semantic merging. We can merge two CS's on semantic grounds when both CS class labels that we found were based on ontology information. Obviously, two

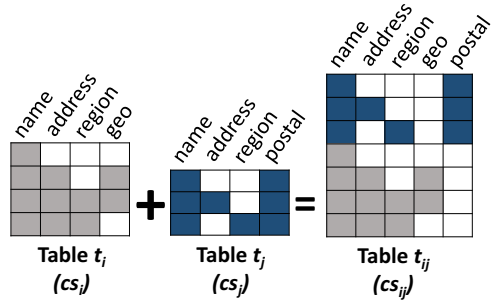


Figure 3.8: Example of merging CS's

CS's whose labels were taken from the same ontology class URI represent the same concept, and thus can be merged. If the labels stem from *different* ontology classes we can examine the class hierarchy and identify the common concept/class shared by both CS's, if any, and then justify whether these CS's are similar based on the "generality" of the concept. Here the "generality" score of a concept is computed by the percentage of instances covered by it and its subclasses among all the instances covered by that ontology (Equation 3.1).

$$g_{score}(O_c) = \frac{\#instances\_coveredby(O_s)}{\#instances\_coveredby\_ontology} \quad where\ O_s\ is\ O_c\ or\ a\ subclass\ of\ O_c \tag{3.1}$$

Figure 3.5 showed an example of an ontology class hierarchy from DBpedia. Consider two CS labels such as `RadioStation` and `TelevisionStation` assigned by using ontology class names. By following the ontology's class hierarchy, it can be found that the corresponding classes of these labels share the same infrequent superclass `Broadcaster`. Therefore, these CS's can be considered as semantically similar, and could be merged with `Broadcaster` as new label.

More formally, there are two rules for semantic merging:

Rule 1 *If an ontology class URI exists equal to the labels of both cs_i and cs_j then merge cs_i and cs_j. ($S1$)*

Rule 2 *If there exists an ontology class O_c being an ancestor of the labels of cs_i and cs_j and $g_{score}(O_c)$ is less than $\frac{1}{Ub_{tbl}}$ then merge cs_i and cs_j. ($S2$)*

In $S2$, $\frac{1}{Ub_{tbl}}$ is used as the threshold for the generality score based on Ub_{tbl}, the upper bound for the number of tables in the schema – which is one of the only three parameters of emergent relational schemas, see Table 3.3.

Figure 3.9 demonstrates the modifications to the explored CS's of the example dataset and their relationships when sequentially applying merging rules $S1$ and $S2$. Here, since cs_2 and cs_3 both derived their label `Offering` from the `Offering` class of the GoodRelation ontology, according to $S1$, they are merged

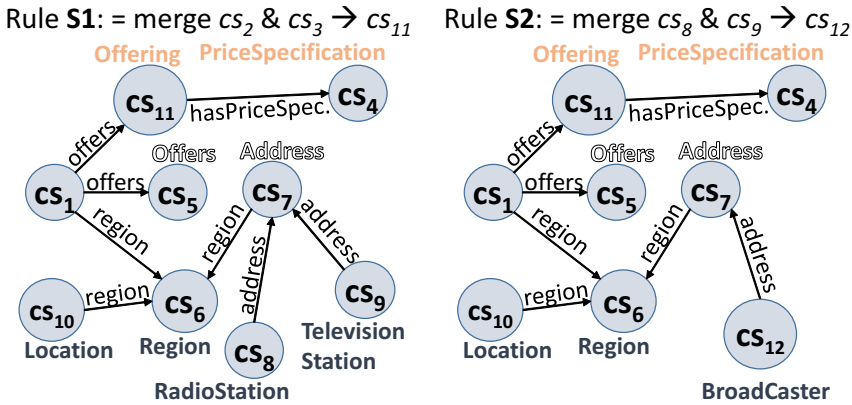


Figure 3.9: Example of merging CS's by using rules S1, S2

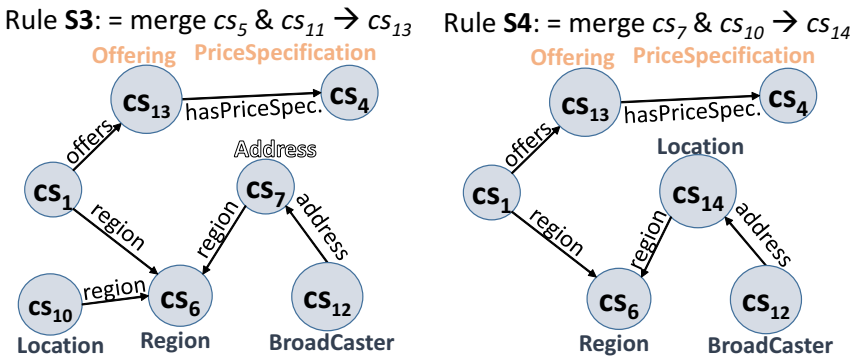


Figure 3.10: Example of merging CS's by using rules S3, S4

into a new CS (cs_{11}). The references from/to cs_2 and cs_3 are also updated for cs_{11}. Besides, since the labels of cs_8 and cs_9 have `Broadcaster` as their non-general common ancestor in the ontology hierarchy, they are merged into cs_{12} according to S2. The label of cs_{12} is assigned by using the name of the common ancestor ontology class. The full description about updating the label of a CS after merging can be found in [157].

Structural merging. While semantic merging is a relatively safe bet, it may not always be applicable or effective enough to reduce the amount of merged CS's. Therefore, we also look at the *structure* of the CS's and their relationships to see if these can be merged. The idea here is to identify CS's that denote the same concept

based on so-called "discriminative" properties, which are those with a high TF/IDF score (see the previous section). If the overlap between two CS's contains enough "discriminative" properties, we can merge them.

Figure 3.11 shows an example where the overlapping properties of cs_7 and cs_{10} indicate that these CS's both originate from the "Location" entity. Here, the property `rdfd:name` is not discriminative since it appears in most of the CS's. However, properties `rdfd:street-address` and `rdfd:region` give evidence that both represent a "Location".

Equations 3.2 and 3.3 formally show the detailed computations for the TF/IDF score of each property in a cs and the cosine similarity score (sim_{ij}) between two cs's (cs_i and cs_j), respectively. In these functions $total\#CSs$ is the total number of CS's, $\#containedCSs(p)$ is the number of CS having property p in their property list.

$$tfidf(p, cs) = \frac{1}{|D_p(cs)|} \times log \frac{total\#CSs}{1 + \#containedCSs(p)} \tag{3.2}$$

$$sim_{ij} = \frac{\sum_{p \in (cs_i \cap cs_j)} tfidf(p, cs_i) \times tfidf(p, cs_j)}{\sqrt{\sum_{p_i \in cs_i} tfidf(p_i, cs_i)^2} \times \sqrt{\sum_{p_j \in cs_j} tfidf(p_j, cs_j)^2}} \tag{3.3}$$

In addition to the set of properties in a CS, incoming relationship references from other CS's can also be used as an evidence in identifying similar CS's. Normally, a subject refers to only one specific entity via a property. For example, the property `has_author` of the subject "Book" always refers to an "Author" entity. Thus, if one CS, e.g., cs_1, refers to several different CS's e.g., cs_2 and cs_3, via a property p, this hints at cs_2 and cs_3 being similar.

In summary, two CS's are considered *structurally similar* if they are both referred from the same CS via the same property (rule $S3$) or their property sets have a high TF/IDF similarity score (rule $S4$). In Rule 3, $ref(cs, p, cs_i)$ is the number of references from cs to cs_i via property p, and τ_{inf} is the infrequent threshold which is used to prevent non-frequent references from being considered in applying the rule $S3$. In Rule 4, τ_{sim} is the similarity threshold above which we decide to merge two CS's.

Rule 3 *If cs and p exist with $\frac{ref(cs,p,cs_i)}{freq(cs)}$ and $\frac{ref(cs,p,cs_j)}{freq(cs)}$ greater than τ_{inf} then merge cs_i and cs_j. ($S3$)*

Rule 4 *If the similarity score sim_{ij} between cs_i and cs_j is greater than τ_{sim} then merge cs_i and cs_j. ($S4$)*

Figure 3.10 shows the updates to the CS's and their relationships when we continue applying the rules $S3$ and $S4$. In this figure, cs_5 and cs_{11} are merged according to the rule $S3$ as they are both referred by cs_1 via the property `offers`. Besides, since cs_7 and cs_{10} have high similarity score (as shown in Figure 3.11), they are merged into cs_{14}.

We experimentally observed that the best order of applying the rules for merging CS's is $S1$, $S3$, $S2$, $S4$. Further details can be found in [157].

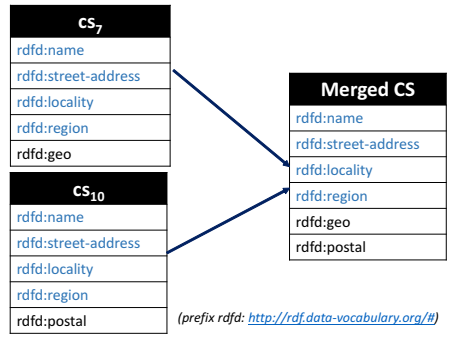


Figure 3.11: Merging CS's based on discriminative properties

3.2.4 Schema Filtering

Our goal is to represent a large portion of the input triples in a compact, human-friendly, relational schema. After CS merging, most of these merged classes[12] cover a large amount of triples. However, it may happen that some classes still cover a limited number of RDF subjects, so if the merged CS covers $< min_t$ (e.g. 1000, see Table 3.3) subjects, it is removed from the schema; and we limit the UML class diagram to the merged Ub_{tbl} CS's with highest frequency. Note that omitting CS's with low frequency will only marginally reduce overall coverage.

Preserving Dimension Tables. However, for this removal of classes (merged CS's) we make one exception, namely we conserve CS's that – although small in terms of covered subjects – are referred to many times from other tables. The rationale is that such CS's thanks to the large amount of incoming references represent important information of the dataset that should be part of the schema. This is similar to a *dimension table* in a relational data warehouse, which may be small itself, but is referred to by many millions of tuples in large fact tables over a foreign key. Thus, combining the information of basic CS detection and relationship detection, we preserve CS's with a high frequency of incoming references. However, detecting dimension tables should not be handled just based on the number of *direct* relationship references. The relational analogy here are *snowflake* schemas, where a finer-grained dimension table like CITY refers to an even smaller coarse-grained dimension table COUNTRY. To find the transitive relationships and their relative importance, we use the PageRank [154] algorithm on the graph formed by all CS's (vertexes) and relationships (edges, regardless of direction). In each iteration, the score of a merged CS is computed based on the references from other merged CS's and their scores computed in the previous iteration. Equation 3.4 shows the formula

[12] At this stage, we also refer surviving merged CS's as classes, similar to UML classes.

for each iteration:

$$IR_k(cs_i) = \sum_{cs_j \to cs_i} IR_{k-1}(cs_j) \times \frac{ref(cs_j, cs_i)}{refsTo(cs_i)} \times \frac{ref(cs_j, cs_i)}{freq(cs_j)} + refsTo(cs_i)$$
$$\text{If}(IR_k(cs_i) \geq Ub_{tbl}) \rightarrow cs_i \; is \; a \; dimension \; CS. \tag{3.4}$$

The merged CS's having a score higher than a threshold Ub_{tbl} will be selected for inclusion in the schema. in which $IR_k(cs_i)$ is the indirect-referenced score of cs_i after k iterations, $ref(cs_j, cs_i)$ is the number of references from cs_j to cs_i, $freq(cs_j)$ is the frequency of cs_j, and $refsTo(cs_i)$ is the total number of direct references to cs_i.

Specifically, the number of iterations k is set the same as the diameter of the CS graph. It is because, with that value, after k iterations, the IR_k score of each CS will get computed from all the CS's. To compute the diameter of the graph, we implemented a fast and simple algorithm described by [57].

Minimizing the number of infrequent properties. A final step of schema filtering considers eliminating CS properties, which as column in a relational table would have many NULL values. If the property coverage ratio (see Equation 3.5) is less than the infrequent threshold τ_{inf}, that property is infrequent and it gets removed from the CS.

$$coverageRatio(p, cs) = \frac{freq(p, cs)}{freq(cs)} \tag{3.5}$$

3.2.5 Instance Filtering

The output after labeling, merging, and schema filtering is a compact relational emergent schema. In the instance filtering phase, all RDF triples are visited again, and either stored in relational tables (typically $> 90\%$ of the triples, which we consider *regular*), or (the remainder) separately in a PSO table. Hence, our final result is a set of relational tables with foreign keys between them, and a single triple table in PSO format. In principle, the regular triples are those belonging to a merged CS (that survived schema filtering). However, not all such triples are considered regular in the end, as we perform three types of *instance filtering*, described next.

Maximizing type homogeneity. Literal object values corresponding to each property in a CS can have several different types e.g., number, string, dateTime. The relational model can only store a single type in each column, so in case of type diversity, a relational system like MonetDB must use multiple columns for a single property. They contain the type-cast value of the literal, if possible, and NULL otherwise. The number of columns needed for representing the data from a cs_i hence is $\sum_{p \in cs_i} \#ofTypes(p)$. This number can be large just due to a few triples having the wrong type (dirty data). To minimize the number of such columns, for each property, we filter out all the infrequent literal types (types that appear in $< \tau_{inf}$ percent of all instances). All triples of class instances with infrequent types are moved to the PSO table.

Parameter	Default	Description
Ub_{tbl}	1000	number of tables upper bound
$\min_t$	1000	minimum table size
τ_{inf}	5%	infrequent threshold

Table 3.3: Emergent Relational Schema Detection Parameters

Relationship Filtering. We further filter out *infrequent* or "dirty" relationships between classes. A relationship between cs_i and cs_j is infrequent if the number of references from cs_i to cs_j is much smaller than the frequency of cs_i (i.e., less than τ_{inf} percent of the CS's frequency). A relationship is considered dirty if the majority but not all the object values of the referring class (e.g., cs_i) refer to the instances of the referred class (cs_j). In the former case, we simply remove the relationship information between two classes. In the latter case, the triples in cs_i that do not refer to cs_j will be filtered out (placed in the separate PSO table).

We note that in the general case of n-m cardinality relationships, the relational model requires to create a separate *mapping table* that holds just the keys of both relations. However, in case one of the sides is $0 \ldots 1$, this is generally avoided by attaching a FK column to the table representing the other side. We try to optimize for this, by observing whether a multi-valued relationship is infrequent ($< \tau_{inf}$). If so, we remove the excess relationship to the separate PSO table, such that all remaining subjects in the class have maximally one relationship destination. Finally, if almost all instances of one class have *exactly* one match in the other class but a few ($< \tau_{inf}$) have none, we move *all* triples with that subject to the separate PSO table to preserve the exact n-1 cardinality (which keeps the FK column non-NULLable).

Multi-valued attributes. The same subject may have 0, 1 or even multiple triples with the same property, which in our schema leads to an attribute with cardinality > 1. While this is allowed in UML class diagrams, direct storage of such values is not possible in relational databases. Practitioners handle this by creating a separate table that contains the primary key (subject OID) and the value (which given literal type diversity may be multiple columns). The RDF bulk-loader of MonetDB does this, but only creates such separate storage if really necessary. That is, we analyze the mean number of object values ($meanp$) per property. If the $meanp$ of a property p is not much greater than 1 (e.g., less than (1+τ_{inf}/100), we consider p as a single-valued property and only keep the first value of that property in each tuple while moving all the triples with other object values of this property to the non-structured part of the RDF dataset. Otherwise, we will add a table for storing all the object values of each multi-valued property.

$$\begin{aligned} meanp(p) &= \sum p(k) \times k \\ \text{where } p(k) &= \frac{\#times\ p\ has\ k\ object\ values}{freq(p)} \end{aligned} \tag{3.6}$$

3.2.6 Parameter Tuning

An important question that we needed to address is how the various parameters guiding the recognition process should be set. Choosing improper parameters might result in a "bad" final schema with e.g., small data coverage, lots of NULLs, etc. Further, since each input dataset can have different characteristics, it would be unfeasible to find a fixed parameter set that works optimally for all datasets.

The most dataset sensitive parameter we found to be the τ_{sim}, used in labeling while matching ontologies using discriminative properties, as well as in the CS merging Rule 4 that determines up until which point merging should continue. It is a control on the strictness of finding equivalences between structures and ontologies, at 1 it is very strict while at 0 it is very lax. We evaluate the quality of the relational schema on two dimensions, namely (i) the number of tables (compactness of the schema) and (ii) its *precision*, which is the number of NOT-NULL cells, $fill(t)$, divided by the total number of cells, $cap(t)$, in all tables, as in Equation 3.7. There is a clear trade-off between having a compact schema and higher precision, depending on τ_{sim}.

Our auto-tuned algorithm iteratively re-runs the labeling and merging steps with different values of τ_{sim}. In each run, we measure the number of tables and the precision; we also compute a delta of these between successive values of τ_{sim}. In Equation 3.8, k is the total number of runs; nT_i ($nTnom_i$) and $prec_i$ ($prNom_i$) are the (normalized) number of tables and the schema precision at the i^{th} run; $nTdelta_i$ and $prDelta_i$ are the relative change in the normalized number of tables and the precision at the i^{th} run, respectively. We use the lowest value of $\tau_{sim} > 0$ where $nTdelta_i > prDelta_i$.

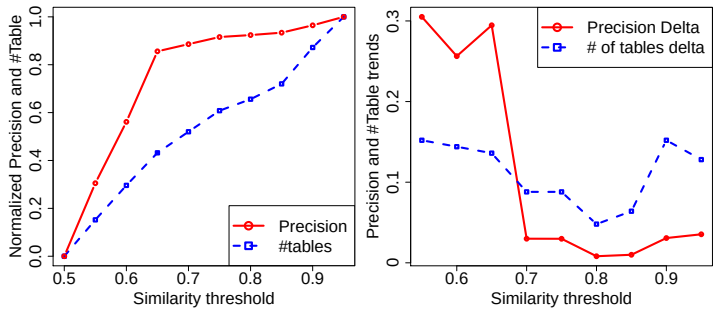
$$prec = \frac{\sum_t fill(t)}{\sum_t cap(t)} \tag{3.7}$$

$$\begin{aligned} nTnom_i = \frac{nT_i - nT_1}{nT_k - nT_1} \quad prNom_i = \frac{prec_i - prec_1}{prec_k - prec_1} \\ nTdelta_i = nTnom_i - nTnom_{i-1} \\ prDelta_i = prNom_i - prNom_{i-1} \end{aligned} \tag{3.8}$$

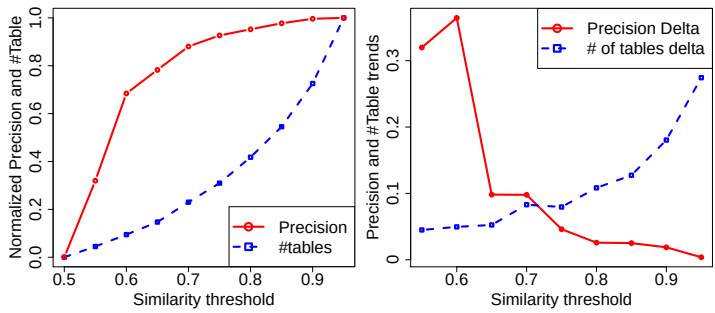
The left of Figure 3.12 shows normalized nT_i and $prec_i$ for WebData Commons as a function of τ_{sim} in steps of 0.05, while the right side shows the deltas between steps. Auto-tuning chooses the cross-over point of the deltas (τ_sim=0.7).

3.3 Experimental Evaluation

Metrics. We propose several metrics for evaluating the quality of the emergent schema. These metrics rely on the fact that a structure is considered to be good if it is *compact* (few and thin tables), *precise* (few NULLs) and has *large coverage* (few triples that have to be moved to separate PSO storage). Given an RDF dataset R and its total number of triples $|R|$, the first performance metric, C, is the percentage of



(a) DBpedia.



(b) WebData.

Figure 3.12: Left: τ_{sim} steps on X, #Tables&Precision on Y. Right: step deltas, auto-tuning selects cross-over

input triples covered by the schema:

$$C = \frac{\sum_1^n cov(t_i)}{|R|} \tag{3.9}$$

Each class in the structure is physically stored in a separate relational table. We define worth $w(t_i)$ of table t_i as:

$$w(t_i) = \overbrace{\frac{cov(t_i)}{\sum_1^n cov(t_i)}}^{(I)} \times \overbrace{\left(prec(t_i) + \frac{ref(t_i)}{\sum_1^n ref(t_i)}\right)}^{(II)} \tag{3.10}$$
$$where\ \ prec(t_i) = \frac{fill(t_i)}{cap(t_i)}$$

The precision $prec(t_i)$ of the table t_i is the fraction of non-NULL values in table ti, $cov(t_i)$ is the number of RDF triples stored in t_i; n is the number of tables and

labels	WebData.	DBpedia
top 3	3.6	3.8
final	4.1	4.6

Table 3.4: Human survey results on Likert scale

$ref(t_i)$ is the number of FK's referring to t_i. Here, (II) sums the precision and the relative importance of the table considering the relationships between tables, while (I) denotes the contribution of the table for the coverage of the schema. As the schema is only compact if n is small, the quality of the explored structure, Q, is defined as: $Q = \frac{\sum_1^n w(t_i)}{n}$.

3.3.1 Experimental Results

Labeling evaluation. We presented the emergent schemas of the DBpedia and WebDataCommons datasets to 19 humans and asked them to rate the labels.On a 5-point Likert scale from 1 (bad) to 5 (excellent) label quality, the top 3 labels of each table were scored by at least 3 persons. As shown in Table 3.4, the top 3 label candidates received an average rating of 3.6 for WebDataCommons and 3.8 for the DBpedia dataset. The finally chosen labels (one among the top 3) got better scores (4.1 and 4.6, respectively). We therefore conclude that the ordering of label candidates created by our algorithms produces encouraging results, as the chosen labels get higher ratings than the other candidates. Furthermore, our evaluation shows that 78% (WebDataCommons) and 90% (DBpedia) of the labels are rated with 4 points or better, hence are considered "good" labels by the users. The emergent relational schemas for the nine datasets we tested are too large to include in this chapter, Figure 3.13 shows EuroStat, one of the simpler schemas.[13]

Merging/Filtering performance. Figure 3.14 and Table 3.5 show the performance of the proposed merging algorithms and the filtering techniques for detecting a compact relational emergent schema with high coverage. According to Figure 3.14, the metric Q of the explored structure, except for WebDataCommons, always increases after the merging and filtering steps. For WebDataCommons, the value of Q decreases when merging CS's using rule S1. This stems from the fact that in WebDataCommons dataset each CS describing a certain entity such as `Website` may have many additional properties describing application attached to the website, and even use various properties for the same attribute (e.g., `ogp.me/ns#url`, `opengraphprotocol.org/schema/url`, `rdf.data-vocabulary.org/#url` for the website's URL), and thus, their merged CS's may contain properties with lots of NULLs values, causing the decrease of the metric Q. Nevertheless, the filtering step, by refining infrequent properties in the explored structure, can help addressing this issue and significantly increases the score of the metric Q. Comparing to the basic structure, the final schema of each experimental dataset is several orders of magnitude better in this metric.

[13] See `www.cwi.nl/~boncz/emergent` for the other datasets.

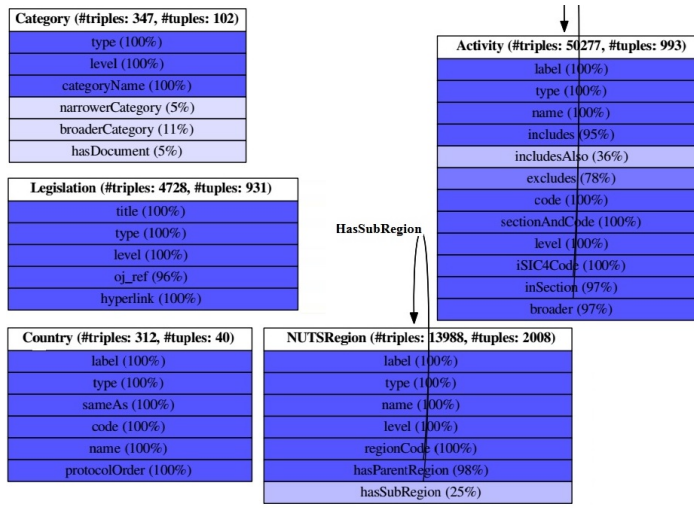


Figure 3.13: Final emergent schema for EuroStat – the lighter a column, the more NULLs (percentage in parentheses).

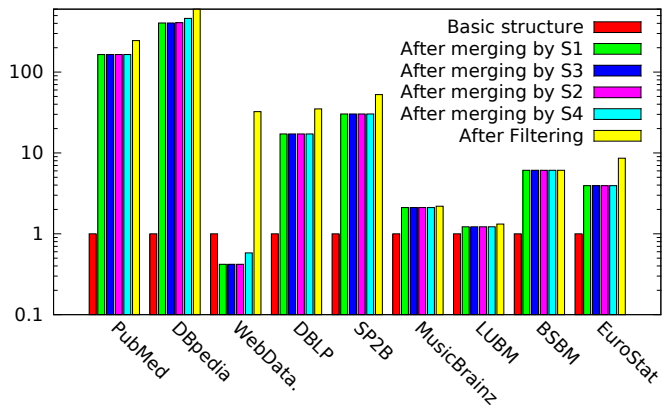


Figure 3.14: Schema quality Q during merging & filtering

Table 3.5 also shows that after the schema filtering, the final schema in all cases achieves very high coverage. We see that synthetic RDF benchmark data (BSBM, SP2B, LUBM) is fully relational, and also all dataset with non-RDF roots (PubMed, MusicBrainz, EuroStat) get $> 99\%$ coverage. Most surprisingly, the RDFa data that dominates WebDataCommons and even DBpedia is more than 90% regular. Further, a non-complete manual inspection of the $< 10\%$ irregular triples in these datasets appeared to show mainly mistyped properties, so our suspicion is that much of this irregularity is in fact data "dirtiness".

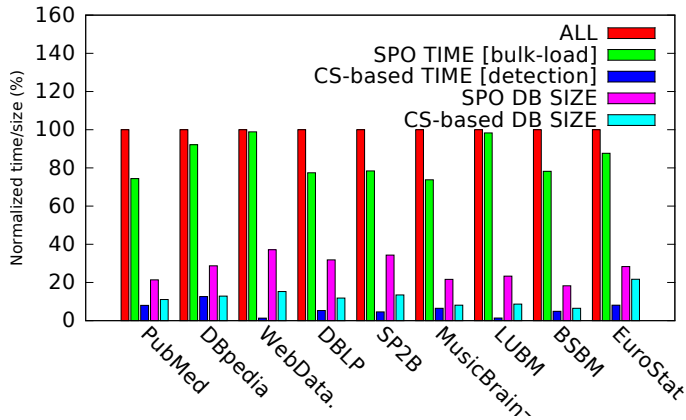


Figure 3.15: Building time & database size for single triple table (SPO) and reorganized relational tables (CS-based) (normalized by bulk-load time and database size for all six S,P,O table permutations (ALL)).

	Number of tables			Coverage – Metric C (%)		
Datasets	before merging	after merging	remove small tables	remove small tables	prune infreq. prop.	final schema
LUBM	17	13	12	100	100	**1**00.00
BSBM	49	8	8	100	100	**1**00.00
SP2B	554	13	10	99.99	99.65	**9**9.65
MusicBrainz	27	12	12	100	99.9	**9**9.60
EuroStat	44	10	5	99.73	99.53	**9**9.53
DBLP	249	9	6	100	99.68	**9**9.60
PubMed	3340	14	12	100	99.75	**9**9.73
WebData.	13354	3000	253	98.17	94.37	**9**2.79
DBpedia	439629	542	234	99.12	96.68	**9**5.82

Table 3.5: #tables and metric C after merging & filtering

Computational cost. Figure 3.15 shows that the time for detecting the emerging schema is negligible compared to bulk-loading time for building a single SPO table as well as building all the six permutations of `S`, `P`, `O` (marked [ALL]). Thus, MonetDB integrates emergent schema detection into its RDF bulk-loading without recognizable delay.

Compression. Figure 3.15 shows that the database size stored using relational tables can be two times smaller than the database size of a single SPO triple table. The reason is that in the relational representation the `S` and `P` columns effectively get compressed away, only the `O` columns remain.

Query processing. As a proof that the recognized emergent schema can be easily integrated and boost the performance of existing RDF stores, we report on the ef-

Queries	Cold		Hot		Opt.	
	Virt-Quad	Virt-CS	Virt-Quad	Virt-CS	Virt-Quad	Virt-CS
Q2	11567	2485	7	6	4.2	3.5
Q3	4210	2965	53	9	40.2	5.4
Q5	3842	2130	1350	712	18.6	4.2
Q7	19401	11642	9	6	5.3	4.5
Q8	14644	5370	9	5	4.4	3.3

Table 3.6: Query time (msecs) w/wo the recognized schema

(Cold: First query runtime after re-starting the server; Hot: Run the query 3 times and get the last runtime; Opt.: Query optimization time)

fort at OpenLink to integrate emergent relational schema technology in one of the state-of-the-art RDF stores, Virtuoso [87][14] It was a few months work to integrate Characteristic Set based storage, query execution and query optimization in Virtuoso. We compare a classic Virtuoso RDF quad table (Virt-Quad) and this CS-based implementation (Virt-CS) on the BSBM benchmark at 10 billion triples scale.

The Virtuoso CS implementation makes a column-wise table for each entity type with all single-valued or sparse properties as columns. The primary key of each CS table is `SG` (Subject-Graph). Exceptions, i.e. "irregular" triples as defined in Section 3.2.5, are represented by standard rows in the RDF quad table (i.e. default RDF storage). The implementation recognizes groups of triple patterns sharing a subject and having properties associated with CS's and treats these as a table in query optimization. At run time, the abstract table, which may match multiple CS's, is expanded into a per-CS plan for each CS. The per-CS plan is in concept a full outer join of the CS table and the exceptions for each property. A Bloom filter on `S` is used to indicate the possibility of exceptions for a given `P` in a CS, so the quads are most often not accessed at all.

Multi-valued and very sparse properties are represented as quads. Some `P`'s and common `O` values e.g, common RDF types, are omitted from Virtuoso's POSG index and a scan of the CS is used instead. Rare values may still exist in the POSG index. The just-in-time plan generation may alternate between scans of CS + PSOG and POSG index lookups depending on values seen at run time.

The experimental results in Table 3.6 show that exploiting the emergent relational schema even in this very preliminary implementation already improves the performance of Virtuoso on a number of BSBM Explore queries by up to a factor of 5.8 (Q3, Hot run). Note that the Cold run is much slower comparing to the Hot run as most of the time goes in the statistics gathering, not in the execution. We see less gain from CS's in other queries, e.g., Q5, since the first condition on the BSBM products (on a range of numeric property) is selective, so the other columns of the CS (or self-joins to RDF quads) are done on a small fraction of the subjects of the first range check. In Q3 more single-valued properties are accessed per subject, resulting in much more gain.

[14] https://github.com/v7fasttrack/virtuoso-opensource

By collapsing multiple triple patterns into a single abstract CS table, query optimization gets a plan search space of the same order as for the equivalent SQL. For Q3, the compilation time drops from 40.2 msecs to 5.4 msecs when using the recognized schema. In many RDF applications, e.g. Open PHACTS[15], query optimization time dominates and can run into the tens of seconds. Due to the extreme search space resulting from triple patterns, there are often ad hoc restrictions on plans, e.g. no hash join or no joins on hash build sides. With CS, a more thorough search of the plan space becomes again practicable and we expect qualitatively better plans to result.

We also note that the space consumption in allocated pages is 59 GB for Virt-CS and 116 GB for Virt-Quad. This comes in part from no indexing of low-cardinality O's and from not storing IRI string to ID mapping for IRI's which consist of a fixed text plus a numeric id. The Virt-CS is nearly two times more space-efficient, which directly translates to lower cost of deployment. The larger the data, the more important space efficiency becomes.

3.4 Related Work

We note that previous work has already proposed building relational-resembling RDF stores [184, 76, 128, 181, 62]. However, these proposals either demand the presence of an all-explaining ontology (which then gets remapped to relational tables), or ask the database system administrator to create and maintain "property tables" explicitly. Our approach, in contrast, does not require any form of explicit schema ingestion. Second, since these approaches just use the structure internally in the SPARQL engine to make things faster, they do not address the challenge of making the schema understandable to humans (compactness, finding short aliases). For the latter, a related line of work is creating summaries of the graph structure to aid query formulation [71], yet these do not focus on making RDF database systems faster, and typically require a cluster to compute, whereas our approach is cheap and can piggyback on RDF bulk-loading. Related to the automatic structure exploration from data is work on ontology mining [129] which discovers ontologies from unstructured text on the web. In our approach, we recognize the emergent structure in RDF data (e.g., mixing of ontologies), and do not change the semantics, and focus on providing a relational view of it.

Frequent itemset mining, which has been studied in many data mining papers [36, 70, 98], is equivalent to the basic CS recognition, originally proposed by [149]. We use this technique but go beyond that by finding a schema graph with cross-CS relationships, and we employ a host of techniques to make this schema graph compact and human-friendly (finding labels).

A recent study on the structure refinement for the RDF data, [41] proposed an integer linear programming (ILP)-based algorithm which allows an RDF dataset being partitioned into a number of "sorts" where each sort satisfies a predefined structured-ness fitting threshold. This approach, relying mainly on the similarity and correlation between the properties of sorts, may merge subjects describing

[15] http://www.openphacts.org/

unrelated entities but having many common properties into a single sort (as also shown in their experiment with Drug Company and Sultan), while our solution only merges related CS's together by exploiting the discriminating properties and the availability of the semantics/ontologies information. Besides, no relationship exploration as well as labeling for the sorts are considered in this work, and thus, no relational schema is recognized.

Consulting external resources for entity labeling is suggested by [180] in the context of table data reconstruction as well as by [176] a study on labeling hierarchical clusters. The former study shows that column names and table names usually cannot be found in table data itself. To reconstruct HTML table data they therefore rely on an external database with hyponym information. The latter study also mentions that documents often do not contain self-descriptive terms. To overcome this, they suggest using "anchor texts" as an additional resource in their document labeling task. Anchor texts are pieces of text on and next to hyperlinks to a specific document. The relational equivalent of anchor texts are names of foreign key relationships. In our case we rely on property names that refer to other tables for name suggestions, supplemented by ontology information when present.

3.5 Conclusions

In this chapter, we introduced the notion of – and demonstrated practical techniques for – discovering an *emergent* relational schema in RDF datasets, that recovers a compact and precise relational schema with high coverage and useful labels as alias for all machine-readable URIs (which it preserves). The functional benefit of an emergent relational schema for RDF datasets is both in giving users better understanding of the structure of an RDF dataset, while also allowing the often $> 90\%$ of regular triples to be queried from existing SQL applications, which still dominate the IT industry. Our MonetDB RDF bulk loader enables this. We think that this also provides impetus to make SQL more semantic, e.g. stimulating usage of URIs in SQL metadata.

The emergent relational schema can also be used under the cover of an SPARQL engine as a new storage approach, where the 90% regular triples are stored in tabular structures and the rest in SPO format. We think that the knowledge of an emergent schema gives SPARQL engines just what they need to close the performance gap with SQL systems. This we demonstrated in Virtuoso, with gains both in compression, query execution and query optimization. The tabular structure opens up many opportunities to improve physical access patterns using (partial) clustered indexes, zone maps, table partitioning and even database cracking [114].

Looking ahead, the prospect of people supporting SQL applications on top of RDF data raises many new questions. Users will desire to tweak a found emergent schema by hand, e.g. by manually improving some labels. We propose making a found emergent schema explicit using a vocabulary, and researching techniques to control schema evolution to preserve schema stability while the emergent schema adapts over time to changes in the underlying RDF datasets.

Chapter 4

Exploiting Emergent Schemas to make RDF systems more efficient

We build on our earlier finding that more than 95% of the triples in actual RDF triple graphs have a remarkably tabular structure, whose schema does not necessarily follow from explicit metadata such as ontologies, but for which an RDF store can automatically derive by looking at the data using so-called "emergent schema" detection techniques. In this chapter we investigate how computers and in particular RDF stores can take advantage from this emergent schema to more compactly store RDF data and more efficiently optimize and execute SPARQL queries. To this end, we contribute techniques for efficient emergent schema aware RDF storage and new query operator algorithms for emergent schema aware scans and joins. In all, these techniques allow RDF schema processors fully catch up with relational database techniques in terms of rich physical database design options and efficiency, without requiring a rigid upfront schema structure definition.

4.1 Emergent Schema Introduction

In previous chapter, we introduced *emergent schemas*: finding that >95% of triples in all LOD datasets we tested, including noisy data such as WebData Commons and DBpedia, conform to a small relational tabular schema. We provided techniques to automatically and at little computational cost find this "emergent" schema, and also to give the found columns, tables, and "foreign key" relationships between them short *human-readable labels*. This label-finding, and in fact the whole process of emergent schema detection, exploits not only value distributions and connection patterns between the triples, but also additional clues provided by RDF ontologies and vocabularies.

A significant insight from Chapter 3 is that relational and semantic practitioners give different meanings to the word "schema". It is thus a misfortune that these two communities are often distinguished from each other by their different attitude to this ambiguous concept of "schema" – the semantic approach supposedly requiring no upfront schema ("schema-last") as opposed to relational databases only working with a rigid upfront schema ("schema-first").

Semantic schemas, primarily ontologies and vocabularies, aim at modeling a knowledge universe in order to allow diverse current and future users to denote these concepts in a universally understood way in many different contexts. Relational database schemas, on the other hand, model the structure of one particular dataset (i.e., a database), and are not designed with a purpose of re-use in different contexts. Both purposes are useful: relational database systems would be easier to integrate with each other if the semantics of a table, a column and even individual primary key values (URIs) would be well-defined and exchangeable. Semantic data applications would benefit from knowledge of the actual patterns of co-occurring triples in the LOD dataset one tries to query, e.g. allowing users to more easily formulate SPARQL queries with a non-empty result (this often results from using a non-occurring property in a triple pattern).

In [160], we observed *partial* and *mixed* usage of ontology classes across LOD datasets: even if there is an ontology closely related to the data, only a small part of its class attributes actually occur as triple properties (partial use), and typically many of the occurring attributes come from different ontologies (mixed use). DBpedia on average populates $<30\%$ of the class attributes it defines [160], and each actually occurring class contains attributes imported from no less than 7 other ontologies on average. This is not necessarily bad design, rather good re-use (e.g. foaf), but it underlines the point that any single ontology class is a poor descriptor of the actual structure of the data (i.e., a "relational" schema). Emergent schemas are helpful for human RDF users, but in this chapter, we investigate how RDF stores can exploit emergent schemas for efficiency.

We address three important problems faced by RDF stores. The first and foremost problem is the high execution cost resulting from the large amount of self-joins that the typical SPARQL processor (based on some form of triple table storage) must perform: one join per additional triple pattern in the query. It has been noted [99] that SPARQL queries very often contain star-patterns (triple patterns that share a common subject variable), and if the properties of the patterns in these stars reference attributes from the same "table", the equivalent relational query can be solved with a table scan, not requiring *any* join. Our work achieves the same reduction of the amount of joins for SPARQL.

The second problem we solve is the low quality of SPARQL query optimization. Query optimization complexity is *exponential* in the amount of joins [179]. In queries with more than 12 joins or so, optimizers cannot analyze the full search space anymore, potentially missing the best plan. Note that SPARQL query plans typically have F times more joins than equivalent SQL plans. Here F is the average size of a star pattern[1]. This leads to a 3^F times larger search space. Additionally, query optimizers depend on cost models for comparing the quality of query plan candidates, and these cost models assume independence of (join) predicates. In case of star patterns on "tables", however, the selectivity of the predicates is heavily correlated (e.g. subjects that have an `ISBN` property, typically instances of the class `Book`, have a much higher join hit ratio with `AuthoredBy` triples than the

[1] A query of X stars has $X \times F$ triple patterns, so needs $P_1 = X \times F - 1$ joins. When each star is collapsed into one tablescan, just $P_2 = (X - 1)$ joins remain: $\frac{P_1}{P_2} \geq F$ times.

independence assumption would lead to predict) which means that the cost model is often wrong. Taken together, this causes the quality of SPARQL query optimization to be significantly lower than in SQL. Our work eliminates many joins, making query optimization exponentially easier, and eliminates the biggest source of correlations that disturb cost modeling (joins between attributes from the same table).

The third problem we address is that mission-critical applications that depend on database performance can be optimized by database administrators using a plethora of physical design options in relational systems, yet RDF system administrators lack all of this. A simple example are *clustered indexes* that store a table with many attributes in the value order of one or more sort key attributes. For instance, in a data warehouse one may store sales records ordered by Region first and ProductType second – since this accelerates queries that select on a particular product or region. Please note that not only the Region and ProductType properties are stored in this order, but *all* attributes of the sales table, which are typically retrieved together in queries (i.e. via a star pattern). A similar relational physical design optimization is table-partitioning or even database cracking [115]. Up until this chapter, one cannot even think of the RDF equivalent of these, as table clustering and partitioning implies an understanding of the structure of an RDF graph. Emergent schemas allow to leave the "pile of triples" quagmire, so one can enter structured data management territory where advanced physical design techniques become applicable.

In all, we believe our work brings RDF datastores on par with SQL stores in terms of performance, without losing any of the flexibility offered by the RDF model, thus without introducing a need to create upfront or enforce subsequently any explicit relational schema.

4.2 Emergent Schema Aware RDF Storage

The original emergent schema work allows to store and query RDF data with SQL systems, but in that case the SQL query answers account for only those "regular" triples that fit in the relational tables. In this work, our target is to answer SPARQL queries over 100% of the triples correctly, but still improve the efficiency of SPARQL systems by exploiting the emergent schema.

RDF systems store triple tables T in multiple orders of Subject (S), Property (P) and Object (O), among which typically T_{PSO} ("column-wise"), T_{SPO} ("row-wise") and either T_{OSP} or T_{OPS} ("value-indexed") – or even all permutations.[2]

In our proposal, RDF systems storage should become emergent schema aware by only changing the T_{PSO} representation. Instead of having a single T_{PSO} triple table, it gets stored as a set of wide relational tables in a *column*-store – we use MonetDB here. These tables represent only the regular triples, the remaining $< 5\%$ of "exception" triples that do not fit the schema (or were updated recently) remain in a smaller PSO table T_{pso}. Thus, T_{PSO} is replaced by the union of a smaller T_{pso} table and a set of relational tables.

[2]To support named RDF graphs, the triples are usually extended to quads. Our approach trivially extends to that but we discuss triple storage here for brevity.

Relational storage of triple data has been proposed before (e.g. property tables [184]), though these prior approaches advocated an explicit and human-controlled mapping to a relational schema, rather than a transparent, adaptive and automatic approach, as we do. While such relational RDF approaches have performance advantages, they remained vulnerable in case SPARQL queries *do not* consist mainly of star patterns and in particular when they have triple patterns where the P is a variable. This would mean that many, if not all, relational tables could contribute to a query result, leading to huge generated SQL queries which bring the underlying SQL technology to its knees.

Our proposal hides relational storage behind T_{PSO}, and has as advantage that SPARQL query execution can always fall back on existing mechanisms – typically MergeJoins between scans of T_{SPO}, T_{PSO} and T_{OPS}. Our approach at no loss of flexibility, just makes T_{PSO} storage more compact as we will discuss here, and creates opportunities for better handling of star patterns, both in query optimization and query execution, as discussed in the following sections.

Formal definition. Given the RDF triple dataset $\Delta = \{t|t = (t_S, t_P, t_O)\}$, an emergent schema $(\Delta, \mathcal{E}, \mu)$ specifies the set $\mathcal{E}$ of emergent tables T_k, and mapping μ from triples in Δ to emergent tables in $\mathcal{E}$. A common idea we apply is rather than storing URIs as some kind of string, to represent them as an OID (object identifier) – in practice as a large 64-bit integer. The RDF system maintains a dictionary $\mathcal{D}$: $OID \rightarrow URI$ elsewhere. We use this $\mathcal{D}$ dictionary creatively, adapting it to the emergent schema.

Definition 8 *Emergent tables ($\mathcal{E} = \{T_1, ..\}$): Let s, p_1, p_2,..., p_n be subject and properties with associated data types OID and D_1, D_2, ..., D_n, then T_k = ($T_k.s$:OID, $T_k.p_1$:D_1, $T_k.p_2$:D_2, ..., $T_k.p_n$:D_n) is an emergent table where $T_k.p_j$ is a column corresponding to the property p_j and $T_k.s$ is the subject column.*

Definition 9 *Dense subject columns: $T_k.s$ consists of densely ascending numeric values β_k, .. $\beta_k + |T_k| - 1$, so s is something like an array index, and we denote $T_k[s].p$ as the cell of row s and column p. For each T_k its base OID $\beta_k = k * 2^{40}$. By choosing β_k to be sufficiently apart, in practice the values of column $T_i.s$ and $T_j.s$ never overlap when $i \neq j$.*[3]

Definition 10 *Triple-Table mapping ($\mu : \Delta \rightarrow \mathcal{E}$): For each table cell $T_k[s].p_j$ with non-NULL value o, $\exists(s, p_j, o) \in \Delta$ and $\mu(s, p_j, o) = T_k$. These triples we call "regular" triples. All other triples $t \in \Delta$ are called "exception" triples and $\mu(t) = T_{pso}$. In fact T_{pso} is exactly the collection of these exception triples.*

The emergent schema detection algorithm [160] assigns each subject to at most 1 emergent table – our storage exploits this by manipulating the URI dictionary $\mathcal{D}$ so that it gives dense numbers to all subjects s assigned to the same T_k.

[3] In our current implementation with 64-bit OIDs we thus can support up to 2^{16} emergent tables with each up to 2^{40}=1 trillion subjects, still leaving the highest 8 bits free, which are used for type information – see footnote 4.

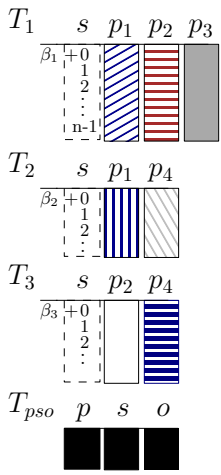


Figure 4.1: Columnar Storage of Emergent Tables T_k and exception table T_{pso}

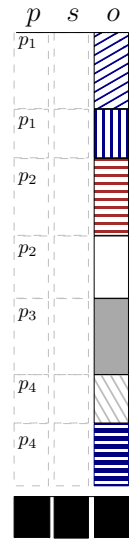


Figure 4.2: PSO as view $P_{PSO} \cup T_{pso}$

	Except%	Null%	Compr
Synthetic RDF datasets			
LUBM	0.0%	6.0%	1.8x
BSBM	0.0%	4.2%	2.5x
SP2Bench	0.4%	5.2%	2.0x
LDBC SNB	0.0%	12.2%	2.0x
RDF datasets with Relational Roots			
MusicBrz	0.4%	3.9%	2.2x
Eurostat	0.5%	3.8%	1.4x
DBLP	0.4%	12.6%	1.7x
PubMed	0.3%	15.3%	1.9x
Native RDF datasets			
WebData	7.5%	42.7%	1.4x
DBpedia	3.8%	32.2%	1.4x

Table 4.1: Exception percentage, NULL percentage and Compression Factor achieved by Emergent Table-aware PSO storage, over normal PSO storage.

Columnar relational storage. On the physical level of bytes stored on disk, columnar databases can be thought of as storing all data of one column consecutively. Column-wise data generally compresses better than row-wise data because data from the same distribution appears consecutively, and column-stores exploit this by having advanced data compression methods built-in in their storage and query execution infrastructure. In particular, the dense property of the columns $T_k.s$ will cause column-stores to compress it down to virtually nothing, using a combina-

tion of delta encoding (the difference between subsequent values is always 1) and run-length encoding (RLE), encoding these subsequent 1's in just a single run. Our evaluation platform MonetDB supports densely ascending OIDs natively with its VOID (virtual OID) type, that requires no storage.

Figure 4.1 shows an example of representing RDF triples using the emergent tables $\{T_1, T_2, T_3\}$ and the triple table of exception data T_{pso} (in black, below). We have drawn the subject columns $T_k.s$ transparent and with dotted lines to indicate that there is no physical storage needed for them.

For each individual property column $T_k.p_j$, we can define a triple table view $P_{j,k}$=(p_j,$T_k.s$,$T_k.o$), the first column being a constant value (p_j) which thanks to RLE compression requires negligible storage and the other two reusing storage from emergent table T_k. If we concatenate these views $P_{j,k}$ ordered by j and k, we obtain table $P_{PSO} = \cup_{j,k} P_{j,k}$. This P_{PSO} is shown in Figure 4.2. Note that P_{PSO} is simply a re-arrangement of the columns $T_k.p_j$. Thus, with emergent schema aware storage, one can always access the data P_{PSO} as if it were a PSO table at no additional cost.[4] In the following, we show this cost is actually less.

Space Usage Analysis. P_{PSO} storage is more efficient than PSO storage in an efficient columnar RDF store such as Virtuoso would be. Normally in a PSO table, the P is highly repetitive and will be compressed away. The S column is ascending, so delta-compression will apply. However, it would not be dense and it will take some storage ($\log_2(W)$ bits per triple, where W is the average gap width between successive s values[5]) – while a dense S column takes no storage.

Compressing-away the S column is only possible for the regular part P_{PSO}, whereas the exception triples in T_{pso} must fall back to normal PSO triple storage. However, the left table column of Figure 4.1 shows that the amount of exception triples is negligible anyway – it is almost 0 in synthetic RDF data (stemming from the LUBM, BSBM, SP2Bench and LDBC Social Network Benchmark), as well as in RDF data with relational roots (EuroStat, PubMed, DBLP, MusicBrainz), and is limited to $< 10\%$ in more noisy "native" RDF data (WebData Commons and DBpedia). A more serious threat to storage efficiency could be the NULL values that emergent tables introduce, which are table cells for which no triple exists. In the middle column we see that the first-generation RDF benchmarks (LUBM, BSBM, SP2Bench) ignore the issue of missing values. The more recent LDBC Social Network benchmark better models data with relational roots where this percentage is roughly 15%. Webdata Commons, which consists of crawled RDFa, has most NULL values (42 percent) and DBpedia roughly one third. We note that the percentage of NULLs is a consequence of the emergent table algorithm trying to create a compact schema that consists of relatively few tables. This process makes it merge initial tables of property-combinations into tables that store the union of those properties: less, wider, tables means more NULLs. If human understandability were not a goal of emergent table detection, parameters could be changed to let it generate more tables with less NULLs. Still, space saving is not really an argu-

[4] SQL-based SPARQL systems (MonetDB, Virtuoso) still allow SQL on T_k tables.

[5] W=$\frac{1}{n-1}\sum_1^{n-1}(s_{i+1} - s_i)$ where s_i is the subject OID at row i (table with n rows)

ment for doing so, as the rightmost table column of Figure 4.1 shows that emergent table storage is overall at least a factor 1.4 more compact than default PSO storage.

Query Processing Microbenchmark. While the emergent schema can be physically viewed as a compressed PSO representation, we now will argue that every use a RDF store will give to a PSO table can be supported at least as efficiently on emergent table aware storage.

Typically, the PSO table is used for three access patterns during SPARQL processing: i) Scanning all the triples of a particular property p (i.e., p is known), ii) Scanning with a particular property p and a range of object value (i.e., p is known + condition on o), and iii) Having a subset of S as the input for the scan on a certain p value (i.e., typically s is sorted, and the system performs a filtering MergeJoin). The first and the second access patterns can be processed on the emergent schema in the similar way as with the original PSO representation by using a UNION operator: $\sigma(pso, p, o) = \sigma(P_{PSO}, p, o) \cup \sigma(T_{pso}, p, o)$

The third access pattern, which is a JOIN with s candidate OIDs is very common in SPARQL queries with star patterns. We test two different cases: with and without of exceptions (i.e. T_{pso}).

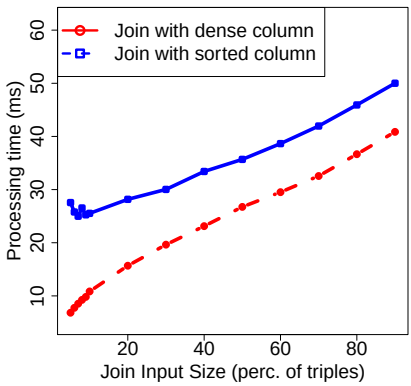


Figure 4.3: PSO join performance vs input size (no exceptions)

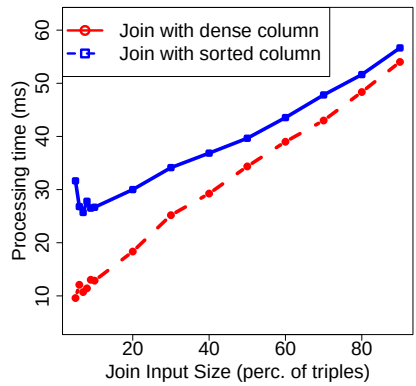


Figure 4.4: PSO join performance vs input size (with exceptions)

Without T_{pso}. In this case, the JOIN can be pushed through the P_{PSO} view and is simply the UNION of JOINs between the s candidates and dense $T_k.s$ columns in each emergent table T_k. MonetDB supports joins into VOID columns very efficiently, essentially this is sequential array lookup.

We conducted a micro-benchmark to compare the emergent schema aware performance with normal PSO access. It executes the JOIN between a set of $I.s$ input OIDs with two different $T_k.s$ columns: a dense column and a sorted (but non-dense) column; in both cases retrieving the $T_k.o$ object values. The benchmark data is extracted from the subjects corresponding to the `Offer` entities in BSBM benchmark, containing ≈5.7 million triples. Each JOIN is executed 10 times and the minimum

running time is recorded. Figure 4.3 shows that dense OID joins are 3 times faster on small inputs: array lookup is faster than MergeJoin.

With T_{pso}. Handling exception data requires merging the result produced by the JOIN between input ($I.s$) and the dense S column of emergent table $T_k.s$ with the result produced by the JOIN between $I.s$ and the exception table $T_{pso}.s$ – the latter requires an actual MergeJoin. We implemented an algorithm that performs both tasks simultaneously. In order to form the JOIN result between $I.s$ with both $T_k.s$ and $T_{pso}.s$ simultaneously, we modify the original MergeJoin algorithm by checking for each new index of $I.s$, whether the current element from $I.s$ belongs to the dense range of $T_k.s$.

We conducted another micro-benchmark using the same 5.7 million triples. The exception data is created by uniformly sampling 3% of the regular data (BSBM itself is perfectly tabular and has no exceptions). We note that 3% is already more than the average percentage of exception data in all our tested datasets. The list of input $I.s$ candidates is also generated by sampling from 5% to 90% of the regular data. Figure 4.4 shows that the performance of the JOIN operator on the emergent schema still outperforms that on the original PSO representation even though it needs to handle exception data.

The conclusion of this section is that emergent schema aware storage reduces space by 1.4 times, provides faster PSO access, and importantly hides the relational table storage from the SPARQL processor – such that query patterns that would be troublesome for property tables (e.g. unbound property variables) can still be executed without complication. We take further advantage of the emergent schema in many common query plans, as described next.

4.3 Emergent Schema Aware SPARQL Optimization

The core of each SPARQL query is a set of (s,p,o) triple patterns, in which s, p, o are either literal values or *variables*. Viewing each pattern as a property-labeled edge between a subject and object, these triples form a *SPARQL graph*. We *group* these triple patterns, where originally each triple pattern is a group of one.

Definition 11 *Star Pattern (ρ=($\$s, p_1, o_1$), ($\$s, p_2, o_2$), ...): A star pattern is a collection of* more than one *triple patterns from the query, that each have a* constant *property p_i and an* identical subject variable $\$s$.

To exploit the emergent schema, we identify star patterns in the query and at the query optimization, group query's triple patterns by each star. Joins are needed only between these triple pattern groups. Each group will be handled by one table scan subplan that uses a new "$RDFscan$" operator described further on. SPARQL query optimization then largely becomes a *join reordering* problem. The complexity of join reordering is exponential in the number of joins.

To show the effects on query optimization performance, we created a micro-benchmark that forms queries consisting of (small) stars of size F=4. The smallest query is a single star, followed by one with two stars that are connected by sharing the same variable for an object in the first star and the subject of the star star, etc

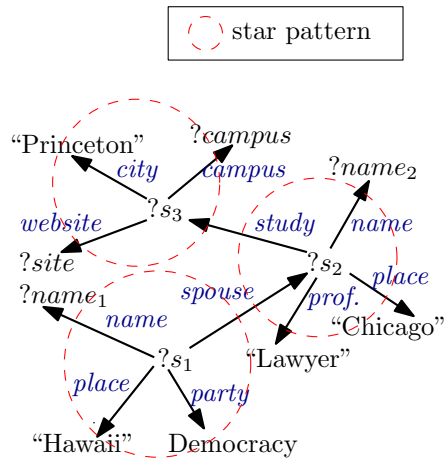


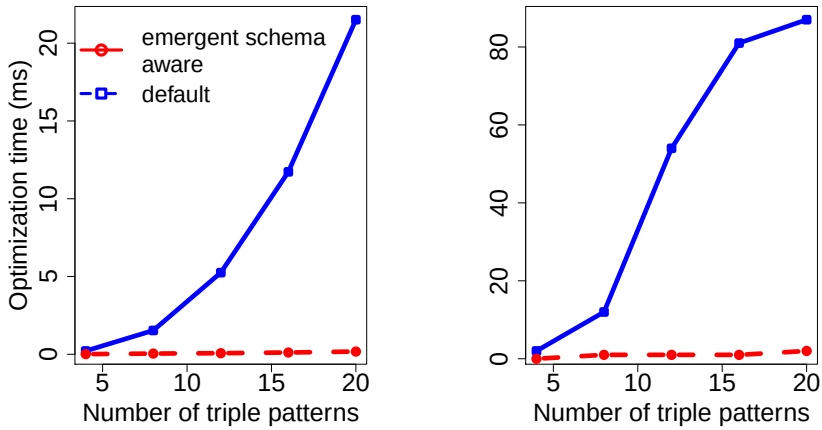
Figure 4.5: Example SPARQL graph with three star patterns

(hence queries have 4, 8, 12, 16 and 20 triple patterns). Our optimization identifies these stars, hence after grouping star patterns their join graph reduces to 0, 1, 2 and 3 joins respectively. We ran the resulting queries through MonetDB and Virtuoso and measured *only* query optimization time. Figure 4.6 shows that emergent schema aware SPARQL query optimization becomes orders of magnitude faster thanks to its simplification of the join ordering problem. The flattening Virtuoso default line beyond 15 patterns suggests that with large amount of joins, it stops to fully traverse the search space using cutoffs, introducing the risk of choosing a sub-optimal plan.

4.4 Emergent Schema Aware SPARQL Execution

The basic idea of emergent schema aware query execution is to handle a complete star pattern ρ with one relational table scan(T_i,[p_1,p_2,..]) on the emergent table T_i with whose properties p_i from ρ. Assuming a SQL-based SPARQL engine, as is the case in Virtuoso and MonetDB, it is crucial to rely on the existing relational table scan infrastructure, so that advanced relational access paths (clustered indexes, partitioned tables, cracking [115]) get seamlessly re-used.

In case of multiple emergent tables matching star pattern ρ, the *scan plan* (denoted ϑ_ρ) we generate consists of the UNION of such table scans. Details on generating ϑ_ρ can be found in the Appendix A. We note that in ϑ_ρ we also push-down certain relational operators (at least simple filters) below these UNIONs – a standard relational optimization. This push-down means that selections are executed before the UNIONs and optimized relational access methods can be used to e.g. perform IndexScans. Moreover, we should mention that OPTIONAL triple patterns in ρ are marked and can be ignored in the generated scans (because missing property values are already represented as NULL in the relational tables). Another detail is that on top of ϑ_ρ, we must introduce a Project operator to cast SQL literal types to a special SPARQL value type, that allows multiple literal types as well as URIs



(a) MonetDB (b) Virtuoso

Figure 4.6: Optimization time as a function of query size (#triple patterns)

to be present in one binding column.[6] Executing (pushed-down) filter operations while values are still SQL literals allows to avoid most casting effort, since after selections much fewer tuples remain.

This whole approach will still only create bindings for the "regular" triples. To generate the 100% correct SPARQL result, we introduce an operator called **RDFscan**, that produces *only* the missing bindings. The basic idea is to put another UNION on top of the scan plan ϑ_ρ that adds the RDFscan(ρ) bindings to the output stream, as shown in Figure 4.7. Unlike normal scans, we cannot push down filters below the RDFscan - hence these selections remain placed above it, at least until optimization 1 (see later).

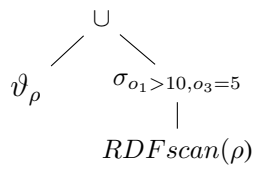


Figure 4.7: Query plan for handing exception

Generating Exception Bindings. Correctly generating all result bindings that SPARQL semantics expect is non-trivial, since the exception triples in T_{pso} when combined

[6] In our MonetDB implementation, the 64-bits OID that encodes (subject) URIs, also encodes literals by using other patterns in its highest 8 bits.

with *any* emergent table T_k (not only those covering ρ) could produce valid bindings. Consider the example SPARQL query, consisting of a single star pattern and two selections ($o_1 > 10$, $o_3 = 5$):

T_1

s	p_1	p_2	p_3
100	11	2	5
101	13	4	6
102	14		5
103	9	6	
104	15	8	5

T_2

s	p_1	p_3	p_4
200	11	7	1
201		5	2
202	13	9	3

T_{pso}

p	s	o
p_1	0	20
p_1	1	9
p_1	201	15
p_2	0	8
p_2	102	6
p_2	201	4
p_3	0	5
p_6	6	7

Result

s	o_1	o_2
100	11	2
104	15	8
0	20	8
102	14	6
201	15	4

Figure 4.8: Example RDF data and expected query result.

```
SELECT ?s ?o1 ?o2 WHERE {
   ?s  p1  ?o1 .
   ?s  p2  ?o2 .
   ?s  p3    5 .
   FILTER (?o1 > 10) }
```

Figure 4.8 shows the expected result of this query on an example data. (For a better view of the example, we assume s base OID of T_1, T_2 are 100, 200, respectively). In this result, the first two tuples come from the regular triples while the last three tuples is the combination of triples stored in T_{pso} table (i.e., in red color) with those stored in tables T_1 and T_2.

Basic approach. RDFscan returns all the bindings for a star pattern, in which each binding is generated by at least one irregular triple (the *missing* bindings). Formally, given a star pattern $\rho = \{(s, p_i, o_i), i = 1,..,k\}$, the RDF dataset Δ, the output of the RDFscan operator for this star pattern is defined as:

$$RDFscan(\rho) = \{(s, o_1, ..., o_k)\} | (s, p_i, o_i) \in \Delta \wedge (\exists i : (s, p_i, o_i) \in T_{pso}) \quad (4.1)$$

RDFscan generates the "exception" bindings in 2 steps:
Step 1: Get all possible bindings (s, o_1, ..., o_k) where each o_i stems from triple (s,p_i,o_i) $\in T_{pso}$ (for those p_i from ρ), or o_i=NULL if such a triple does not exist, with the constraint that at least one of the object values o_i is non-NULL.
Step 2: Merge each binding (s, o_1, ...,o_k) with the emergent table T_k corresponding to s ($\beta_k \leq s < \beta_k + |T_k|$) to produce output bindings for $RDFscan$.

Step 1 is implemented by first extracting the set E_i of all $\{(s, o_i)\}$ corresponding to each property p_i from the T_{pso}: $E_i = \sigma_{p=p_i}(T_{pso})$. Then, it returns the output, S_1, by performing a relational OuterJoin on s between all E_i. We note that, as T_{pso} table is sorted by p, extracting E_i from T_{pso} can be done with no cost by reading a slice of T_{pso} from the starting row of p_i and the ending row of p_i (the information

E_1

s	o_1
0	20
1	9
201	15

E_2

s	o_2
0	8
102	6
201	4

E_3

s	o_3
0	5

Output(S_1)

s	o_1	o_2	o_3
0	20	8	5
1	9		
102		6	
201	15	4	

Figure 4.9: Step 1 on example data & query

on starting, ending rows of each p in T_{pso} table is pre-loaded before any query processing). Furthermore, as for each p in T_{pso}, {(s,o)} are sorted according to s, E_i are also sorted by s. Thus, the full OuterJoin of all E_i can be efficiently done by using a multi-way sort merging algorithm. Figure 4.9 demonstrates Step 1 for the example query.

Algorithm 1 Merge-exception-regular algorithm

Input: S_1: Step 1 output
$lstP$: List of required properties
$\mathcal{E}$: Emergent tables
Output: T_{fin}: Merging results

```
1:  for each tuple t=(s, o1,...,ok) in S1 do
2:    id, r = getT_row(t.s) /*table & row id*/
3:    accept = true
4:    for each pi in lstP do
5:      if t.oi = null & E[id] [r].pi = null then
6:        accept = false
7:        Continue next tuple
8:      else
9:        store_cand(bind, t.oi, E[id] [r].pi)
10:   if accept = true then
11:     append(Tfin, bind)
```

Step 2 merges each tuple in S_1 with a tuple of the same s in the regular table in order to form the final output of $RDFscan$. For example, the 4th tuple of S_1 (201, 15, 4, $null$) merged with the 2nd tuple of T_2 (201, $null$, 5, 2) returns a valid binding (201, 15, 4) for the (s, o_1, o_2) of the example query. Figure 1 shows the detailed algorithm of Step 2. For each tuple t in S_1, it first extracts the corresponding regular table and row Id of the current $t.s$ from encoded information inside each s OID (Line 2). Then, for each property p_i, the algorithm will check whether there is any non-NULL object value appearing in either t (i.e., $t.o_i$) or the regular column p_i (i.e., $\mathcal{E}[id]\,[r].p_i$) (Line 5). If yes, the non-NULL value will be placed in the binding

for p_i (Line 9). Otherwise, if both of the values are NULL, there will be no valid binding for the current checking tuple t. Finally, the binding that has non-NULL object values for all non-optional properties will be appended to the output table T_{fin}.

Optimization 1: Selection push-down. Pushing selection predicates down in the query plan is an important query optimization technique to apply filters as early as possible. This technique can be applied to RDFscan when there is any selection predicate on the object values of the input star pattern (e.g., $o_1 > 10$, o_3=5 in the example query). Specifically, we push the selection predicates down in Step 1 of the RDFscan operator to reduce the size of each set E_i (i.e., $\sigma_{p=p_i}(T_{pso})$), accordingly returning a smaller output S_1 of this step. Formally, given λ_i being a selection predicate on the object o_i, the set E_i of {(s,o_i)} from T_{pso}) is computed as: E_i = $\sigma_{p=p_i,\lambda_i}(T_{pso})$. In the example query, E_1 = $\sigma_{p=p_1,o_1>10}(T_{pso})$. Figure 4.10 shows that the size of E_1 and the output S_1 are reduced after applying the selection push-down optimization, which thus improves the processing time of RDFscan operator.

E_1

s	o_1
0	20
201	15

Output(S_1)

s	o_1	o_2	o_3
0	20	8	5
102		6	
201	15	4	

Figure 4.10: Step 1 output with pushing down Selection predicates

Optimization 2: Early check for missing property. If a regular table T_k does not have p_i in its list of columns, to produce a valid binding by merging a tuple t of S_1 (i.e., output of Step 1) and T, the exception object value $t.o_i$ must be non-NULL. Thus, we can quickly check whether t is an invalid candidate without looking into the tuple from T_k by verifying whether t contains non-NULL object values for all missing columns of T_k. We implement this by modifying the algorithm for Step 2. Before considering the object values of all properties from both exception and regular data (Line 4), we first check exception object value $t.o_i$ of each missing property to prune the tuple if any $t.o_i$ is NULL. Then, we continue the original algorithm with the remaining properties.

Optimization 3: Prune non-matching tables. The exception table T_{pso} mostly contains triples whose subject was mapped to some emergent table. For example, the triple (201, p_2, 4) refers to the emergent table T_2 because $s \geq 200$=β_2. During the emergent schema exploration process [160] this triple was temporarily stored in the *initial* emergent table T_2', but was then moved to T_{pso} during the so-called "schema and instance filtering" step. This filtering moves not only triples but also whole columns from initial emergent tables to T_{pso}, in order to derive a compact and precise emergent schema. Assume column p_2 was removed from T_2 during schema filtering. We observe that before filtering, *all* triples (*regular + exception triples*) of subject s were part of the initial emergent table which means that had

a particular set of properties. Accordingly, if C is the set of columns of an initial emergent table T' and if C does not contain the set of properties in ρ, there cannot be a matching subject with all properties of ρ stemming from T' even with the help of T_{pso}. This observation can be exploited to prune all subject ranges corresponding to (initial) emergent tables that cannot have any matching for ρ from the pass over T_{pso}.

Specifically, we pre-store, for each emergent table, its set of columns C before schema and instance filtering was applied during emergent schema detection. Then, given the input star pattern ρ, the possible matching tables for ρ are those tables whose set of columns C contain all properties in ρ. Finally, Step 1 is optimized by removing from E_i all the triples that the subject does not refer to any of the matching tables.

4.5 Performance Evaluation

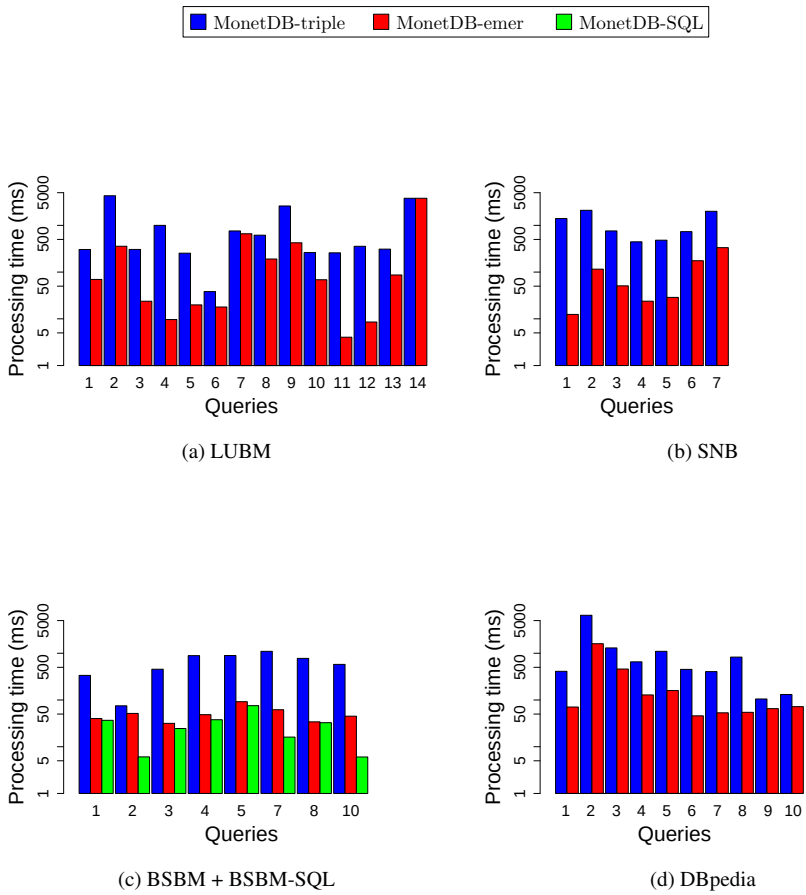
We tested with both synthetic and real RDF datasets BSBM [55], LUBM [100], LDBC-SNB[88] and DBpedia (DBPSB) [147]; and their respective query workloads. For BSBM, we also include its relational version, namely BSBM-SQL, in order to compare the performance of the RDF store against a SQL system (i.e., *MonetDB-SQL*). We used datasets of 100 million triples for LUBM and BSBM, and scale factor 3 (≈200 million triples) for LDBC-SNB. The experiments were conducted on a Linux 4.3 machine with Intel Core i7 3.4Ghz CPU and 16 GBytes RAM. All approaches are implemented in the RDF experimental branch of MonetDB.

Query workload. For BSBM, we use the SELECT queries from Explore workload (ignoring the queries with DESCRIBE and CONSTRUCT). For LUBM, we use its published queries and rewrite some queries (i.e., Q4, Q7, Q8, Q9, Q10, Q13) that requires certain ontology reasoning capabilities in order to account for the ontology rules and implicit relationships. For LDBC-SNB, we use its short read queries workload. DBPSB exploits the actual query logs of the DBpedia SPARQL endpoints to build a set of templates for the query workload. Using these templates, we create 10 non-empty result queries w.r.t DBpedia 3.9 dataset[7]. Table 4.2 show the features of tested DBpedia queries. In Figures 4.11, 4.12 and 4.13, X-axis holds query-numbers: 1 means Q1. For each benchmark query we run three times and record the last query execution (i.e., Hot run).

Emergent schema aware vs Triple-based RDF stores. We perform the benchmarks against two different approaches of MonetDB RDF store: the original triple-based store (*MonetDB-triple*) and the emergent schema-based store (*MonetDB-emer*).

Figure 4.11 shows the query processing time using two approaches over four benchmarks. For BSBM and LDBC-SNB, the emergent schema aware approach significantly outperforms the triple-based approach in all the queries, by up to two orders of magnitude faster (i.e., Q1 SNB). In a real workload such as DBpedia

[7]The detailed DBpedia queries can be found at goo.gl/RxzOmy



(a) LUBM (b) SNB

(c) BSBM + BSBM-SQL (d) DBpedia

Figure 4.11: Query processing time: Emergent schema-based vs triple-based

where there is significant amount of exception triples, our approach is still much faster (note: logscale) by up to more than an order of magnitude (Q8). We also note that multi-valued properties appear in most of DBpedia queries, and this is costly for the emergent schema aware approach as it requires additional MergeJoins to retrieve the object values. In Figure 4.11d, the best-performing query Q8 is the one having no multi-valued property.

For LUBM, a few queries (i.e. 7, 14) show comparable processing times for triple-table based and emergent schema aware query processing. The underlying reason is that each subject variable in these queries only contains one or two common properties (e.g., Q14 only contains one triple pattern with the properties `rdf:type`). Thus, the emergent schema aware approach will not improve the query execution time – however as the optimization does not trigger then it also does not degrade performance in absence of fruitful star patterns. For the queries having *discriminative* properties [160] in a star pattern (e.g., Q4, 11, 12), the emergent

Queries	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
Operators: **O**PTIONAL, **F**ILTER, **U**NION	-	-	O	-	U	F	-	F,U	O,F,U	O
Modifiers: **D**istinct, **L**imit, **O**RDER	D	D	D,L,O	D	D,L	D	D,L	-	-	D
# of triple pattern	4	5	5	3	10	3	6	4	6	7
# constraints on O?	1	0	1	1	2	2	1	4	2	0
Has multi-valued prop.?	√	√	√	√	√	√	√	-	√	-

Table 4.2: Properties of DBpedia queries

schema aware approach significantly outperforms the original triple-based version, by up to two order of magnitude (i.e, Q4).

Emergent schema-based RDF store vs RDBMS As shown in Figure 4.11c, the emergent schema aware SPARQL processing (*MonetDB-emer*) provides comparable performance on most queries (i.e., Q1,Q3,Q4,Q5,Q8) compared to *MonetDB-SQL*. In other queries (Queries 7,10), the emergent schema aware approach also significantly reduces the performance gap between SPARQL and SQL, from almost two orders of magnitude slower (*MonetDB-triple* vs *MonetDB-SQL*) to a factor of 3.8 (*MonetDB-emer* vs *MonetDB-SQL*).

RDFscan optimizations. Figure 4.12 shows the effects of each of the three described RDFscan optimization by running the DBpedia benchmark without using with each of them. All optimizations have positive effects, though in different queries, and the longer running queries show stronger effects. Selection push-down (Opt. 1) has most influence, while the early check in T_{pso} to see if it delivers missing properties has the least influence. Obviously, selection push-down does not give any performance boost when there is no constraint on the object variables in the queries (e.g., Query 2). For queries having constraints on the object variables, which are quite common in any query workload, it does speed up query processing by up to a factor of 24 (i.e., Q8).

Query optimization time. Figure 4.13 shows query optimization time on LDBC-SNB and DBPSB (due to lack of space, we omit similar results for BSBM and LUBM). For all queries, the emergent schema aware approach significantly lowers optimization time, by even up to two orders of magnitude (Q1 SNB) or a factor of 37 (Q7 DBPSB). Note also that due to the smaller plan space and strong reduction of join correlations, query optimization also qualitatively improves, a claim supported by its performance improvements across the board.

4.6 Related Work

Most state-of-the-art RDF systems store their data in triple- or quad-tables creating indexes on multiple orders of S,P,O[183, 87, 150, 177]. However, according to [99, 160], these approaches have several RDF data management problems including unpredictably bad query plans and low storage locality.

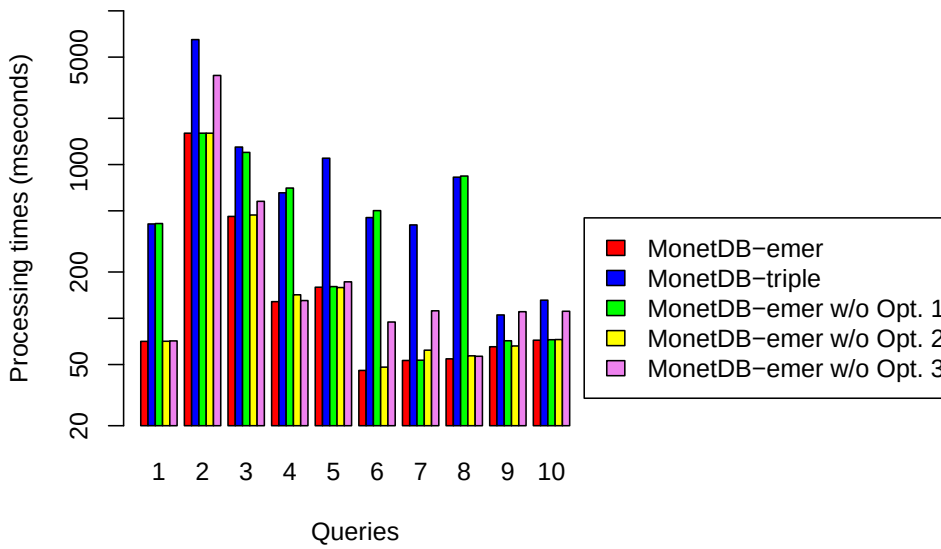
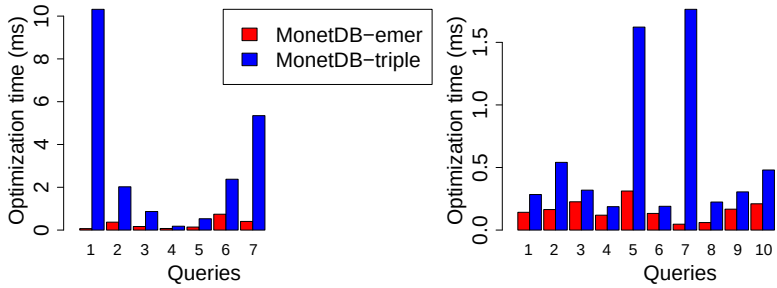


Figure 4.12: Query processing with/with-out optimizations

Structure-aware storage was first exploited in RDF stores with the“property tables” approach[184, 76, 128, 181]. However, early systems using this approach [184, 76] do not support automatic structure recognition, but rely on a database administrator doing the table modeling manually. Automatic recognition is introduced in some newer systems[128, 181, 134], however unlike emergent schemas these structures are not apt for human usage, nor did these papers research in depth integration with relational systems in terms of storage, access methods or query optimization. Recently, Bornea et al.[62] built an RDF store, DB2RDF, on top of a relational



(a) SNB (b) DBpedia

Figure 4.13: Optimization time: Emergent schema-based vs triple-based

system using hash functions to shred RDF data into multiple multi-column tables. This approach (nor any of the others) allows both SQL and SPARQL access to the same data, as emergent schemas do. Gubichev et al. [99] and Neumann et al.[149] use structure recognition to improve join ordering in SPARQL queries alone. Brodt et al. [66] proposed a new operator, called *Pivot Index Scan*, to efficient deliver attribute values for a resource (i.e., subject) with less joins using something similar to a SPO index – as such it does not recognize structure in RDF to leverage it on the physical level.

4.7 Conclusion

Emergent Schema detection is a recent technique that automatically analyzes the actual structure an RDF graph, and creates a compact relational schema that fits most of the data. We investigate here how these Emergent Schemas, beyond helping humans to understand a RDF dataset, can be used to make RDF stores more efficient. The basic idea is to store the majority of data, the "regular" triples (typically $>$95% of all data) in relational tables under the hood, and the remaining "exception" triples in a reduced PSO triple table. This storage still allows to see the relational data as if it were a PSO table, but is in fact $>$ 1.4x more compact and faster to access than a normal PSO table. Furthermore, we provide a simple optimization heuristic that groups triple patterns by star-shape. This reduces the complexity of query optimization by often more than a magnitude, since the size of the join graph is reduced thanks to only joining these groups. Finally, we contribute the RDFscan algorithm with three important optimizations. It is designed to work in conjunction with relational scans, which perform most of the heavy-lifting, and

can benefit from existing physical storage optimizations such as table clustering and partitioning. RDFscan keeps the overhead of generating additional binding results for "exception" triples low, yielding overall speed improvements of 3-10x on a wide variety of datasets and benchmarks, closing the performance gap between SQL and SPARQL.

Chapter 5

Benchmarking RDF stores

In this chapter, we present our work on RDF benchmarking performed while participating in the LDBC Social Network Benchmark (SNB) task force. In particular, we first describe our main contribution on this benchmark focusing specifically in designing and developing its scalable correlated graph generator. Then, we shortly introduce the benchmark and its workloads.

5.1 S3G2: A Scalable Structure-correlated Social Graph Generator

Benchmarking graph-oriented database workloads and graph-oriented database systems is increasingly becoming relevant in analytical Big Data tasks, such as social network analysis. In graph data, structure is not mainly found inside the nodes, but especially in the way nodes happen to be connected, i.e. structural correlations. Because such structural correlations determine join fan-outs experienced by graph analysis algorithms and graph query executors, they are an essential, yet typically neglected, ingredient of synthetic graph generators. To address this, we present S3G2: a Scalable Structure-correlated Social Graph Generator. This graph generator creates a synthetic social graph, containing non-uniform value distributions and structural correlations, which is intended as test data for scalable graph analysis algorithms and graph database systems. We generalize the problem by decomposing correlated graph generation in multiple passes that each focus on one so-called *correlation dimension*; each of which can be mapped to a MapReduce task. We show that S3G2 can generate social graphs that (i) share well-known graph connectivity characteristics typically found in real social graphs (ii) contain certain plausible structural correlations that influence the performance of graph analysis algorithms and queries, and (iii) can be quickly generated at huge sizes on common cluster hardware.

5.1.1 Introduction

Data in real life is correlated; e.g. people living in Germany have a different distribution in names than people in Italy (location), and people who went to the same

university in the same period have a much higher probability to be friends in a social network. Such correlations can strongly influence the intermediate result sizes of query plans, the effectiveness of indexing strategies, and cause absence or presence of locality in data access patterns. Regarding intermediate result sizes of selections, consider:

```
SELECT personID FROM person
WHERE firstName = 'Joachim' AND addressCountry = 'Germany'
```

Query optimizers commonly use the *independence assumption* for estimating the result size of conjunctive predicates, by multiplying the estimates for the individual predicates. This would underestimate this result size, since Joachim is more common in Germany than in most other countries; similar would happen e.g. when querying for `firstName` 'Cesare' from 'Italy'. Overestimation can also easily happen, if we would query for 'Cesare' from 'Germany' or 'Joachim' from 'Italy' (i.e. *anti-correlation*).

This correlation problem has been recognized in relational database systems as relevant, and some work exists to detect correlated properties inside the same table (e.g., see [173]). Still, employing techniques for the detection of correlation is hardly mainstream in relational database management, and this is even more so when we start considering correlations between predicates that are separated by joins. Consider for instance the DBLP example of co-authorship of papers that counts the number of authors that have published both in TODS and in the VLDB Journal:

```
SELECT COUNT(*)
FROM paper pa1 JOIN journal jn1 ON pa1.journal = jn1.ID
     paper pa2 JOIN journal jn2 ON pa2.journal = jn2.ID
WHERE pa1.author = pa2.author AND
      jn1.name = 'TODS' AND jn2.name = 'VLDB Journal'
```

The above query is likely to have a larger result size than a query that substitutes 'TODS' for 'Bioinformatics', even though Bioinformatics is a much larger publication than TODS. The underlying observation is that database researchers are likely to co-publish in TODS and The VLDB Journal, but are much less likely to do cross-disciplinary work. For database technology, this example poses (i) a challenge to the optimizer to adjust the estimated join hit ratio of `pa1.author = pa2.author` downwards or upwards depending on other (selection or join) predicates in the query (ii) provide indexing support that can accelerate this query: the anti-correlated query (Bioinformatics and The VLDB Journal) has a very small result size and thus could theoretically be answered very quickly. However, just employing standard join indices will generate a large intermediate result for the Bioinformatics sub-plan containing all Bioinformatics authors, of which only a minute fraction is actually useful for the final answer.

Summarizing, correlated predicates are still a frontier area in database research, and such queries are generally not well-supported yet in mature relational systems. This holds still more strongly in the emerging class of graph database systems,

where we argue the need for correlation-awareness in query processing is even higher.

In the particular case of RDF, its graph data model is expressly chosen to work without need for an explicit schema, such that graph datasets get stored as one big pile of edges (in particular, subject-property-object "triples"). Here we see a dualism between structure and correlation: in the relational model, certain structure is explicit in the schema, whereas in RDF such structure only re-surfaces as structural *correlation*. That is, it will turn out a journal paper (subject) always happens to have one `title` property, one `issue` property, one `journalName`, etc; and that these properties exclusively occur in connection to journal issues. The extreme flexibility of RDF systems in the data they can store, thus poses a significant challenge to SPARQL query optimizers, as they need to understand such correlations to get the planning of even basic queries right. Other graph database systems which use a richer data model, where nodes have a declared structure, suffer less from this problem. Still, when considering that graph analysis queries often involve a combination of (property) value constraints and structural constraints (pattern matching), it is likely that correlations between the structure of the graph and the values in them will strongly affect the performance of systems and algorithms. Yet, systems are not sufficiently aware of this, and existing graph benchmarks do not specifically test for this; and synthetic graphs used for benchmarking do not have such structure correlations. As such, we argue that for *benchmarking* graph data analysis systems and algorithms, it would be very worthwhile if a data generator could generate *synthetic graphs* which such *correlated structure*. To our knowledge, there exists no solution for generating a scalable random graph with value and structure correlations. Existing literature on random graph generation [47, 126, 58, 94] either does not consider node properties at all or ignores correlations between them.

In this chapter, we describe the Scalable Structure-correlated Social Graph Generator (S3G2), and its underlying generic conceptual correlated graph generation framework. This framework organizes data generation in multiple phases that each center around a *correlation dimension*. In the case of our social graph use case, these dimensions are (i) education and (ii) personal interests. The data generation workflow is constrained by *correlation dependencies*, where certain already generated data influences the generation of additional data. A graph generator generates new nodes (with property values), and edges between these nodes and existing nodes. The probability to choose a certain value from a dictionary, or the probability to connect two nodes with an edge are thus influenced by existing data values. For instance, the birth location of a person influences probability distribution of the `firstName` and `university` dictionaries. As another example, the probability to create a friendship edge is influenced by (dis)agreement on `gender`, `birthYear` and `university` properties of two person nodes.

A practical challenge in S3G2 is that a naive approach to correlated graph generation would continuously access possibly any node and any edge in order to make decisions regarding the generation of a next node or edge. For generating graphs of a size that exceeds RAM, such a naive algorithm would grind down due to expensive random I/O. To address this challenge, we designed a S3G2 graph generation algorithm following the MapReduce paradigm. Each pass along one correlation di-

mension is a Map phase in which data is generated, followed by a Reduce phase that sorts the data along the correlation dimension that steers the next pass. We show that this algorithm achieves good parallel scale-out, allowing it e.g. to generate 1.2TB of correlated graph data in half an hour on a Hadoop cluster of 16 machines.

Contributions of our work are the following: (1) we propose a novel framework for specifying the generation of correlated graphs, (2) we show the usefulness of this framework in its ability to specify the generation of a social network with certain plausible correlations between values and structure, and (3) we devise a scalable algorithm that implements this generator as a series of MapReduce tasks, and verify both quality of its result as well as its scalability. In our vision, this data generator is a key ingredient for new benchmarks for graph query processing.

Outline. In Section 5.1.2, we present our framework for the generation of correlated graphs, and describe how such it maps on a MapReduce implementation. In Section 5.1.4 we use our framework to generate a synthetic social network graph. In Section 5.1.5 we evaluate our approach, confirming that the generated data has typical social network characteristics, and showing the scalability of our generator. Finally, in Section 5.1.6, we review related work before concluding in Section 5.1.7.

5.1.2 Scalable Structure-correlated Social Graph Generator (S3G2)

We first formally define the end product of S3G2 which is essentially a directed graph of objects, and introduce the main ingredients of the S3G2 framework. Then, we describe the MapReduce-based generation algorithm that follows from these ingredients.

S3G2 generates a directed labeled graph, where the nodes are objects with property values, and their structure is determined by the *class* a node belongs to. Such a data model is common in graph database systems, and is more structured than RDF (though it can be represented in RDF, as our S3G2 implementation in fact does).

Definition 12 *S3G2 produces a graph G(V, E, P, C) where V is a set of nodes, E is a set of edges, P is a set of properties and C is a set of classes.*

$$V = L \cup \bigcup_{c \in C} O^c$$
$$P = \left\{ P^{L(x)} \,|x \in C \right\} \cup \left\{ P^{E(x,y)} \,|x, y \in C \right\}$$
$$E = \left\{ (n_1, n_2, p) | n_1 \in O^x \wedge ((n_2 \in L \wedge p \in P^{L(x)}) \vee (n_2 \in O^y \wedge p \in P^{E(x,y)})) \right\}$$

in which O^c is an object of class c in C; L is the set of literals; $P^{L(x)}$ is set of literal properties of class x in C; $P^{E(x,y)}$ is the set of properties representing relationship edges that go from instances of class x to class y.

We now discuss the main concepts in S3G2, which are (i) property dictionaries, (ii) simple subgraph generation, and (iii) edge generation along correlation dimensions.

Property Dictionary. Property values for each literal property $l \in P^{L(x)}$ are generated following a property dictionary specification $PD_l(D, R, F)$, consisting of a

dictionary D, a ranking function R and a probability function F (if the context is unclear, we can also write D_l, R_l and F_l).

A dictionary D is simply a fixed set of values: $D = \{v_1, .., v_{|D|}\}$. The ranking function R is a bijection $R : D \rightarrow \{1, .., |D|\}$ which gives each value in a dictionary a unique rank between 1 and $|D|$. The probability density function $F : \{1, .., |D|\} \rightarrow [0, 1]$ steers how the generator chooses values; i.e. by having it draw random numbers $0 \leq p \leq 1$, it chooses the largest rank r such that $F'(r) < p$, where F' is the cumulative version of F, that is $F' = \sum_{i=1}^{r} F(i)$. It finally emits the value v_{pos} from dictionary D from position $pos = R(r)$. Thus, our framework can generate data corresponding to any discrete probability distribution.

The idea to have a separate ranking and probability function comes from generating correlated values. In particular, the ranking function $R[z](c)$ is typically parametrized by some parameters z; which means that depending on the parameter z, the value ranking is different. For instance, in case of a dictionary of `firstName` we could have $R[g, c, y]$; e.g. the popularity of first names, depending on `gender` g, `country` c and the year y from the `birthDate` property (let's call this `birthYear`). Thus, the fact that the popularity of first names in different countries and times is different, is reflected by the different ranks produced by function $R()$ over the full dictionary of names. Name frequency distributions do tend to be similar in shape, which is guaranteed by the fact that we use the same probability distribution $F()$ for all data of a property.

Thus, the S3G2 data generator must contain property dictionaries D_l for all literal properties in $l \in P^{L(x)}$, and it also must contain the ranking functions R_l, for all literal properties defined in all classes $x \in C$. When designing correlation parameters for a ordering function R_l, one should ensure that the amount of parameter combinations such as (g, c, y) stays limited, in order to keep the representation of such functions compact. We want the generator to be a relatively small program and not depend needlessly on huge data files with dictionaries and ranking functions.

Figure 5.2 shows how S3G2 compactly represents $R[g, c, y]$, by keeping for each combination of (g, c, y) a small table with only the top-N dictionary values (here N=10 for presentation purposes, but it is typically larger). Rather than storing an ordering of all values, a table like $R[male, Germany, 2010]$ is just an array of N integers. A value j here simply identifies value v_j in dictionary D. The values ranked lower than N get their rank assigned randomly. Given that in a

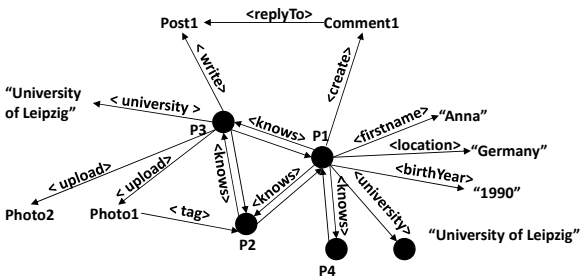


Figure 5.1: Example S3G2 graph: Social Network with Person Information.

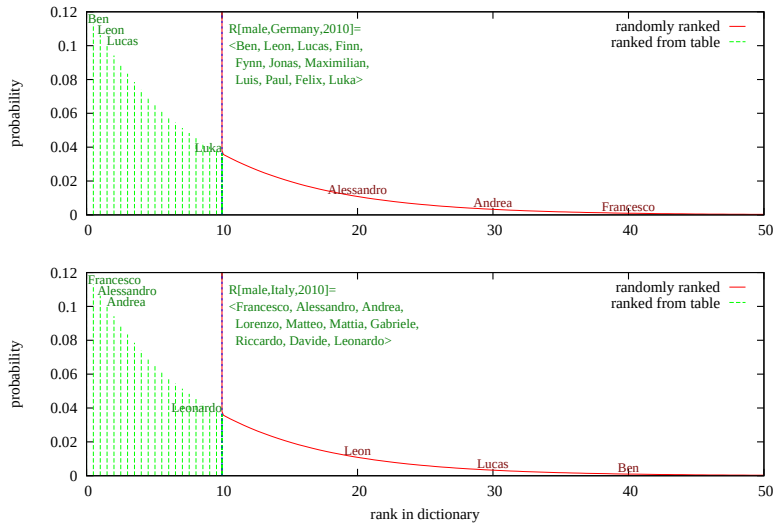


Figure 5.2: Compact Correlated Dictionary Distributions: boy names in Germany (up) vs. Italy (lo)

monotonically decreasing probability function like the geometric distribution used here, the probabilities below that rank are very small anyway, this approximation only slightly decreases the plausibility of the generated values. In Figure 5.2 we see in the top graph that for *(male,Germany,2010)* we keep the 10 most popular boys names, which get mapped on the geometric distribution. All other dictionary values (among which Italian names) get some random rank > 10. In the lower graph, we see that for *(male,Italy,2010)* these Italian names are actually the most popular, and the German names get arbitrary (and low) probabilities.

Simple Graph Generation. Edges are often generated in one go together with new nodes, essentially starting with an existing node n, and creating new nodes to which it gets connected. This process is guided by a degree distribution function $N : h \rightarrow [0, 1]$ that first determines how many h such new children (or descendants) to generate. In many social networks, the amount of neighbour edges h is distributed following a *power law* distribution (the probability that a node has degree $h \sim \gamma.h^{-\lambda}$).

In the S3G2 framework, it is possible to have a correlated the degree distribution function $N[n_i](h)$, from which the degree of each nodes n_i is generated, correlated with properties of node n_i, e.g. by having these properties influence λ or γ. For instance, people with many friends in a social network will typically post more pictures than people with few friends (hence, the amount of friend nodes in our use case influences the amount of posted comment and picture nodes).

Generating new nodes and connecting them on the fly among mostly themselves and to an existing node n_i leads to isolated subgraphs that are dangling off the main graph connected to it by n_i. Typically, such subgraphs are small or have the shape

of shallow trees if they are larger.

Correlation Dimensions. To generate correlated and highly connected graph data, we need a different approach that generates edges *after* generating many nodes. This is computationally harder than generating edges towards new nodes. The reason is that if node properties influence their connectivity, a naive implementation would have to compare the properties of all existing nodes with all nodes, which could lead to quadratic computational cost and a random access pattern, so the generation algorithm would only be fast as long as the data fits in RAM (to avoid a random I/O access pattern).

Data correlation actually alleviates this problem. We observe that the probability that two nodes are connected is typically skewed with respect to some similarity between the nodes. Given node n_i, for a small set of nodes that are somehow similar to it, there is a high connectivity probability, whereas for most other nodes, this probability is quite low. This observation can be exploited by a graph data generator by identifying *correlation dimensions*.

For a certain edge label $e \in P^{E(x,y)}$ between node classes O^x and O^y, a correlation dimension $CD_e(M^x, M^y, F)$ consists of two *similarity metric* functions $M^x : n \rightarrow [0, \infty]$, $M^y : n \rightarrow [0, \infty]$, and a probability distribution F :[1,$W.t$]→[0,1]. Here the $W.t$ is a window size, of W tiles with each t nodes, as explained later. Note that in case of friends in a social network, both start and end of the edges are of the same class persons ($O^x = O^y$), so a single metric function would typically be used. For simplicity of discussion we will assume $M = M^x = M^y$ in the sequel.

We can compute the similarity metric by invoking $M(n_i)$ on all nodes n_i, and sort all nodes on this score. This means that similar nodes are brought near each other, and we observe that the larger the distance between two nodes, their similarity difference monotonically increases. Again, we use a geometric probability distribution for $F()$ that provides a probability for picking nodes to connect with that are between 1 and $W.t$ positions apart in this similarity ranking. To fully comply with a geometric distribution, we should not cut short at $W.t$ positions apart, but consider even further apart nodes. However, we observe that for a skewed monotonically decreasing distribution like geometric, the probability many positions away will be minute, i.e. $\leq \epsilon$ ($F(W.t) = \epsilon$). The advantage of this window shortcut is that after sorting the data, it allows S3G2 to generate edges using a fully sequential access pattern that needs little RAM resources (it only buffers $W.t$ nodes). An example of a similarity function $M()$ could be `location`. Location, i.e., a place name, can be mapped to (longitude,latitude) coordinates, yet for $M()$ we need a single-dimensional metric that can be sorted on. In this case, one can keep (longitude,latitude) at 16-bits integer resolution and mix these by bit-interleaving into one 32-bits integer. This creates a two-dimensional space filling curve called Z-ordering, also known in geographic query processing as QuadTiles[1]. Such a space filling curve "roughly" provides the property that points which are near each other in the Euclidean space have a small z-order difference.

[1] See http://wiki.openstreetmap.org/wiki/QuadTiles

Note that the use of similarity functions and probability distribution functions over ranked distance drives *what* kind of nodes get connected with an edge, not *how many*. The decision on the degree of a node is made prior to generating the edges, using the previously mentioned degree function $N[n_i](h)$, which in social networks would typically be a power-law function. During data generation, this degree $n_i.h$ of node n_i is determined by randomly picking the required number of edges according to the correlated probability distributions as described before in the example with person who have many friends generating more discussion posts. In case of multiple correlations, we use another probability function to divide the intended number of edges between the various correlation dimensions. Thus, we have a power-law distributed node degree, and a predictable (but not fixed) average split between the causes for creating edges.

Random Dimension. The idea that we only generate edges between the $W.t$ most similar nodes in all correlation dimensions is too restrictive: unlikely connections in a social network that the data model would not explain or make plausible, will occur in practice. Such random noise can be modeled by partly falling back onto uniformly random data generation. In the S3G2 framework this can be modeled as a special case of a correlation dimension, by using a purely random function as similarity metric, and a uniform probability function. Hence, data distributions can be made more noisy by making a pass in random order over the data and generating (a few) additional random edges.

5.1.3 MapReduce S3G2 Algorithm

In the previous discussion we have introduced the main concepts of the S3G2 framework: (i) correlated data dictionaries (ii) simple graph generation (iii) edge generation according to correlation dimensions. We now describe how a MapReduce algorithm is built using these ingredients.

In MapReduce, a Map function is run on different parts of the input data on many cluster machines in parallel. Each Map function processes its input data item and produces for each a result with a key attached. MapReduce sends all produced results to Reduce functions that also run on many cluster machines; the key determines to which Reducer each item is sent. The Reduce function then processes this stream of data.

In the S3G2 algorithm, the key generated between Map and Reduce is used to *sort* the data for which edges need to be generated according to the similarity metric (the M^x, M^y functions) of the next correlation dimension. As mentioned, there may be multiple correlation dimensions, hence multiple successive MapReduce phases. Both the Map and Reduce functions can perform simple graph generation, which includes generation of (correlated) property values using dictionaries, as described before in the example with boys names in Germany vs. Italy. The main task of the Reduce function is sorting on correlation dimension and subsequent edge generation between existing nodes using a sliding window algorithm described in Algorithm 2.

The main idea of the *sliding window* approach to correlated edge generation is that when generating edges, we only need to consider nodes that are sufficiently

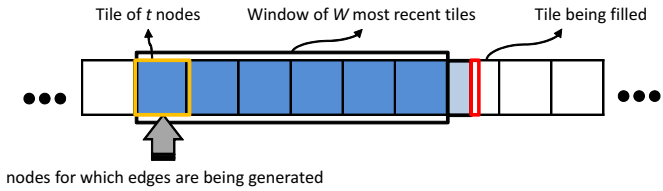


Figure 5.3: Sliding window of W tiles along the graph.

similar. By ordering the nodes according to this similarity (the metric M^x, M^y) we can keep a sliding window of nodes (plus their properties and edges) in RAM, and only consider generating edges between nodes that are cached in this window. If multiple correlations influence the presence of an edge, multiple full data sorts and sequential sliding window passes are needed (i.e. multiple MapReduce jobs). Thus, each correlation dimension adds one MapReduce job to the whole process, that basically re-sorts the data. One may remark that if the simple graph generation activities that kick off graph generation already generate data ordered along the first correlation dimension, we can save one MapReduce job (as data is already sorted).

The sliding window approach is implemented by dividing the sorted nodes conceptually in tiles of t nodes. When the Reduce function accepts a data item, it adds it to the current tile (an in-memory data structure). If this tile is full, and it has W tiles already in memory, the oldest tile is dropped from memory. This is visualized in Figure 5.3.

The Reduce function generates edges for all nodes in the oldest tile right before it is dropped, using Algorithm 2, implementing the windowing approach and generating edges along a correlation dimension. For each node u in this tile, it sequentially scans nodes in the window, and picks a node to be connected based on a probability function $F()$, until $N(u)$ nodes are connected. Function $F()$ computes the probability of connecting two nodes based on their absolute distance in the window. Using this function nearby nodes are most likely to be picked; since successive nodes do the same, there is a high likelihood that similar (nearby) nodes have some overlapping neighbours (e.g. friends).

In principle, simple graph generation only requires local information (the current node), and can be performed as a Map task, but also as a post-processing job in the Reduce function. Note that node generation also includes the generation of the (correlated) properties of the new nodes.

We should mention that data correlations introduce *dependencies*, that impose constraints on the order in which generation tasks have to be performed. For instance, if the `firstName` property of a person node depends on the `birthYear` and `university` properties, then within simple node generation, the latter properties need to be generated first. Also, if the discussion posts forum that a user might have below a posted picture involves the friends of that user, the discussion node generation should follow the generation of all `friend` edges. Thus, the correlation rules one introduces, naturally determine the amount of MapReduce jobs needed, as well as the order of actions inside the Map and Reduce functions.

Algorithm 2 GenerateEdges(t, $N()$, $F()$)

Input: t: tile of nodes to generate edges for
Input: N: a function determines the degree of a node
Input: F: computes probability of connecting two nodes based on their distance

```
1:  for each node u in tile t do
2:    for each node v in window do
3:      if numOfEdges(v) = N(v) then
4:        continue
5:      generate a uniform random number p in [0,1)
6:      distance = position of v - position of u;
7:      if (F(distance) < p) & (u not yet connected to v) then
8:        createEdge(u,v)
9:      if numOfEdges(u) = N(u) then
10:       break
11: flushTile(t);
```

5.1.4 Case study: generating social network data

In this section, we show how we applied the S3G2 framework for creating a social network graph generator. The purpose of this generator is to provide a dataset for a new graph benchmark, called the Social Intelligence Benchmark (SIB).[2] As we focus here on correlated graph generation, this benchmark is out of scope for this chapter. Let us state clearly that the purpose of this generator is *not* to generate "realistic" social network data. Determining the characteristics of social networks is the topic of a lot of research, and we use some of the current insights as inspiration (only). Our data generator introduces some plausible correlations, but we believe that real life (social network) data is riddled with many more correlations; it is a true data mining task to extract these. Given that we want to use the generated dataset for a graph database benchmark workload, having only a limited set of correlations is not a problem; as in a benchmark query workload only a limited set of query patterns will be tested.

Figure 5.4 shows the ER diagram of the social network. It contains persons and entities of social activities (posted pictures, and comments in discussions in the user's forum) as the object classes of C. These object classes and their properties (e.g., user name, post creation date, ...) form the set of nodes V. E contains all the connection between two persons including their friendship edges and social activity edges between persons and a social activity when they all join a social activity (e.g., persons discussing about a topic). P contains all attributes of a user profile, the properties of user friendships and social activities.

Correlated Dictionaries. A basic task is to establish a plausible dictionary (D) for every property in our schema. For each dictionary, we subsequently decide on a frequency distribution. As mentioned, in many cases we use a geometric distribution, which is the discrete equivalent of the exponential distribution, known to accurately

[2] See: www.w3.org/wiki/Social_Network_Intelligence_Benchmark

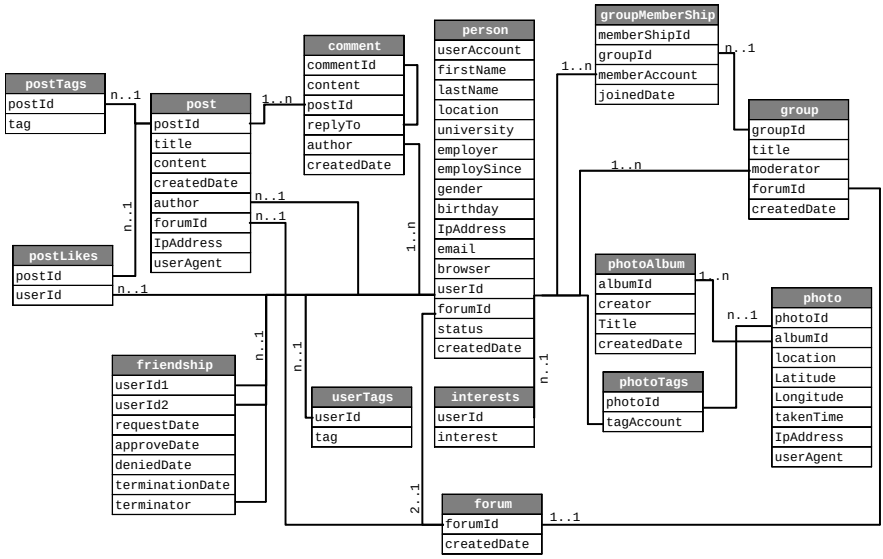


Figure 5.4: The Generated Social Network Schema (SIB).

model many natural phenomena. Finally, we need to determine a ranking of these values in the probability distribution (the $R()$ function). For correlated properties, this function is parameterized ($R[z]()$) and is different for value of z. Our compact approximation stores for each z value a top-N (typically N=30) of dictionary values.

The following property value correlations are built in ($R_x[z]$ denoted as $z \rightsquigarrow x$):

- `(person.location,person.gender,person.birthDay)` $\rightsquigarrow$ `person.firstName`
- `person.location` $\rightsquigarrow$ `person.lastName`
- `person.location` $\rightsquigarrow$ `person.university`
- `person.location` $\rightsquigarrow$ `person.employer`
- `person.location` $\rightsquigarrow$ `person.employSince`
- `(person.location,person.Gender,person.birthDay)` $\rightsquigarrow$ `person.interests.interest`
- `person.location` $\rightsquigarrow$ `person.photoAlbum.photo.location`
- `person.employer` $\rightsquigarrow$ `person.email`
- `person.birthDate` $\rightsquigarrow$ `person.createdDate`
- `person.createdDate` $\rightsquigarrow$ `person.photoAlbum.createdDate`
- `photoAlbum.createdDate` $\rightsquigarrow$ `photoAlbum.photo.takenTime`

- `photoAlbum.photo.location` $\rightsquigarrow$ `photoAlbum.photo.latitude`
- `photoAlbum.photo.location` $\rightsquigarrow$ `photoAlbum.photo.longitude`
- `friendship.requestDate` $\rightsquigarrow$ `friendship.approveDate`
- `friendship.requestDate` $\rightsquigarrow$ `friendship.deniedDate`
- `(friendship.userId1,friendship.userId2)` $\rightsquigarrow$ `friendship.terminator`
- `person.createdDate` $\rightsquigarrow$ `person.forum.createdDate`
- `forum.createdDate` $\rightsquigarrow$ `forum.groupmembership.joinedDate`
- `forum.createdDate,forum.post.author.createdDate` $\rightsquigarrow$ `forum.post.createdDate`
- `post.createdDate` $\rightsquigarrow$ `post.comment.createdDate`

Our main source of dictionary information is DBpedia [44], an online RDF version of Wikipedia, extended with some ranking information derived with internet search engine scripts. From DBpedia one can obtain a collection of place names with population information, which is used as `person.location`. For the place names, DBpedia also provides population distributions. We use this actual distribution as found in DBpedia to guide the generation of `location`.

The `person.university` property is filled with university names as found in DBpedia. The sorting function $R_{university}[location]$ ranks the universities by distance from the person location, and we keep for each location the top-10 universities. The geometric distribution is used as $F_{university}$ and its parameters are tuned such that over 90% of persons choose one of the top-10. Arguably, it is not plausible that all persons have gone to university, but absolute realism is not the point of our exercise.

From the cities, DBpedia allows to derive country information. DBpedia contains a large collection of person names (first and lastnames) and their country of birth, plus certain explicit information on popularity of first-names per country, which was used as well. Other information was collected manually on the internet, such as a distribution of browser usage, which is not correlated with anything, currently. A special rule for dates is applied that ensures that certain dates (e.g. the date a user joined the network) precede another date (the date that a user became friends with someone). This is simply done by repeating the process of randomly picking a date until it satisfies this constraint.

Correlation Dimensions. In our social network graph, the graph with most complex connectivity is the friends graph. The main correlations we have built in are (i) having studied together (ii) having common interests (hobbies). Arguably, the current schema allows more plausible correlations like working in the same company, or living really close, but these can easily be added following our framework. Further, the concept of `interest` is currently highly simplified to favorite musical artists/composers. Consequently, there are three correlation dimensions, where the first is studying together, the second is musical interests and the third is random (this will create random connections). The degree of the persons (function $N[n](h)$) is

a power-law distribution that on average produces h=30 friends per person node n; it is divided over the three correlation dimensions in a 45%, 45%, 10% split: on average we generate 13.5 study friends, 13.5 friends with similar interests and 3 random friends. For having studied together we use the $M_{study}()$ function described before, It depends on `gender`, `university` and `birthYear`, to give highest probability for people of same gender who studied together to be friends. The similarity metric $M_{study}()$ hashes the `university` to the highest 20 bits of an integer; the following 11 bits are taken by filled with the `birthYear` and the lowest bit by `gender`. The musical-interests correlation dimension is also a multi-valued function, because the persons have a list of favorite artists/composers. The similarity metric $M_{interests}$ creates a vector that holds a score for each genre (S3G2 has predetermined genre vectors for all artists, and the result vector contains the maximum value of all favorite artists for each genre). Then, like the previous example with `location`, z-ordering is used to combine the various genre scores (the genre vector) into a single integer metric.

Graph Generation. The generation of the social graph kicks off by generating person nodes; and all its properties. This "simple graph" generation process forms part of the first MapReduce job and is executed in its Map function. The data *is* generated in a specific order: namely `location`. From `location`, we generate `university` in the Map phase and with that (and the uncorrelated `gender` and `birthYear` we are able to emit an M_{study} key, that the first Reduce phase sorts on. Because the members of the forum groups of a user (who tag photos and comment on discussions of the user page) and their activity levels are correlated with the user's friends, the objects for these "social activities" cannot be generated before all friends have been generated. Therefore, the algorithm first continues with all correlation dimensions for friendship. The second MapReduce job generates the first 45% percent of friendship edges using the F_{study} probability distribution in its Map function, and emits the $M_{interest}$ keys. Note that we sort person objects that include all their properties and all their generated friendship edges (user IDs); which are stored twice, once with the source node and once at the destination node. The third MapReduce job generates the second 45% percent of friendship edges in its Map function using the $F_{interests}$ probability distribution, and emits the M_{random} keys. The key produced is simply a random number (note that all randomness is deterministic, so the generated dataset is always identical for identical input parameters). The Reduce phase of the third MapReduce job sorts the data on M_{random}, but as this is the last sort, it runs the window edge-generation algorithm right inside the Reduce function. This Reduce function further performs *simple graph generation* for the social activities. These social activities are subgraphs with only "local" connections and shallow tree-shape, hence can be generated on-the-fly with low resource consumption. Here, the discussion topics are topics from DBpedia articles, and the comments are successive sentences from the article body (this way the discussions consist of real English text, and is kind-of on-topic). The forum group members are picked using a ranking function that puts the friends of a user first, and adds some persons that are in the window at lower ranks; using a geometric probability distribution.

5.1.5 Evaluation

We evaluate S3G2 both qualitatively and quantitatively (scalability). Existing literature studying social networks has shown that popular real social networks have the characteristics of a small-world network [144, 186, 49]. We consider the three most robust measures, i.e. the social degrees, the clustering coefficient, and the average path length of the network topology. We empirically show that S3G2 generates a social graph with such characteristics. In this experiment, we generated small social graphs of 10K, 20K, 40K, 80K, and 160K persons, which on average have 30 friends.

Table 5.1: Graph measurements of the generated social network.

# users	Diameter	Avg. Path Len.	Avg. Clust. Coef.
10000	5	3.13	0.224
20000	6	3.45	0.225
40000	6	3.77	0.225

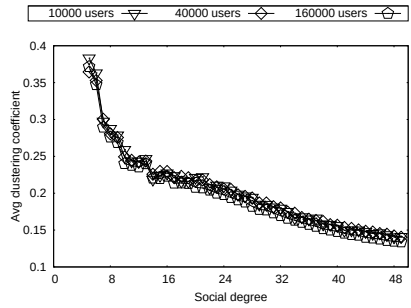
Clustering coefficient. Table 5.1 shows the graph measurements of the generated social network while varying the number of users. According to the experimental results, the generated social networks have high clustering coefficients of about 0.22 which adequately follow the analysis on real social networks in [186] where the clustering coefficients range from 0.13 to 0.21 for Facebook, and 0.171 for Orkut. Figure 5.5a shows the typical clustering coefficient distribution according to the social degrees that indicates the small-world characteristic of social networks.

Average path length. Table 5.1 shows that the average path lengths of generated social graphs range from 3.13 to 3.77 which are comparable to the average path lengths of real social networks observed in [186]. These experimental results also conform to the aforementioned observations that average path length is logarithmically proportional to the total number of users. Since we used a simple all-pair-shortest-path algorithm which consumes a lot of memory for analyzing large graphs, Table 5.1 only shows the results of the average path length for a social graph of 40K users.

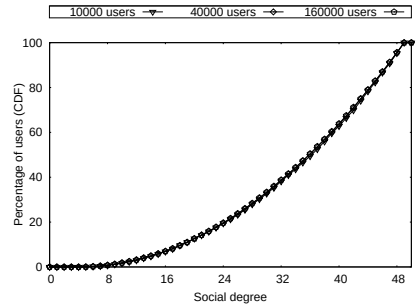
Social degree distributions. Figure 5.5b shows the distribution of the social degree with different number of users. All our experimental results show that the social degree follows a power-law distribution with an alpha value of roughly 2.0.

Scalability. We conducted scalability experiments generating up to 1.2TB of data on Hadoop a cluster of 16 nodes. Each node is a PC with an Intel i7-2600K, 3.40GHz CPU, 4-core CPU and 16 GB RAM. [3] The intermediate results in the MapReduce program use Java object serialization, and the space occupancy of a person profile+friends is 2KB. The final datasize per person is 1MB: most is in the few hundred comments and picture tags each person has (on average), which contain largish text fields.

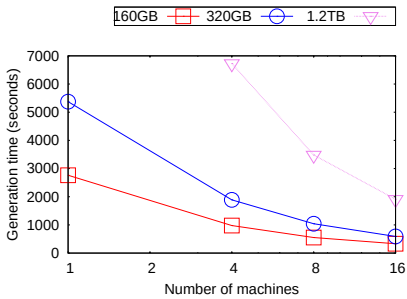
[3] We used the SciLens cluster at CWI: www.scilens.org



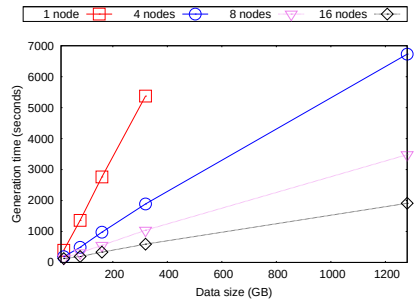
(a) Clustering coefficient



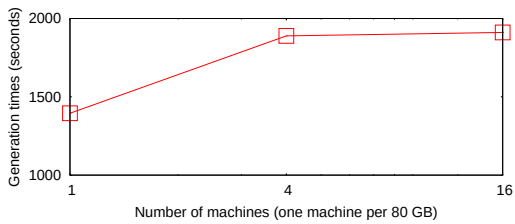
(b) User distribution



(c) Speed-Up Experiments



(d) Scale-Up Experiments



(e) Scale-Out Experiments

Figure 5.5: Experimental Evaluation of S3G2

In Figure 5.5d, for a specific number of nodes, we increase the data size. These results shows that the generation increases linearly with data size. Most of the computational effort is in the first Map function that generates all person nodes and its properties. Further, most data volume (and I/O) appears in the last Reduce that generates the social activities (photos, forum posts). Both these first and last steps are time intensive and benefit strongly from parallel execution. Therefore, the cost of data sorting, which is the mainstay of the intermediate steps, and which due to its $N.log(N)$ complexity should causes less than linear scaling, is not visible yet at these data sizes.

Figure 5.5c shows the speed-up of the generator when adding nodes and keeping

data size fixed. It shows the MapReduce approach works well, and speed-up is especially good at the larger data sizes.

Figure 5.5e shows the scale-out capability of S3G2 increasing together the dataset size and amount of cluster machines. In these experiments we keep the data generated per machine at 80GB; hence with 4 machines we generate 320GB and with 16 this is 1.2TB. The experimental result shows that performance remains constant at half an hour when scaling out from 4 machines to 16 nodes. This suggests that S3G2 can generate extremely large graphs quickly on a Hadoop cluster with large resources.

5.1.6 Related Work

There is a lot of work studying the characteristics of social networks [143, 82, 144, 186, 49, 37, 122] and also on the generation of random graphs having global properties similar to a social network [182, 46, 47, 126, 58, 94]. However, to the best of our knowledge, there is no generator that creates a synthetic social graph with correlations. The existing graph generators mostly consider the topology and the structures of the generated graph, i.e., global properties, not the individual connections of nodes and their correlations.

One of the first studies to generate social-network-like random graph is [182]. This graph generator with small world properties such as a high clustering coefficient and low path lengths, by connecting a node with its k-nearest-neighbors and then rewiring edges. To satisfy the degree distributions [46] introduced the model of preferential attachment which was subsequently improved by [47]. The main idea of this model is that, for a new vertex, the probability that an edge is created between this vertex to an existing vertex depends on the degree of that vertex. Leskovec et al.[126] proposed a tractable graph that matches several properties of a social graph such as small diameter, heavy-tails in/out degree distribution, heavy-tails eigenvalues and eigenvectors by recursively creating a self-similar graph based on Kronecker[4] multiplication. None of these algorithms considers the correlation of a node attributes in the social graph.

Recently, Bonato et al.[58] studied the link structure of a social network and provided a model that can generate a graph satisfying many social graph properties by considering the location of each graph node by ranking each node. In this model, each node is randomly assigned a unique rank value and has a region of influence according to its rank. The probability that an edge is created between a new node and an existing node depends on the ranking of the existing node. Similar to the approach of using influent regions [94] constructed a set of cliques (i.e., groups) over all the users. For each new node (i.e., a new user), an edge to an existing node is created based on the size of cliques they have in common. These models are approaching the realistic observation that users tend to join and connect with people in a group of same properties such as the same location. However, the simulation of realistic data correlations is quite limited and both do not address the correlations between different attributes of the users.

[4] http://en.wikipedia.org/wiki/Kronecker_product

Additionally, all of the existing models need a large amount of memory for storing either the whole social graph or its adjacency matrix. Leskovec et al. [126] may need to store all stages of their recursive graph. Although Batagelj et al. aimed at providing a efficient space-requirement algorithm, the space-requirement is $O(|V| + |E|)$ where V is the set of vertices and E is the set of edges [47].

5.1.7 Conclusion

In this chapter, we have proposed S3G2, a novel framework for scalable graph generator that can generate huge graphs having correlations between the graph structure and graph data such as node properties. While current approaches at generating graphs require holding it in RAM, our graph generator can generate the graph with little memory by using a sliding window algorithm, and exploit parallelism offered by the MapReduce paradigm. It thus was able to generate in half an hour 1.2TB of tightly connected, correlated social graph data, on 16 cluster machines using only limited RAM.

In order to address the problem of generating correlated data and structure together, which has not been handled in existing generators, we propose an approach that separates value generation (data dictionaries) and probability distribution, by putting in between a value ranking function that can be parametrized by correlating factors. We also showed a compact implementation of such correlated ranking functions.

Further, we address correlated structure generation by introducing the concept of correlation dimensions. These correlation dimensions allow to generate edges efficiently by relying on multiple sorting passes; which map naturally on MapReduce jobs.

We demonstrate the utility of the S3G2 framework by applying it to the scenario of modeling a social network graph. The experiments show that our generator can easily generate a graph having important characteristics of a social network and additionally introduce a series of plausible correlations in it.

Future work, is to apply the S3G2 framework to other domains such as telecommunications networks, and a possible direction is to write a compiler that automatically generates a MapReduce implementation from a set of correlation specifications. As we believe that correlations between value and structure are an important missing ingredient in today's graph benchmarks, we intend to introduce a RDF/graph benchmark that uses S3G2 as data generator to fill that gap. By the time of writing this thesis, S3G2 has been extended and become the main data generator, DATAGEN, for LDBC Social Network Benchmark (SNB) which we will introduce in the next section. Our work on DATAGEN and its updated features can be found in Appendix C.

5.2 LDBC Social Network Benchmark (SNB)

Managing and analyzing graph-shaped data is an increasingly important use case for many organizations, in for instance marketing, fraud detection, logistics, pharma,

healthcare but also digital forensics and security. People have been trying to use existing technologies, such as relational database systems for graph data management problems. It is perfectly possible to represent and store a graph in a relational table, for instance as a table where every row contains an edge, and the start and end vertex of every edge are a foreign key reference (in SQL terms). However, what makes a data management problem a graph problem is that the data analysis is not only about the values of the data items in such a table, but about the *connection patterns* between the various pieces. SQL-based systems were not originally designed for this – though systems have implemented diverse extensions for navigational and recursive query execution.

In recent years, the database industry has seen a proliferation of new graph-oriented data management technologies. Roughly speaking, there are four families of approaches. One are pure graph database systems, such as Neo4j, Sparksee and Titan, which elevate graphs to first class citizens in their data model ("property graphs"), query languages, and APIs. These systems often provide specific features such as breadth-first search and shortest path algorithms, but also allow to insert, delete and modify data using transactional semantics. A second variant are systems intended to manage semantic web data conforming to the RDF data model, such as Virtuoso or OWLIM. Although RDF systems emphasize usage in semantic applications (e.g. data integration), RDF is a graph data model, which makes SPARQL the only well-defined standard query language for graph data. A third kind of new system targets the need to compute certain complex graph algorithms, that are normally not expressed in high-level query languages, such as Community Finding, Clustering and PageRank, on huge graphs that may not fit the memory of a single machine, by making use of cluster computing. Example systems are GraphLab, Stratosphere and Giraph, though this area is still heavily in motion and does not yet have much industrial installed base. Finally, recursive SQL, albeit not very elegant, is expressive enough to construct a large class of graph queries (variable length path queries, pattern matching, etc.). One of the possibilities (exemplified by Virtuoso RDBMS) is to introduce vendor-specific extensions to SQL, which are basically shortcuts for recursive SQL subqueries to run specific graph algorithms inside SQL queries (such as shortest paths).

The Linked Data Benchmark Council[5] (LDBC) is an independent authority responsible for specifying benchmarks, benchmarking procedures and verifying/publishing benchmark results. Benchmarks on the one hand allow to quantitatively compare different technological solutions, helping IT users to make more objective choices for their software architectures. On the other hand, an important second goal for LDBC is to stimulate technological progress among competing systems and thereby accelerate the maturing of the new software market of graph data management systems.

This section introduces the Social Network Benchmark (SNB), the first LDBC benchmark, which models a social network akin to Facebook. The dataset consists of persons and a *friendship network* that connects them; whereas the majority of the data is in the *messages* that these persons post in discussion trees on their forums.

[5] ldbcouncil.org - LDBC originates from the EU FP7 project (FP7-317548) by the same name.

While SNB goes through lengths to make its generated data more realistic than previous synthetic approaches, it should not be understood as an attempt to fully model Facebook – its ambition is to be as realistic as necessary for the benchmark queries to exhibit the desired effects – nor does the choice for social network data as the scenario for SNB imply that LDBC sees social network companies as the primary consumers of its benchmarks – typically these internet-scale companies do not work with standard data management software and rather roll their own. Rather, the SNB scenario is chosen because it is an appealing graph-centric use case, and in fact social network analysis on data that contains excerpts of social networks is a very common marketing activity nowadays.

There are in fact three SNB benchmarks on one common dataset (generated by DATAGEN), since SNB has three different *workloads*. Each workload produces a single metric for performance at the given scale and a price/performance metric at the scale and can be considered a separate benchmark. The full disclosure further breaks down the composition of the metric into its constituent parts, e.g. single query execution times. These workload have been carefully designed according to so-called *choke-point* analysis that identifies important technical challenges to evaluate in a workload. Specifically, a choke point is an aspect of query execution or optimization which is known to be problematical for the present generation of various DBMS (relational, graph and RDF). It generally covers the "usual" challenges of query processing (e.g., subquery unnesting, complex aggregate performance, detecting dependent group-by keys etc.)), as well as some hard problems that are usually not part of synthetic benchmarks. Some examples of the choke points are "estimating cardinality in graph traversals with data skew and correlations", "choosing the right join order and type", "handling scattered index access patterns", "parallelism and result reuse", ...

SNB-Interactive. This workload consists of a set of relatively complex read-only queries, that touch a significant amount of data, often the two-step friendship neighborhood and associated messages. Still these queries typically start at a single point and the query complexity is sublinear to the dataset size. Associated with the complex read-only queries are simple read-only queries, which typically only lookup one entity (e.g. a person). Concurrent with these read-only queries is an insert workload, under at least read committed transaction semantics. All data generated by the SNB data generator is timestamped, and a standard scale factor covers three years. Of this 32 months are bulkloaded at benchmark start, whereas the data from the last 4 months is added using individual DML statements.

The goal of SNB-Interactive is to test graph data management systems that combine transactional update with query capabilities. A well-known graph database system that offers this is neo4j, but SNB-Interactive is formulated such that many systems can participate, as long a they support transactional updates allowing simultaneous queries. The query workload focus on interactivity, with the intention of sub-second response times and query patterns that start typically at a single graph node and visit only a small portion of the entire graph. One could hence position it as OLTP, even though the query complexity is much higher than TPC-C and does include graph tasks such as traversals and restricted shortest paths. The ra-

tionale for this focus stems from LDBC research among its vendor members and the LDBC Technical User Community of database users. This identified that many interactive graph applications currently rely on key-value data management systems without strong consistency, where query predicates that are more complex than a key-lookup are answered using offline pre-computed data. This staleness and lack of consistency both impact the user experience and complicate application development, hence LDBC hopes that SNB-Interactive will lead to the maturing of transactional graph data management systems that can improve the user experience and ease application development.

SNB-BI. This workload consists of a set of queries that access a large percentage of all entities in the dataset (the "fact tables"), and groups these in various dimensions. In this sense, the workload has similarities with existing relational Business Intelligence benchmarks like TPC-H and TPC-DS; the distinguishing factor is the presence of graph traversal predicates and recursion. Whereas the SNB Interactive workload has been fully developed, the SNB BI workload is a working draft, and the concurrent bulk-load workload has not yet been specified.

SNB-Algorithms. This workload is under construction, but is planned to consist of a handful of often-used graph analysis algorithms, including PageRank, Community Detection, Clustering and Breadth First Search. While we foresee that the two other SNB workloads can be used to compare graph database systems, RDF stores, but also SQL stores or even noSQL systems; the SNB-Algorithms workload primary targets graph programming systems or even general purpose cluster computing environments like MapReduce. It may, however, be possible to implement graph algorithms as iterative queries, e.g. keeping state in temporary tables, hence it is possible that other kinds of systems may also implement it.

Given that graph queries and graph algorithm complexity is heavily influenced by the complex structure of the graph, we specifically aim to run all three benchmarks on the same dataset. In the process of benchmark definition, the dataset generator is being tuned such that the graph, e.g. contains communities, and clusters comparable to clusters and communities found on real data. These graph properties cause the SNB-Algorithms workload to produce "sensible" results, but are also likely to affect the behavior of queries in SNB-Interactive and SNB-BI. Similarly, the graph degree and value/structure correlation (e.g. people having names typical for a country) that affect query outcomes in SNB-Interactive and BI may also implicitly affect the complexity of SNB-Algorithms. As such, having three diverse workloads on the same dataset is thought to make the behavior of all workloads more realistic, even if we currently would not understand or foresee how complex graph patterns affect all graph management tasks.

By the time of writing this thesis, SNB-Interactive and SNB-BI have been completed. The detail on LDBC SNB and its workloads can be found from its official website [17].

Chapter 6

Conclusions

One of the prime goals of much research in the Semantic Web community is to improve the performance of Semantic Web data management, particularly, RDF data management systems. This thesis has argued that the main problems in current RDF data management systems are excessive join complexity, low storage locality, and lack of user schema insight and empty query results, and has presented techniques to tackle these problems by automatically deriving an emergent relational schema from RDF data and leveraging the derived schema to make RDF stores efficient in terms of RDF storage, query execution, and query optimization. Additionally, we also described our work on evaluating RDF stores by creating an scalable data generator that can generate realistic RDF/graph data having specific characteristics and data corrections of a social network, and building a RDF/graph benchmark for challenging query processing over the generated data. In this chapter, we summarize the research presented in this thesis and iterate over the major contributions. We also sketch a roadmap for future research on the presented subjects.

6.1 Contributions

In this section we summarize the contributions of this thesis by recapitulating the semantic web/database "schema" difference and answering the research questions raised in the Chapter 1.4.

6.1.1 The difference between the semantic web and database schemas

An important contribution in this thesis is identifying the difference between the notions of schema in the database (i.e., relational schema) and in the semantic web (i.e., ontologies and vocabularies) communities. In the semantic web, schemas are intended to allow diverse organizations to consistently denote certain concepts in a variety of contexts, and are not required to be defined upfront ("schema-last"). In contrast, database relational schemas describe the structure of one database, designed without regard for reuse in other databases. The relational schema gives the query writer a clear idea of what the data looks like and must be declared before the data can be used ("schema first"). In this thesis we also find that ontologies are often

mixed and their classes only partially used in describing each RDF resource, so any single ontology class is a poor descriptor of the actual structure of the data. Nevertheless, we also argued that these schema notions are valuable and one can profit from the other: The semantic web applications can use database schemas to provide better understand the dataset, allow users to formulate non-empty-result SPARQL query, and make the systems more efficient (as shown in Chapter 4), while database practitioners can create well-defined and exchangeable semantics for the relational schema by extending its components with URI links to the semantic web schemas, allowing easier data and instance integration with other datasets.

Future work. Following the above discussions, our proposal would be to extend the tables, columns and even individual primary key values (URIs) of the relational schema with the links to the semantic web schemas. In some database systems it is already possible to add a "comment" to each column name, which could be used to link to a semantic web schema resource (like an RDF schema or class property). However rather than using such hacks, it would be desirable that the ISO SQL standard be augmented with semantic annotations for each table, column and foreign key relationship. In addition, the SQL query language could also be extended to better support RDF data, by supplying its type system (prominently URLs), as well as multi-valued properties.

6.1.2 What is an "emergent" relational schema exactly and how to efficiently and scalably discover it from actual RDF datasets? (i.e., Questions 1+2)

An emergent relational schema is a compact and precise relational schema (containing "emergent" tables, columns and relationships between tables) with high coverage (i.e., covering more than 90% of RDF data) and useful labels from RDF data. In particular, an emergent schema is considered to be compact and precise if it contains few and thin tables and has small number of NULL cells in each table. The formal definition of emergent schema is provided in the Section 4.2 of Chapter 4.

One of our major contributions in this thesis is providing practical techniques (presented in Chapter 3) for efficiently and scalably deriving an emergent relational schema from RDF data. Particularly, we first recognize all basic emergent classes consisting of properties that frequently co-occur with the same subject from a bulk-loaded SPO table, and analyze non-literal properties in each class (i.e., the properties that may refer to subject URIs of other classes) in order to explore the relationships between these classes. To make the schema compact, we proposed multiple algorithms for merging classes which are semantically or structurally similar. Finally, we build several filtering approaches (i.e., schema filtering and instance filtering) in order to further optimize the schema and reduce NULLs. The algorithms that we described are efficient and can be executed during the bulk-load of an RDF database with little overhead. We showed that on a wide variety of datasets, the derived schema explains well over 90% of the RDF triples, emphasizing the fact that a great majority of RDF triples do conform to regular structural patterns and provide useful information to improve the performance of RDF stores. We also note that our presented techniques in exploring emergent relational schema from RDF data

also exploit and respect semantic web schemas (e.g., ontologies) when present, but do not require their presence. Because of the partial and mixed ontology class usage in each RDF resource, the emergent relational schema actually presents a more realistic picture of an RDF dataset than one ontology does on its own.

Future work. As deriving the emergent schema, in principle, is to explore the structural regularity of the RDF data, other related research in finding regularities in data can also benefit from our work. An example of the related research is a recent work of Bloem et al. [56] on detecting network motifs (i.e., the subgraph patterns or graphlets which occur more frequently in the data than expected) by using Minimum Description Length (MDL) principle [165]. In particular, each table in the emergent schema can serve as a basic star-shape motif in the network. Besides, in order to achieve a compressed description in designing a code, this research may also exploit our metrics on the compactness (few and thin tables), preciseness (few NULL values), and data coverage of the schema. Thus, future work would be to make use of our techniques on deriving emergent schema in other related graph analysis research.

6.1.3 How to derive human-friendly names for the tables and columns in the emergent relational schema? (Question 3)

We presented multiple methods (in Section 3.2.2) that exploit semantic information (ontologies) as well as structural information for assigning human-friendly labels to the tables and columns of the emergent relational schema. Specifically, these methods mainly exploited either the type properties (e.g., `rdf:type`) available in a emergent class, or the discriminative properties (i.e., properties which appear in few ontology classes only and can give a strong hint for the membership of a specific class), or the URI of the relationship between classes.

Future work. In the emergent relational schema, there can be tables without any label assigned as our current labeling approach either relies on ontology matching (of a limited number of known ontologies) or the encountered relationships between tables. Thus, a future approach might be to look for the other labeling sources on the web e.g., from search engines or from online open database (e.g., freebase [12]).

6.1.4 How to exploit the emergent schema in order to make RDF stores efficient in terms of storage? (Question 4)

The ultimate goal of our research is to build a high-performance RDF store that can efficiently address the existing RDF data management problems by exploiting the emergent schema inside the RDF system (in storage, optimization, and execution). We now summarize our contributions in creating such efficient RDF storage. Specifically, for storing RDF triples, we created an emergent schema-aware RDF storage in which the regular triples which conform to the emergent schema (the majority of RDF data) are represented by relational tables stored in a column-store database under the hood, while the remaining "exception" triples (typically less than 5% of the RDF data) are stored in a small pso triple table. The columnar stor-

age allows to see all this relational data as if it were a PSO table (see Chapter 4 for explanation) and thus SPARQL query execution can always fall back on existing mechanisms (e.g., triple-table-based storage) if the emergent schema-aware query execution is not applicable (e.g., when handling hard cases in SPARQL queries if it contains unbound variables that bind to a property). As column-wise data generally compresses better than row-wise data, this RDF storage was realized in a columnar database system i.e., MonetDB, showing $>$ 1.4x more compact storage than default triple storage (i.e., PSO).

Future work. In the Conclusion section of the Chapter 3, we have shortly discussed the issue of controlling the evolution of the emergent schema in the RDF storage over time as the RDF datasets are exposed to updates. Particularly, in order to update the schema when new data arrives, we propose to export the found schema explicitly using a vocabulary, and then, use that as the "ontology" information for re-running the schema recognition process with the newly updated data to enforce that certain structures remain recognized (schema stability). However, the details of this proposal as well as the influence of updating emergent schema on the performance of the RDF system has not been carefully analyzed yet. Thus, this should be one of the improvements which needs to be addressed in the future work.

6.1.5 How to exploit the emergent schema in order to make RDF stores efficient in terms of query execution and optimization? (Questions 5 + 6)

Together with the novel RDF storage, we contributed so-called emergent schema aware SPARQL optimization and execution to exploit the derived emergent schema in improving the performance SPARQL query. Specifically, the emergent schema aware optimization groups triple patterns in the query by each star-shape pattern so that the SPARQL optimization only needs to be performed on a reduced join graph (of joins between these groups). Consequently, thanks to the smaller number of joins, this can significantly reduce the complexity of the query optimization as well as the optimization time by more than an order of magnitude. The emergent schema aware SPARQL query execution basically handles each star pattern group by using a single relational table scan on its matching emergent tables (i.e., tables which has columns corresponding to the properties in the star pattern), then return the result by joining these table scans.

Future work. As a complex SPARQL query typically contains a conjunction of multiple star-patterns, the emergent schema can be further exploited to efficiently answer multi-star patterns queries. Given the fact that a conjunction of two star-patterns corresponds to the relationship between their matching emergent tables, a future approach might be to build an index over the relationships of emergent tables to efficiently filter out non-matching bindings of the query.

6.1.6 How do we exploit the emergent schema with minimum impact to RDBMS kernel? (Question 7)

To exploit the emergent schema with minimum impact to existing RDBMS kernels, we contributed a new database operator, *RDFscan* that generates complete bindings (including exception data) for a star pattern. This operator performs most of the heavy-lifting and is designed in conjunction with existing relational table scan infrastructure so that advanced relational access paths such as clustered indexes and partitioned tables can seamlessly be re-used. The idea is that most data is scanned with traditional relational scan methods so that existing physical optimizations can be re-used. The RDFscan then augments scan data with missing bindings involving the exception triples. An alternative approach that would generate all RDF bindings in one go would not be able to profit from existing relational infrastructure. This RDFscan operator is further optimized with three important optimizations (i.e., selection push-down, early check for missing property, and pruning non-matching tables), efficiently yielding an overall speed improvement of 3-10x over a wide variety of RDF benchmarks and closing the performance gap between SQL and SPARQL.

Future work. Even though RDFscan was designed to efficiently leverage existing relational access methods, the performance of this operator as well as our emergent schema aware SPARQL optimization/execution in using advanced relational database techniques such as database cracking, vectorized execution have not been carefully evaluated yet. Thus, a future work is to evaluate the efficiency of our proposed emergent schema aware SPARQL optimization/execution in using advanced relational database techniques in our prototype systems (i.e., MonetDB, Virtuoso).

6.1.7 How to scalably generate realistic RDF/graph data that simulates the skewed data distribution and plausible structural correlation in a real network graph? (Question 8)

Lots of our research effort has been put into RDF benchmarking. Our main contribution in this research topic is developing a novel data generator called S3G2 (Scalable Structure-correlated Social Graph Generator) that can scalably generate realistic RDF/graph data simulating the skewed data distributions and plausible structural correlations in a real social network graph. Specifically, in order to generate correlated data with realistic skew distribution, we separated the data dictionary (i.e., set of values) of each literal property and its probability distribution function by associating each data dictionary with a ranking function that can be parameterized by correlated factors. Values in a data dictionary thus will be ranked differently depending on their correlated parameters (e.g., the ranking of each `firstName` depends on `gender`, `country`, and `birthYear` properties). Moreover, as the distribution function steers each value generation by choosing a position in the dictionary without regard to the specific value at that position, our approach allows to generate correlated data with any discrete probability distribution. To obtain realistic structural correlations for the generated graph, we introduced the concept of *correlation*

dimension and decomposed the graph generation into multiple passes in which each pass focuses on one correlation dimension. Particularly, in each pass, the nodes are sorted according to a certain correlation dimension and subsequent edge are generated between existing nodes based on the nodes' properties and their positions along that dimension using a sliding window algorithm. Besides, in order to build a scalable data generator, we leveraged parallelism through Hadoop and naturally mapped each correlation dimension-based generating pass to a MapReduce job. As a result, being designed to create synthetic data that can be representative of a real social network, the generated data of S3G2 exhibited interesting realistic value correlations (e.g., German people having predominantly German names), structural correlations (e.g., friends being mostly people living close to one another), and statistical distributions (e.g., the friendship relationship between people follows a power-law distribution). The data generator also showed fast and scalable generation of huge datasets, allowing to generate a social network structure with millions of user profiles, enriched with interests/tags, posts, and comments using a cluster of commodity hardwares. Moreover, this data generator has been the basis for subsequent research by others. It has been further developed and become the main data generator, DATAGEN [16], of the LDBC Social Network Benchmark (SNB).

Future work. The current extended version of the social graph generator (i.e., DATAGEN) has allowed to generate a social graph with a set of certain characteristics (e.g., max number of friends per user, max number of groups per user, probability of correlation between tags and interests,...). However, this is still limited as e.g., it does not allow to generate the social graph conforming a given specific shape or distribution models (although generating a graph following a certain distribution model is already a big research topic). Thus, the generator can be improved so that it will be robust and support more models or configurations that are theoretically possible to be generated. Nevertheless, we have to acknowledge that this is a different topic although related to our research on social graph generator.

6.1.8 How to design an RDF/graph benchmark over the realistic dataset so that important technical challenges for RDF/graph database systems will be evaluated? (Question 9)

To answer this question, we introduced the LDBC Social Network Benchmark (LDBC SNB) which is designed to evaluate important technical challenges in RDF/graph database based on the realistic dataset (generated from DATAGEN). This benchmark development has been carefully driven by a so-called "choke-point" based design which covers the usual challenges or known-problematical aspects of query execution and optimization in various DBMS (e.g., choosing the right join order and type, estimating cardinality in graph traversals with data skew and correlations). This choke-point analysis requires both user input as well as expert input from database systems architects. Based on the choke-point design and realistic generated social graph data, LDBC SNB has provided three different workloads for covering all the main aspects of social network data management: an interactive workload, oriented to test a systems throughput with relatively simple queries with concurrent updates; a business intelligence workload, consisting of complex struc-

tured queries for analyzing online behavior of users for marketing purposes; and a graph analytics workload, thought to test the functionality and scalability of the systems for graph analytics that typically cannot be expressed in a query language (this workload is being constructed). Further description on the LDBC SNB and its workloads can be found in [88] and from its official website [17].

6.2 Future research directions

In the previous section, we have summarized our contributions and also discussed future works on several research questions. Though, exploiting emergent schema for building efficient RDF stores and the further development of the social graph data generator still opened many other interesting improvements and challenges for future research.

6.2.1 Emergent schema aware distributed system

In Chapter 3 we shortly presented the effort of integrating the emergent schema into the storage, query execution, and query optimization of a distributed system, Virtuoso. Even though the experiment was performed on a single machine, it showed a great possibility of exploiting emergent schema in improving a distributed RDF system. However, this is still a research prototype and the queries were only executed on completely regular RDF data (i.e., BSBM) which in fact does not require the use of an operator like RDFscan as there are no exception triples. Thus, there are several open research questions for future work in leveraging the use of emergent schema in distributed storage, query execution, and optimization such as "How RDF data can be partitioned and stored based on to the emergent relational schema?", "How RDFscan operator will be designed and implemented in a distributed system?".

6.2.2 Exploiting emergent schema for stream RDF engines

As streaming RDF engines are not the main focus of this thesis, we have not discussed these systems and their performance on executing stream SPARQL in detail. However, with its large market of interesting application (e.g., event-based applications, financial trading floors, ecommerce purchases, or geospatial services, ...), we do not ignore the research topics on these systems. Specially, during the time of my PhD, together with external research institutes, we have done several researches on benchmarking and analyzing the performance of these systems [81, 191]. Our basic idea is to incorporate the work originally designed for the non-stream RDF systems in order to evaluate and improve stream RDF systems. However, while the stream version of our social graph generator, S3G2, and its benchmark prototype, Social Network Interlligence Benchmark (SIB) [27], showed a very good testcase for analyzing RDF streaming engines, we did not have time to exploit the idea of emergent relational schema in improving the performance of these engines. Realizing the emergent relational schema-aware execution and optimization on a stream RDF engine is actually more challenging because of the dynamic properties of the

streaming RDF/graph-based data which are produced and changed over time. Thus, it should be a interesting and challenging research topic.

6.3 Summary

In this thesis, we have provided hopefully valuable insights and contributions on developing a high performance RDF store as well as on material for evaluating the technical challenges of RDF/Graph systems. Particularly, we have characterized the differences between semantic web and database schemas, and addressed the main problems in current RDF data management systems by exploiting the emergent relational schema automatically derived from RDF data. Beyond the use of the derived emergent relational schema for conveying the structure information of RDF dataset to users and allowing humans to understand RDF dataset better, we has exploited this emergent schema internally inside the RDF system (in storage, optimization, and execution) in order to build an efficient *self-organizing structure RDF store*. The use of emergent relational schemas has opened a promising direction in developing efficient RDF stores and has shown to close the performance gap between SQL and SPARQL systems. Additionally, we have developed a scalable graph data generator which can generate synthetic RDF/graph data having skew data distributions and plausible structural correlations of a real social network. This data generator has become a core ingredient of an RDF/graph benchmark (LDBC SNB) which is designed to evaluate technical challenges in RDF/graph systems.

List of Figures

List of Tables

Appendix A

Query plan transformation for star pattern

Considering the following example SPARQL query:

```
Select ?s ?o where {
    ?s   <birthPlace> ``Hawaii".
    ?s   <spouse>     ?o.
    ?s   <party>  <Democratic>
}
```

This example query contains three triple patterns (*tp's*) Figure A.1 show the SPARQL graph of the query which contains one star pattern.

A "canonical" un-optimized query plan can be derived easily from a SPARQL graph by creating an index scan for each node in the SPARQL graph and adding a join for each edge in it. Figures A.2 and A.3 show the query graph and an un-optimized plan execution built for the example query.

In this section, we represent the query plan transformation for the star pattern without considering exception data. Basically, the canonical query plan for a star pattern can be re-written as a single Select operator on the matching emergent table. For example, the query plan in Figure A.3 can be rewritten as $\sigma_e(T_1)$ where e is {`place`="Hawaii", `party=Democracy`}, given T_1 is the only matching emergent table of the star pattern. However, since there can be multiple matching tables and the star pattern may contain properties of different types e.g., multi-valued

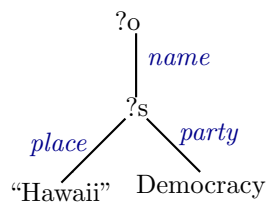


Figure A.1: Example SPARQL graph

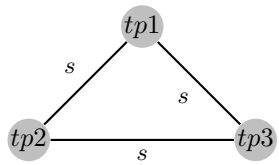


Figure A.2: Join query graph

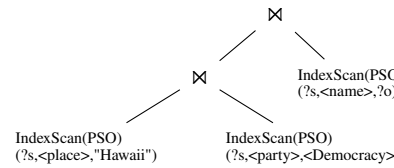


Figure A.3: Example query plan

property or optional property, the query plan transformation is much more complicated. In the following, we formally represent the query plan transformation for a star pattern in different cases, from the simple to the complicated ones.

Case 1. **Single matching table, no multi-valued or optional property**.

Given the star pattern ρ = {t_i=(s, p_i, o_i), i = 1,..,k} and T is the only matching table for ρ. The transformed query plan of ρ will be $\sigma_e(T)$ in which the Select predicate e is generated by aggregating all the Select predicates on the subject and object values of each triple pattern t_i. Specifically, a Select predicate on o_i (e.g., o_i = `Professor0`) will become a Select predicate on the p_i column of T (e.g., p_i = `Professor0`), and a Select predicate on s will become a predicate on the s column of T. For example, the plan transformation for the ρ of three triple patterns {($?s$, `type`, `Publication`), ($?s$, `author`, `Professor0`), ($?s$, `name`, $?o$)} will be $\sigma_{type=Publication\ \&\ author=Professor0}(T)$

Case 2. **Single matching table, having multi-valued property**

Given a emergent table T and a property p. If p is multi-valued property, the object values of p will be stored in a separated table (T_p) having foreign key relationship on the s column with T. Thus, retrieving object values of the multi-valued property p requires a Join between T and T_p. To generate the plan for a star pattern having multi-valued properties, we first transform the plan without considering triple pattern of multiple-valued property using the approach in handling Case 1. Note that, the matching table is identified not only based on single-valued properties, but on all the properties in the star pattern. Then, for each triple pattern having a multi-valued property, a Join is added to join the previously created plan with the multi-valued table of that property.

Given the star pattern ρ = {t_i=(s, p_i, o_i), i = 1,..,k} and the only matching table T. We assume that p_k is a multi-valued property and T_{p_k} is the "multi-valued" table storing the object values of p_k. The query plan for this star pattern will be $\sigma_e(T) \bowtie_s \sigma_{e_k}(T_{p_k})$. Here, e, e_k are the Select predicates generated from the Select predicates of triple patterns {t_i, i = 1,..,$k-1$} and triple pattern (t_k), respectively.

Case 3. **Single matching table, having Optional filter**

The OPTIONAL filter in a SPARQL query allows the RDF/SPARQL engine to return results even without having bindings of a certain triple pattern group by using NULL value for the bindings. In the query graph representation, the optional pattern group is connected from the required pattern groups via an outer join edge. Given a star pattern and its matching emergent table T, we can first transform the required

pattern group as well as the optional pattern group using the transformation process in Case 2, and then, add a Outer join on the column T.s to join these transformed query plans.

We recognized that the OUTER JOIN is created on the s column of the same table. Thus, query plan can be re-written without using OUTER JOIN by having if-then-else clause on the PROJECT operator for optional columns. Specifically, given $\{(s, p_i, o_i)|i = 1,..,h\}$ being an OPTIONAL pattern group, if any of object value the oi (i = 1,.., h) is null, it returns null value for all the columns p_i (i = 1,..,h) in the final output result. The following example demonstrates this PROJECT operator where columns $p3$, $p4$ belong to an OPTIONAL binding.

```
(SELECT p1, p2 FROM T) as t1
LEFT JOIN
(SELECT p3, p4 FROM T) as t2
ON t1.s = t2.s
```

will have the same result as

```
SELECT p1, p2,
     (if (p3 or p4 is null) return null, else p3),
     (if (p3 or p4 is null) return null, else p4)
FROM T
```

Formally, Given the star pattern ρ = $\{t_i$=$(s, p_i, o_i), i = 1,..,k\}$ having OPTIONAL binding on triple patterns $\{t_j, j$=$h + 1,...,k)\}$, ϑ is the query plan generated for ρ with assumption that all triple patterns in ρ are required. ϑ is generated using the transformation process in Case 2. The query plan for ρ will be $\Pi_{p_1,...,p_h,cond(p_{h+1}),...,cond(p_k)}(\vartheta)$. Here, $cond(p_j)$ = "if (p_{h+1} or p_{h+2} or ... or p_k is $null$) return $null$, else p_j" (j = $(h + 1)$,...,k).

Case 4. **Multiple matching tables**

If there are multiple matching relational tables for a star pattern, we create the transformed plan w.r.t each matching table by applying the transformation process in Case 3. Then, we add an UNION operator for combining all the generated plans.

Appendix B

DBpedia queries

In this appendix, we includes the list of queries that we used for DBpedia dataset.
Query 1.

```
SELECT DISTINCT ?var0 ?var1
WHERE {
    ?var2 a <http://dbpedia.org/ontology/Organisation> .
    ?var2 <http://dbpedia.org/ontology/foundationPlace> ?var0 .
    ?var4 <http://dbpedia.org/ontology/developer> ?var2 .
    ?var4 <http://dbpedia.org/ontology/location> ?var1 .
}
```

Query 2.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpowl:<http://dbpedia.org/ontology/>
SELECT DISTINCT ?var
WHERE {
        ?var6 rdf:type ?var.
        ?var6 <http://xmlns.com/foaf/0.1/name> ?var0.
        ?var6 dbpowl:numberOfPages ?var1.
        ?var6 dbpowl:isbn ?var2.
        ?var6 dbpowl:author ?var3.
}
```

Query 3.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX space: <http://purl.org/net/schemas/space/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?var
WHERE { ?var5 dbpedia-owl:thumbnail ?var4 .
        ?var5 rdf:type dbpedia-owl:Person .
        ?var5 rdfs:label ?var .
        ?var5 dbpedia-owl:battle ?battle .
        OPTIONAL { ?var5 foaf:homepage ?var10 .} . }
ORDER BY ?var
LIMIT 10
```

Query 4.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
```

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX space: <http://purl.org/net/schemas/space/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia-prop: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?var
WHERE {
        ?var2 rdf:type dbpedia-owl:Person .
        ?var2 rdfs:label ?var .
        ?var2 dbpedia-owl:worldChampionTitleYear ?var4.
}
```

Query 5.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX space: <http://purl.org/net/schemas/space/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?var0 ?var1 ?var3
WHERE { { ?s    foaf:givenName ?var0;
                dbpedia-owl:team ?var1 ;
                dbpedia-owl:careerStation ?var2 ;
                dbpedia-owl:position ?var3 ;
                dbpedia-owl:number 9 .
        } UNION {
                ?s     foaf:givenName ?var0;
                dbpedia-owl:team ?var1 ;
                dbpedia-owl:careerStation ?var2 ;
                dbpedia-owl:position ?var3 ;
                dbpedia-owl:number 8 .
        }

}
LIMIT 100
```

Query 6.

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var3 ?var5 ?var7
WHERE { ?var3 rdf:type <http://dbpedia.org/class/yago/Company108058098> .
        ?var3 dbpedia-owl:numberOfEmployees ?var5
        FILTER ( ?var5 > 100 ) .
        ?var3 foaf:homepage ?var7 .
}
```

Query 7.

```
SELECT DISTINCT ?var0 ?var2 ?var3 ?var4 ?var5
WHERE {
        ?s <http://xmlns.com/foaf/0.1/homepage> ?var0 .
        ?s <http://dbpedia.org/ontology/location> <http://dbpedia.org/resource/Cannes> .
        ?s <http://dbpedia.org/ontology/startDate> ?var2 .
        ?s <http://dbpedia.org/ontology/endDate> ?var3 .
        ?s <http://dbpedia.org/ontology/openingFilm> ?var4 .
        ?s <http://dbpedia.org/ontology/closingFilm> ?var5 .
```

```
}
LIMIT 20
```

Query 8.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
SELECT ?var2 ?var4
WHERE {
        { ?var2 rdf:type <http://dbpedia.org/class/yago/UrbanArea108675967> .
                ?var2 dbpedia-owl:populationRural ?var4.
                FILTER (?var4 > 2000)
        } UNION {
                ?var2 rdf:type <http://dbpedia.org/class/yago/UrbanArea108675967>.
                ?var2 dbpedia-owl:populationUrban ?var4.

                FILTER (?var4 > 4000)
        }
}
```

Query 9.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?var3 ?var4 ?var5
WHERE {
        {  ?var1 foaf:name ?var3 .
                ?var1 dbpedia-owl:numberOfPostgraduateStudents ?var4.
                FILTER (?var4 > 10000)
                ?var1 foaf:homepage ?var5
        } UNION {
                ?var2 foaf:name ?var3.
                ?var2 dbpedia-owl:numberOfGraduateStudents ?var4.
                FILTER (?var4 < 200)
                OPTIONAL {?var2 foaf:homepage ?var5 .}
        }
}
```

Query 10.

```
SELECT DISTINCT ?var0 ?var1 ?var2 ?var3 ?var4 ?var5 ?var6 ?var7
WHERE {
        ?var0 <http://dbpedia.org/ontology/numberOfUndergraduateStudents> ?var1 .
        ?var0 <http://dbpedia.org/ontology/numberOfPostgraduateStudents> ?var2 .
        OPTIONAL {?var0 <http://dbpedia.org/ontology/mascot> ?var3. }
        OPTIONAL {?var0 <http://dbpedia.org/ontology/staff> ?var4. }
        OPTIONAL{?var0 <http://dbpedia.org/ontology/president> ?var5.}
        OPTIONAL {?var0 <http://www.w3.org/2003/01/geo/wgs84_pos#lat> ?var6. }
        OPTIONAL {?var0 <http://www.w3.org/2003/01/geo/wgs84_pos#long> ?var7. }
}
```

Appendix C

LDBC Datagen

The LDBC SNB data generator (DATAGEN) evolved from the S3G2 generator [159] and simulates the user's activity in a social network during a period of time. Its schema has 11 entities connected by 20 relations, with attributes of different types and values, making for a rich benchmark dataset. The main entities are: Persons, Tags, Forums, Messages (Posts, Comments and Photos), Likes, Organizations, and Places.

The dataset forms a graph that is a *fully connected component* of persons over their *friendship relationships*. Each person has a few forums under which the messages form large discussion *trees*. The messages are further connected to posts by authorship but also likes. These data elements scale linearly with the amount of friendships (people having more friends are likely more active and post more messages). Organization and Place information are more dimension-like and do not scale with the amount of persons or time. Time is an implicit dimension (there is no separate time entity) but is present in many timestamp attributes.

C.0.1 Correlated Attribute Values

An important novelty in DATAGEN is the ability to produce a highly correlated social network graph, in which attribute values are correlated among themselves and also influence the connection patterns in the social graph. Such correlations clearly occur in real graphs and influence the complexity of algorithms operating on the graph.

A full list of attribute correlations is given in Table C.1. For instance, the top row in the table states that the place where a person was born and gender influence the first name distribution. An example is shown in Table C.2, which shows the top-10 most occurring first names for people from Germany vs China. The actual set of attribute values is taken from DBpedia, which also is used as a source for many other attributes. Similarly, the location where a person lives influences his/her interests (a set of tags), which in turn influences the topic of the discussions (s)he opens (i.e., Posts), which finally also influences the text of the messages in the discussion. This is implemented by using the text taken from DBpedia pages closely related to a topic as the text used in the discussion (original post and comments on

(person.location, person.gender)	person.firstName *(typical names)*
	person.interests *(popular artist)*
person.location	person.lastName *(typical names)*
	person.university *(nearby universities)*
	person.company *(in country)*
	person.languages *(spoken in country)*
person.language	person.forum.post.language *(speaks)*
person.interests	person.forum.post.topic *(in)*
post.topic	post.text *(DBpedia article lines)*
	post.comment.text *(DBpedia article lines)*
person.employer	person.email *(@company, @university)*
post.photoLocation	post.location.latitude *(matches location)*
	post.location.longitude *(matches location)*
person.birthDate	person.createdDate *(>)*
person.createdDate	person.forum.message.createdDate *(>)*
	person.forum.createdDate *(>)*
forum.createdDate	post.photoTime *(>)*
	forum.post.createdDate *(>)*
	forum.groupmembership.joinedDate *(>)*
post.createdDate	post.comment.createdDate *(>)*

Table C.1: Attribute Value Correlations: left determines right

Name	Number
Karl	215
Hans	190
Wolfgang	174
Fritz	159
Rudolf	159
Walter	150
Franz	115
Paul	109
Otto	99
Wilhelm	74

Name	Number
Yang	961
Chen	929
Wei	887
Lei	789
Jun	779
Jie	778
Li	562
Hao	533
Lin	456
Peng	448

Table C.2: Top-10 person.firstNames (SF=10) for persons with person.location=Germany (left) or China (right).

it).

Person location also influences last name, university, company and languages. This influence is not full, there are Germans with Chinese names, but these are infrequent. In fact, the shape of the attribute value distributions is equal (and skewed), but the order of the values from the *value dictionaries* used in the distribution, changes depending on the correlation parameters (e.g. location).

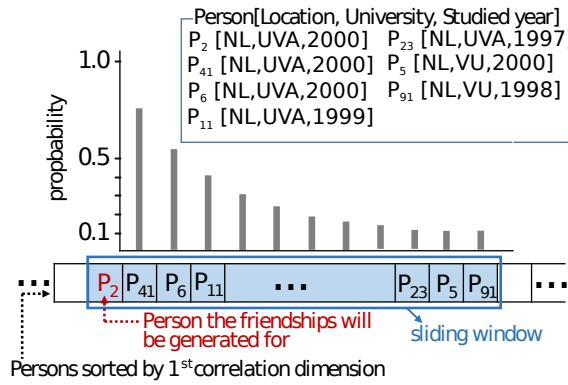


Figure C.1: Friendships generation (NL: The Netherlands, UVA: University of Amsterdam, VU: Vrij University)

C.0.2 Time Correlation and Spiking Trends

Almost all entities in the SNB dataset have timestamp attributes, since time is an important phenomenon in social networks. The latter correlation rules in Table C.1 are related to time, and ensure that events in the social network follow a logical order: e.g., people can post a comment only after becoming a friend with someone, and that can only happen after both persons joined the network.

The volume of person activity in a real social network, i.e., number of messages created per unit of time, is not uniform, but driven by real world events such as elections, natural disasters and sport competitions. Whenever an important real world event occurs, the amount of people and messages talking about that topic spikes – especially from those persons interested in that topic. We introduced this in DATAGEN by simulating events related to certain tags, around which the frequency of posts by persons interested in that tag is significantly higher (the topic is "trending"). Figure C.2(a) shows the density of posts over time with and without event-driven post generation, for SF=10. When event driven post generation is enabled, the density is not uniform but spikes of different magnitude appear, which correspond to events of different levels of importance. The activity volume around an event is implemented as proposed in [127].

C.0.3 Structure Correlation: Friendships

The "Homophily Principle" [141] states that similar people have a higher probability to be connected. This is modeled by DATAGEN by making the probability that people are connected dependent on their characteristics (attributes). This is implemented by a multi-stage edge generation process over two **correlation dimensions**: (i) *places where people studied* and (ii) *interests of persons*.

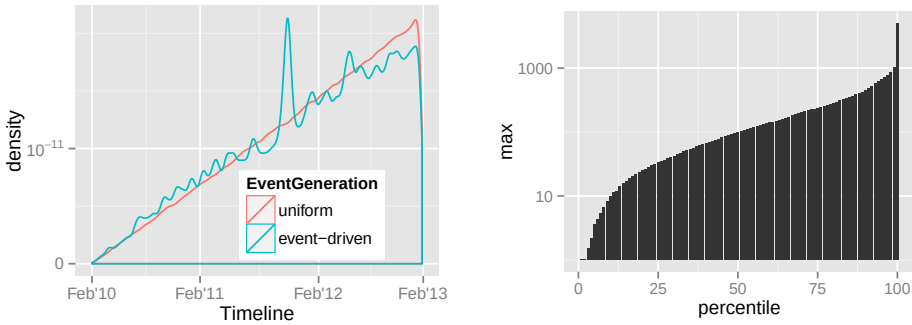
In other words, people that are interested in a topic and/or have studied in the

same university at the same year, have a larger probability to be friends. Furthermore, in order to reproduce the inhomogeneities found in real data, a third dimension consisting of a random number is also used.

In each edge generation stage the persons are re-sorted on one dimension (first stage: study location, second: interests, last: random). Each worker processes a disjunct range of these persons sequentially, keeping a window of the persons in memory – the entire range does not have to fit – and picks friends from the window using a geometric probability distribution that decreases with distance in the window. The probability for generating a connection during this stage drops from very low at window boundary to zero outside it (since the generator is not even capable of generating a friendship to data dropped from its window). All this makes the complex task of generating correlated friendship edges scalable, as it now only depends on parallel sorting and sequential processing with limited memory. We note that one dimension may have the form of multiple single-dimensional values bitwise appended. In the particular case of the *studied location*, these are the Z-order location of the university's city (bits 31-24), the university ID (bits 23-12), and the studied year (bits 11-0). This is exemplified at Figure C.1 where we show a sliding window along the first correlation dimension (i.e., *studied location*). As shown in this figure, those persons closer to person P_2 (the person generating friends for) according to the first dimension (e.g., P_{41}, P_6) have a higher probability to be friends of P_2.

The correlations in the friends graph also propagate to the messages. A person location influences on the one hand interests and studied location, so one gets many more like-minded or local friends. These persons typically have many more common interests (tags), which become the topic of posts and comment messages.

The *number* of friendship edges generated per person (friendship degree) is skewed [77]. DATAGEN discretizes the power law distribution given by Facebook graph [178], but scales this according to the size of the network. Because in smaller networks, the amount of "real" friends that is a member and to which one can connect is lower, we adjust the mean average degree logarithmically in terms of person membership, such that it becomes (somewhat) lower for smaller networks. A target average degree of the friendship graph is chosen using the following formula: $avg\_degree = n^{0.512-0.028 \cdot log(n)}$, where n is the number of persons in the graph. That is, when the size of the SNB dataset would be that of Facebook (i.e. 700M persons) the average friendship degree would be around 200. Then, each person is first assigned to a percentile p in the Facebook's degree distribution and second, a target degree uniformly distributed between the minimum and the maximum degrees at percentile p. Figure C.2(b) shows the maximum degree per percentile of the Facebook graph, used in DATAGEN. Finally, the person's target degree is scaled by multiplying it by a factor resulting from dividing avg_degree by the average degree of the real Facebook graph. Figure C.3(a) shows the friendship degree distribution for SF=10. Finally, given a person, the number of friendship edges for each correlation dimension is distributed as follows: 45%, 45% and 10% out of the target degree, for the first, the second and the third correlation dimension, respectively.



(a) (b)

Figure C.2: (a) Post distribution over time for event-driven vs uniform post generation on SF=10. (b) Maximum degree of each percentile in the Facebook graph.

SFs	Number of entities (x 1000000)					
	Nodes	Edges	Persons	Friends	Messages	Forums
30	99.4	655.4	0.18	14.2	97.4	1.8
100	317.7	2154.9	0.50	46.6	312.1	5.0
300	907.6	6292.5	1.25	136.2	893.7	12.6
1000	2930.7	20704.6	3.60	447.2	2890.9	36.1

Table C.3: SNB dataset statistics at different Scale Factors

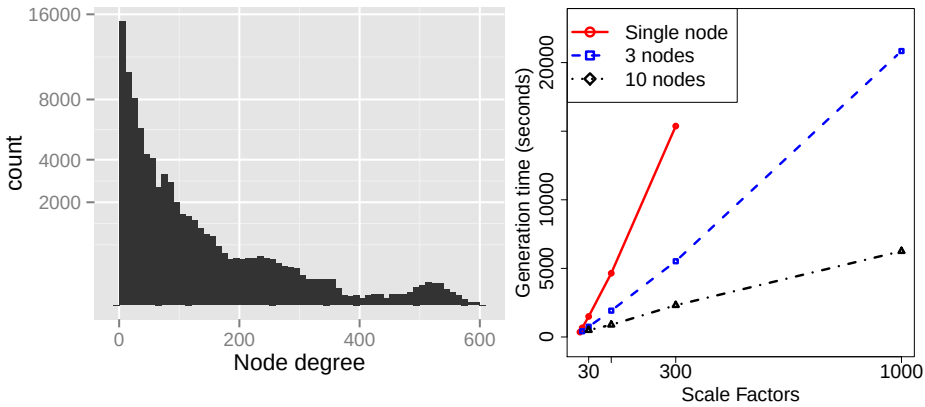
C.0.4 Scales & Scaling

DATAGEN can generate social networks of arbitrary size, however for the benchmarks we work with standard scale-factors (SF) valued 1,3,10,30,.. as indicated in Table C.3. The scale is determined by setting the amount of persons in the network, yet the scale factor is the amount of GB of uncompressed data in comma separated value (CSV) representation. DATAGEN can also generate RDF data in Ntriple[1] format, which is much more verbose.

DATAGEN is implemented on top of Hadoop to provide scalability. Data generation is performed in three steps, each of them composed of more MapReduce jobs.

person generation: In this step, the people of the social network are generated, including the personal information, interests, universities where they studied and companies where they worked at. Each mapper is responsible of generating a subset of the persons of the network.

[1]When generating URIs that identify entities, we ensure that URIs for the same kind of entity (e.g. person) have an order that follows the time dimension. This is done by encoding the timestamp (e.g. when the user joined the network) in the URI string in an order-preserving way. This is important for URI compression in RDF systems where often a correlation between such identifying URIs and time is present, yet it is not trivial to realize since we generate data in correlation dimension order, not logical time order.



(a) (b)

Figure C.3: (a) Friendship degree distribution for scale factor 10. (b) DATAGEN scale-up.

friendship generation: As explained above, friendship generation is split into a succession of stages, each of them based on a different correlation dimension. Each of these stages consists of two MapReduce jobs. The first is responsible for sorting the persons by the given correlation dimension. The second receives the sorted people and performs the sliding window process explained above.

person activity generation: this involves filling the forums with posts comments and likes. This data is mostly tree-structured and is therefore easily parallelized by the person who owns the forum. Each worker needs the attributes of the owner (e.g. interests influence post topics), the friend list (only friends post comments and likes) with the friendship creation timestamps (they only post after that); but otherwise the workers can operate independently.

We have paid specific attention to making data generation *deterministic*. This means that regardless the Hadoop configuration parameters (#node, #map and #reduce tasks) the generated dataset is always the same.

On a single 4-core machine (Intel i7-2600K@3.4GHz, 16GB RAM) that runs MapReduce in "pseudo-distributed" mode – where each CPU core runs a mapper or reducer – one can generate a SF=30 in 20 minutes. For larger scale factors it is recommended to use a true cluster; SF=1000 can be generated within 2 hours with 10 such machines connected with Gigabit ethernet (see Figure C.3(b)).

Bibliography

[1] Allegrograph. www.franz.com/agraph/allegrograph/.

[2] Amazon simpledb. https://aws.amazon.com/simpledb/.

[3] Apache accumulo. https://accumulo.apache.org/.

[4] Apache cassandra. cassandra.apache.org.

[5] Apache hbase. https://hbase.apache.org.

[6] Apache titan. http://titan.thinkaurelius.com.

[7] Bigdata. http://www.systap.com/bigdata.htm.

[8] Bigowlim. www.ontotext.com/owlim/.

[9] Blaze graph. https://www.blazegraph.com.

[10] D-gap compression scheme.

[11] Eclipse rdf4j. rdf4j.org.

[12] Freebase. https://developers.google.com/freebase/.

[13] Graphdb. http://graphdb.ontotext.com.

[14] Infinitegraph. http://www.objectivity.com.

[15] Jena tdb. jena.apache.org.

[16] Ldbc datagen. https://github.com/ldbc/ldbc_snb_datagen.

[17] Ldbc snb. http://ldbcouncil.org/benchmarks/snb.

[18] Marklogic. www.marklogic.com.

[19] Metis software. http://glaros.dtc.umn.edu/gkhome/views/metis.

[20] Monetdb. https://www.monetdb.org/.

[21] Mysql. www.mysql.com.

[22] Neo4j. neo4j.org.

[23] Oracle database semantic technologies. http://www.oracle.com.

[24] Oracle spatial and graph. http://www.oracle.com/technetwork/database/options/spatialandgraph.

[25] Orientdb. orientdb.com.

[26] Postgresql. www.postgresql.org.

[27] Social network interlligence benchmark. https://www.w3.org/wiki/Social_Network_Intelligence_BenchMark.

[28] Sparksee graph database. http://www.sparsity-technologies.com.

[29] Stardog. www.stardog.com.

[30] Trree engine. https://ontotext.com/trree.

[31] Virtuoso. http://virtuoso.openlinksw.com/.

[32] Jans Aasman. Allegro graph: Rdf triple database. *Cidade: Oakland Franz Incorporated*, 2006.

[33] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. *The design and implementation of modern column-oriented database systems*. Now, 2013.

[34] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[35] Riham Abdel Kader, Peter Boncz, Stefan Manegold, and Maurice Van Keulen. Rox: run-time optimization of xqueries. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 615–626. ACM, 2009.

[36] R. Agrawal et al. Fast algorithms for mining association rules. In *VLDB*, 1994.

[37] Y.Y. Ahn, S. Han, H. Kwak, S. Moon, and H. Jeong. Analysis of topological characteristics of huge online social networking services. In *Proc. WWW*, 2007.

[38] Sofia Alexaki, Vassilis Christophides, Gregory Karvounarakis, Dimitris Plexousakis, and Karsten Tolle. The ics-forth rdfsuite: Managing voluminous rdf description bases. In *Proceedings of the Second International Conference on Semantic Web-Volume 40*, pages 1–13. CEUR-WS. org, 2001.

[39] Güneş Aluç, M Tamer Özsu, and Khuzaima Daudjee. Workload matters: Why rdf databases need a new design. *Proceedings of the VLDB Endowment*, 7(10):837–840, 2014.

[40] Andrés Aranda-Andújar, Francesca Bugiotti, Jesús Camacho-Rodríguez, Dario Colazzo, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. Amada: web data repositories in the amazon cloud. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2749–2751. ACM, 2012.

[41] M. Arenas et al. A principled approach to bridging the gap between graph data and their schemas. In *VLDB*, 2014.

[42] Mario Arias, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. *arXiv preprint arXiv:1103.5043*, 2011.

[43] Medha Atre, Jagannathan Srinivasan, and James A Hendler. Bitmat: A main memory rdf triple store. *Proc. of SSWS 2009*, pages 33–48, 2009.

[44] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A nucleus for a web of open data. *Semantic Web Journal*, pages 722–735, 2007.

[45] Liu Baolin and Hu Bo. Hprd: a high performance rdf database. In *IFIP International Conference on Network and Parallel Computing*, pages 364–374. Springer, 2007.

[46] A.L. Barabási, R. Albert, and H. Jeong. Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics and its Applications*, 281(1-4):69–77, 2000.

[47] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

[48] David Beckett. The design and implementation of the redland rdf application framework. *Computer Networks*, 39(5):577–588, 2002.

[49] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *Proc. SIGCOMM*, 2009.

[50] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[51] Mark Birbeck et al. Curie syntax 1.0, a syntax for expressing compact uris. *W3C Recommendations*, (curie):1–7, 2007.

[52] Paul Biron, Ashok Malhotra, World Wide Web Consortium, et al. Xml schema part 2: Datatypes. *World Wide Web Consortium Recommendation REC-xmlschema-2-20041028*, 2004.

[53] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. Owlim: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011.

[54] C BIZER. Linked data-the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[55] C. Bizer and A. Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst*, 5(2):1–24, 2009.

[56] Peter Bloem and Steven de Rooij. Finding network motifs in large graphs using compression as a measure of relevance. *arXiv preprint arXiv:1701.02026*, 2017.

[57] K. Boitmanis et al. Fast and simple approximation of the diameter and radius of a graph. In *Experimental Algorithms*, pages 98–108. Springer, 2006.

[58] A. Bonato, J. Janssen, and P. Prałat. A geometric model for on-line social networks. In *Proc. Conf. on Online Social networks*, 2010.

[59] Peter Boncz, Orri Erling, and Minh-Duc Pham. Advances in large-scale rdf data management. In *Linked Open Data–Creating Knowledge Out of Interlinked Data*, pages 21–44. Springer, 2014.

[60] Peter Boncz, Orri Erling, and Minh-Duc Pham. Experiences with virtuoso cluster rdf column store. In *Linked Data Management*, pages 239–259. Chapman and Hall/CRC, 2014.

[61] Peter A Boncz, Martin L Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Communications of the ACM*, 51(12):77–85, 2008.

[62] A. Bornea et al. Building an efficient RDF store over a relational database. In *SIGMOD*, 2013.

[63] Mihaela A Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 121–132. ACM, 2013.

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

[65] Dan Brickley, Ramanathan V Guha, and Brian McBride. Rdf vocabulary description language 1.0: Rdf schema. w3c recommendation (2004). *URL http://www. w3. org/tr/2004/rec-rdf-schema-20040210*, 2004.

[66] Andreas Brodt, Oliver Schiller, and Bernhard Mitschang. Efficient resource attribute retrieval in rdf triple stores. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1445–1454. ACM, 2011.

[67] Jeen Broekstra and Arjohn Kampman. Serql: a second generation rdf query language. In *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, pages 13–14, 2003.

[68] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *International semantic web conference*, pages 54–68. Springer, 2002.

[69] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: An architecture for storing and querying rdf data and schema information. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, 197, 2003.

[70] D. Burdick et al. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, 2001.

[71] S. Campinas et al. Introducing RDF graph summary with application to assisted SPARQL formulation. In *DEXA Workshops*, 2012.

[72] Stephane Campinas, Thomas E Perry, Diego Ceccarelli, Renaud Delbru, and Giovanni Tummarello. Introducing rdf graph summary with application to assisted sparql formulation. In *2012 23rd International Workshop on Database and Expert Systems Applications*, pages 261–266. IEEE, 2012.

[73] Jeremy J Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83. ACM, 2004.

[74] Jeremy J Carroll and Graham Klyne. Resource description framework ({RDF}): Concepts and abstract syntax. 2004.

[75] Rick Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[76] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An efficient sql-based rdf querying scheme. In *Proceedings of the 31st international conference on Very large data bases*, pages 1216–1227. VLDB Endowment, 2005.

[77] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[78] D Connolly, F Harmelen, I Horrocks, DL McGuinnes, PF Patel-Schneider, and L Andrea Stein. Daml+ oil reference description. march 2001. Technical report, W3C Note 18 December 2001. http://www. w3. org/TR/2001/NOTE-daml+ oil-reference-20011218, 2009.

[79] World Wide Web Consortium et al. Owl 2 web ontology language document overview. 2012.

[80] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[81] LP Danh, DT Minh, P Minh Duc, PA Boncz, E Thomas, F Michael, et al. Linked stream data processing: Facts and figures. 2012.

[82] I. de Sola Pool and M. Kochen. *Contacts and influence*. Elsevier, 1978.

[83] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[84] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[85] Christos Doulkeridis and Kjetil Nørvåg. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal?The International Journal on Very Large Data Bases*, 23(3):355–380, 2014.

[86] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *SIGMOD*, pages 145–156. ACM, 2011.

[87] Orri Erling. Virtuoso, a hybrid RDBMS/graph column store. *IEEE Data Eng. Bull*, 2012.

[88] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 619–630. ACM, 2015.

[89] Orri Erling and Ivan Mikhailov. Integrating open sources and relational data with sparql. *The Semantic Web: Research and Applications*, pages 838–842, 2008.

[90] Orri Erling and Ivan Mikhailov. Towards web scale rdf. *Proc. SSWS*, 2008.

[91] Orri Erling and Ivan Mikhailov. Virtuoso: Rdf support in a native rdbms. In *Semantic Web Information Management*, pages 501–519. Springer, 2010.

[92] David C Faye, Olivier Curé, and Guillaume Blin. A survey of rdf storage approaches. *Arima Journal*, 15:11–35, 2012.

[93] George HL Fletcher and Peter W Beck. Scalable indexing of rdf graphs for efficient join processing. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1513–1516. ACM, 2009.

[94] I. Foudalis, K. Jain, C. Papadimitriou, and M. Sideri. Modeling social networks through user background and behavior. *Algorithms and Models for the Web Graph*, pages 85–102, 2011.

[95] Luis Galarraga, Katja Hose, and Ralf Schenkel. Partout: a distributed engine for efficient rdf processing. In *Proceedings of the 23rd International Conference on World Wide Web*, pages 267–268. ACM, 2014.

[96] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. An empirical study of real-world sparql queries. In *1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) at the 20th International World Wide Web Conference (WWW 2011), Hydebarabad, India*, 2011.

[97] Aurona Gerber, Alta Van der Merwe, and Andries Barnard. A functional semantic web architecture. In *European Semantic Web Conference*, pages 273–287. Springer, 2008.

[98] K. Gouda and M Zaki. Efficiently mining maximal frequent itemsets. In *ICDM*, 2001.

[99] Andrey Gubichev and Thomas Neumann. Exploiting the query structure for efficient join ordering in sparql queries. In *EDBT*, volume 14, pages 439–450, 2014.

[100] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

[101] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300. ACM, 2014.

[102] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk rdf storage. In *The First International Workshop on Practical and Scalable Semantic Systems*, 2003.

[103] Stephen Harris and Nigel Shadbolt. Sparql query processing with conventional relational database systems. In *International Conference on Web Information Systems Engineering*, pages 235–244. Springer, 2005.

[104] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The design and implementation of a clustered rdf store. In *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, page 81.

[105] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The design and implementation of a clustered rdf store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, pages 94–109, 2009.

[106] Steve Harris, Andy Seaborne, and Eric Prudhommeaux. Sparql 1.1 query language. *W3C recommendation*, 21(10), 2013.

[107] Andreas Harth and Stefan Decker. Optimized index structures for querying rdf from the web. In *Web Congress, 2005. LA-WEB 2005. Third Latin American*, pages 10–pp. IEEE, 2005.

[108] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. Yars2: a federated repository for querying graph structured data from the web. In *Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, pages 211–224. Springer-Verlag, 2007.

[109] Oktie Hassanzadeh, Anastasios Kementsietsidis, and Yannis Velegrakis. Data management issues on the semantic web. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1204–1206. IEEE, 2012.

[110] Ivan Herman. Sparql is a recommendation. *W3C Semantic Web Activity News*, 2008.

[111] Katja Hose and Ralf Schenkel. Warp: Workload-aware replication and partitioning for rdf. In *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on*, pages 1–6. IEEE, 2013.

[112] Jiewen Huang, Daniel J Abadi, and Kun Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.

[113] Mohammad Husain, James McGlothlin, Mohammad M Masud, Latifur Khan, and Bhavani M Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1312–1327, 2011.

[114] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Conference on Innovative Data Systems Research (CIDR)*, pages 68–78, Asilomar, California, 2007.

[115] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. Database cracking. In *CIDR*, volume 7, pages 68–78, 2007.

[116] Milena G Ivanova, Martin L Kersten, Niels J Nes, and Romulo AP Gonçalves. An architecture for recycling intermediates in a column-store. *ACM Transactions on Database Systems (TODS)*, 35(4):24, 2010.

[117] Maciej Janik and Krys Kochut. Brahms: a workbench rdf store and high performance memory system for semantic association discovery. In *International Semantic Web Conference*, pages 431–445. Springer, 2005.

[118] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 29. ACM, 1995.

[119] Atanas Kiryakov. Owlim: balancing between scalable repository and lightweight reasoner. *Proc. of WWW2006, Edinburgh, Scotland*, 2006.

[120] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. Owlim–a pragmatic semantic repository for owl. In *Web Information Systems Engineering–WISE 2005 Workshops*, pages 182–192. Springer, 2005.

[121] Dave Kolas, Ian Emmons, and Mike Dean. Efficient linked-list rdf indexing in parliament. *SSWS*, 9:17–32, 2009.

[122] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proc. WWW*, 2010.

[123] Günter Ladwig and Andreas Harth. Cumulusrdf: linked data management on nested key-value stores. In *The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011)*, volume 30, 2011.

[124] Jonathan K Lawder and Peter JH King. Using space-filling curves for multi-dimensional indexing. In *British National Conference on Databases*, pages 20–35. Springer, 2000.

[125] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[126] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. *PKDD*, 2005.

[127] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. Meme-tracking and the dynamics of the news cycle. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 497–506. ACM, 2009.

[128] J. Levandoski and M. Mokbel. RDF data-centric storage. In *ICWS*, 2009.

[129] Yuefeng Li et al. Mining ontology for automatically acquiring web user information needs. *KDE*, 2006.

[130] Li Ma, Zhong Su, Yue Pan, Li Zhang, and Tao Liu. Rstar: An rdf storage and query system for enterprise resource management. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 484–491. ACM, 2004.

[131] Frank Manola, Eric Miller, Brian McBride, et al. Rdf primer. *W3C recommendation*, 10(1-107):6, 2004.

[132] Miguel A Martínez-Prieto, Javier D Fernández, and Rodrigo Cánovas. Compression of rdf dictionaries. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 340–347. ACM, 2012.

[133] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. Uniform resource identifier (uri): Generic syntax. 2005.

[134] Akiyoshi Matono and Isao Kojima. Paragraph tables: A storage scheme based on RDF document structure. In *DEXA*, pages 231–247. Springer, 2012.

[135] Brian McBride. Jena: Implementing the rdf model and syntax specification. In *Proceedings of the Second International Conference on Semantic Web-Volume 40*, pages 23–28. CEUR-WS. org, 2001.

[136] Brian McBride. Jena: A semantic web toolkit. *IEEE Internet computing*, 6(6):55–59, 2002.

[137] Brian McBride. The resource description framework (rdf) and its vocabulary description language rdfs. In *Handbook on ontologies*, pages 51–65. Springer, 2004.

[138] JAMES McGlothlin and L Khan. Rdfjoin: A scalable data model for persistence and efficient querying of rdf datasets. *Database*, 2009.

[139] James P McGlothlin and Latifur R Khan. Rdfkb: efficient support for rdf inference queries and knowledge management. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, pages 259–266. ACM, 2009.

[140] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[141] Miller McPherson, Lynn Smith-Lovin, and James M Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, 27(1):415–444, 2001.

[142] Alistair Miles and Sean Bechhofer. Skos simple knowledge organization system reference. 2009.

[143] S. Milgram. The small world problem. *Psychology today*, 2(1):60–67, 1967.

[144] A. Mislove, M. Marcon, K.P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proc. SIGCOMM*, 2007.

[145] Gianfranco E Modoni, Marco Sacco, and Walter Terkaj. A survey of rdf store solutions. In *Engineering, Technology and Innovation (ICE), 2014 International ICE Conference on*, pages 1–7. IEEE, 2014.

[146] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd international conference on Very large data bases*, pages 930–941. VLDB Endowment, 2006.

[147] Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. Dbpedia sparql benchmark–performance assessment with real queries on real data. *The Semantic Web–ISWC 2011*, pages 454–469, 2011.

[148] Sivaramakrishnan Narayanan, Tahsin Kurc, and Joel Saltz. Dbowl: Towards extensional queries on a billion statements using relational databases.

[149] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.

[150] Thomas Neumann and Gerhard Weikum. Rdf-3x: a risc-style engine for rdf. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.

[151] Thomas Neumann and Gerhard Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal?The International Journal on Very Large Data Bases*, 19(1):91–113, 2010.

[152] R. Neumayer et al. When simple is (more than) good enough: Effective semantic search with (almost) no semantics. In *Advances in Information Retrieval*. Springer, 2012.

[153] M Tamer Özsu. A survey of rdf data management systems. *Frontiers of Computer Science*, 3(10):418–432, 2016.

[154] L. Page et al. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford, 1999.

[155] Zhengxiang Pan and Jeff Heflin. Dldb: Extending relational databases to support semantic web queries. In *In PSSS*, 2003.

[156] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2rdf: adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st International Conference on World Wide Web*, pages 397–400. ACM, 2012.

[157] Linnea Passing. Recognizing, naming and exploring structure in RDF data. Master's thesis, Technische Universität München, 2014.

[158] Minh-Duc Pham and Peter Boncz. Exploiting emergent schemas to make rdf systems more efficient. In *International Semantic Web Conference*, pages 463–479. Springer, 2016.

[159] Minh-Duc Pham, Peter Boncz, and Orri Erling. S3g2: A scalable structure-correlated social graph generator. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 156–172. Springer, 2012.

[160] Minh-Duc Pham, Linnea Passing, Orri Erling, and Peter Boncz. Deriving an emergent relational schema from rdf data. In *Proceedings of the 24th International Conference on World Wide Web*, pages 864–874. ACM, 2015.

[161] Eric PrudHommeaux, Andy Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

[162] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: a scalable rdf triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, page 4. ACM, 2012.

[163] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill, 2003.

[164] Alberto Reggiori, Dirk-Willem van Gulik, and Zavisa Bjelogrlic. Indexing and retrieving semantic web resources: the rdfstore model. In *SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, pages 13–14, 2003.

[165] Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[166] Kurt Rohloff and Richard E Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In *Programming support innovations for emerging distributed applications*, page 4. ACM, 2010.

[167] Sherif Sakr and Ghazi Al-Naymat. Relational processing of rdf queries: a survey. *ACM SIGMOD Record*, 38(4):23–28, 2010.

[168] Gerard Salton and Michael J McGill. Introduction to modern information retrieval. 1983.

[169] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp^2bench: A sparql performance benchmark. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 222–233. IEEE, 2009.

[170] Lefteris Sidirourgos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store support for rdf data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2):1553–1563, 2008.

[171] Michael Sintek and Malte Kiesel. Rdfbroker: A signature-based high-performance rdf store. In *European Semantic Web Conference*, pages 363–377. Springer, 2006.

[172] Raffael Stein and Valentin Zacharias. Rdf on cloud number nine. In *4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic*, pages 11–23, 2010.

[173] M. Stillger, G. Lohman, V. Markl, and M. Kandil. Leo-db2's learning optimizer. In *Proc. VLDB*, 2001.

[174] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of*

the 31st international conference on Very large data bases, pages 553–564. VLDB Endowment, 2005.

[175] Thanh Tran, GUNTER Ladwig, and Sebastian Rudolph. Istore: efficient rdf data management using structure indexes for general graph structured data. *Institute AIFB, Karlsruhe Institute of Technology*, 2009.

[176] P. Treeratpituk and J. Callan. Automatically labeling hierarchical clusters. In *DGSNA*, 2006.

[177] Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. Heuristics-based query optimisation for sparql. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 324–335. ACM, 2012.

[178] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[179] J Ullman and J Widom. Database systems: The complete book, 2008.

[180] P. Venetis et al. Recovering semantics of tables on the web. In *VLDB*, 2011.

[181] Y. Wang et al. Flextable: using a dynamic relation model to store RDF data. In *DASFAA*, 2010.

[182] D.J. Watts and S.H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393(6684):440–442, 1998.

[183] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[184] K. Wilkinson. Jena property table implementation. Technical report, HP Labs, 2006.

[185] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient rdf storage and retrieval in jena2. In *Proceedings of the First International Conference on Semantic Web and Databases*, pages 120–139. CEUR-WS. org, 2003.

[186] C. Wilson, B. Boe, A. Sala, K.P.N. Puttaswamy, and B.Y. Zhao. User interactions in social networks and their implications. In *Proc. European Conference on Computer Systems*, 2009.

[187] David Wood, Paul Gearon, and Tom Adams. Kowari: A platform for semantic web storage and analysis. In *XTech 2005 Conference*, pages 05–0402, 2005.

[188] Buwen Wu, Hai Jin, and Pingpeng Yuan. Scalable saprql querying processing on large rdf data in cloud computing environment. In *Proceedings of the 2012 international conference on Pervasive Computing and the Networked World*, pages 631–646. Springer-Verlag, 2012.

[189] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. Eagre: Towards scalable i/o efficient sparql query evaluation on the cloud. In *Data engineering (ICDE), 2013 ieee 29th international conference on*, pages 565–576. IEEE, 2013.

[190] Xiaofei Zhang, Lei Chen, and Min Wang. Towards efficient join processing over large rdf graph using mapreduce. In *SSDBM*, pages 250–259. Springer, 2012.

[191] Ying Zhang, Pham Minh Duc, Oscar Corcho, and Jean-Paul Calbimonte. Srbench: a streaming rdf/sparql benchmark. In *International Semantic Web Conference*, pages 641–657. Springer, 2012.

[192] Marcin Zukowski, Peter A Boncz, et al. Vectorwise: Beyond column stores. 2012.

[193] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. Monetdb/x100-a dbms in the cpu cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.

SIKS Dissertation Series

2011 01 Botond Cseke (RUN), Variational Algorithms for Bayesian Inference in Latent Gaussian Models
02 Nick Tinnemeier (UU), Organizing Agent Organizations. Syntax and Operational Semantics of an Organization-Oriented Programming Language
03 Jan Martijn van der Werf (TUE), Compositional Design and Verification of Component-Based Information Systems
04 Hado van Hasselt (UU), Insights in Reinforcement Learning; Formal analysis and empirical evaluation of temporal-difference
05 Bas van der Raadt (VU), Enterprise Architecture Coming of Age - Increasing the Performance of an Emerging Discipline.
06 Yiwen Wang (TUE), Semantically-Enhanced Recommendations in Cultural Heritage
07 Yujia Cao (UT), Multimodal Information Presentation for High Load Human Computer Interaction
08 Nieske Vergunst (UU), BDI-based Generation of Robust Task-Oriented Dialogues
09 Tim de Jong (OU), Contextualised Mobile Media for Learning
10 Bart Bogaert (UvT), Cloud Content Contention
11 Dhaval Vyas (UT), Designing for Awareness: An Experience-focused HCI Perspective
12 Carmen Bratosin (TUE), Grid Architecture for Distributed Process Mining
13 Xiaoyu Mao (UvT), Airport under Control. Multiagent Scheduling for Airport Ground Handling
14 Milan Lovric (EUR), Behavioral Finance and Agent-Based Artificial Markets
15 Marijn Koolen (UvA), The Meaning of Structure: the Value of Link Evidence for Information Retrieval
16 Maarten Schadd (UM), Selective Search in Games of Different Complexity
17 Jiyin He (UVA), Exploring Topic Structure: Coherence, Diversity and Relatedness
18 Mark Ponsen (UM), Strategic Decision-Making in complex games
19 Ellen Rusman (OU), The Mind's Eye on Personal Profiles
20 Qing Gu (VU), Guiding service-oriented software engineering - A view-based approach
21 Linda Terlouw (TUD), Modularization and Specification of Service-Oriented Systems
22 Junte Zhang (UVA), System Evaluation of Archival Description and Access
23 Wouter Weerkamp (UVA), Finding People and their Utterances in Social Media
24 Herwin van Welbergen (UT), Behavior Generation for Interpersonal Coordination with Virtual Humans On Specifying, Scheduling and Realizing Multimodal Virtual Human Behavior

25 Syed Waqar ul Qounain Jaffry (VU), Analysis and Validation of Models for Trust Dynamics
26 Matthijs Aart Pontier (VU), Virtual Agents for Human Communication - Emotion Regulation and Involvement-Distance Trade-Offs in Embodied Conversational Agents and Robots
27 Aniel Bhulai (VU), Dynamic website optimization through autonomous management of design patterns
28 Rianne Kaptein (UVA), Effective Focused Retrieval by Exploiting Query Context and Document Structure
29 Faisal Kamiran (TUE), Discrimination-aware Classification
30 Egon van den Broek (UT), Affective Signal Processing (ASP): Unraveling the mystery of emotions
31 Ludo Waltman (EUR), Computational and Game-Theoretic Approaches for Modeling Bounded Rationality
32 Nees-Jan van Eck (EUR), Methodological Advances in Bibliometric Mapping of Science
33 Tom van der Weide (UU), Arguing to Motivate Decisions
34 Paolo Turrini (UU), Strategic Reasoning in Interdependence: Logical and Game-theoretical Investigations
35 Maaike Harbers (UU), Explaining Agent Behavior in Virtual Training
36 Erik van der Spek (UU), Experiments in serious game design: a cognitive approach
37 Adriana Burlutiu (RUN), Machine Learning for Pairwise Data, Applications for Preference Learning and Supervised Network Inference
38 Nyree Lemmens (UM), Bee-inspired Distributed Optimization
39 Joost Westra (UU), Organizing Adaptation using Agents in Serious Games
40 Viktor Clerc (VU), Architectural Knowledge Management in Global Software Development
41 Luan Ibraimi (UT), Cryptographically Enforced Distributed Data Access Control
42 Michal Sindlar (UU), Explaining Behavior through Mental State Attribution
43 Henk van der Schuur (UU), Process Improvement through Software Operation Knowledge
44 Boris Reuderink (UT), Robust Brain-Computer Interfaces
45 Herman Stehouwer (UvT), Statistical Language Models for Alternative Sequence Selection
46 Beibei Hu (TUD), Towards Contextualized Information Delivery: A Rule-based Architecture for the Domain of Mobile Police Work
47 Azizi Bin Ab Aziz (VU), Exploring Computational Models for Intelligent Support of Persons with Depression
48 Mark Ter Maat (UT), Response Selection and Turn-taking for a Sensitive Artificial Listening Agent
49 Andreea Niculescu (UT), Conversational interfaces for task-oriented spoken dialogues: design aspects influencing interaction quality

2012 01 Terry Kakeeto (UvT), Relationship Marketing for SMEs in Uganda
02 Muhammad Umair (VU), Adaptivity, emotion, and Rationality in Human and Ambient Agent Models
03 Adam Vanya (VU), Supporting Architecture Evolution by Mining Software Repositories
04 Jurriaan Souer (UU), Development of Content Management System-based Web Applications
05 Marijn Plomp (UU), Maturing Interorganisational Information Systems

06 Wolfgang Reinhardt (OU), Awareness Support for Knowledge Workers in Research Networks
07 Rianne van Lambalgen (VU), When the Going Gets Tough: Exploring Agent-based Models of Human Performance under Demanding Conditions
08 Gerben de Vries (UVA), Kernel Methods for Vessel Trajectories
09 Ricardo Neisse (UT), Trust and Privacy Management Support for Context-Aware Service Platforms
10 David Smits (TUE), Towards a Generic Distributed Adaptive Hypermedia Environment
11 J.C.B. Rantham Prabhakara (TUE), Process Mining in the Large: Preprocessing, Discovery, and Diagnostics
12 Kees van der Sluijs (TUE), Model Driven Design and Data Integration in Semantic Web Information Systems
13 Suleman Shahid (UvT), Fun and Face: Exploring non-verbal expressions of emotion during playful interactions
14 Evgeny Knutov (TUE), Generic Adaptation Framework for Unifying Adaptive Web-based Systems
15 Natalie van der Wal (VU), Social Agents. Agent-Based Modelling of Integrated Internal and Social Dynamics of Cognitive and Affective Processes.
16 Fiemke Both (VU), Helping people by understanding them - Ambient Agents supporting task execution and depression treatment
17 Amal Elgammal (UvT), Towards a Comprehensive Framework for Business Process Compliance
18 Eltjo Poort (VU), Improving Solution Architecting Practices
19 Helen Schonenberg (TUE), What's Next? Operational Support for Business Process Execution
20 Ali Bahramisharif (RUN), Covert Visual Spatial Attention, a Robust Paradigm for Brain-Computer Interfacing
21 Roberto Cornacchia (TUD), Querying Sparse Matrices for Information Retrieval
22 Thijs Vis (UvT), Intelligence, politie en veiligheidsdienst: verenigbare grootheden?
23 Christian Muehl (UT), Toward Affective Brain-Computer Interfaces: Exploring the Neurophysiology of Affect during Human Media Interaction
24 Laurens van der Werff (UT), Evaluation of Noisy Transcripts for Spoken Document Retrieval
25 Silja Eckartz (UT), Managing the Business Case Development in Inter-Organizational IT Projects: A Methodology and its Application
26 Emile de Maat (UVA), Making Sense of Legal Text
27 Hayrettin Gurkok (UT), Mind the Sheep! User Experience Evaluation & Brain-Computer Interface Games
28 Nancy Pascall (UvT), Engendering Technology Empowering Women
29 Almer Tigelaar (UT), Peer-to-Peer Information Retrieval
30 Alina Pommeranz (TUD), Designing Human-Centered Systems for Reflective Decision Making
31 Emily Bagarukayo (RUN), A Learning by Construction Approach for Higher Order Cognitive Skills Improvement, Building Capacity and Infrastructure
32 Wietske Visser (TUD), Qualitative multi-criteria preference representation and reasoning
33 Rory Sie (OUN), Coalitions in Cooperation Networks (COCOON)
34 Pavol Jancura (RUN), Evolutionary analysis in PPI networks and applications
35 Evert Haasdijk (VU), Never Too Old To Learn – On-line Evolution of Controllers in Swarm- and Modular Robotics
36 Denis Ssebugwawo (RUN), Analysis and Evaluation of Collaborative Modeling Processes

37 Agnes Nakakawa (RUN), A Collaboration Process for Enterprise Architecture Creation
38 Selmar Smit (VU), Parameter Tuning and Scientific Testing in Evolutionary Algorithms
39 Hassan Fatemi (UT), Risk-aware design of value and coordination networks
40 Agus Gunawan (UvT), Information Access for SMEs in Indonesia
41 Sebastian Kelle (OU), Game Design Patterns for Learning
42 Dominique Verpoorten (OU), Reflection Amplifiers in self-regulated Learning
43 Withdrawn
44 Anna Tordai (VU), On Combining Alignment Techniques
45 Benedikt Kratz (UvT), A Model and Language for Business-aware Transactions
46 Simon Carter (UVA), Exploration and Exploitation of Multilingual Data for Statistical Machine Translation
47 Manos Tsagkias (UVA), Mining Social Media: Tracking Content and Predicting Behavior
48 Jorn Bakker (TUE), Handling Abrupt Changes in Evolving Time-series Data
49 Michael Kaisers (UM), Learning against Learning - Evolutionary dynamics of reinforcement learning algorithms in strategic interactions
50 Steven van Kervel (TUD), Ontologogy driven Enterprise Information Systems Engineering
51 Jeroen de Jong (TUD), Heuristics in Dynamic Sceduling; a practical framework with a case study in elevator dispatching

2013 01 Viorel Milea (EUR), News Analytics for Financial Decision Support
02 Erietta Liarou (CWI), MonetDB/DataCell: Leveraging the Column-store Database Technology for Efficient and Scalable Stream Processing
03 Szymon Klarman (VU), Reasoning with Contexts in Description Logics
04 Chetan Yadati (TUD), Coordinating autonomous planning and scheduling
05 Dulce Pumareja (UT), Groupware Requirements Evolutions Patterns
06 Romulo Goncalves (CWI), The Data Cyclotron: Juggling Data and Queries for a Data Warehouse Audience
07 Giel van Lankveld (UvT), Quantifying Individual Player Differences
08 Robbert-Jan Merk (VU), Making enemies: cognitive modeling for opponent agents in fighter pilot simulators
09 Fabio Gori (RUN), Metagenomic Data Analysis: Computational Methods and Applications
10 Jeewanie Jayasinghe Arachchige (UvT), A Unified Modeling Framework for Service Design.
11 Evangelos Pournaras (TUD), Multi-level Reconfigurable Self-organization in Overlay Services
12 Marian Razavian (VU), Knowledge-driven Migration to Services
13 Mohammad Safiri (UT), Service Tailoring: User-centric creation of integrated IT-based homecare services to support independent living of elderly
14 Jafar Tanha (UVA), Ensemble Approaches to Semi-Supervised Learning Learning
15 Daniel Hennes (UM), Multiagent Learning - Dynamic Games and Applications
16 Eric Kok (UU), Exploring the practical benefits of argumentation in multi-agent deliberation
17 Koen Kok (VU), The PowerMatcher: Smart Coordination for the Smart Electricity Grid
18 Jeroen Janssens (UvT), Outlier Selection and One-Class Classification
19 Renze Steenhuizen (TUD), Coordinated Multi-Agent Planning and Scheduling
20 Katja Hofmann (UvA), Fast and Reliable Online Learning to Rank for Information Retrieval

21 Sander Wubben (UvT), Text-to-text generation by monolingual machine translation
22 Tom Claassen (RUN), Causal Discovery and Logic
23 Patricio de Alencar Silva (UvT), Value Activity Monitoring
24 Haitham Bou Ammar (UM), Automated Transfer in Reinforcement Learning
25 Agnieszka Anna Latoszek-Berendsen (UM), Intention-based Decision Support. A new way of representing and implementing clinical guidelines in a Decision Support System
26 Alireza Zarghami (UT), Architectural Support for Dynamic Homecare Service Provisioning
27 Mohammad Huq (UT), Inference-based Framework Managing Data Provenance
28 Frans van der Sluis (UT), When Complexity becomes Interesting: An Inquiry into the Information eXperience
29 Iwan de Kok (UT), Listening Heads
30 Joyce Nakatumba (TUE), Resource-Aware Business Process Management: Analysis and Support
31 Dinh Khoa Nguyen (UvT), Blueprint Model and Language for Engineering Cloud Applications
32 Kamakshi Rajagopal (OUN), Networking For Learning; The role of Networking in a Lifelong Learner's Professional Development
33 Qi Gao (TUD), User Modeling and Personalization in the Microblogging Sphere
34 Kien Tjin-Kam-Jet (UT), Distributed Deep Web Search
35 Abdallah El Ali (UvA), Minimal Mobile Human Computer Interaction
36 Than Lam Hoang (TUe), Pattern Mining in Data Streams
37 Dirk Börner (OUN), Ambient Learning Displays
38 Eelco den Heijer (VU), Autonomous Evolutionary Art
39 Joop de Jong (TUD), A Method for Enterprise Ontology based Design of Enterprise Information Systems
40 Pim Nijssen (UM), Monte-Carlo Tree Search for Multi-Player Games
41 Jochem Liem (UVA), Supporting the Conceptual Modelling of Dynamic Systems: A Knowledge Engineering Perspective on Qualitative Reasoning
42 Léon Planken (TUD), Algorithms for Simple Temporal Reasoning
43 Marc Bron (UVA), Exploration and Contextualization through Interaction and Concepts

2014 01 Nicola Barile (UU), Studies in Learning Monotone Models from Data
02 Fiona Tuliyano (RUN), Combining System Dynamics with a Domain Modeling Method
03 Sergio Raul Duarte Torres (UT), Information Retrieval for Children: Search Behavior and Solutions
04 Hanna Jochmann-Mannak (UT), Websites for children: search strategies and interface design - Three studies on children's search performance and evaluation
05 Jurriaan van Reijsen (UU), Knowledge Perspectives on Advancing Dynamic Capability
06 Damian Tamburri (VU), Supporting Networked Software Development
07 Arya Adriansyah (TUE), Aligning Observed and Modeled Behavior
08 Samur Araujo (TUD), Data Integration over Distributed and Heterogeneous Data Endpoints
09 Philip Jackson (UvT), Toward Human-Level Artificial Intelligence: Representation and Computation of Meaning in Natural Language
10 Ivan Salvador Razo Zapata (VU), Service Value Networks
11 Janneke van der Zwaan (TUD), An Empathic Virtual Buddy for Social Support

12 Willem van Willigen (VU), Look Ma, No Hands: Aspects of Autonomous Vehicle Control
13 Arlette van Wissen (VU), Agent-Based Support for Behavior Change: Models and Applications in Health and Safety Domains
14 Yangyang Shi (TUD), Language Models With Meta-information
15 Natalya Mogles (VU), Agent-Based Analysis and Support of Human Functioning in Complex Socio-Technical Systems: Applications in Safety and Healthcare
16 Krystyna Milian (VU), Supporting trial recruitment and design by automatically interpreting eligibility criteria
17 Kathrin Dentler (VU), Computing healthcare quality indicators automatically: Secondary Use of Patient Data and Semantic Interoperability
18 Mattijs Ghijsen (UVA), Methods and Models for the Design and Study of Dynamic Agent Organizations
19 Vinicius Ramos (TUE), Adaptive Hypermedia Courses: Qualitative and Quantitative Evaluation and Tool Support
20 Mena Habib (UT), Named Entity Extraction and Disambiguation for Informal Text: The Missing Link
21 Kassidy Clark (TUD), Negotiation and Monitoring in Open Environments
22 Marieke Peeters (UU), Personalized Educational Games - Developing agent-supported scenario-based training
23 Eleftherios Sidirourgos (UvA/CWI), Space Efficient Indexes for the Big Data Era
24 Davide Ceolin (VU), Trusting Semi-structured Web Data
25 Martijn Lappenschaar (RUN), New network models for the analysis of disease interaction
26 Tim Baarslag (TUD), What to Bid and When to Stop
27 Rui Jorge Almeida (EUR), Conditional Density Models Integrating Fuzzy and Probabilistic Representations of Uncertainty
28 Anna Chmielowiec (VU), Decentralized k-Clique Matching
29 Jaap Kabbedijk (UU), Variability in Multi-Tenant Enterprise Software
30 Peter de Cock (UvT), Anticipating Criminal Behaviour
31 Leo van Moergestel (UU), Agent Technology in Agile Multiparallel Manufacturing and Product Support
32 Naser Ayat (UvA), On Entity Resolution in Probabilistic Data
33 Tesfa Tegegne (RUN), Service Discovery in eHealth
34 Christina Manteli (VU), The Effect of Governance in Global Software Development: Analyzing Transactive Memory Systems.
35 Joost van Ooijen (UU), Cognitive Agents in Virtual Worlds: A Middleware Design Approach
36 Joos Buijs (TUE), Flexible Evolutionary Algorithms for Mining Structured Process Models
37 Maral Dadvar (UT), Experts and Machines United Against Cyberbullying
38 Danny Plass-Oude Bos (UT), Making brain-computer interfaces better: improving usability through post-processing.
39 Jasmina Maric (UvT), Web Communities, Immigration, and Social Capital
40 Walter Omona (RUN), A Framework for Knowledge Management Using ICT in Higher Education
41 Frederic Hogenboom (EUR), Automated Detection of Financial Events in News Text
42 Carsten Eijckhof (CWI/TUD), Contextual Multidimensional Relevance Models
43 Kevin Vlaanderen (UU), Supporting Process Improvement using Method Increments
44 Paulien Meesters (UvT), Intelligent Blauw. Met als ondertitel: Intelligence-gestuurde politiezorg in gebiedsgebonden eenheden.

45 Birgit Schmitz (OUN), Mobile Games for Learning: A Pattern-Based Approach
46 Ke Tao (TUD), Social Web Data Analytics: Relevance, Redundancy, Diversity
47 Shangsong Liang (UVA), Fusion and Diversification in Information Retrieval

---

2015 01 Niels Netten (UvA), Machine Learning for Relevance of Information in Crisis Response
02 Faiza Bukhsh (UvT), Smart auditing: Innovative Compliance Checking in Customs Controls
03 Twan van Laarhoven (RUN), Machine learning for network data
04 Howard Spoelstra (OUN), Collaborations in Open Learning Environments
05 Christoph Bösch (UT), Cryptographically Enforced Search Pattern Hiding
06 Farideh Heidari (TUD), Business Process Quality Computation - Computing Non-Functional Requirements to Improve Business Processes
07 Maria-Hendrike Peetz (UvA), Time-Aware Online Reputation Analysis
08 Jie Jiang (TUD), Organizational Compliance: An agent-based model for designing and evaluating organizational interactions
09 Randy Klaassen (UT), HCI Perspectives on Behavior Change Support Systems
10 Henry Hermans (OUN), OpenU: design of an integrated system to support lifelong learning
11 Yongming Luo (TUE), Designing algorithms for big graph datasets: A study of computing bisimulation and joins
12 Julie M. Birkholz (VU), Modi Operandi of Social Network Dynamics: The Effect of Context on Scientific Collaboration Networks
13 Giuseppe Procaccianti (VU), Energy-Efficient Software
14 Bart van Straalen (UT), A cognitive approach to modeling bad news conversations
15 Klaas Andries de Graaf (VU), Ontology-based Software Architecture Documentation
16 Changyun Wei (UT), Cognitive Coordination for Cooperative Multi-Robot Teamwork
17 André van Cleeff (UT), Physical and Digital Security Mechanisms: Properties, Combinations and Trade-offs
18 Holger Pirk (CWI), Waste Not, Want Not! - Managing Relational Data in Asymmetric Memories
19 Bernardo Tabuenca (OUN), Ubiquitous Technology for Lifelong Learners
20 Lois Vanhée (UU), Using Culture and Values to Support Flexible Coordination
21 Sibren Fetter (OUN), Using Peer-Support to Expand and Stabilize Online Learning
22 Zhemin Zhu (UT), Co-occurrence Rate Networks
23 Luit Gazendam (VU), Cataloguer Support in Cultural Heritage
24 Richard Berendsen (UVA), Finding People, Papers, and Posts: Vertical Search Algorithms and Evaluation
25 Steven Woudenberg (UU), Bayesian Tools for Early Disease Detection
26 Alexander Hogenboom (EUR), Sentiment Analysis of Text Guided by Semantics and Structure
27 Sándor Héman (CWI), Updating compressed colomn stores
28 Janet Bagorogoza (TiU), Knowledge Management and High Performance; The Uganda Financial Institutions Model for HPO
29 Hendrik Baier (UM), Monte-Carlo Tree Search Enhancements for One-Player and Two-Player Domains
30 Kiavash Bahreini (OU), Real-time Multimodal Emotion Recognition in E-Learning
31 Yakup Koç (TUD), On the robustness of Power Grids
32 Jerome Gard (UL), Corporate Venture Management in SMEs

33 Frederik Schadd (TUD), Ontology Mapping with Auxiliary Resources
34 Victor de Graaf (UT), Gesocial Recommender Systems
35 Jungxao Xu (TUD), Affective Body Language of Humanoid Robots: Perception and Effects in Human Robot Interaction

2016 01 Syed Saiden Abbas (RUN), Recognition of Shapes by Humans and Machines
02 Michiel Christiaan Meulendijk (UU), Optimizing medication reviews through decision support: prescribing a better pill to swallow
03 Maya Sappelli (RUN), Knowledge Work in Context: User Centered Knowledge Worker Support
04 Laurens Rietveld (VU), Publishing and Consuming Linked Data
05 Evgeny Sherkhonov (UVA), Expanded Acyclic Queries: Containment and an Application in Explaining Missing Answers
06 Michel Wilson (TUD), Robust scheduling in an uncertain environment
07 Jeroen de Man (VU), Measuring and modeling negative emotions for virtual training
08 Matje van de Camp (TiU), A Link to the Past: Constructing Historical Social Networks from Unstructured Data
09 Archana Nottamkandath (VU), Trusting Crowdsourced Information on Cultural Artefacts
10 George Karafotias (VUA), Parameter Control for Evolutionary Algorithms
11 Anne Schuth (UVA), Search Engines that Learn from Their Users
12 Max Knobbout (UU), Logics for Modelling and Verifying Normative Multi-Agent Systems
13 Nana Baah Gyan (VU), The Web, Speech Technologies and Rural Development in West Africa - An ICT4D Approach
14 Ravi Khadka (UU), Revisiting Legacy Software System Modernization
15 Steffen Michels (RUN), Hybrid Probabilistic Logics - Theoretical Aspects, Algorithms and Experiments
16 Guangliang Li (UVA), Socially Intelligent Autonomous Agents that Learn from Human Reward
17 Berend Weel (VU), Towards Embodied Evolution of Robot Organisms
18 Albert Meroño Peñuela (VU), Refining Statistical Data on the Web
19 Julia Efremova (Tu/e), Mining Social Structures from Genealogical Data
20 Daan Odijk (UVA), Context & Semantics in News & Web Search
21 Alejandro Moreno Célleri (UT), From Traditional to Interactive Playspaces: Automatic Analysis of Player Behavior in the Interactive Tag Playground
22 Grace Lewis (VU), Software Architecture Strategies for Cyber-Foraging Systems
23 Fei Cai (UVA), Query Auto Completion in Information Retrieval
24 Brend Wanders (UT), Repurposing and Probabilistic Integration of Data; An Iterative and data model independent approach
25 Julia Kiseleva (TU/e), Using Contextual Information to Understand Searching and Browsing Behavior
26 Dilhan Thilakarathne (VU), In or Out of Control: Exploring Computational Models to Study the Role of Human Awareness and Control in Behavioural Choices, with Applications in Aviation and Energy Management Domains
27 Wen Li (TUD), Understanding Geo-spatial Information on Social Media
28 Mingxin Zhang (TUD), Large-scale Agent-based Social Simulation - A study on epidemic prediction and control
29 Nicolas Höning (TUD), Peak reduction in decentralised electricity systems - Markets and prices for flexible planning
30 Ruud Mattheij (UvT), The Eyes Have It

31 Mohammad Khelghati (UT), Deep web content monitoring
32 Eelco Vriezekolk (UT), Assessing Telecommunication Service Availability Risks for Crisis Organisations
33 Peter Bloem (UVA), Single Sample Statistics, exercises in learning from just one example
34 Dennis Schunselaar (TUE), Configurable Process Trees: Elicitation, Analysis, and Enactment
35 Zhaochun Ren (UVA), Monitoring Social Media: Summarization, Classification and Recommendation
36 Daphne Karreman (UT), Beyond R2D2: The design of nonverbal interaction behavior optimized for robot-specific morphologies
37 Giovanni Sileno (UvA), Aligning Law and Action - a conceptual and computational inquiry
38 Andrea Minuto (UT), Materials that Matter - Smart Materials meet Art & Interaction Design
39 Merijn Bruijnes (UT), Believable Suspect Agents; Response and Interpersonal Style Selection for an Artificial Suspect
40 Christian Detweiler (TUD), Accounting for Values in Design
41 Thomas King (TUD), Governing Governance: A Formal Framework for Analysing Institutional Design and Enactment Governance
42 Spyros Martzoukos (UVA), Combinatorial and Compositional Aspects of Bilingual Aligned Corpora
43 Saskia Koldijk (RUN), Context-Aware Support for Stress Self-Management: From Theory to Practice
44 Thibault Sellam (UVA), Automatic Assistants for Database Exploration
45 Bram van de Laar (UT), Experiencing Brain-Computer Interface Control
46 Jorge Gallego Perez (UT), Robots to Make you Happy
47 Christina Weber (UL), Real-time foresight - Preparedness for dynamic innovation networks
48 Tanja Buttler (TUD), Collecting Lessons Learned
49 Gleb Polevoy (TUD), Participation and Interaction in Projects. A Game-Theoretic Analysis
50 Yan Wang (UVT), The Bridge of Dreams: Towards a Method for Operational Performance Alignment in IT-enabled Service Supply Chains

---

2017 01 Jan-Jaap Oerlemans (UL), Investigating Cybercrime
02 Sjoerd Timmer (UU), Designing and Understanding Forensic Bayesian Networks using Argumentation
03 Daniël Harold Telgen (UU), Grid Manufacturing; A Cyber-Physical Approach with Autonomous Products and Reconfigurable Manufacturing Machines
04 Mrunal Gawade (CWI), Multi-core Parallelism in a Column-store
05 Mahdieh Shadi (UVA), Collaboration Behavior
06 Damir Vandic (EUR), Intelligent Information Systems for Web Product Search
07 Roel Bertens (UU), Insight in Information: from Abstract to Anomaly
08 Rob Konijn (VU) , Detecting Interesting Differences:Data Mining in Health Insurance Data using Outlier Detection and Subgroup Discovery
09 Dong Nguyen (UT), Text as Social and Cultural Data: A Computational Perspective on Variation in Text
10 Robby van Delden (UT), (Steering) Interactive Play Behavior
11 Florian Kunneman (RUN), Modelling patterns of time and emotion in Twitter #anticipointment
12 Sander Leemans (TUE), Robust Process Mining with Guarantees

13 Gijs Huisman (UT), Social Touch Technology - Extending the reach of social touch through haptic technology
14 Shoshannah Tekofsky (UvT), You Are Who You Play You Are: Modelling Player Traits from Video Game Behavior
15 Peter Berck (RUN), Memory-Based Text Correction
16 Aleksandr Chuklin (UVA), Understanding and Modeling Users of Modern Search Engines
17 Daniel Dimov (UL), Crowdsourced Online Dispute Resolution
18 Ridho Reinanda (UVA), Entity Associations for Search
19 Jeroen Vuurens (UT), Proximity of Terms, Texts and Semantic Vectors in Information Retrieval
20 Mohammadbashir Sedighi (TUD), Fostering Engagement in Knowledge Sharing: The Role of Perceived Benefits, Costs and Visibility
21 Jeroen Linssen (UT), Meta Matters in Interactive Storytelling and Serious Gaming (A Play on Worlds)
22 Sara Magliacane (VU), Logics for causal inference under uncertainty
23 David Graus (UVA), Entities of Interest — Discovery in Digital Traces
24 Chang Wang (TUD), Use of Affordances for Efficient Robot Learning
25 Veruska Zamborlini (VU), Knowledge Representation for Clinical Guidelines, with applications to Multimorbidity Analysis and Literature Search
26 Merel Jung (UT), Socially intelligent robots that understand and respond to human touch
27 Michiel Joosse (UT), Investigating Positioning and Gaze Behaviors of Social Robots: People's Preferences, Perceptions and Behaviors
28 John Klein (VU), Architecture Practices for Complex Contexts
29 Adel Alhuraibi (UvT), From IT-BusinessStrategic Alignment to Performance: A Moderated Mediation Model of Social Innovation, and Enterprise Governance of IT"
30 Wilma Latuny (UvT), The Power of Facial Expressions
31 Ben Ruijl (UL), Advances in computational methods for QFT calculations
32 Thaer Samar (RUN), Access to and Retrievability of Content in Web Archives
33 Brigit van Loggem (OU), Towards a Design Rationale for Software Documentation: A Model of Computer-Mediated Activity
34 Maren Scheffel (OU), The Evaluation Framework for Learning Analytics
35 Martine de Vos (VU), Interpreting natural science spreadsheets
36 Yuanhao Guo (UL), Shape Analysis for Phenotype Characterisation from High-throughput Imaging
37 Alejandro Montes Garcia (TUE), WiBAF: A Within Browser Adaptation Framework that Enables Control over Privacy
38 Alex Kayal (TUD), Normative Social Applications
39 Sara Ahmadi (RUN), Exploiting properties of the human auditory system and compressive sensing methods to increase noise robustness in ASR
40 Altaf Hussain Abro (VUA), Steer your Mind: Computational Exploration of Human Control in Relation to Emotions, Desires and Social Support For applications in human-aware support systems
41 Adnan Manzoor (VUA), Minding a Healthy Lifestyle: An Exploration of Mental Processes and a Smart Environment to Provide Support for a Healthy Lifestyle
42 Elena Sokolova (RUN), Causal discovery from mixed and missing data with applications on ADHD datasets
43 Maaike de Boer (RUN), Semantic Mapping in Video Retrieval
44 Garm Lucassen (UU), Understanding User Stories - Computational Linguistics in Agile Requirements Engineering
45 Bas Testerink (UU), Decentralized Runtime Norm Enforcement

	46	Jan Schneider (OU), Sensor-based Learning Support
	47	Jie Yang (TUD), Crowd Knowledge Creation Acceleration
	48	Angel Suarez (OU), Collaborative inquiry-based learning
2018	01	Han van der Aa (VUA), Comparing and Aligning Process Representations
	02	Felix Mannhardt (TUE), Multi-perspective Process Mining
	03	Steven Bosems (UT), Causal Models For Well-Being: Knowledge Modeling, Model-Driven Development of Context-Aware Applications, and Behavior Prediction
	04	Jordan Janeiro (TUD), Flexible Coordination Support for Diagnosis Teams in Data-Centric Engineering Tasks
	05	Hugo Huurdeman (UVA), Supporting the Complex Dynamics of the Information Seeking Process
	06	Dan Ionita (UT), Model-Driven Information Security Risk Assessment of Socio-Technical Systems
	07	Jieting Luo (UU), A formal account of opportunism in multi-agent systems
	08	Rick Smetsers (RUN), Advances in Model Learning for Software Systems
	09	Xu Xie (TUD), Data Assimilation in Discrete Event Simulations