MANCHESTER 1824

The University of Manchester

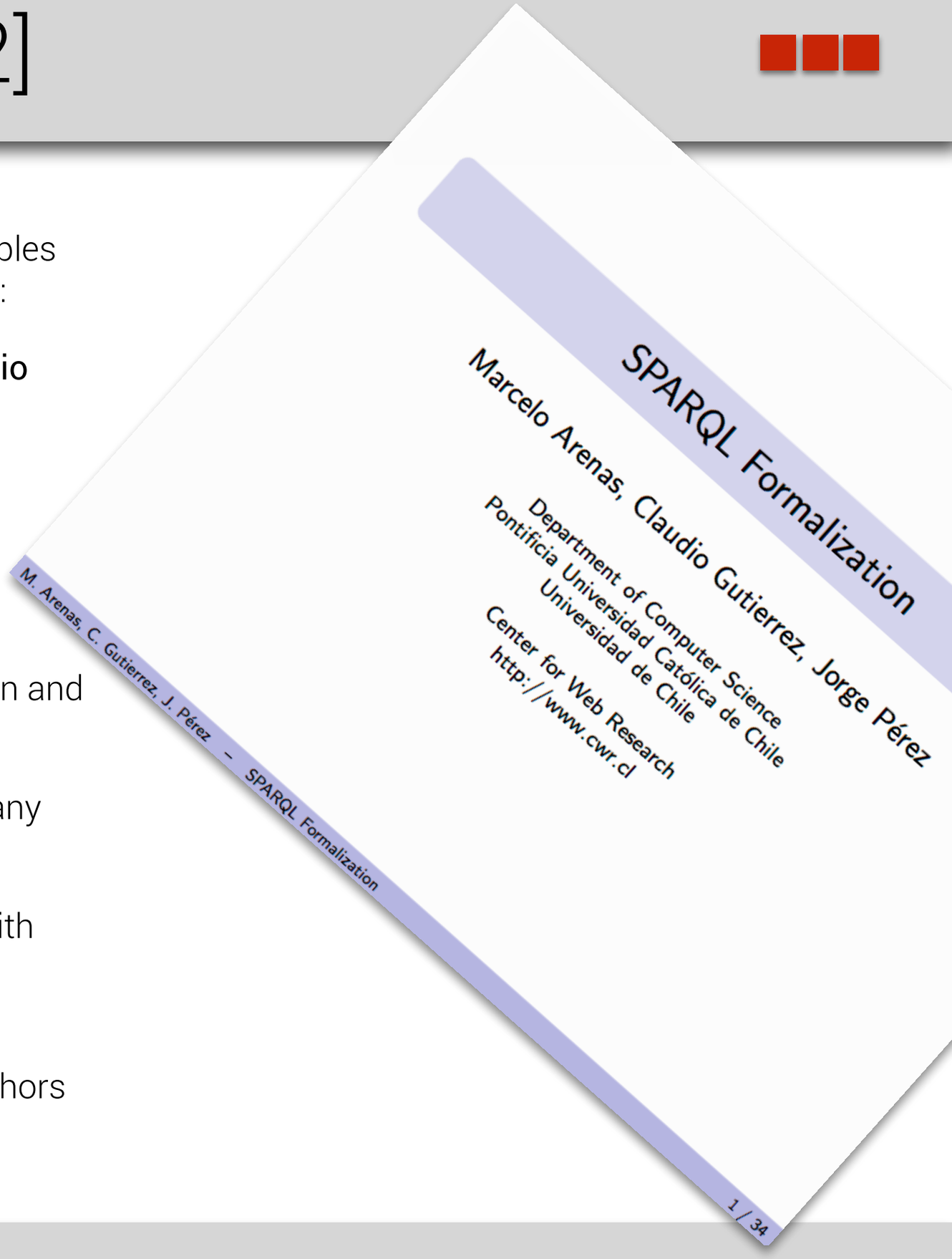
Querying Data on the Web

Alvaro A A Fernandes
School of Computer Science
University of Manchester

Acknowledgements [1]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the tutorial:
  - SPARQL. Andy Seaborne. ESWC 2007 Tutorial. https://ai.wu.ac.at/~polleres/sparqltutorial/
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect recent updates to SPARQL or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above author is acknowledged.
- The author of these slides is very grateful to the author of the work above.

Acknowledgements [2]



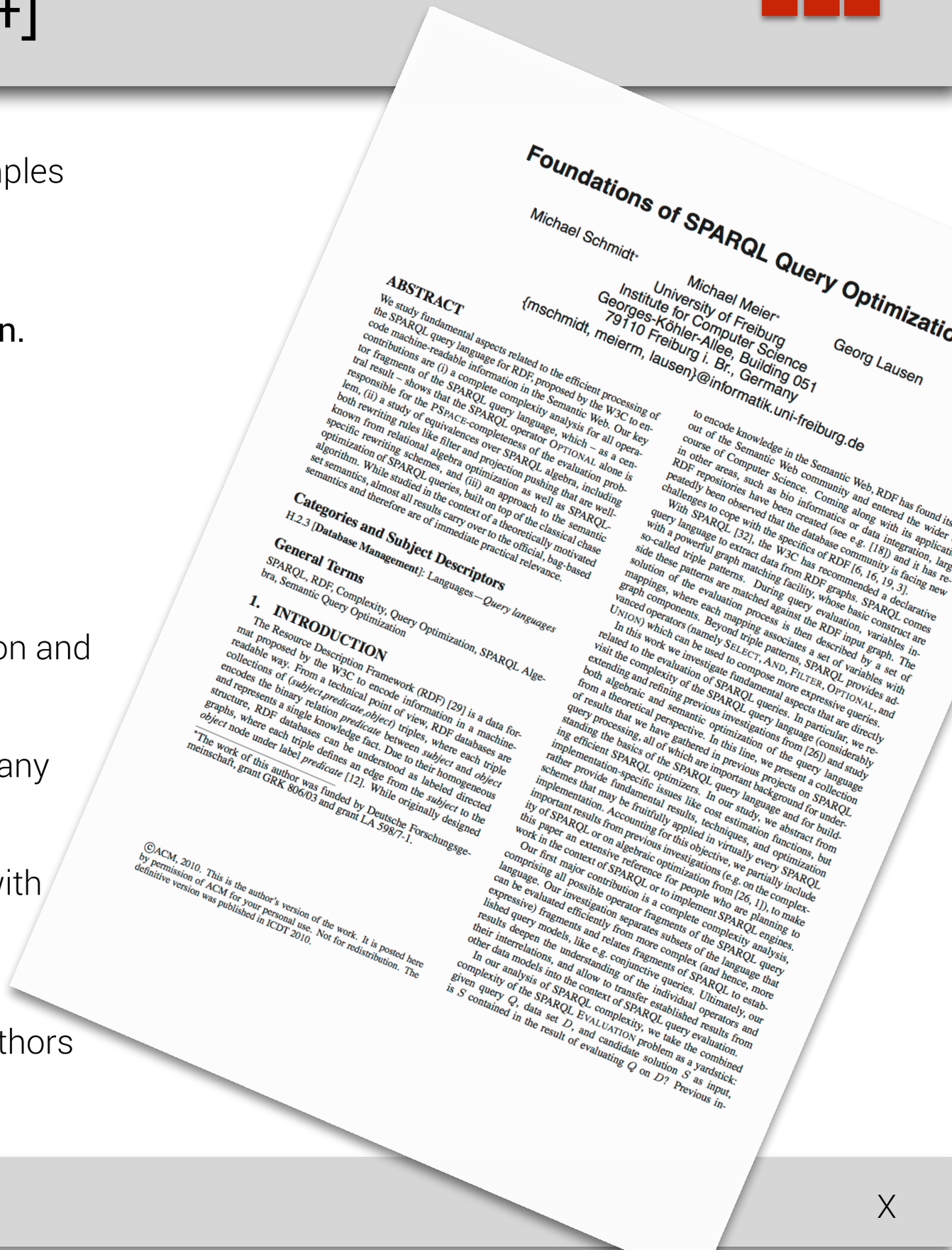
- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the tutorial:
  - SPARQL Formalization. Marcelo Arenas, Claudio Gutierrez, Jorge Pérez. ESWC 2007 Tutorial. https://ai.wu.ac.at/~polleres/sparqltutorial/
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect recent updates to SPARQL or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above authors are acknowledged.
- The author of these slides is very grateful to the authors of the work above.

Acknowledgements [3]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the tutorial:
  - SPARQL Algebra. Andy Seaborne. ESWC 2007 Tutorial. https://ai.wu.ac.at/~polleres/sparqltutorial/
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect recent updates to SPARQL or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above author is acknowledged.
- The author of these slides is very grateful to the author of the work above.

Acknowledgements [4]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the paper:
  - Foundations of SPARQL Query Optimization/ Michael Schmidt, Michael Maier, Georg Lausen. ICDT 2010: 4-33. http://doi.acm.org/10.1145/1804669.1804675
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect recent updates to SPARQL or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above authors are acknowledged.
- The author of these slides is very grateful to the authors of the work above.



Foundations of SPARQL Query Optimizatio

Michael Schmidt* Michael Meier* Georg Lausen

University of Freiburg
Institute for Computer Science
Georges-Köhler-Allee, Building 051
79110 Freiburg i. Br., Germany
{mschmidt, meierm, lausen}@informatik.uni-freiburg.de

ABSTRACT

Categories and Subject Descriptors

General Terms

1. INTRODUCTION

Acknowledgements [5]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the paper:
  - Artem Chebotko, Shiyong Lu, Farshad Fotouhi: Semantics preserving SPARQL-to-SQL translation. Data Knowl. Eng. 68(10): 973-1000 (2009) http://dx.doi.org/10.1016/j.datak.2009.04.001
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect recent updates to SPARQL or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above authors are acknowledged.
- The author of these slides is very grateful to the authors of the work above.

Data & Knowledge Engineering 68 (2009) 973–1000

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

ELSEVIER

DKE

Semantics preserving SPARQL-to-SQL translation

Artem Chebotko [a,*], Shiyong Lu [b], Farshad Fotouhi [b]

[a] Department of Computer Science, University of Texas-Pan American, 1201 West University Drive, Edinburg, TX 78539, USA
[b] Department of Computer Science, Wayne State University, 431 State Hall, 5143 Cass Avenue, Detroit, MI 48202, USA

ARTICLE INFO

Article history:
Received 6 July 2008
Received in revised form 2 April 2009
Accepted 3 April 2009
Available online 16 April 2009

Keywords:
SPARQL-to-SQL translation
SPARQL semantics
SPARQL
SQL
RDF
query
RDF store
RDBMS

ABSTRACT

Most existing RDF stores, which serve as metadata repositories on the Semantic Web, use an RDBMS as a backend to manage RDF data. This motivates us to study the problem of translating SPARQL queries into equivalent SQL queries, which further can be optimized and evaluated by the relational query engine and their results can be returned as SPARQL query solutions. The main contributions of our research are: (i) We formalize a relational algebra based semantics of SPARQL, which bridges the gap between SPARQL and SQL query languages, and prove that our semantics is equivalent to the mapping-based semantics of SPARQL; (ii) Based on this semantics, we propose the first provably semantics preserving SPARQL-to-SQL translation for SPARQL triple patterns, basic graph patterns, optional graph patterns, alternative graph patterns, and value constraints; (iii) Our translation algorithm is generic and can be directly applied to existing RDBMS-based RDF stores; and (iv) We outline a number of simplifications for the SPARQL-to-SQL translation to generate simpler and more efficient SQL queries and extend our defined semantics and translation to support the bag semantics of a SPARQL query solution. The experimental study showed that our proposed generic translation can serve as a good alternative to existing schema dependent translations in terms of efficient query evaluation and/or ensured query result correctness.

1. Introduction

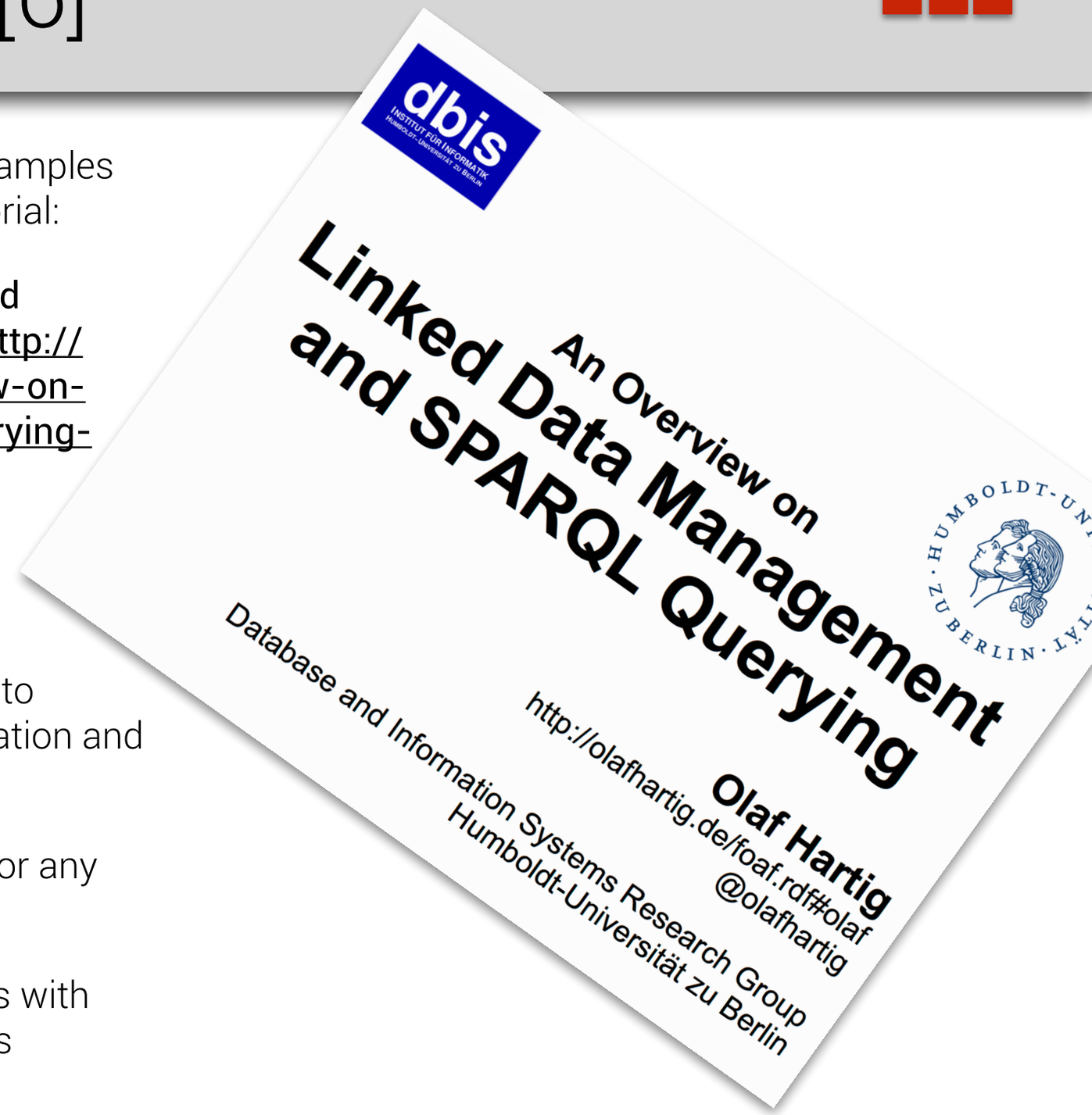
The Semantic Web [7,47] has recently gained tremendous momentum due to its great potential for providing a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. Semantic annotations for various heterogeneous resources on the Web are represented in Resource Description Framework (RDF) [56,58], the standard language for annotating resources on the Web, and searched using the query language for RDF, called SPARQL [59], that has been proposed by the World Wide Web Consortium (W3C) and has recently achieved the recommendation status. Essentially, RDF data is a collection of statements, called *triples*, of the form (s, p, o), where s is called *subject*, p is called *predicate*, and o is called *object*, and each triple states the relation between a subject and an object. Such a collection of triples can be viewed as a directed graph, in which nodes represent subjects and objects, and edges represent predicates connecting from subject nodes to object nodes. To query RDF data, SPARQL allows the specification of triple and graph patterns to be matched over RDF graphs.

Explosive growth of RDF data on the Web drives the need for novel database systems, called *RDF stores*, that can efficiently store and query large RDF datasets. Most existing RDF stores, including Jena [63,62], Sesame [9], 3store [27,28], KAON [54], RStar [35], OpenLink Virtuoso [22], DLDB [38], RDFSuite [3,52], DBOWL [37], PARKA [50], RDFProv [12], and RDFBroker [48] use a relational database management system (RDBMS) as a backend to manage RDF data. The main advantage of the

* Corresponding author. Tel.: +1 956 381 2577; fax: +1 956 384 5099.
E-mail addresses: artem@cs.panam.edu (A. Chebotko), shiyong@wayne.edu (S. Lu), fotouhi@wayne.edu (F. Fotouhi).

0169-023X/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2009.04.001

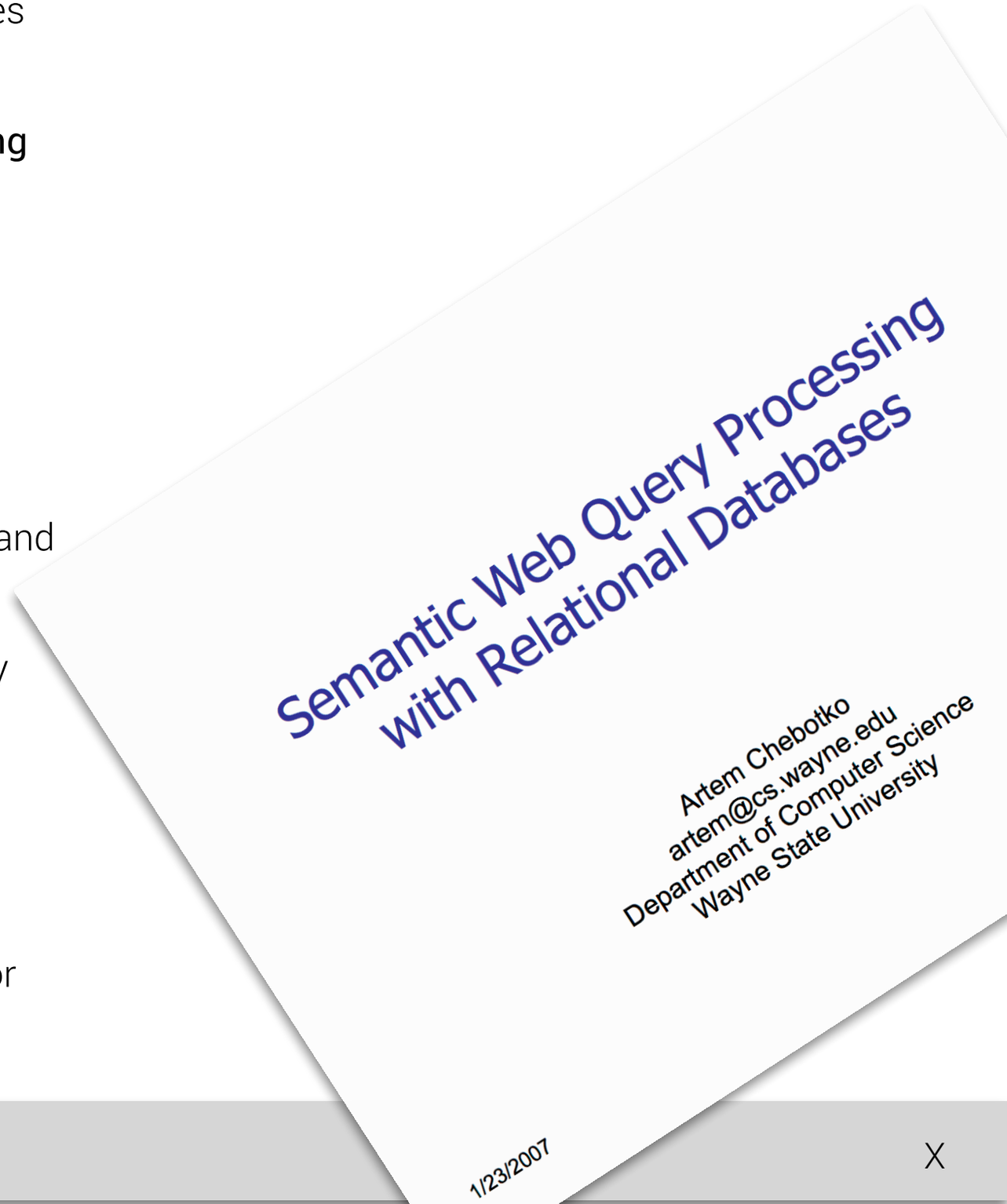
Acknowledgements [6]



- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the tutorial:
  - Olaf Hartig. An Overview on Linked Data and SPARQL Querying. ISSLOD 2011 Tutorial. http://www.slideshare.net/olafhartig/an-overview-on-linked-data-management-and-sparql-querying-isslod2011
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect recent updates to SPARQL or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above author is acknowledged.
- The author of these slides is very grateful to the author of the work above.

Acknowledgements [7]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the slide set:
  - Artem Chebotko. Semantic Web Query Processing with Relational Databases. (2007) http://www.cs.wayne.edu/graduateseminars/gradsem_w07/slides/artem_talk.ppt
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect recent updates to SPARQL or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above author is acknowledged.
- The author of these slides is very grateful to the author of the work above.



Acknowledgements [8]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from the slide set:
  - Bart Wolff, George Fletcher, James Lu. An Extensible Framework for Query Optimization on TripleT-Based RDF Stores. (2015) http://www.win.tue.nl/~gfletche/papers-final/lwdm2015.pdf
- These slides were put together from the above publication for educational purposes only.
- Any changes made either reflect recent updates to SPARQL or are the result of minor editing, adaptation and extension for use in teaching.
- The author of these slides is solely responsible for any errrors they contain.
- No claims are made by the author of these slides with regards to any form of intellectual property rights wherever the above author is acknowledged.
- The author of these slides is very grateful to the author of the work above.

Acknowledgements [8]

- Some of these slides mostly contain text and examples that are most often taken *verbatim* from W3C documents mentioned in the slide titled **W3C Documents**.

Topics So Far [1]

- We have revised the basic notions of what a database management system (DBMS) is, what role the languages a DBMS supports play, and how DBMS-centred applications are designed.
- We have adopted an approach to describing the internal architecture of DBMSs that allowed us to discuss the strengths and weaknesses of classical DBMSs and the recent trends in the way organizations use DBMS how architectures have been diversifying recently.
- We have revised the various kinds of query languages by looking into the relational calculi, a relational algebra and SQL
- We have looked in detail into logical optimization, i.e., heuristic-based rewriting of algebraic plans.
- We have explored some of the basic strategies for designing physical operators and for combining them into query evaluation plans (QEPs).
- We have briefly discussed some of the challenges in cost-based QEP selection, in particular the hard problem of choosing a join order.
- We have found out how data partitioning strategies, allied with the algebraic properties of query languages, make it possible to automatically parallelize QEP execution.

Topics So Far [2]

- We have taken a quick, example-driven tour of XQuery, to remind ourselves of the basic characteristics of the language.
- We have introduced and briefly explored an algebraic view of XQuery, and some equivalence laws that arise, on the basis of which one can conceive of a rewriting approach to logical optimization for the language.
- We then studied storage, optimization and evaluation of XQuery as implemented in one specific XML native DBMS, viz., BaseX.

Learning Objectives So Far

- We have aimed to revise and reinforce undergraduate-level understanding of DBMS as software systems, rather than as software development components.
- We have aimed to start a postgraduate-level exploration of query languages.
- We have aimed to explore classical query processing, including optimization and evaluation, both centralized and parallel.
- We have aimed to acquire a deeper understanding of some of the challenges arising in querying over large XML collections using XQuery.

Topics This Week

- We will take a quick, example-driven tour of SPARQL, to remind ourselves of the basic characteristics of the language.
- We will introduce and briefly explore an algebraic view of SPARQL, and some equivalence laws that arise, on the basis of which one can conceive of a rewriting approach to logical optimization for the language.
- We will then study storage, optimization and evaluation of SPARQL, specially in the context of the SPARQL-through-SQL strategy.

Learning Objectives This Week

- We will aim to acquire a deeper understanding of some of the challenges arising in querying over large RDF collections using SPARQL.

Teaching Day 4 Outline

1. A Quick Tour of SPARQL
   I. SPARQL in Broad Strokes
   II. Example SPARQL Queries
   III. SPARQL Evaluation Steps
2. An Algebraic View of SPARQL
   I. Graph Patterns (GPs)
   II. Negation in SPARQL
   III. Abstract Syntax Tree and Query Plan
   IV. Translation Procedure
   V. Example GP Translations
   VI. SA Core Abstract Syntax
   VII. A Formal Semantics for SPARQL
3. Logical Optimization in SPARQL
   I. Properties of Variables
   II. Algebraic Equivalences
   III. Example Rewritings
4. SPARQL Query Evaluation [1]
   I. Basic Perspectives
   II. SPARQL Through SQL
   III. Modelling Triples with Tables
5. SPARQL Query Evaluation [2]
   I. Translating SPARQL into SQL
   II. Examples
   III. Empirical Evidence of Efficiency
   IV. Index-Centric SPARQL Evaluation

Lecture 16

MANCHESTER 1824

The University of Manchester

A Quick Tour of SPARQL

The University of Manchester

SPARQL in Broad Strokes

SPARQL in Broad Strokes: Capabilities

- SPARQL is a declarative query language over RDF datasets.
- Here, the focus is on querying (and on SPARQL 1.0), but SPARQL (1.1) comes with capabilites related to:
  - Updates
  - Entailment regimes
  - Federation
  - Service description and discovery
  - HTTP-based interaction with endpoints (i.e., front-ends to RDF-graph collections)
  - HTTP-based management of RDF-graph collections
  - Format management (e.g., CSV, XML, JSON.
- In terms of format, these slides mostly use Turtle notation.

from: SPARQL. Andy Seaborne (2007)

SPARQL in Broad Strokes: Patterns and Operators

- Building on the notion of a subject-predicate-object (s-p-o) RDF triple, the core notion in a SPARQL query is that of a *triple pattern*, in which variables can appear in one or more of subject, predicate or object position.
- A *basic graph pattern* (BGP) is a set of triple patterns.
- In the formalization that will be given later, the following operators will be used to combine graph patterns: AND, UNION, OPTIONAL and FILTER, plus SELECT and ASK as query modifiers.
- These operators are defined for an internal form, i.e., an abstract syntactic object as generated by a parser.
- The W3C standard (see the bibliography slides) covers, of course, all the possible operators.

SPARQL in Broad Strokes: Declarations, Query Forms, Datasets, Solution Modifiers

- A PREFIX declaration allows the abbreviation of namespace URIs.
- The query itself can take one of the following query forms: SELECT, ASK, CONSTRUCT and DESCRIBE.
- The dataset over which a query is evaluated is a collection of graphs.
- There exists one, unnamed, default graph.
- There exist zero or more named graphs accessible via the GRAPH keyword.
- The keyword FROM (and FROM NAMED) can be used to bind the query to URIs denoting stored graphs.
- The result of evaluating a SPARQL query can be modified by the following directives: DISTINCT, REDUCED, ORDER BY and LIMIT/OFFSET.
- Note that SPARQL 1.1 goes beyond the above.

MANCHESTER 1824
The University of Manchester

Example SPARQL Queries

SPARQL Example Queries: Triple Patterns

```
# data

@prefix person: <http://example/person/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

person:A foaf:name "Alice" .
person:A foaf:mbox <mailto:alice@example.net> .
person:B foaf:name "Bob" .

#query

PREFIX person: <http://example/person/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
WHERE  {
         ?x foaf:name ?name
       }

#result

+---------+
| ?name   |
+---------+
| "Bob"   |
| "Alice" |
+---------+
```

a TP

a SELECT query returns bindings

SPARQL Example Queries: Basic Graph Patterns

```
# data

@prefix person: <http://example/person/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

person:A foaf:name "Alice" .
person:A foaf:mbox <mailto:alice@example.net> .
person:B foaf:name "Bob" .

#query

PREFIX person: <http://example/person/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
WHERE { ?person foaf:mbox mailto:alice@example.net> .
        ?person foaf:name ?name .
      }

#result

+---------+
| ?name   |
+---------+
| "Alice" |
+---------+
```

a BGP comprising two TPs

dot acts as AND

curly brackets are used for grouping

from: SPARQL. Andy Seaborne (2007)

SPARQL Example Queries: FILTER

```
# data

@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix stock: <http://example.org/stock#> .
@prefix inv: <http://example.org/inventory#> .

stock:book1 dc:title "SPARQL Tutorial" .
stock:book1 inv:price 10 .
stock:book1 inv:quantity 3 .
stock:book2 dc:title "SPARQL (2nd ed)" .
stock:book2 inv:price 20 ; inv:quantity 5 .
stock:book3 dc:title "From SQL to SPARQL" .
stock:book3 inv:price 5 ; inv:quantity 0 .
stock:book4 dc:title "Applying XQuery" .
stock:book4 inv:price 20 ; inv:quantity 8 .
```

the semicolon allows for lists of property-object pairs to share a subject

```
#query

PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX stock: <http://example.org/stock#>
PREFIX inv: <http://example.org/inventory#>

SELECT ?book ?title
WHERE { ?book dc:title ?title .
        ?book inv:price ?price .
           FILTER ( ?price < 15 )
        ?book inv:quantity ?num .
           FILTER ( ?num > 0    )
      }

#result

+-------------+-------------------+
| ?book       | ?title            |
+-------------+-------------------+
| stock:book1 | "SPARQL Tutorial" |
+-------------+-------------------+
```

FILTER takes as argument a predicate on query-occurring variables

SPARQL Example Queries: UNION, DISTINCT

```
# data

@prefix book: <http://example/book/> .
@prefix dc10: <http://purl.org/dc/elements/1.0/> .
@prefix dc11: <http://purl.org/dc/elements/1.1/> .

book:a  dc10:title "SPARQL Tutorial" .
book:b  dc11:title "SPARQL (2nd ed)" .
book:c  dc10:title "SPARQLing" .
book:c  dc11:title "SPARQLing" .
```

Brackets make the WHERE keyword optional.

```
#query

PREFIX dc10: <http://purl.org/dc/elements/1.0/>
PREFIX dc11: <http://purl.org/dc/elements/1.1/>

SELECT DISTINCT ?title
  { { ?book dc10:title ?title }
    UNION
    { ?book dc11:title ?title }
  }
```

DISTINCT is used if set semantics is needed.

UNION combines the solutions to both the left and the right graph patterns.

```
#result

+--------------------+
| ?title             |
+--------------------+
| "SPARQL Tutorial"  |
| "SPARQLing"        |
| "SPARQL (2nd ed)"  |
+--------------------+
```

from: SPARQL. Andy Seaborne (2007)

SPARQL Example Queries: OPTIONAL

```
# data

@prefix person: <http://example/person/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

person:a foaf:name "Alice" .
person:a foaf:nick "A-online" .
person:b foaf:name "Bob" .
```

```
#query

PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name ?nick
  {            ?x foaf:name ?name .
    OPTIONAL {?x foaf:nick ?nick }
  }

#result

+---------+------------+
| ?name   | ?nick      |
+---------+------------+
| "Alice" | "A-online" |
| "Bob"   |            |
+---------+------------+
```

OPTIONAL indicates that partial bindings are allowed.

from: SPARQL. Andy Seaborne (2007)

SPARQL Example Queries: FROM/GRAPH

```
# Named graph: http://example.org/foaf/aliceFoaf

@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#> .

_:a  foaf:name     "Alice" .
_:a  foaf:mbox     <mailto:alice@work.example> .
_:a  foaf:knows    _:b .

_:b  foaf:name     "Bob" .
_:b  foaf:mbox     <mailto:bob@work.example> .
_:b  foaf:nick     "Bobby" .
_:b  rdfs:seeAlso  <http://example.org/foaf/bobFoaf> .

<http://example.org/foaf/bobFoaf>
     rdf:type      foaf:PersonalProfileDocument .

# Named graph: http://example.org/foaf/bobFoaf

@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#> .

_:z  foaf:mbox     <mailto:bob@work.example> .
_:z  rdfs:seeAlso  <http://example.org/foaf/bobFoaf> .
_:z  foaf:nick     "Robert" .

<http://example.org/foaf/bobFoaf>
     rdf:type      foaf:PersonalProfileDocument .
```

Two distinct named graphs

```
# Q1

PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?src ?bobNick
FROM NAMED <http://example.org/foaf/aliceFoaf>
FROM NAMED <http://example.org/foaf/bobFoaf>
WHERE
  {
    GRAPH ?src
    { ?x foaf:mbox <mailto:bob@work.example> .
      ?x foaf:nick ?bobNick
    }
  }
```

declares two data sources

a variable in GRAPH ranges over both graphs

```
#Q2

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

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

GRAPH ranges over one graph only

from: SPARQL. Andy Seaborne (2007)

SPARQL Example Queries: ASK

```
# data

@prefix person: <http://example/person/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

person:a foaf:name "Alice" .
person:a foaf:nick "A-online" .
person:b foaf:name "Bob" .
```

ASK returns true if the query pattern has a solition, and false otherwise.

```
#query

PREFIX foaf: <http://xmlns.com/foaf/0.1/>

ASK
  {            ?x foaf:name ?name .
    OPTIONAL {?x foaf:nick ?nick }
  }

#result

+------+
| true |
+------+
```

SPARQL Example Queries: CONSTRUCT

```
# data

@prefix person: <http://example/person/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

person:a foaf:name "Alice" .
person:a foaf:mbox <mailto:alice@example.net> .
person:b foaf:name "Bob" .
```

CONSTRUCT returns an RDF graph, i.e., a set of triples instantiating the pattern passed as argument.

```
#query

PREFIX person:<http://example/person/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
PREFIX vc:<http://www.w3.org/2001/vcard-rdf/3.0#>

CONSTRUCT { ?person vc:FN ?name }
WHERE
  {?person foaf:name ?name . }

#result


@prefix person: <http://example/person/> .
@prefix vc: <http://www.w3.org/2001/vcard-rdf/3.0#>

person:a vcard:FN "Alice" .
person:b vcard:FN "Bob" .
```

from: SPARQL. Andy Seaborne (2007)

MANCHESTER 1824

The University of Manchester

SPARQL Evaluation Steps

Steps in Evaluating a SPARQL Query [1]

- The outcome of executing a SPARQL query is defined by a series of steps.
- A SPARQL *query string* is turned into an abstract syntax form.
- The *abstract syntax form* is then mapped to a SPARQL *abstract query (plan)* comprising operators from the **SPARQL algebra (SA)**.
- This query plan is then available for evaluation on an RDF dataset.
- Before evaluation takes place, logical and physical optimization of the query plan may be performed, but this is not part of the standard description.
- We will first describe the translation procedure from query strings to an abstract query plan.
- Then, we will formally describe how an abstract query plan is evaluated.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Steps in Evaluating a SPARQL Query [2]

- The complete SA operates on
  - multisets/sets of mappings for evaluating GP expressions
  - lists (more precisely, sequences) for evaluating solution modifiers
- The W3C spec defines the translation algorithms in detail.
- Note that the formal semantics as presented in later slides does not cover all of SPARQL (1.0 nor 1.1) but can be extended to do so.
- Those later slides will also focus on set (as opposed to bag, or multiset) semantics.
- The W3C spec defines the result of evaluation whilst implementations are free to carry out the evaluation in any way that leads to the specified result.
- We will look at implementation challenges later.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Lecture 17

MANCHESTER 1824

The University of Manchester

An Algebraic View of SPARQL

MANCHESTER 1824

The University of Manchester

Graph Patterns

Graph Patterns

- SPARQL is based on graph pattern matching.
- More complex patterns are formed by combining argument patterns.
- An informally defined classification of graph patterns (GPs) is as follows:
  - A basic graph pattern (BGP) is a set of triple patterns (TPs) that must match.
  - A group graph pattern (GGP) is a set of graph patterns that must match.
  - An optional graph (OGP) pattern possibly extends the solutions contributed by another graph pattern.
  - An alternative (or union) graph pattern (AGP) consists of two graph patterns that can, each independently of the other, contribute solutions.
  - A named (or graph) graph pattern (NGP) is matched against a specific, explicitly-named graph.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Basic Graph Patterns

- Thus, SPARQL GP matching is defined in terms of the combination of the results obtained by matching BGPs.
- A sequence of TPs, possibly interspersed with filters, is a BGP.
- The occurrence of any other type of GP terminates the definition of the BGP.
- Blank nodes are scoped to the BGP they occur in.
- A blank node label can occur only once, i.e., in a single BGP.
- The evaluation of a BGP is based on subgraph matching.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Group Graph Patterns

- In SPARQL, a GGP is syntactically characterized by being enclosed in curly brackets.
  - If P and P′ are TPs, then P P′ is a BGP and {P P′} is a GGP.
- Nested GGPs are semantically equivalent to the GGP that results from flattening them.
  - If P, P′ and P″ are (singleton) BGPs, then the GGP {{P P′} P″} returns the same results as the GGP {P P′ P″}.
  - The empty GGP { } matches any graph, including the empty graph, with exactly one solution that binds no variable.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Scope of Filters

- A constraint introduced by the keyword FILTER is a restriction, sometimes referred to as a value constraint, on the solutions returned by the GGP in which the FILTER expression occurs, irrespective of (precisely, linearly) where, in the GGP, it occurs.

- If P and P′ are BGPs and F is a filter, then following GGPs all return the same solutions:
  - {F P P′}
  - {P F P′}
  - {P P′ F}

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

GGP Structures

- It follows that the occurrence of a FILTER in a GGP does not break the GGP into two.
  - If P and P′ are BGPs and F is a filter, then {P F P′} is one GGP containing one BGP (with two TPs) and one filter.
- Nested groups give structure to the GGP.
  - If P and P′ are BGPs, then {P { } P′} is one GGP containing one singleton BGP, one (empty) GGP and another singleton BGP.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Optional Graph Patterns [1]

- A GP that is made entirely of BGPs requires all the BGPs to match for there to be a solution to the query as a whole.
- This implies a regularity of structure in the data that cannot be assumed in every RDF graph.
- Thus, it is useful to specify in a query that should a given GP match, then it should contribute to the result without, however, causing the query as whole to lack a solution if it does not match.
- Such a GP is an OGP, i.e., if it has no match, it contributes no bindings but does not nullify the solution as a whole.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Optional Graph Patterns [2]

- If P and P′ are GPs, then then P OPTIONAL {P′} is said to be an OGP.
- Thus, if P′ matches, the resulting bindings contribute to the solutions, otherwise it leaves the solutions unchanged with respect to already existing bindings.
- {OPTIONAL {P} is shorthand for {{ } OPTIONAL {P}}
- Syntactically, OPTIONAL is left-associative, i.e., if P, P′ and P″ are GPs, then P OPTIONAL {P′} OPTIONAL {P″} is equivalent to {P OPTIONAL {P′}} OPTIONAL {P″}

Optional Graph Patterns [3]

- OGPs can be constrained with FILTER.
- If P and P′ are GPs and F is a filter, then then P OPTIONAL {P′ F} is a GP the right-hand side of which is a constrained (or conditional) OGP.
- GPs are defined recursively and a GP may have zero or more OGPs, as already said.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

X

Alternative Graph Patterns

- SPARQL provides the keyword UNION to denote to combine the left- and right-hand sided GPs and allow both to match and contribute to the solutions.
- If P and P′ are GPs, then then {{P} UNION {P′}} is an AGP.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

MANCHESTER 1824

The University of Manchester

Negation in SPARQL

Negation in SPARQL [1]

- SPARQL (1.1) incorporates two styles for expressing negation.
- One is based on filtering results depending on whether a GP does or does not match in the context of the solution being filtered.
- The other is based on removing solutions that are related to another GP.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

X

Negation in SPARQL [2]

- Filter-based negation is done with a FILTER expression using NOT EXISTS and EXISTS.
- Recall that the scope of the FILTER is the whole GGP in which the filter occurs.
- The NOT EXISTS filter expression tests whether a GP does not match the dataset, given the values of variables in the GGP in which it occurs.
- The EXISTS filter expression tests whether a GP does match the dataset, given the values of variables in the GGP in which it occurs.
- Neither the EXISTS nor the NOT EXISTS filter expression generates additional bindings.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

X

Negation in SPARQL [3]

- Removal-based negation is done with MINUS, which evaluates both its arguments, then removes solutions to the left-hand side that are not compatible with solutions to the right-hand side.
- Filter-based negation can produce a different answer from removal-based negation.
- The next slide illustrates some such cases.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

X

Negation in SPARQL [4]

- In SELECT * {?s ?p ?o} FILTER NOT EXISTS {?x ?y ?z}, because no variables are shared, NOT EXISTS eliminates every solution, whereas, for the same reason, we have that SELECT * {?s ?p ?o} MINUS {?x ?y ?z} eliminates no solutions.

- When the argument to NOT EXISTS is a ground (i.e., literal-only, no-variable) pattern, no solutions are returned, whereas if the right-hand side of MINUS is a ground pattern, no solutions are eliminated.

- Differences also arise from the fact that in a FILTER expression, all variables in the group are in scope, which is not the case for MINUS.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

MANCHESTER 1824

The University of Manchester

Abstract Syntax Tree and Query Plan

SPARQL Abstract Syntax Tree

We will not cover the ones in italics.

Patterns	Modifiers	Query Forms	Other
RDF terms	*DISTINCT*	SELECT	*VALUES*
Property Path Expressions	*REDUCED*	*CONSTRUCT*	*SERVICE*
Property Path Patterns	*Projection*	*DESCRIBE*	
Groups	*ORDER BY*	*ASK*	
OPTIONAL	*LIMIT*		
UNION	*OFFSET*		
GRAPH	*Select Expressions*		
BIND			
GROUP BY			
HAVING			
MINUS			
FILTER			

The translation procedure we're about to describe maps from these constructs to the operators in the next slide.

The output of the parser has these constructs.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

X

SPARQL Abstract Query Plan

We will not cover the ones in italics.

Graph Patterns	Solution Modifiers	Property Paths
BGP	ToList	*PredicatePath*
Join	OrderBy	*InversePath*
LeftJoin	Project	*SequencePath*
Filter	Distinct	*AlternativePath*
Union	Reduced	*ZeroOrMorePath*
Graph	Slice	*ZeroOrOnePath*
Extend	*ToMultiset*	*NegatedPropertySet*
Minus		
Group		
Aggregation		
AggregateJoin		

The formal semantics of the GP (non-italicized) operators is given later.

The output of the translation procedure has these constructs.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

MANCHESTER 1824

The University of Manchester

Translation Procedure

Translation Procedure: Signature + Note

- If P is a graph pattern, then the procedure to translate GPs is denoted by
  - translate(P)
- Note that in SPARQL 1.0, it is significant whether the simplification step is applied either
  - after all translations or not at all, or else
  - during translation
- An ambiguity would occur in queries involving a doubly nested filter and pattern in an optional:
  - OPTIONAL { { … FILTER (… ?x …) } }..
- Applying the simpification step ***after all translations*** of graph patterns is the preferred reading.

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Translation Procedure: Steps 1-5

- FILTER expressions apply to the whole GGP in which they appear.
- The operators to perform filtering are added to the group after translation of each group element.
- We collect the filters together here and remove them from the group element, then apply them to the whole translated GGP.
- In this step, we also translate GPs within FILTER expressions EXISTS and NOT EXISTS.
- The set of filter expressions FS is used later.

```
# Step 1: Expand

Expand abbreviations in IRIs and TPs

# Step 2: Collect

Let FS := empty set
For each form FILTER(e) in the GGP:
   In e, replace NOT EXISTS{P}
         with fn:not(exists(translate(P)))
   In e, replace EXISTS{P}
         with exists(translate(P))
   FS := FS ∪ {e}

# Step 3: Translate Property Path Expressions

# We omit this procedure here.

# Step 4: Translate Property Path Patterns

# We omit this procedure here.

# Step 5. Translate Basic Graph Patterns

Any adjacent TPs t1 … tn are collected and
replaced with BGP(t1, …, tn).
```

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Translation Procedure: Step 6.1

- Next, we translate each remaining form of GP, recursively applying the translation procedure.
- This and the next few slides each treat one kind of GP.
- Note that, following the standard grammar for SPARQL, AGP is also referred to as Union Graph Pattern and, therefore, what we have called (to the right) GGP-or-AGP is also referred to as Graph-Or-Union Graph Pattern.

```
# Step 6: Translate Other Graph Patterns

# Step 6.1: Translate GGP or AGP

If GP is of the form GGP-or-AGP:
   Let A := undefined
   For each element G in the GP:
      If A is undefined:
         A := Translate(G)
      Else:
         A := Union(A, Translate(G))


The result is A
```

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Translation Procedure: Step 6.2

- Note that, following the standard grammar for SPARQL, NGP is also referred to as Graph Graph Pattern (which leads to acronymic confusion with Group Graph Pattern, potentially).

```
# Step 6.2: Translate NGP

If the form is NGP:
   If the form is GRAPH IRI GGP:
      The result is Graph(IRI, Translate(GGP))
   If the form is GRAPH Var GGP:
      The result is Graph(Var, Translate(GGP))
```

Translation Procedure: Step 6.3-6.5

- Here, more kinds of algebraic operators are introduced.

```
# Step 6.3: Translate GGP

If the form is GGP:
   Let FS := the empty set
   Let G  := the empty pattern, the empty BGP
   For each element E in the GGP:
      If E is of the form OPTIONAL{P}:
         Let A := Translate(P)
         If A is of the form Filter(F, A')
            G := LeftJoin(G, A', F)
         Else
            G := LeftJoin(G, A, true)
      If E is of the form MINUS{P}:
         G := Minus(G, Translate(P))
      If E is of the form BIND(expr AS var):
         G := Extend(G, var, expr)
      If E is any other form:
         Let A := Translate(E)
         G := Join(G, A)

The result is G.

# Step 6.4
# We omit the case where the form is InlineData
# Step 6.5
# We omit the case where the form is SubSelect
```

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

X

Translation Procedure: Step 7

- The set of filter expressions FS is now used.
- After the group has been translated, the filter expressions are conjoined and given as scope the whole of the rest of the group.

```
If FS is not empty:
   Let G := output of preceding step
   Let X := Conjunction of expressions in FS
   G := Filter(X, G)
```

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Translation Procedure: Step 8

- Some groups with one GP become Join(Z, A), where Z is the empty BGP (which is the empty set).
- A simplification step is carried out because the empty graph pattern Z is the algebraic identity for join.

```
Replace Join(Z, A) with A
Replace Join(A, Z) with A
```

MANCHESTER 1824

The University of Manchester

Example GP Translations

Examples of Translated GPs [1]

SPARQL (sub)string

```
{ ?s ?p ?o }
```

group with a BGP consisting of a single TP

Translation

Join with Z, the identity

```
   Join( Z , BGP( ?s ?p ?o ) )
↝ BGP( ?s ?p ?o )
```

simplification removes Join

operator tree

triple patterns here are interpreted as scans that touch the data

from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Examples of Translated GPs [2]

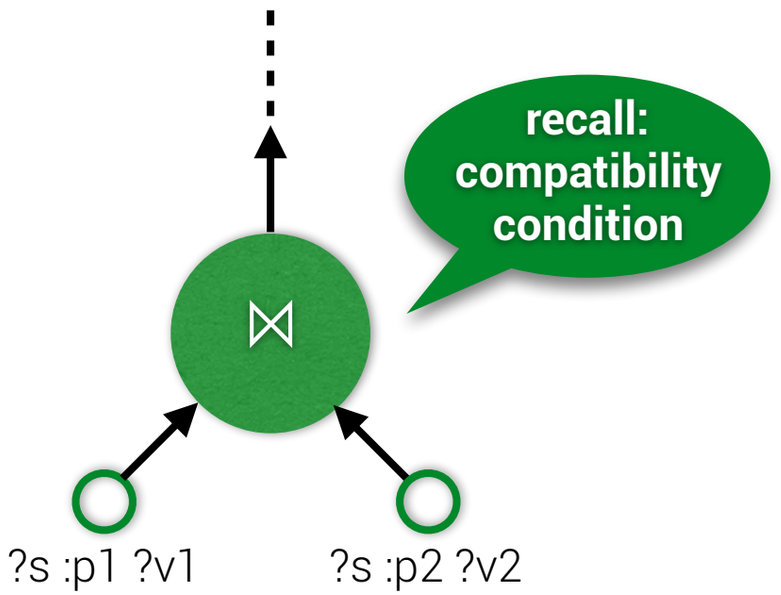
SPARQL (sub)string

```
{ ?s :p1 ?v1 ;
     :p2 ?v2 }
```

group with a BGP consisting of two TPs

Translation

```
BGP( ?s :p1 ?v1 .
     ?s :p2 ?v2 )
```



from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Examples of Translated GPs [3]

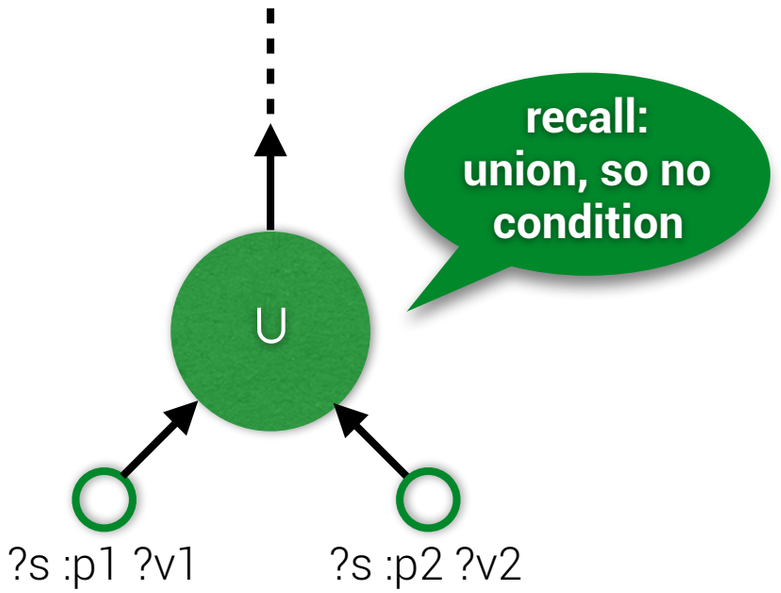
SPARQL (sub)string

```
{
  { ?s :p1 ?v1 }
UNION
  { ?s :p2 ?v2 }
}
```

group consisting of an AGP of two BGPs

Translation

```
  Union( Join( Z , BGP( ?s :p1 ?v1 )) ,
         Join( Z , BGP( ?s :p2 ?v2 )) )
↝ Union( BGP( ?s :p1 ?v1 ) ,
          BGP( ?s :p2 ?v2 ) )
```



from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Examples of Translated GPs [4]

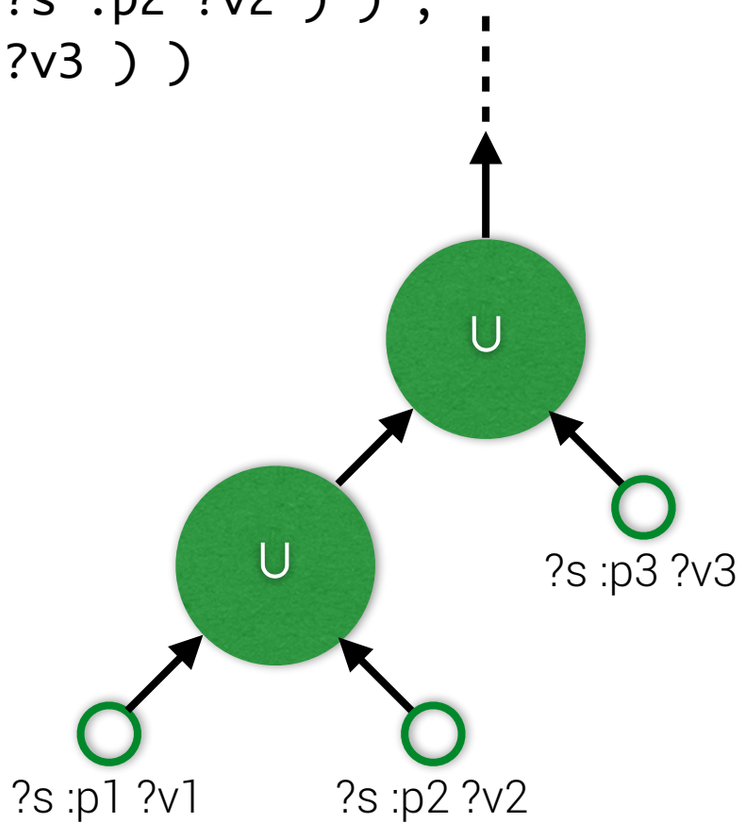
SPARQL (sub)string

```
{
  { ?s :p1 ?v1 }
UNION
  { ?s :p2 ?v2 }
UNION
  { ?s :p3 ?v3 }
}
```

group consisting of an AGP of an AGP and a BGP

Translation

```
Union(
      Union( Join( Z , BGP( ?s :p1 ?v1 ) ) ,
             Join( Z , BGP( ?s :p2 ?v2 ) ) ) ,
      Join( Z , BGP( ?s :p3 ?v3 ) ) )
→ Union(
       Union( BGP( ?s :p1 ?v1 ) ,
              BGP( ?s :p2 ?v2 ) ) ,
BGP( ?s :p3 ?v3 ) )
```



Examples of Translated GPs [5]

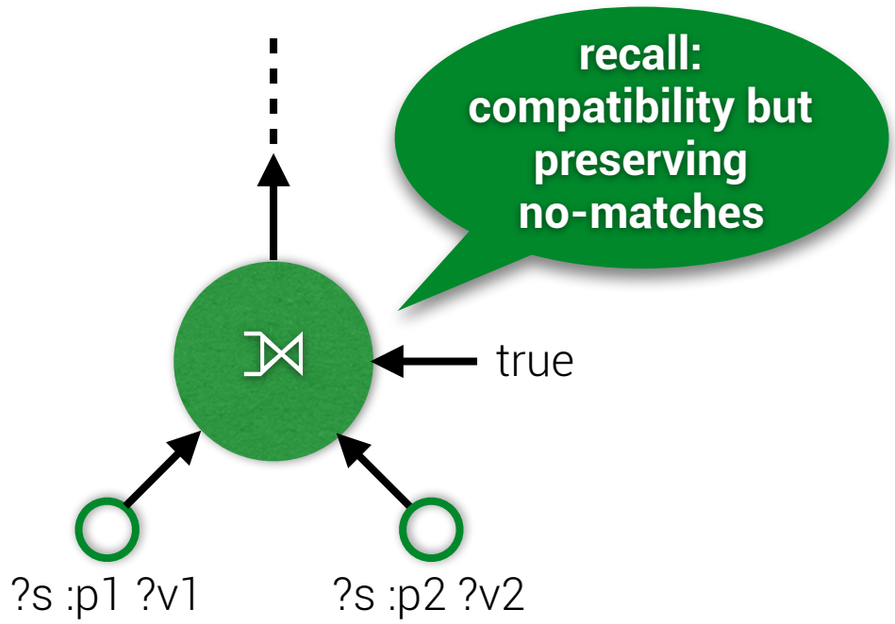
SPARQL (sub)string

```
{
  ?s :p1 ?v1
OPTIONAL
  { ?s :p2 ?v2 }
}
```

group consisting of a BGP and an (unconditional) OGP

Translation

```
  LeftJoin(
            Join( Z , BGP( ?s :p1 ?v1 ) ) ,
            Join( Z , BGP( ?s :p2 ?v2 ) ) ,
            true )
→ LeftJoin(
            BGP( ?s :p1 ?v1 ) ,
            BGP( ?s :p2 ?v2 ) ,
            true )
```



from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Examples of Translated GPs [6]

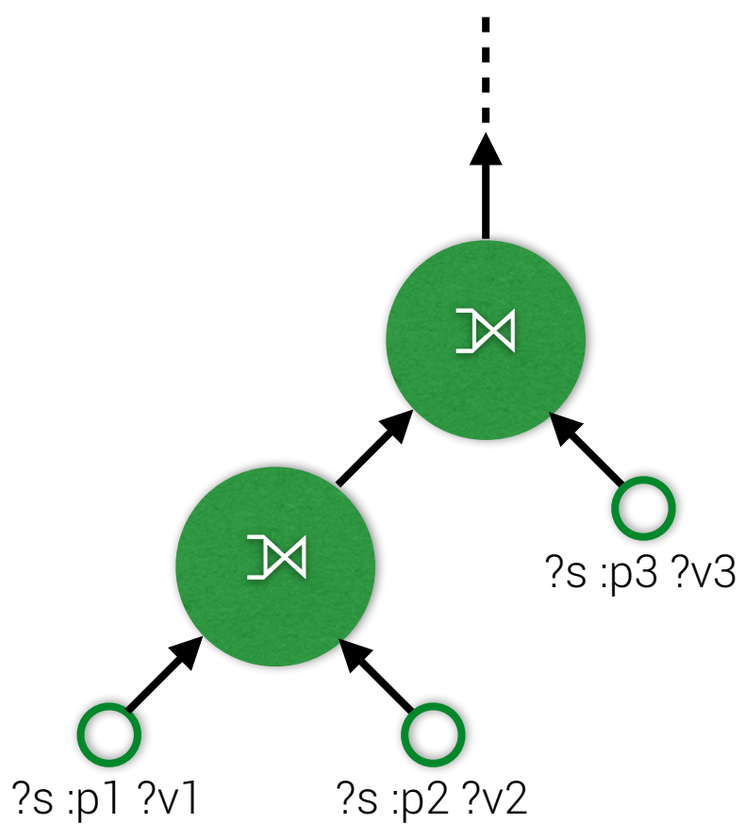
SPARQL (sub)string

```
{
  ?s :p1 ?v1
OPTIONAL
  { ?s :p2 ?v2 }
OPTIONAL
  { ?s :p3 ?v3 }
}
```

group consisting of a BGP and two (unconditional) OGPs

Translation

```
LeftJoin(
        LeftJoin(
           BGP( ?s :p1 ?v1 ) ,
           BGP( ?s :p2 ?v2 ) ,
           true ) ,
        BGP( ?s :p3 ?v3 ) ,
        true )
```



from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Examples of Translated GPs [7]

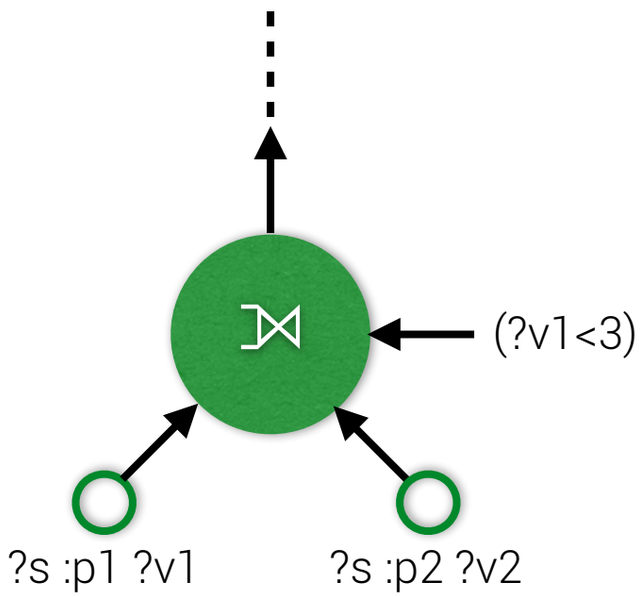
SPARQL (sub)string

```
{
  ?s :p1 ?v1
OPTIONAL
  { ?s :p2 ?v2
    FILTER( ?v1<3 )
  }
}
```

group consisting of a BGP and a conditional OGP

Translation

```
LeftJoin(
          Join( Z, BGP( ?s :p1 ?v1) ) ,
          Join( Z, BGP( ?s :p2 ?v2) ) ,
          ( ?v1<3 ) )
→ LeftJoin(
          BGP( ?s :p1 ?v1 ) ,
          BGP( ?s :p2 ?v2 ) ,
          ( ?v1<3 ) )
```



from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Examples of Translated GPs [8]

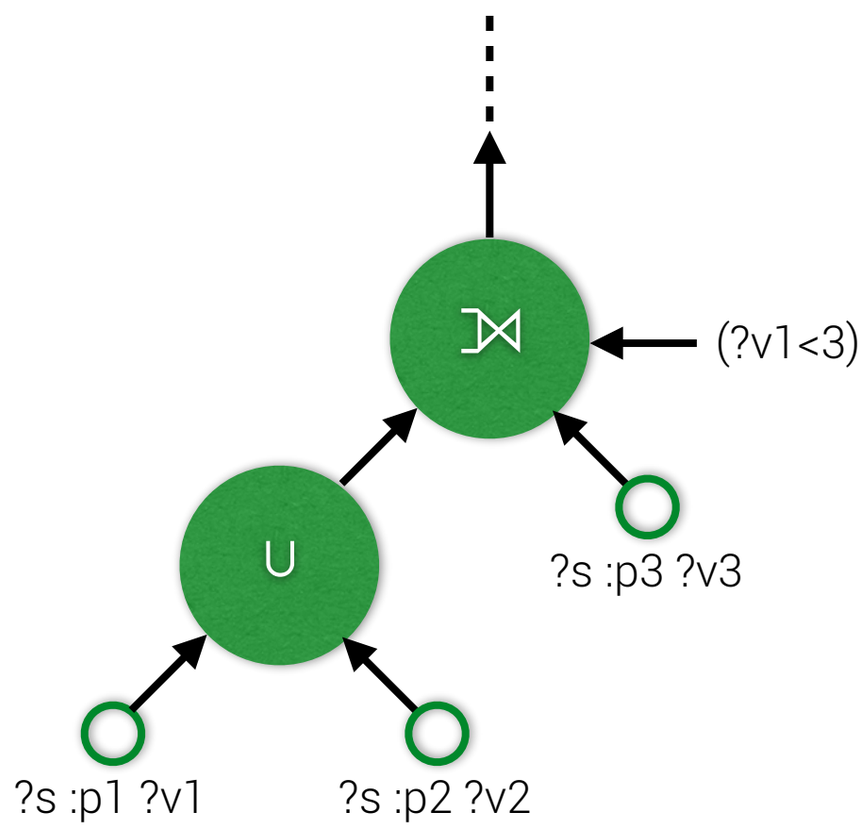
SPARQL (sub)string

```
{
  { ?s :p1 ?v1 }
UNION
  { ?s :p2 ?v2 }
  OPTIONAL
  { ?s :p3 ?v3 }
}
```

group consisting of an AGP and a unconditional OGP

Translation

```
LeftJoin(
        Union( BGP( ?s :p1 ?v1 ) ,
               BGP( ?s :p2 ?v2 ) ) ,
        BGP( ?s :p3 ?v3 ) ,
        true )
```



from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

Examples of Translated GPs [9]

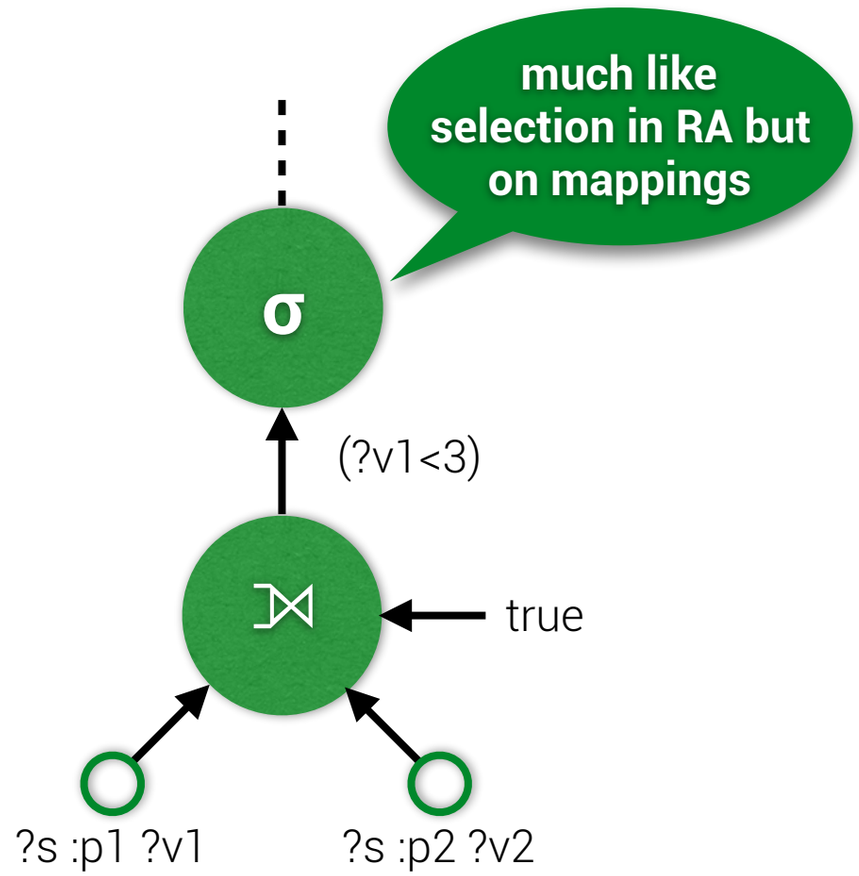
SPARQL (sub)string

```
{ ?s :p1 ?v1
  FILTER ( ?v1 < 3 )
OPTIONAL
  { ?s :p2 ?v2 }
}
```

group consisting of a BGP, a filter and an OGP

Translation

```
Filter( ?v1 < 3 ,
        LeftJoin( BGP( ?s :p1 ?v1 ) ,
                  BGP( ?s :p2 ?v2 ) ,
                  true) )
```

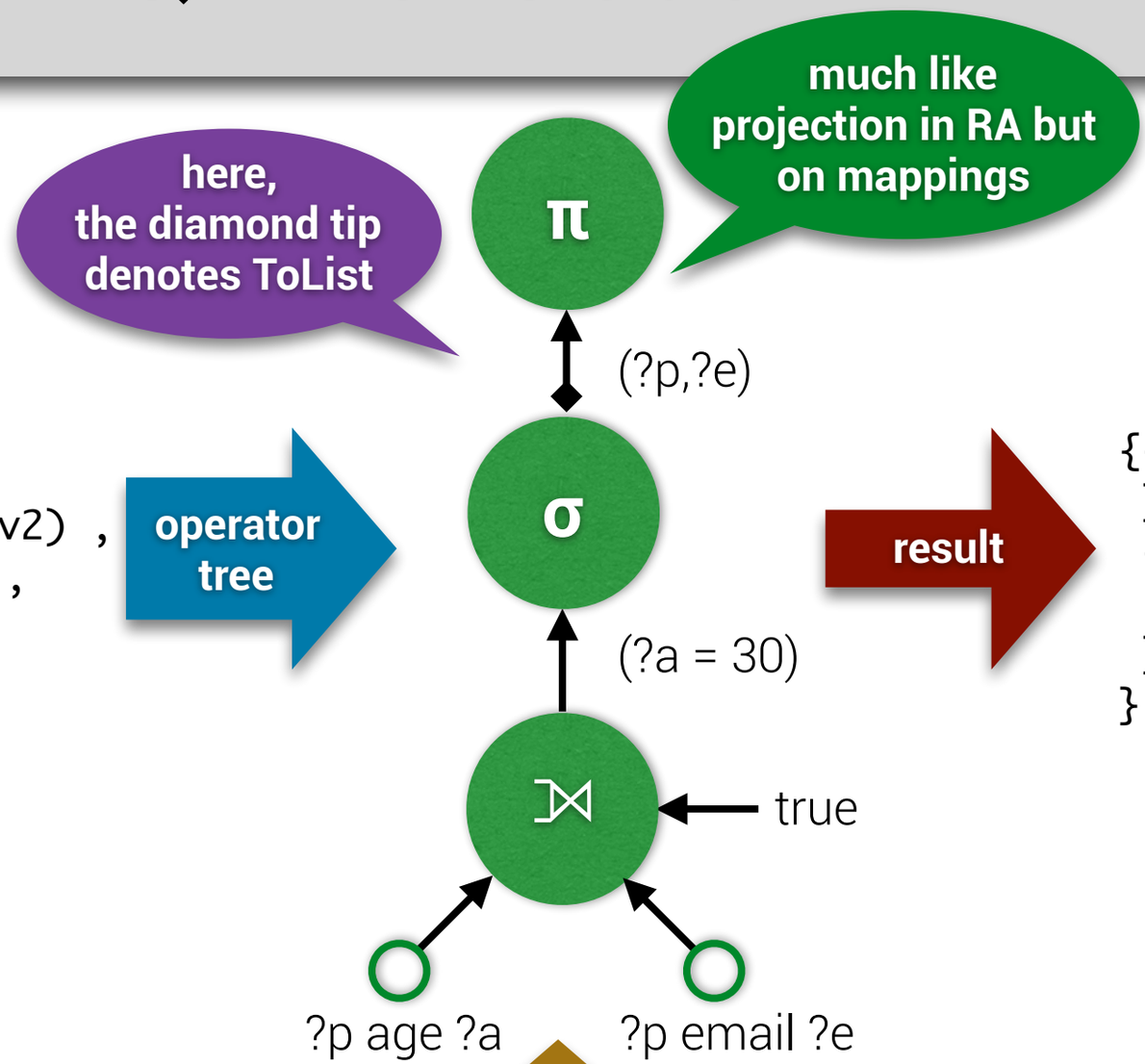


from: SPARQL 1.1 Query Language. S Harris, A Seaborne. (2012)

End-to-End SPARQL Translation

```
( Project
    ToList(
       Filter( ?v2 = 30 ,
          LeftJoin( BGP( ?v1 :p1 ?v2) ,
                BGP( ?v1 :p2 ?v3 ) ,
            true ) ) )
    (?v1 ?v3) )
```

operator tree



result

```
{{?p -> P1
 },
 {?p -> P3,
  ?e -> "joe@tld.com"
 }
}
```

translate

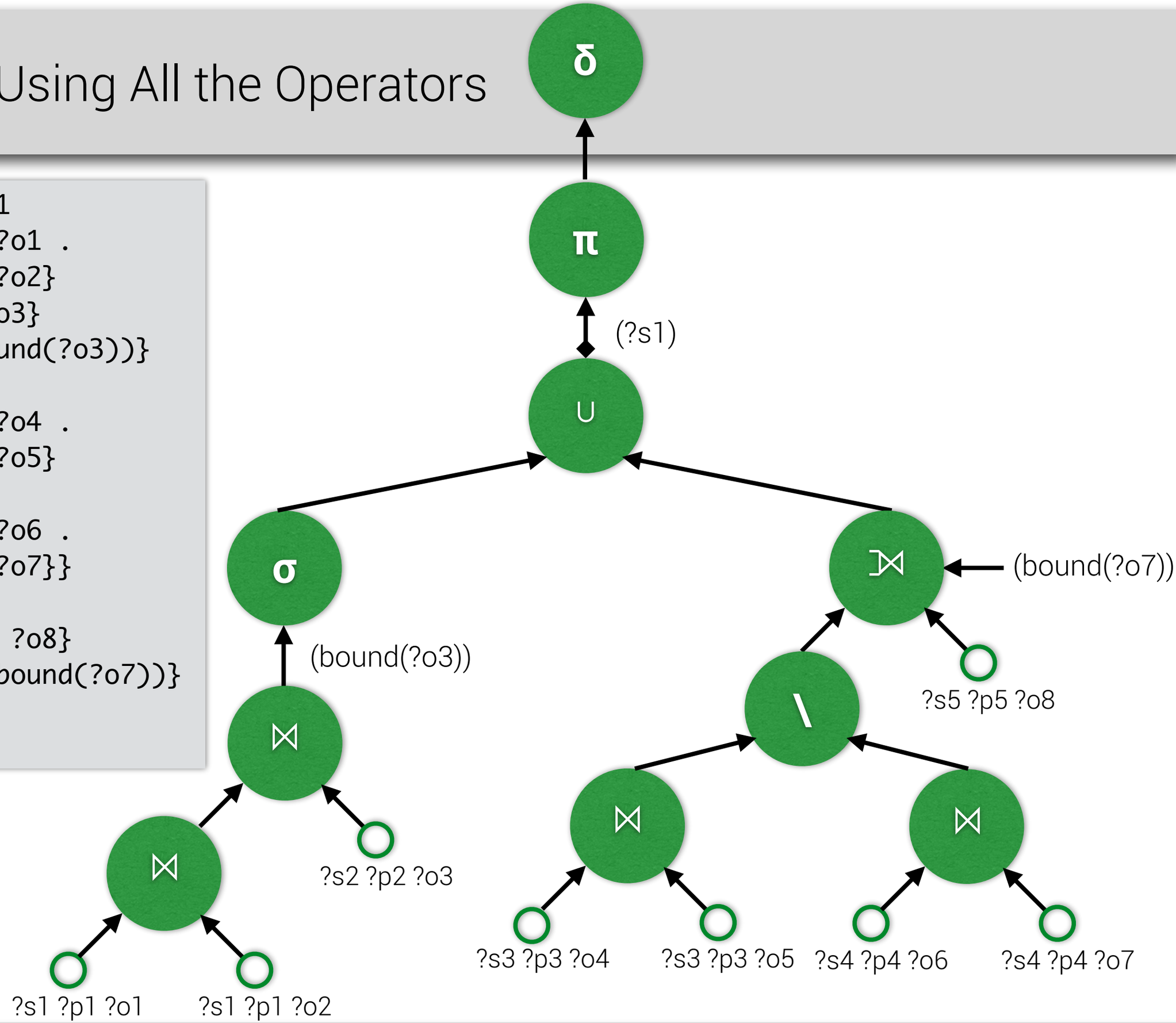
```
SELECT ?p ?e
WHERE
  { ?p age ?a
    FILTER ( ?a = 30 )
  OPTIONAL
    { ?p email ?e }
  }
```

document

```
{(P1, age,   "30"),
 (P2, age,   "29"),
 (P3, age,   "30"),
 (P3, email, "joe@tld.com")}
```

A Example Using All the Operators

```
SELECT DISTINCT ?s1
WHERE {{{{?s1 ?p1 ?o1 .
          ?s1 ?p1 ?o2}
         ?s2 ?p2 ?o3}
        FILTER (bound(?o3))}
      UNION
       {{{?s3 ?p3 ?o4 .
          ?s3 ?p3 ?o5}
        MINUS
         {?s4 ?p4 ?o6 .
          ?s4 ?p4 ?o7}}
       OPTIONAL
         {{?s5 ?p5 ?o8}
          FILTER (bound(?o7))}
       }
      }
```



More Examples and Translation Cases

- The W3C standard has additional examples of patterns involving BIND, MINUS, subqueries, which we omit here.
- We also omit here the translation procedure for grouping, aggregates, the HAVING and the VALUES clause, and of SELECT expressions.
- Solutions modifiers apply to the processing of a SPARQL query after pattern matching.
- The solution modifiers are applied to a query in the following order:
  - Order by, which orders the solutions
  - Projection, which retains only the requested variables
  - Distinct, which removes duplicate solutions
  - Reduced, which allows duplicate solutions to be retained or removed
  - Offset, which defines the start of the requested subsequence of solutions
  - Limit, which defines the length of the requested subsequence of solutions
- The translation of solutions modifiers is a straightforward wrapping (see the W3C standard).
- The minor exception is that Offset and Limit are translated as a pair into a slicing operation.

MANCHESTER 1824

The University of Manchester

SA Core Abstract Syntax

SPARQL as a Query Calculus

- In the same way that SQL is based on the tuple relational calculus, SPARQL is based on the domain relational calculus.
- As will be seen, the evaluation of the most common kind of SPARQL query returns sets of mappings each of which binds a set of variables, each to a value.
- So, roughly speaking, and by way of analogy, while in SQL one obtains a handle on an entire tuple/row, in SPARQL one obtains a handle on an attribute/column.
- This more or less follows from the absence of regular structure in SPARQL, which implies that the properties one knows of an entity of a given type may not be the same as those of another entity of the same type.

SPARQL :: Patterns and Operators

- The semantics of simple SPARQL queries is easy to understand, at least intuitively.
- But, in combination, several features of pattern matching on graphs make the meaning of some queries less immediately obvious.
- Such features include:
  - Grouping
  - Optional parts
  - Nesting
  - Union of patterns
  - Filtering
- A formal semantics becomes crucial as syntactic variety and complexity grows.

```
SELECT ?Name ?Email
WHERE
 {
  ?X :name ?Name
  ?X :email ?Email
 }
```

Return the name and email of the resources in the data source.

Return up to 10 s1 that … what ?

```
SELECT ?s1
FROM <file:timbl-foaf.rdf>
{ { { { ?s1 ?p1 ?o1 ;
        ?p2 ?o2 .
      } OPTIONAL { ?s1 ?p3 ?o3 }
    } .
    { { ?s2 ?p4 ?o4 .
        ?s3 ?p4 ?o4 .
      } OPTIONAL { ?s4 ?p4 ?o4 .}
    } OPTIONAL { ?s5 ?p5 ?o5 . }
  }
UNION
  { { ?s6 ?p6 ?o6 .
      FILTER ( bound(?s6) )
    }
  }
}
LIMIT 10
```

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

A Formal Approach

- A formal approach is beneficial for:
  - Clarifying corner cases
  - Helping in the implementation process
  - Providing sound foundations
- We will briefly look at:
  - A formal compositional semantics based on (Schmidt et al., 2010).
  - This formalization is the starting point of the official semantics of the SPARQL language by the W3C.
  - We will describe the set semantics.
  - For the bag semantics, including conditions where it coincides with the set semantics, see (Schmidt et al., 2010).

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

The SPARQL Language: Data Model

- Let B, L, and U be pairwise disjoint sets, referred as the set of blank nodes, literals and URIs, respectively.
- We notate literals as quoted strings and blank nodes as strings prefixed by the substring "_:".
- When sets are denoted by single letters (as above), the concatenation of their denotations denote their union, so that BLU = B∪L∪U, for example.
- Let (s, p, o) ∈ BU×U×BLU be called a triple, with s, p, and o being referred to as its subject, predicate and object , respectively.
- A database (also called a document) is a finite set of triples.

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

The SPARQL Language: Core Syntax [1]

- Let V be a set of variables disjoint from BLU.
- We notate variables as strings prefixed by the substring "?".
- We first define filter conditions (also called value constraints) recursively, as follows.
- For ?x, ?x′ ∈ V and c, c′ ∈ LU, the following expressions are atomic filter conditions:
  - ?x = c
  - ?x = ?x′
  - c = c′
  - bound(?x)
- If R and R′ are filter conditions, the following expressions are filter conditions:
  - ¬R
  - R∧R′
  - R∨R′
- We denote by vars(R) the set of variables occuring in the filter condition R.

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

The SPARQL Language: Core Syntax [2]

- A SPARQL expression is an expression that is built recursively, as follows:
  - A triple pattern t ∈ UV×UV×LUV is a SPARQL expression.
  - If E and E′ are SPARQL expressions and R is a filter condition, then the following expressions are SPARQL expressions:
    - E FILTER R
    - E AND E′
    - E UNION E′
    - E OPTIONAL E′
  - We abbreviate OPTIONAL to OPT in what follows.

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

The SPARQL Language: Core Syntax [3]

- From SPARQL expressions, four types of SPARQL query can be built, viz., SELECT, ASK, CONSTRUCT and DESCRIBE.
- We will focus on SELECT and ASK.
- The type of a SELECT query is set of mappings; the type of an ASK query is Boolean.
- Let $S \subset V$ be a finite set of variables and let E be a SPARQL expression.
- Then, the following expressions are SPARQL queries (here):
  - $SELECT_S(E)$
  - ASK(E)
- To reduce clutter, we write S without curly brackets.

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

Core Syntax: Examples [1]

- The following are example triple patterns (TPs):
  - (?X, name, ?Name)
  - (?X, ?Y, "Georg")
  - (?X, worksIn, ?W)

Core Syntax: Examples [2]

- The following are example SPARQL expressions:
  - (?X, name, ?Name)
  - ((?X, name, ?Name) AND (?X, worksIn, "Freiburg"))
  - (((?X, name, ?Name) AND (?X, worksIn, "Freiburg")) FILTER bound(?Name))
  - ((((?X, name, ?Name) AND (?X, worksIn, "Freiburg")) FILTER bound(?Name)) OPT (?X position "Professor"))
  - (((((?X, name, ?Name) AND (?X, worksIn, "Freiburg")) FILTER bound(?Name)) OPT (?X position "Professor")) UNION ?Y coAuthor ?X)

Core Syntax: Examples [3]

- The following are example SPARQL queries:
  - ASK((?X, name, ?Name))
  - $SELECT_{?X}$((?X, name, ?Name) AND (?X, worksIn, "Freiburg"))

MANCHESTER 1824

The University of Manchester

A Formal Semantics for SPARQL

Mappings: Building Blocks for the Semantics [1]

- Mappings express variable-to-document bindings produced by evaluation.
- A *mapping* μ is a partial function from variables V to the set of RDF terms BLU, i.e., μ: V → BLU
- We denote the universe of all mappings by $\mathscr{M}$.
- The domain of a mapping μ, denoted by dom(μ), is the subset of V for which μ is defined.

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

Mappings: Building Blocks for the Semantics [2]

- We say that two mappings μ and μ′ are compatible, denoted by μ~μ′, if they agree on their shared variables, i.e.,
  - ∀?x ∈ dom(μ) ∩ dom(μ′)→μ(?x)=μ′(?x).
- By overloading, we denote by vars(t) the set of variables occuring in the triple pattern t.
- By overloading, we denote by μ(t), the triple pattern that results from replacing all variables ?x ∈ dom(μ) ∩ vars(t) in t with μ(?x).

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

Mappings: Examples

- Consider the following mappings:
  - μ1 := {?x↦a1}
  - μ2 := {?x↦a2, ?y↦b2}
  - μ3 := {?x↦a1, ?z↦c1}
- Then:
  - dom(μ1) = {?x}
  - dom(μ2) = {?x, ?y}
  - dom(μ3) = {?x. ?z}
- Furthermore:
  - μ1~μ3≁
  - μ1≁μ2
  - μ2≁μ3
- Finally, given t := (c, ?x, ?y), we have:
  - vars(t) = {?x, ?y}
  - μ2(t) = (c, a2, b2)

The SPARQL Language: Semantics [1]

- We first define the semantics of filter conditions w.r.t. mappings.
- We write $\mu \models R$ to assert that a mapping μ satisfies a filter condition R.
- We now define this relation, recursively.
- If R and R′ are filter conditions, a mapping μ satisfies a filter condition of (recursively) the form:
  - $?x = c$ if $\mu(?x) = c$
  - $?x = ?x'$ if $\mu(?x) = \mu(?x')$
  - $c = c'$ if $c = c'$
  - bound(?x) if $?x \in dom(\mu)$
  - $\neg R$ if $\mu \not\models R$
  - $R \wedge R'$ if $\mu \models R$ and $\mu \models R'$
  - $R \vee R'$ if $\mu \models R$ or $\mu \models R'$

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

Semantics: Examples [1]

- Consider the following mappings:
  - μ1 := {?x↦a1}
  - μ2 := {?x↦a2, ?y↦b2}
  - μ3 := {?x↦a1, ?z↦c1}

- Now, consider the following filter conditions:
  1. ?x = a1
  2. ?x = ?y
  3. a1 = a2
  4. bound(?y)
  5. ¬(?x = a1)
  6. (?x = a2 ∧ bound(?y))
  7. (?x = a2 ∨ ?z = c1)

- Then, we have that:
  - I. μ1 ⊨ (1), μ2 ⊭ (1), μ3 ⊨ (1)
  - II. μ1 ⊭ (2), μ2 ⊭ (2), μ3 ⊭ (2)
  - III. ∀μ ∈ $\mathcal{M}$: μ ⊭ (3)
  - IV. μ1 ⊭ (4), μ2 ⊨ (4) μ3 ⊭ (4)
  - V. μ1 ⊭ (5), μ2 ⊨ (5), μ3 ⊭ (5)
  - VI. μ1 ⊭ (6), μ2 ⊨ (6), μ3 ⊭ (6)
  - VII. μ1 ⊭ (7), μ2 ⊨ (7), μ3 ⊨ (7)

The SPARQL Language: Semantics [2]

- The solution of a SPARQL expression or query over a document D is a set of mappings, each element of which represents a possible answer.
- The semantics of SPARQL query evaluation is then defined using an algebra over sets of mappings.
- Let M and M′ be sets of mappings, let R be a filter condition, and let S⊂V be a finite set of variables.

- We define the SPARQL set algebra (SA) as comprising the following operations:
  - $M \bowtie M' := \{\mu \cup \mu' \mid \mu \in M \wedge \mu' \in M' \wedge \mu \sim \mu'\}$
  - $M \cup M' := \{\mu \mid \mu \in M \vee \mu \in M'\}$
  - $M \setminus M' := \{\mu \in M \mid \forall \mu' \in M' : \mu \not\sim \mu'\}$
  - $M \ltimes M' := (M \bowtie M') \cup (M \setminus M')$
  - $\sigma_R(M) := \{\mu \in M \mid \mu \models R\}$
  - $\pi_S(M) := \{\mu \mid \exists \mu' : \mu \cup \mu' \in M \wedge dom(\mu) \subseteq S \wedge dom(\mu') \cap S = \varnothing\}$

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

The SPARQL Language: Semantics [3]

- We define the result of evaluating SPARQL SELECT and ASQ queries against a document D by defining a function $[\![.]\!]_D$ that translates them into the SA defined above.
- Let D be a document, let t be a triple pattern, let Q and Q′ be SPARQL expressions, let R be a filter condition, and let $S \subset V$ be a finite set of variables.
- We define a function $[\![.]\!]_D$: from SPARQL expressions to SA expressions recursively as follows:

- $[\![t]\!]_D := \{\mu \mid dom(\mu) = vars(t) \text{ and } \mu(t) \in D\}$
- $[\![Q \text{ AND } Q']\!]_D := [\![Q]\!]_D \bowtie [\![Q']\!]_D$
- $[\![Q \text{ OPT } Q']\!]_D := [\![Q]\!]_D \mathbin{⟕} [\![Q']\!]_D$
- $[\![Q \text{ UNION } Q']\!]_D := [\![Q]\!]_D \cup [\![Q']\!]_D$
- $[\![Q \text{ FILTER } R]\!]_D := \sigma_R([\![Q]\!]_D)$
- $[\![SELECT_S(Q)]\!]_D := \pi_S([\![Q]\!]_D)$
- $[\![ASK(Q)]\!]_D := \neg(\varnothing = [\![Q]\!]_D)$

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

Semantics: Examples [2]

- Consider the SPARQL SELECT query Q:

Q := SELECT ?p,?e

(((?p, age, ?a)

OPT

(?p, email, ?e))

FILTER (?a=30))

- Q retrieves all 30-year old persons (?) and, if available, their email (?w).

- Let the database be:

D := {(p1, age, "30"),

(p2, age, "29"),

(p3, age, "30"),

(p3, email, "joe@tld.com")}

- Then,

$[\![Q]\!]_D$ = {{?p↦p1},

{?p↦p3},

{?e↦"joe@tld.com"}}

More Examples: Mappings

μ = {?X → R1,
?Y → R2,
?Name → john,
?Email → J@ed.ex
}

a mapping

Note:
dom(μ) = {?X, ?Y, ?Name, ?Email}

P ={(?X, name, ?Name),
(?X, email, ?Email)
}

a BGP, i.e., a conjunction of TPs

μ(P) = {(R1, name, john),
(R1, email, J@ed.ex)
}

the BGP with every occurrence of every variable in dom(μ) replaced by its corresponding value

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

More Examples: Evaluating BGPs [1]

a graph

G = {(R1, name, john),
(R1, email, J@ed.ex),
(R2, name, paul)
}

a BGP

⟦{(?X, name, ?Y)}⟧G

{μ1 ={?X →R1, ?Y →john}
μ2 ={?X →R2, ?Y →paul}}

the answer, a set of mappings

a set of mappings can be thought of as a table, the domain as column name, each row a mapping, each cell a binding

	?X	?Y
μ1	R1	john
μ2	R2	paul

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

More Examples: Evaluating BGPs [2]

G = {(R1, name, john),
(R1, email, J@ed.ex),
(R2, name, paul)
}

a graph

⟦{(?X, name, ?Y),(?X, email, ?Z)}⟧G

a BGP

{μ = {?X → R1,?Y → john,?Z → J@ed.ex}}

the answer, a set of mappings

the answer, in tabular form

	?X	?Y	?Z
μ1	R1	john	J@ed.ex

More Examples: Evaluating BGPs [3]

a graph

G = {(R1, name, john),
(R1, email, J@ed.ex),
(R2, name, paul)
}

⟦{(R1, webPage, ?W)}⟧G

{ }

evaluates to the empty set of mappings, i.e., no answers

⟦{(R3, name, ringo)}⟧G

{ }

ditto

⟦{(R2, name, paul)}⟧G

{ μ = { } }

evaluates to a singleton whose element is the empty mapping, i.e., true

the empty BGP

⟦{ }⟧G

{ μ = { } }

is true

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

More Examples: Compatible Mappings [1]

μ0: { }
μ1: { ?x → "foo" , ?y → <http://example/y> }
μ2: { ?y → <http://example/y> }
μ3: { ?x →"bar" }

μ0, μ1 and μ2 are compatible
μ0 and μ3 are compatible
μ2 and μ3 are compatible
μ1 and μ1 are not compatible

More Examples: Compatible Mappings [2]

	?X	?Y	?U	?V
μ1	R1	john		
μ2	R2		J@edu.ex	
μ3			P@edu.ex	R2
μ1 ∪ μ2	R1	john	J@edu.ex	
μ1 ∪ μ3	R1	john	P@edu.ex	R2

μ1 and μ2 are compatible

μ1 and μ3 are compatible

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

More Examples: Evaluating AND

G = { (R1, name, john),
(R2, name, paul),
(R3, name, ringo),
(R1, email, J@ed.ex),
(R3, email, R@ed.ex),
(R3, webPage, www.ringo.com) }

⟦{(?X, name, ?N)} AND {(?X, email, ?E)}⟧$_G$ →

⟦{(?X, name, ?N)}⟧$_G$ ⋈ ⟦{(?X, email, ?E)}⟧$_G$ →

a graph

a graph pattern expression

the corresponding SA-algebraic expression

final result

	?X	?Y
μ1	R1	john
μ2	R2	paul
μ3	R3	ringo

evaluated arguments

⋈

	?X	?E
μ4	R1	J@ed.ex
μ5	R3	R@ed.ex

→

	?X	?N	?E
μ1 ∪ μ4	R1	john	J@ed.ex
μ3 ∪ μ5	R3	ringo	R@ed.ex

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

More Examples: Evaluating OPT

G = { (R1, name, john),
(R2, name, paul),
(R3, name, ringo),
(R1, email, J@ed.ex),
(R3, email, R@ed.ex),
(R3, webPage, www.ringo.com) }

⟦{(?X, name, ?N)} OPT {(?X, email, ?E)}⟧$_G$ →

⟦{(?X, name, ?N)}⟧$_G$ ⟕ ⟦{(?X, email, ?E)}⟧$_G$ →

a graph

a graph pattern expression

the corresponding SA-algebraic expression

final result

	?X	?Y
μ1	R1	john
μ2	R2	paul
μ3	R3	ringo

evaluated arguments

⋈

	?X	?E
μ4	R1	J@ed.ex
μ5	R3	R@ed.ex

→

	?X	?N	?E
μ1 ∪ μ4	R1	john	J@ed.ex
μ3 ∪ μ5	R3	ringo	R@ed.ex
μ2	R2	paul	

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

More Examples: Evaluating UNION

G = { (R1, name, john),
(R2, name, paul),
(R3, name, ringo),
(R1, email, J@ed.ex),
(R3, email, R@ed.ex),
(R3, webPage, www.ringo.com) }

a graph

a graph pattern expression

the corresponding SA-algebraic expression

final result

$[\![\{(?X, email, ?I)\}\ \text{UNION}\ \{(?X, webpage, ?I)\}]\!]_G \rightsquigarrow$

$[\![\{(?X, email, ?I)\}]\!]_G \cup [\![\{(?X, webpage, ?I)\}]\!]_G \rightsquigarrow$

	?X	?I
μ1	R1	J@ed.ex
μ2	R3	R@ed.ex

evaluated arguments

$\bowtie$

	?X	?I
μ3	R1	www.ringo.com

$\rightsquigarrow$

	?X	?E
μ1	R1	J@ed.ex
μ2	R3	R@ed.ex
μ3	R3	www.ringo.com

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

More Examples: Evaluating FILTER

G = { (R1, name, john),
(R2, name, paul),
(R3, name, ringo),
(R1, email, J@ed.ex),
(R3, email, R@ed.ex),
(R3, webPage, www.ringo.com) }

$[\![(\{(?X, name, ?N)\}\ FILTER(?N = ringo \vee ?N = paul))]\!]_G \rightsquigarrow$

a graph

a graph pattern expression

the corresponding SA-algebraic expression

final result

evaluated argument

	?X	?N
μ1	R1	john
μ2	R2	paul
μ3	R3	ringo

filter on evaluated argument

$?N = ringo \vee ?N = paul \rightsquigarrow$

	?X	?N
μ2	R2	paul
μ3	R3	ringo

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

More Examples: Evaluating Complex Expressions

G = { (R1, name, john),
(R2, name, paul),
(R3, name, ringo),
(R1, email, J@ed.ex),
(R3, email, R@ed.ex),
(R3, webPage, www.ringo.com) }

$[\![(\{(?X, name, ?N)\} \text{ OPT } \{(?X, email, ?E)\}) \text{ FILTER } \neg bound(?E))]\!]_G \rightsquigarrow$

	?X	?N	?E
μ1 ∪ μ4	R1	john	J@ed.ex
μ3 ∪ μ5	R3	ringo	R@ed.ex
μ3	R2	paul	

¬bound(?E) ⇝

	?X	?N
μ2	R2	paul

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

X

Some Comments [1]

- The result of a join of M and M′ comprises the mappings in M that are compatible with the the mappings in M′.
- This is related but strictly different from the join of two relations in relational database theory.
- The result of a union of M and M′ comprises the mappings in M and all the mappings in M′.
- Therefore, union is indeed classical set union.
- The result of subtracting M′ from M (i.e., M\M′) comprises the mappings in M that are incompatible with every mapping in M′.
- The result of a left outer join of M and M′ comprises the mappings in M that are compatible with mappings in M′, *as well as* all the mappings in M that are incompatible with every mapping in M′.

Some Comments [2]

- The result of a join of M and M′ comprises the mappings in M that are compatible with the the mappings in M′.
- This is related but strictly different from the join of two relations in relational database theory.
- The result of a union of M and M′ comprises the mappings in M and all the mappings in M′.
- Therefore, union is indeed classical set union.
- The result of subtracting M′ from M (i.e., M\M′) comprises the mappings in M that are incompatible with every mapping in M′.
- The result of a left outer join of M and M′ comprises the mappings in M that are compatible with mappings in M′, *as well as* all the mappings in M that are incompatible with every mapping in M′.

Some Comments [3]

- A FILTER expression is said to be *safe* if the set of variables occurring in the value constraint R is a subset of the set of variables occurring in the graph pattern P.
- The account in (Arenas et al, 2007) is restricted to the case in which all FILTER expressions are safe.
- While this restriction is not imposed in the semantics specified by W3C, it does not change the expressive power of the language.
- (Arenas et al, 2007) also give a semantics for queries involving named graphs.

from: SPARQL Formalization. M Arenas, C Gutierrez, J Pérez (2007)

Lecture 18

MANCHESTER 1824

The University of Manchester

Logical Optimization in SPARQL

Algebraic Equivalences in SA [1]

- The next slides present a set of algebraic equivalences over SA expressions as defined above.
- They hold for the full SA $\mathcal{A}$, which comprises expressions in which ∪, ⋈, ⟕, ⟖, σ, π and triple patterns can occur.
- In query optimization, we interpret them, as is the case with similar equivalences between RA expression, as rewrinting rules.

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

Algebraic Equivalences in SA [2]

- We only cover the case of set-based SA, but later we explain which equivalences for set-based SA are also valid for bag-based SA.
- Also, a few (but useful) equivalences that do not hold for $\mathcal{A}$, do hold for a fragment $\mathcal{A}^{\sim} \subset \mathcal{A}$, which is defined in full detail in Schmidt et al. 2010.
- The presentation begins with the definition of certain variables and possible variables.
- This is followed by the presentation of a set of algebraic equivalences that cover all the algebraic operators in $\mathcal{A}$.

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

MANCHESTER 1824

The University of Manchester

Properties of Variables

Certain Variables

- Given SA expressions A, A′, and A″, t a triple pattern, R a filter condition, and S⊂V a finite set of variables, we recursively define a function cVars(.) that returns the set of certain variables in the argument SA expression(s), as follows:
  - cVars($[\![t]\!]_D$) := vars(t)
  - cVars(A⋈A′) := cVars(A) ∪ cVars(A′)
  - cVars(A∪A′) := cVars(A∪A′) ∩ cVars(A∪A′)
  - cVars(A∖A′) := cVars(A)
  - cVars(σ_R(A)) := cVars(A)
  - cVars(π_S(A)) := cVars(A) ∩ S
- The definitions above are for set semantics, but can be generalized.
- Also, cVars(A⟕A′) follows from the cases above.
- Note that the function is in fact independent of any input document D.

Possible Variables

- Similarly, we recursively define a function pVars(.) that returns the set of possible variables in the argument SA expression(s), as follows:
  - $pVars([\![t]\!]_D) := vars(t)$
  - $pVars(A \bowtie A') := pVars(A) \cup pVars(A')$
  - $pVars(A \cup A') := pVars(A) \cup pVars(A')$
  - $pVars(A \setminus A') := pVars(A)$
  - $cVars(\sigma_R(A)) := cVars(A)$
  - $pVars(\pi_S(A)) := cVars(A) \cap S$
- Again, the definitions above are for set semantics, but can be generalized.
- Note that the only non-obvious difference in the definition of pVars w.r.t. that of cVars is for $A \cup A'$.
- Note again the independence of any input document D.

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

Certain v. Possible Variables

- Both functions are efficiently computable independenly of an input document.
- The following two properties hold:
  - If ?x ∈ cVars(A), then ?x belongs to every mapping set that is an answer to A.
  - For every mapping set μ that is an answer to A, if ?x ∈ dom(μ), then ?x ∈ pVars(A)

- As an example, let A be the following SPARQL expression:

$\pi_{?x.?y}((\llbracket(a,q,?x)\rrbracket_D \bowtie \llbracket(a,?y,?x)\rrbracket_D) \cup \llbracket(a,p,?x)\rrbracket_D)$

- Then, it follows that:
  - pVars(A) = {?x, ?y}
  - cVars(A) = {?x}

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

MANCHESTER 1824

The University of Manchester

Algebraic Equivalences

SA Algebraic Equivalences [1]

Idempotence and Inverse	
UI	$A \cup A \equiv A$
IN	$A \setminus A \equiv \varnothing$

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

SA Algebraic Equivalences [2]

Associativity, Commutativity, Distributivity	
UA	$(A \cup A') \cup A'' \equiv A \cup (A' \cup A'')$
JA	$(A \bowtie A') \bowtie A'' \equiv A \bowtie (A' \bowtie A'')$
UC	$A \cup A' \equiv A' \cup A$
JC	$A \bowtie A' \equiv A' \bowtie A$
JUDR	$(A \cup A') \bowtie A'' \equiv (A \bowtie A'') \cup (A' \bowtie A'')$
JUDL	$A \bowtie (A' \cup A'') \equiv (A \bowtie A') \cup (A \bowtie A'')$
MUDR	$(A \cup A') \setminus A'' \equiv (A \setminus A'') \cup (A' \setminus A'')$
LUDL	$(A \cup A') \ltimes A'' \equiv (A \ltimes A'') \cup (A' \ltimes A'')$

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

SA Algebraic Equivalences [3]

Projection Pushing		
PBI	$\pi_{pVars(A) \cup S}(A) \equiv A$	
PBII	$\pi_S(A) \equiv \pi_{pVars(A) \cap S}(A)$	
PFP	$\pi_S(\sigma_R(A)) \equiv \pi_S(\sigma_R(\pi_{vars(R) \cup S}(A)))$	
PM	$\pi_{S1}(\pi_{S2}(A)) \equiv \pi_{S1 \cap S2}(A)$	
PUP	$\pi_S(A \cup A') \equiv \pi_S(A) \cup \pi_S(A')$	
PJP	$\pi_S(A \bowtie A') \equiv \pi_S(\pi_{S1}(A) \bowtie \pi_{S1}(A'))$	S2:=pVars(A)∩pVars(A'); S1:=S∪S2
PMP	$\pi_S(A \setminus A') \equiv \pi_S(\pi_{S1}(A) \setminus \pi_{S2}(A'))$	ditto
PLP	$\pi_S(A \ltimes A') \equiv \pi_S(\pi_{S1}(A) \ltimes \pi_{S1}(A'))$	ditto

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

SA Algebraic Equivalences [4]

Filter Decomposition and Bound-Predicate Elimination		
FDI	$\sigma_{R1 \wedge R2}(A) \equiv \sigma_{R1}(\sigma_{R2}(A))$	
FDII	$\sigma_{R1 \vee R2}(A) \equiv \sigma_{R1}(A) \cup \sigma_{R2}(A)$	
FR	$\sigma_{R1}(\sigma_{R2}(A)) \equiv \sigma_{R2}(\sigma_{R1}(A))$	
FBI	$\sigma_{bound(?x)}(A) \equiv A$	if $?x \in cVars(A)$
FBII	$\sigma_{bound(?x)}(A) \equiv \varnothing$	if $?x \notin pVars(A)$
FBIII	$\sigma_{\neg bound(?x)}(A) \equiv \varnothing$	if $?x \in cVars(A)$
FBIV	$\sigma_{\neg bound(?x)}(A) \equiv A$	if $?x \notin pVars(A)$

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

SA Algebraic Equivalences [5]

Filter Elimination		
FEI	$\pi_{S\setminus\{?x\}}(\sigma_{?x=?y}(A)) \equiv \pi_{S\setminus\{?x\}}(A[?x/?y])$	if A is made of $\bowtie$, $\cup$ and TPs only, then when $W \in V$ or when $W \in LU$, let us denote by A[?x/W] the result of replacing all occurrences of ?x with W in A
FEII	$\pi_{S\setminus\{?x\}}(\sigma_{?x=c}(A)) \equiv \pi_{S\setminus\{?x\}}(A[?x/c])$	

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

SA Algebraic Equivalences [6]

Filter Pushing		
FUP	$\sigma_R(A \cup A') \equiv \sigma_R(A) \cup \sigma_R(A')$	
FMP	$\sigma_R(A \setminus A') \equiv \sigma_R(A) \setminus A'$	
FJP	$\sigma_R(A \bowtie A') \equiv \sigma_R(A) \bowtie A'$	if $\forall ?x \in vars(R)$: $?x \in cVars(A) \vee ?x \notin pVars(A')$
FLP	$\sigma_R(A \ltimes A') \equiv \sigma_R(A) \ltimes A'$	ditto

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

SA Algebraic Equivalences [7]

Minus and Left Outer Join Rewriting		
MR	$(A \setminus A') \setminus A'' \equiv (A \setminus A'') \setminus A'$	
MMUC	$(A \setminus A') \setminus A'' \equiv A \setminus (A' \cup A'')$	
MJ	$A \setminus A' \equiv A \setminus (A \bowtie A')$	
FLBI	$\sigma_{\neg bound(?x)}(A \mathbin{⟕} A') \equiv A \setminus A'$	for $?x \in V$ s.t. $?x \in (cVars(A') \setminus pVars(A))$
FLBII	$\sigma_{bound(?x)}(A \mathbin{⟕} A') \equiv A \bowtie A'$	ditto

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

SA Algebraic Equivalences: Comments [1]

- With respect to Idempotence and Inverse Equivalences, in the case of Join and Left Outer Join, we note that, as shown, they do not hold for the full algebra $\mathcal{A}$, but they do hold for the more restricted fragment $\mathcal{A}\sim$ (see (Schmidt et al., 2010)).
- Likewise, in $\mathcal{A}\sim$, an LJ rule holds that is analogous, for left outer join, to the MJ rule for join shown above.

SA Algebraic Equivalences: Comments [2]

- With respect to Associativity, Commutativity and Distributivity Equivalences, all combination that were not stated above do not hold (again, see (Schmidt et al., 2010)).
- The rules above assume set-based evaluation but all except UI and FDII (highlighted in the tables) also hold for bag-based evaluation.

SA Algebraic Equivalences: Comments [3]

- Note that SPARQL 1.0 lacks a MINUS operator at the surface level.
- Since MINUS essentially implements closed-world negation, it can be simulated using OPTIONAL and FILTER, e.g.:
- (Schmidt et al., 2010) show that, in $\mathcal{A}\sim$, the target expression below can be simplified.

((?p,type,Person) OPT ((?p,type,Person) AND (?p,name,?n))) FILTER (¬bound(?n))

$\sigma_{\neg bound(?n)}([\![(?p,type,Person)]\!]_D \not\bowtie ([\![(?p,type,Person)]\!]_D \bowtie [\![(?p,name,?n)]\!]_D))$

from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

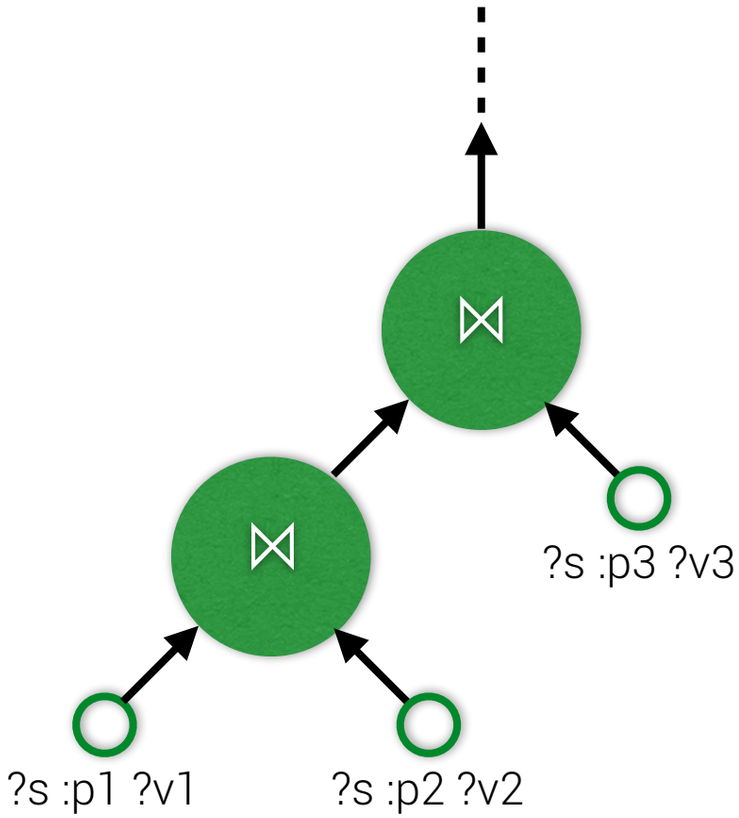
X

MANCHESTER 1824

The University of Manchester

Example Rewritings

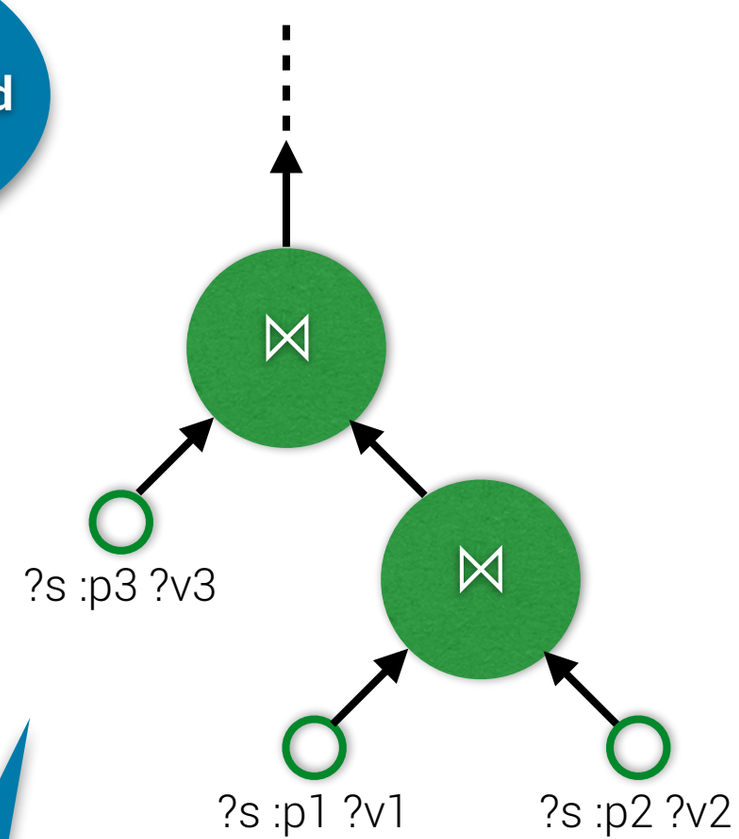
SA Algebraic Equivalences: Examples [1]



Commutativity laws allow us to swap left and right subtrees.

≡

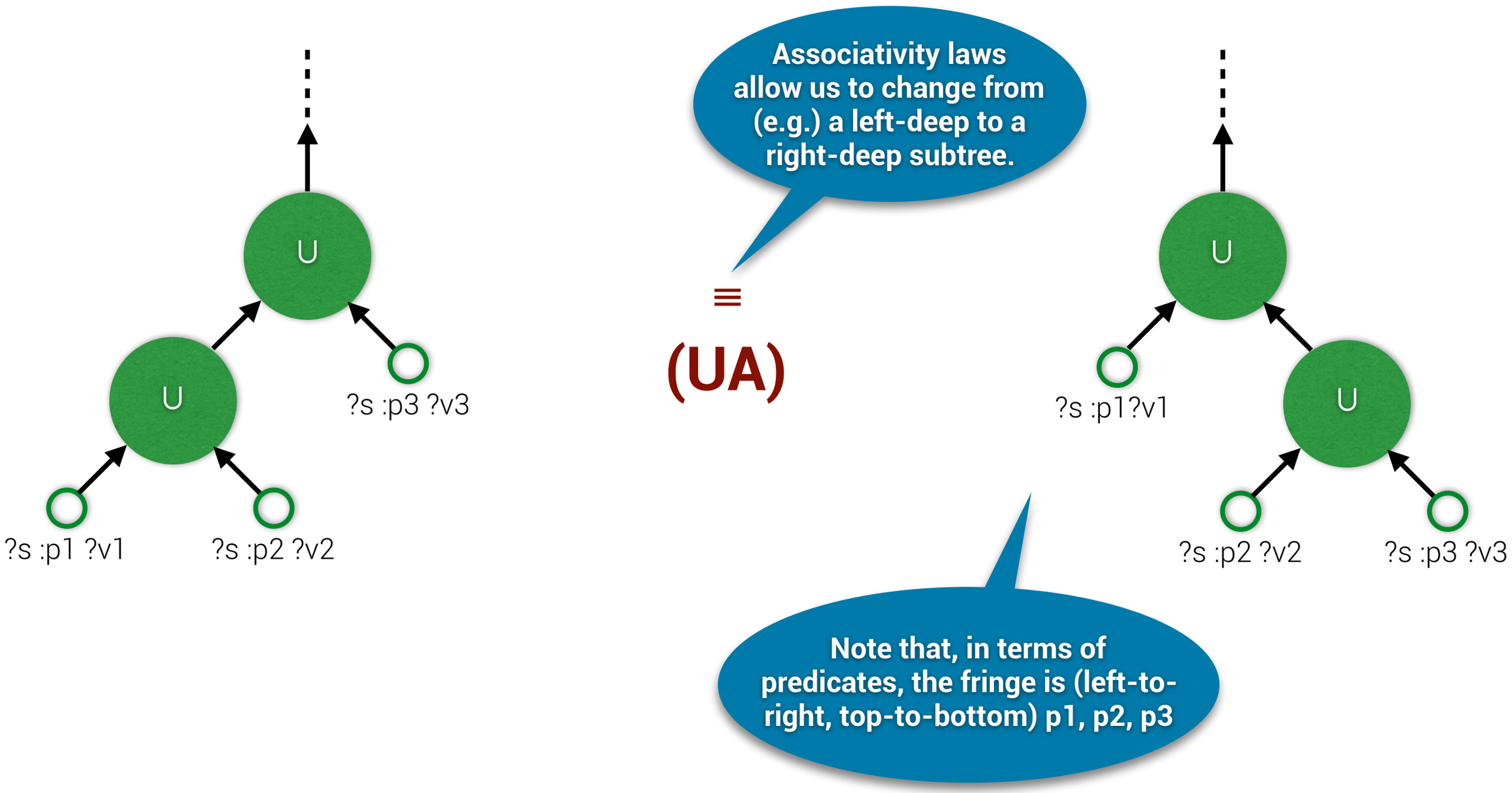
(JC)



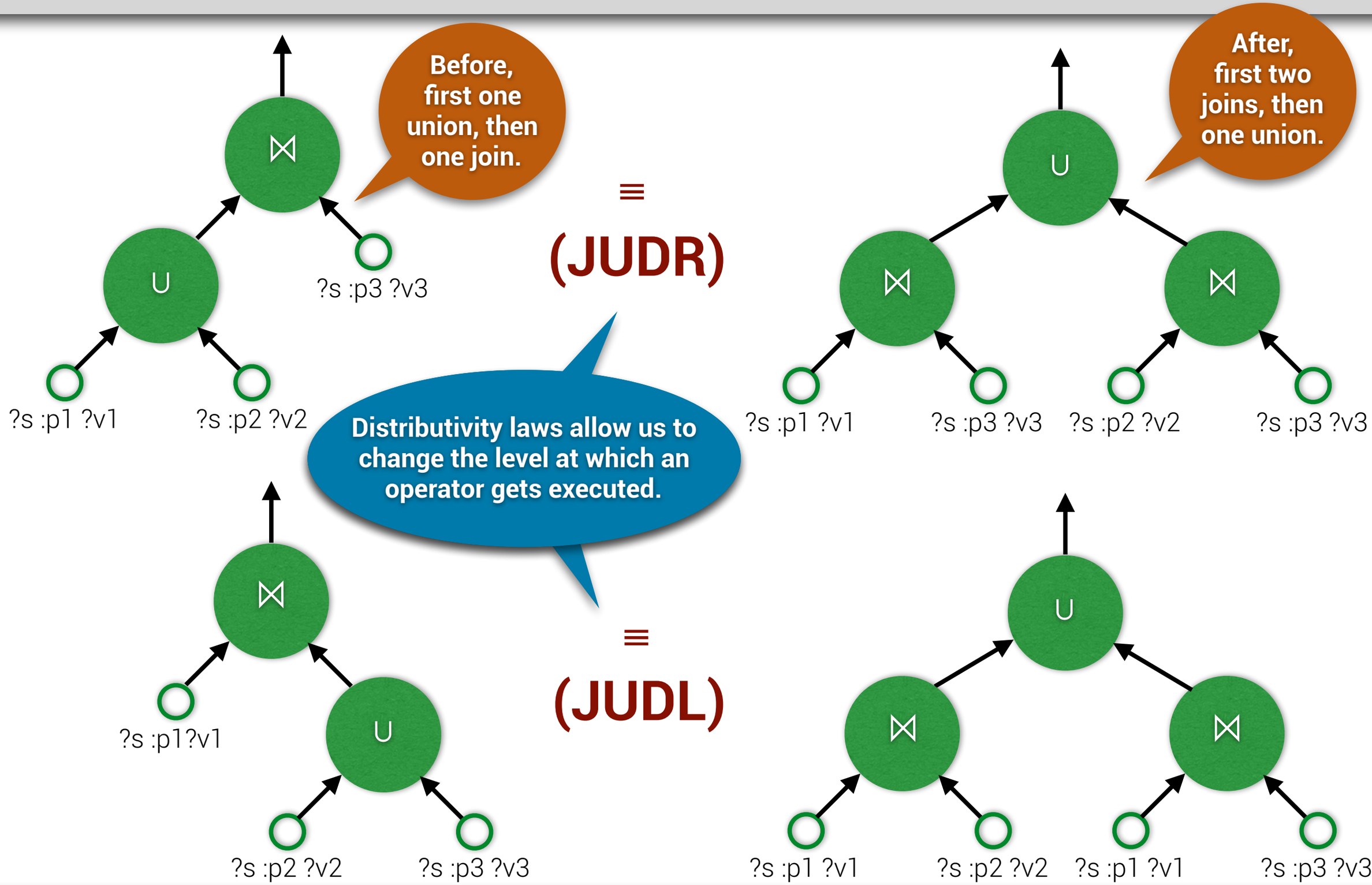
Although this and the next examples show TPs at the fringe, we could have entire subtree as arguments.

Note that, in terms of predicates, the fringe is (left-to-right, top-to-bottom) p3, p1, p2

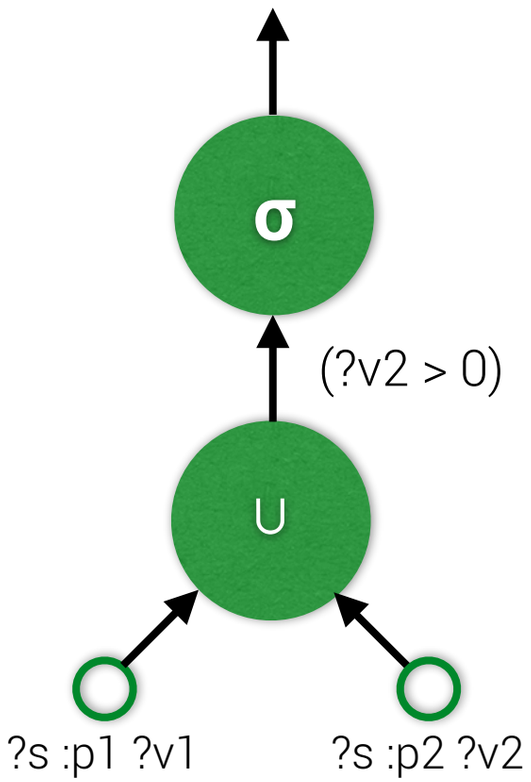
SA Algebraic Equivalences: Examples [2]



SA Algebraic Equivalences: Examples [3]



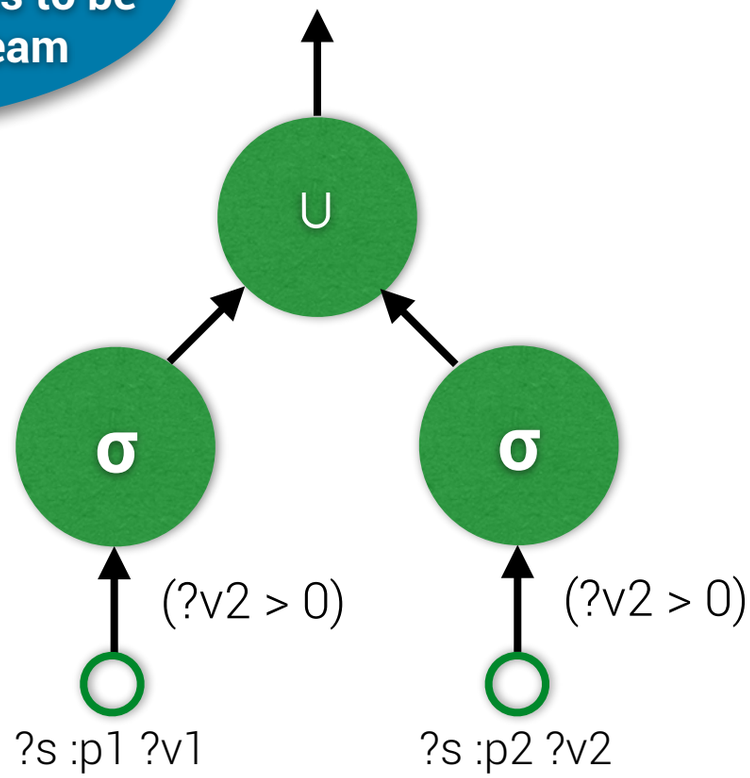
SA Algebraic Equivalences: Examples [4]



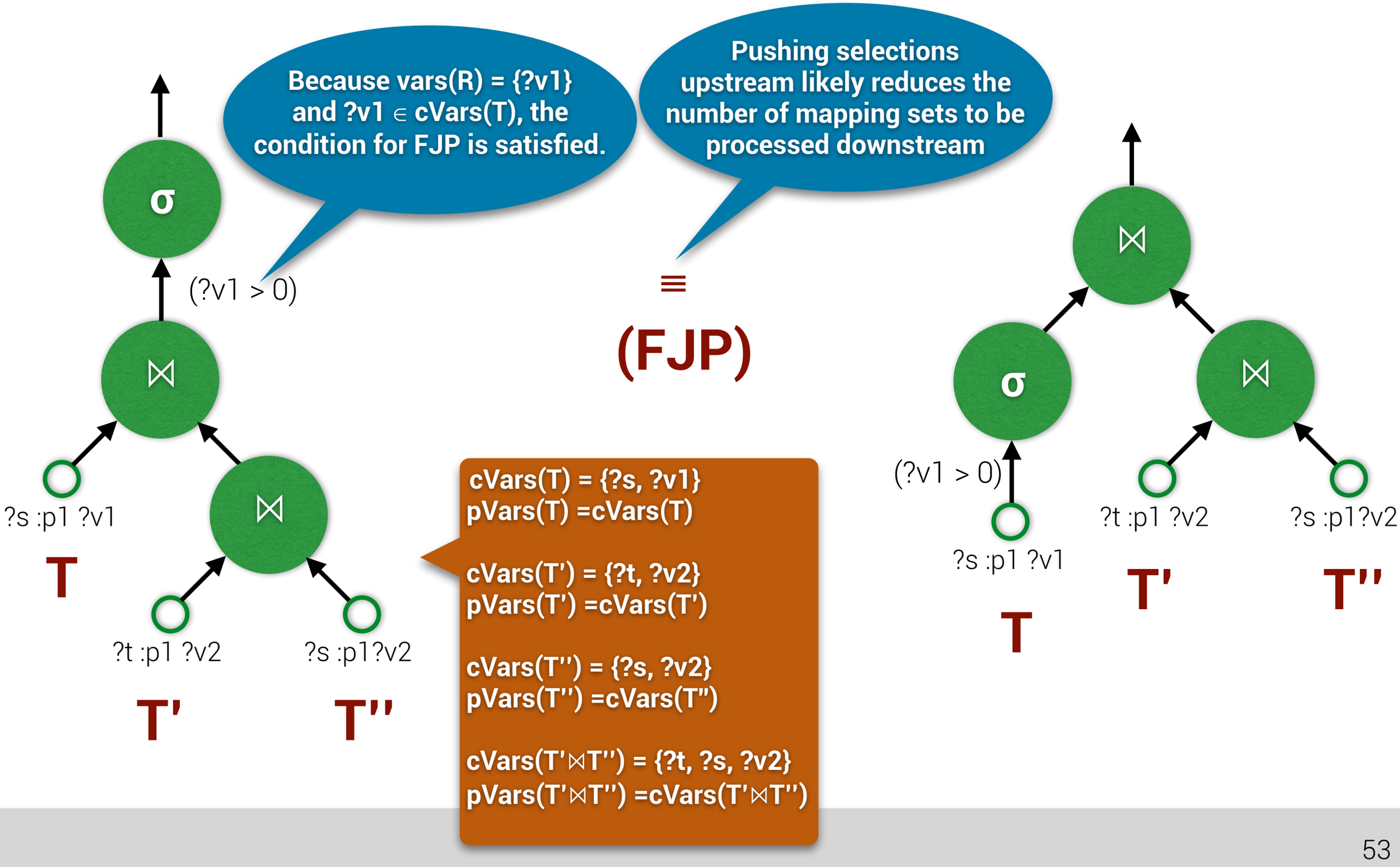
Pushing selections upstream likely reduces the number of mapping sets to be processed downstream

≡

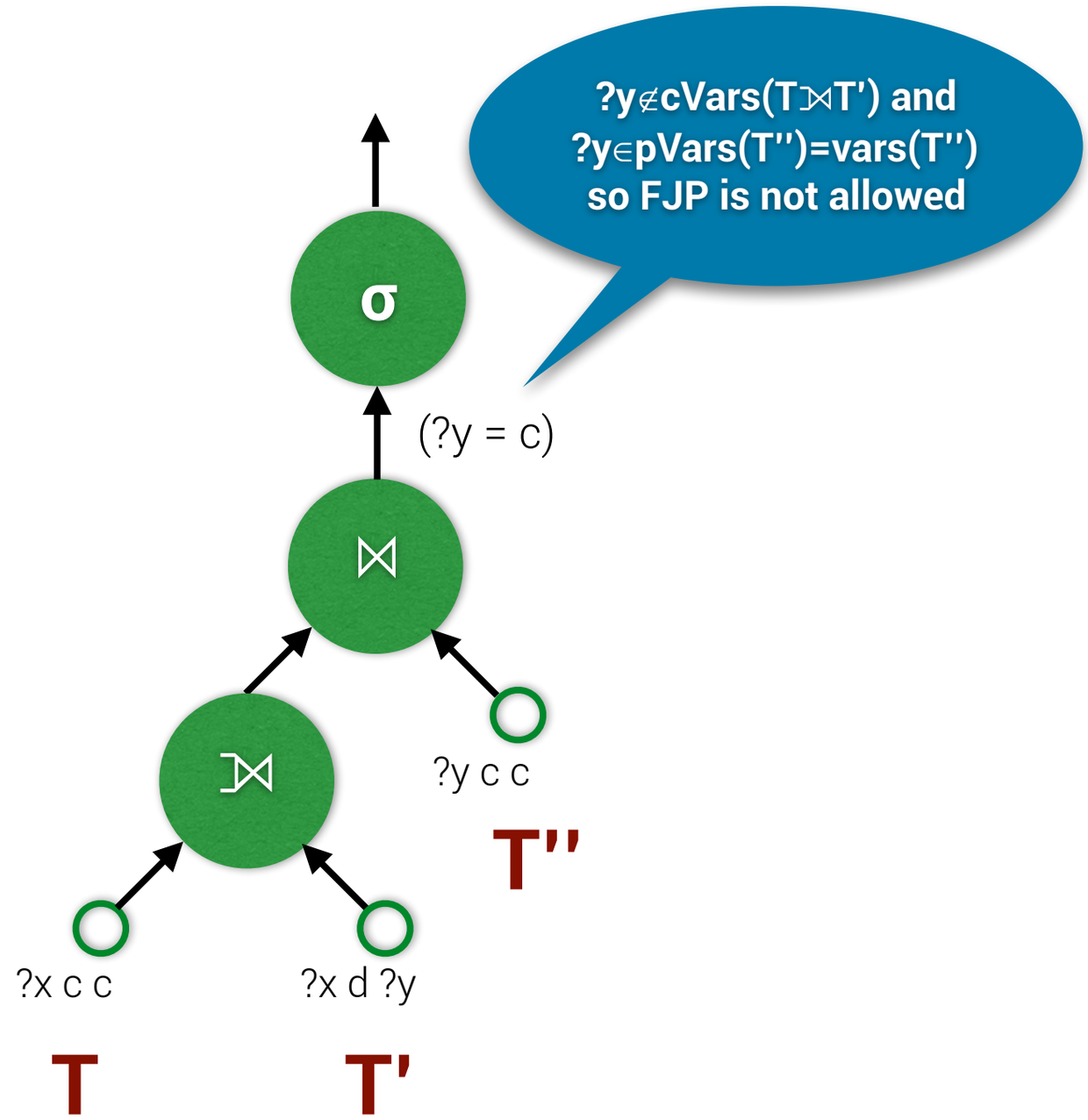
(FUP)



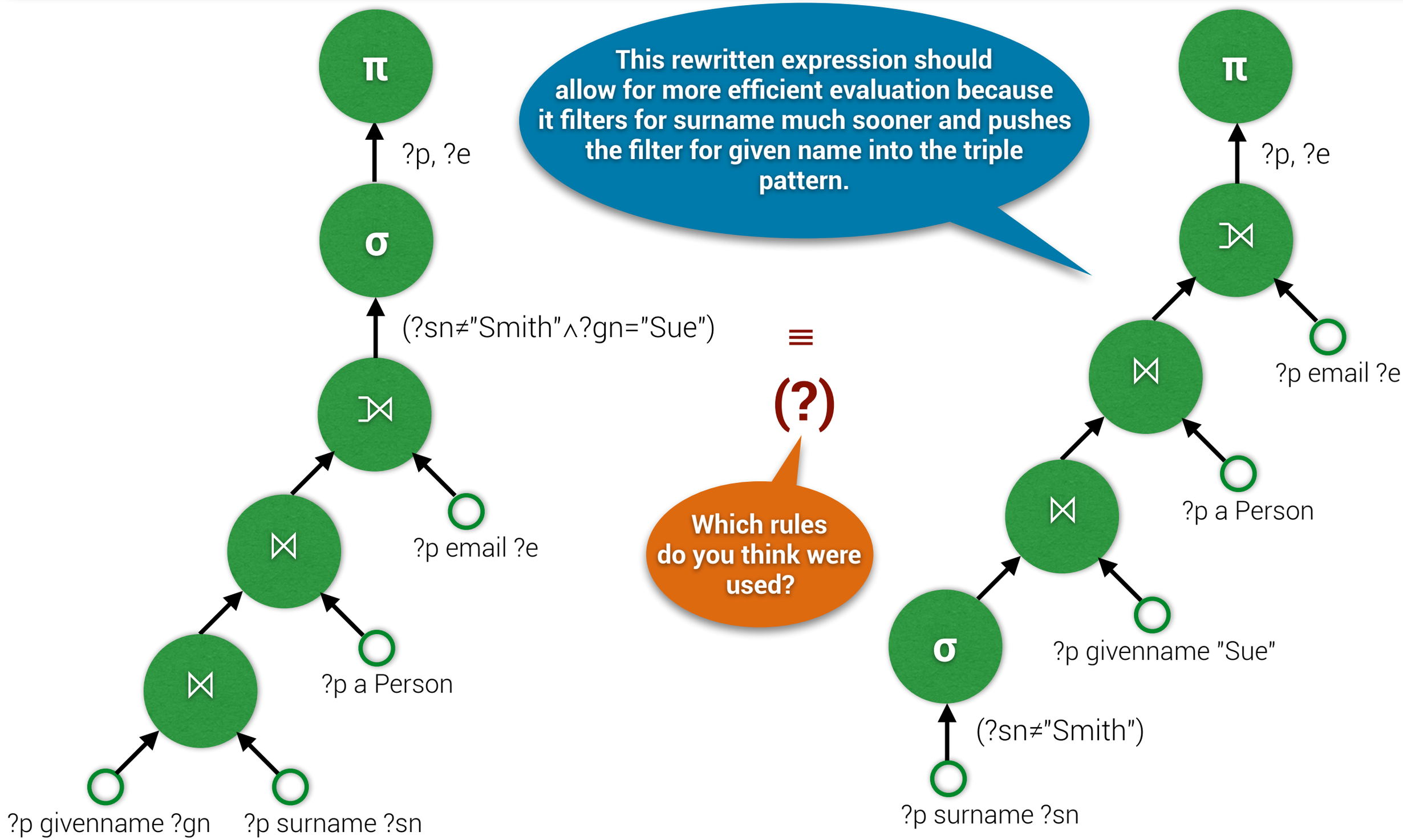
SA Algebraic Equivalences: Examples [5]



SA Algebraic Equivalences: Examples [6]



SA Algebraic Equivalences: Examples [7]



from: Foundations of SPARQL Optimization. M Schimdt et al. (2010)

Equivalence-Based Query Rewriting

- As for RA queries, the equivalences presented for SA queries open the way for developers to a saturation algorithm that by virtue of its design, yields a canonical form that is heuristically more efficient.
- As for RA queries, the goals are similiar, e.g.:
  - to filter mapping sets as stringently as possible as upstream as possible
  - to throw away as many unnecessary bindings as possible as upstream as possible
- As we will see, this is not the route that most current SPARQL engines have followed so far.

Lecture 19

MANCHESTER 1824

The University of Manchester

SPARQL Query Evaluation [1]

MANCHESTER 1824

The University of Manchester

Basic Perspectives

Three Basic Perspectives

- The first basic perspective is referred to as a **relational** perspective.
- It is closer to the database research community.
- It emphasises the fact that an RDF graph can be modelled as relational data.
- From this one would hope that the storage of RDF data and the evaluation of SPARL queries can be a layer over relational DBMS.
- This approach then relies on the DBMS to deliver efficient storage and query processing.

Three Basic Perspectives

- The second basic perspective is referred to as an **entity** perspective.
- It is closer to the information retrieval research community.
- It emphasises the fact that an RDF resource can be modelled a set of predicate-object (i.e., attribute-values) pairs.
- This opens the way for querying to be seen as closest to classical keyword search.

Three Basic Perspectives

- The third basic perspective is referred to as a **graph** perspective.
- It is closer to the semistructured-data research community.
- It emphasises the fact that an RDF document is a graph.
- This opens the way for querying to be seen as closest to navigation processes.

Three Basic Perspectives

- There is also research activity in seeing RDF data storage and SPARQL query processing from a peer-to-peer angle.
- We will focus on the relational perspective in these slides.
- In particular, we will first focus on the most basic strategy for modelling RDF data using relational techniques.
- We will then consider index-centric strategies that depart from the above.

MANCHESTER 1824

The University of Manchester

SPARQL through SQL

SPARQL-to-SQL Approaches to Evaluation [1]

- Most SPARQL engines do not take the algebraic rewriting route yet.
- For practical reasons, the tendency has been to delegate the storage layer to a relational DBMS.
- This implies adopting a strategy for modelling triples with tables.
- Once such a strategy is decided on, it yields a domain-independent relational model of the data, i.e., a database schema consisting of domain-independent tables.
- A SPARQL query is then translated directly into SQL over that schema.

SPARQL-to-SQL Approaches to Evaluation [2]

- In this section, we will briefly consider popular strategies for modelling triples with tables.
- We will then present a SPARQL-to-SQL translation that is agnostic as to the modelling strategy (though, for clarity, we will fix one).
- Finally, relying on empirical performance studies, we will draw some conclusions as to
  - how the modelling strategies compare under benchmark conditions
  - what are the limitations of a SPARQL-to-SQL approach under benchmark conditions
- In the next section, we will move on to consider an alternative approach, sometimes referred to as native, in which the storage layer is not delegated to a relational DBMS

MANCHESTER 1824

The University of Manchester

Modelling Triples with Tables

Strategies for Modelling Triples with Tables [1]

- As is implied by the RDF data model underlying SPARQL and made clear by the algebraic operator trees shown in previous slides, the storage level of a SPARQL engine stores triples
- In other words, the leaves of SA operator trees are matching scans of stored triple sets, i.e., of documents.

- If a SPARQL engine delegates the storage of RDF triples to a relational DBMS store, then, for this to be a domain-independent architecture, two decisions must be made:
  - Which relation(s) will be used to store the triples that triple patterns that will occur in queries will be matched against?
  - Which attributes in such relation(s) will be used to store the triple elements, i.e., subjects, predicates and objects?

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

Strategies for Modelling Triples with Tables [2]

- In the ensuing definitions, the document D to the right will be used in the examples.

D := {(B1, name, paul),
(B2, name, john),
(B3, name, george),
(B4, name, ringo),
(B4, web, www.starr.edu),
(B4, cell, 444-4444),
(B1, phone, 111-1111),
(B2, email, john@john.edu),
(B3, web, www.george.edu),
(B4, email, ringo@ringo.edu),
(B4, phone, 444-4444)
}

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

Strategies for Modelling Triples with Tables [3]

- One (possibly obvious, but certainly difficult) strategy is to attempt to infer from the triples, which entity types the document contains, each with the attributes that describe them.
- This has been referred to as the *n-ary property table* strategy.
- It can be automated by clustering (e.g., see (Christodoulou et al., 2015).
- If the data has clear, regular structure types, this strategy essentially induces the relational model of the data in the document, which is good in principle.
- There are complications in one document models more than on entity type, for example.
- In this strategy, populating the database with the example document D would result in the database instance in the next slide.

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009),
and: An Overview on Linked Data and SPARQL Querying. O Hartig. (2011)

Strategies for Modelling Triples with Tables [4]

It is highly non-trivial to conclude this with confidence.

Note how many NULLs.

Artists					
s	**name**	**web**	**cell**	**phone**	**email**
B1	paul	NULL	NULL	111-1111	NULL
B2	john	NULL	NULL	NULL	john@john.edu
B3	george	www.george.edu	NULL	NULL	NULL
B4	ringo	www.ringo.edu	444-4444	444-4444	ringo@ringo.edu

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009),
and: An Overview on Linked Data and SPARQL Querying. O Hartig. (2011)

Strategies for Modelling Triples with Tables [5]

- The most popular strategy, often referred to (rather ambiguously) as the *triple store* strategy is to design the relational database to contain a single ternary table, with schema **Triples(s, p, o)**.

- In this strategy, populating the database with the example document D would result in the database instance to the right.

Triples		
s	**p**	**o**
B1	name	paul
B2	name	john
B3	name	george
B4	name	ringo
B4	web	www.starr.edu
B4	cell	444-4444
B1	phone	111-1111
B2	email	john@john.edu
B3	web	www.george.edu
B4	email	ringo@ringo.edu
B4	phone	444-4444

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009),
and: An Overview on Linked Data and SPARQL Querying. O Hartig. (2011)

Strategies for Modelling Triples with Tables [6]

- We will see that with one single table, queries will often require multiple self-joins, which are, in principle, often difficult to evaluate efficiently.

- This strategy can be extended to a quaternary table to take names graphs into account.

- Inserts and deletes are not expensive.

Triples		
s	**p**	**o**
B1	name	paul
B2	name	john
B3	name	george
B4	name	ringo
B4	web	www.starr.edu
B4	cell	444-4444
B1	phone	111-1111
B2	email	john@john.edu
B3	web	www.george.edu
B4	email	ringo@ringo.edu
B4	phone	444-4444

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009),
and: An Overview on Linked Data and SPARQL Querying. O Hartig. (2011)

Strategies for Modelling Triples with Tables [7]

- Another complex issue is to decide which indexes to create.
- Note that with indexes, space consumption may become a worry.
- We will consider these issues later.

Triples		
s	**p**	**o**
B1	name	paul
B2	name	john
B3	name	george
B4	name	ringo
B4	web	www.starr.edu
B4	cell	444-4444
B1	phone	111-1111
B2	email	john@john.edu
B3	web	www.george.edu
B4	email	ringo@ringo.edu
B4	phone	444-4444

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009),
and: An Overview on Linked Data and SPARQL Querying. O Hartig. (2011)

Strategies for Modelling Triples with Tables [8]

- One refinement (for all strategies) is to add a dictionary table on terms, called, say, **Terms**, so that the cells in Triples only need store compact system-generated identifiers (IDs).
- The IDs are pointers, i.e., they act as foreign keys in joins onto the actual values.
- This reduces storage but leads to an increase in the need for joins.
- In the limit, the Triples table becomes just an IDTriples table.
- For the example document, the Terms table is shown to the right.

Terms	
id	**value**
1	B1
2	B2
3	B3
4	B4
5	name
6	web
7	cell
8	phone
9	email
10	paul
11	george

Terms (cont.)	
id	**value**
12	ringo
13	john
14	www.starr.edu
15	444-4444
16	111-1111
17	john@john.edu
18	www.george.edu
19	ringo@ringo.edu

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009),
and: An Overview on Linked Data and SPARQL Querying. O Hartig. (2011)

Strategies for Modelling Triples with Tables [9]

- Another strategy, referred to as the *property table* strategy, is to model each property as a table.
- In its simplest form, if there p_1, …, p_n are the distinct properties, in the document, we create n binary tables p_i, $1 \leq i \leq n$, with schema (s,o) for each p_i.
- In this strategy, populating the database with the example document D would result in the database instance to the right.

name	
s	**o**
B1	paul
B2	john
B3	george
B4	ringo

web	
s	**o**
B4	www.starr.edu
B3	www.george.edu

cell	
s	**o**
B4	444-4444

email	
s	**o**
B2	john@john.edu
B4	ringo@ringo.edu

phone	
s	**o**
B1	111-1111
B4	444-4444

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009),
and: An Overview on Linked Data and SPARQL Querying. O Hartig. (2011)

Strategies for Modelling Triples with Tables [10]

- Note that this is tantamount to vertically partitioning the Triples table on the property column.
- No NULLs are need and multivalued properties are supported.
- There is less wasted I/O from reading attributes that cannot contribute to solutions.
- A Terms dictionary can also be used and increases efficiency.

name	
s	**o**
B1	paul
B2	john
B3	george
B4	ringo

web	
s	**o**
B4	www.starr.edu
B3	www.george.edu

cell	
s	**o**
B4	444-4444

email	
s	**o**
B2	john@john.edu
B4	ringo@ringo.edu

phone	
s	**o**
B1	111-1111
B4	444-4444

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009),
and: An Overview on Linked Data and SPARQL Querying. O Hartig. (2011)

Strategies for Modelling Triples with Tables [11]

- However, there remains a problem with matching triples with variables in predicate position.
- So, often, the Triples table is retained for such cases.
- Inserts are more expensive, as is restructuring.
- If many triple patterns have a variable in predicate position, the issues with the triple store strategy dominate.

name	
s	**o**
B1	paul
B2	john
B3	george
B4	ringo

web	
s	**o**
B4	www.starr.edu
B3	www.george.edu

cell	
s	**o**
B4	444-4444

email	
s	**o**
B2	john@john.edu
B4	ringo@ringo.edu

phone	
s	**o**
B1	111-1111
B4	444-4444

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009),
and: An Overview on Linked Data and SPARQL Querying. O Hartig. (2011)

Strategies for Modelling Triples with Tables [12]

- By analogy, another strategy, which we might call the *triple binary tables* strategy is to vertically partition not just on the property but on the subject and the object as well.
- Some SPARQL engines, to reduce the number of joins at the price of making updates more complex and expensive, denormalize the database schema.
- The Triples table again needs to be retained, so the space blow-up becomes really significant.

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009),
and: An Overview on Linked Data and SPARQL Querying. O Hartig. (2011)

Lecture 20

MANCHESTER 1824

The University of Manchester

SPARQL Query Evaluation [2]

MANCHESTER 1824

The University of Manchester

Translating SPARQL into SQL

SPARQL-to-SQL Translation

- In what follows, we will illustrate the SPARQL-to-SQL translation assuming the relational model uses the triple store strategy.
- (Chebotko et al., 2009) show how to effect the translation for other modelling strategies than triple store.
- In these slides, we adapt the simpler, informal, partial description in (Chebotko, 2007) since our purpose is illustrative.
- (Chebotko et al., 2009) should be consulted for a formal, detailed and exhaustive account.

SPARQL-to-SQL Translation [1]

```
WHERE {
       (?a, email, ?e) .
      }
```

```
WHERE {
       (?a, web, ?w) .
      }
```

```
WHERE {
       (?a, email, ?e) .
       (?a, web, ?w) .
      }
```

```
Q1 = SELECT DISTINCT t.s AS a,
                     t.p AS email
                     t.o AS e
       FROM Triple t
      WHERE t.p = 'email'
```

```
Q2 = SELECT DISTINCT t.s AS a,
                     t.p AS web
                     t.o AS w
       FROM Triple t
      WHERE t.p = 'web'
```

```
Q3 = SELECT DISTINCT email, e, web, w,
                COALESCE (t1.a, t2.a) AS a
         FROM (Q1) t1
INNER JOIN (Q2) t2
                ON (    t1.a = t2.a
                     OR  t1.a IS NULL
                     OR  t2.a IS NULL)
```

Renaming using the aliased attribute names in Triple and ?-dropping.

One alias per TP is needed in the FROM clause.

Ground terms lead to conjuncts in the WHERE clause.

AND leads to (inner) joins on shared variables.

Special care with join attributes being NULL, plus coalescing in the SELECT clause.

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

SPARQL-to-SQL Translation [2]

```
WHERE {
        (?a, email, ?e) .
        OPT
        (?a, web, ?w) .
      }
```

```
Q4 =      SELECT DISTINCT email, e, web, w,
                  COALESCE (t1.a, t2.a) AS a
            FROM (Q1) t1
LEFT OUTER JOIN (Q2) t2
                  ON (    t1.a = t2.a
                       OR  t1.a IS NULL
                       OR  t2.a IS NULL)
```

OPT leads to left outer joins on shared variables.

```
WHERE {
        (?a, email, ?e) .
        UNION
        (?a, web, ?w) .
      }
```

```
          SELECT DISTINCT email, e, web, w
            FROM (Q1) t1
LEFT OUTER JOIN (Q2) t2 ON (FALSE)
UNION
          SELECT DISTINCT email, e, web, w
            FROM (Q2) t3
LEFT OUTER JOIN (Q1) t4 ON (FALSE)
```

No need to COALESCE.

UNION takes as arguments two left outer joins on FALSE (so that there is padding with NULLs).

Note how the two left outer joins differ in on the left and right arguments.

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

SPARQL-to-SQL Translation [3]

```
WHERE {
        { (?a, email, ?e) .
          OPT
          (?a, web, ?w) .
        }
        FILTER fn:not(bound(?w))
      }
```

```
SELECT ?a, ?e, ?w
WHERE {
        (?a, email, ?e) .
        (?a, web, ?w) .
      }
```

Note how FILTER causes the GP it applies to become a subquery.

```
SELECT *
  FROM (Q4) t
 WHERE NOT (t.w IS NOT NULL)
```

SPARQL SELECT translates simply to SQL SELECT.

```
SELECT t.a, t.e, t.w
  FROM (Q3) t
```

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

MANCHESTER 1824

The University of Manchester

An Extended Example

An Extended Example Translation [1]

- We now show an example translation of the query in the next slide using the approach in (Chebotko, 2007).
- (Chebotko, 2007) proceeds by example and breaks the process down to two phases:
  - Translation of BGPs into SQL, referred to as BGPtoSQL
  - Translation of other constructs into SQL, referred to as SPARQLtoSQL, of which BGPtoSQL can be seen as a subroutine.
- Therefore, the account in the next few slides does not rigorously match the one given in the above slides with title *SPARQL-to-SQL Translation*, but is compatible with the latter.

An Extended Example Translation [2]

Return the URL of Artem Chebotko's homepage, the URL of his blog if available and if his publications point to it, and the topics in the latter if available, and his blogs in example.org if available.

```
SELECT ?url ?log ?topic
WHERE  { ?someone foaf:name "Artem Chebotko" .
          ?someone foaf:homepage ?url .
          OPTIONAL { ?someone foaf:weblog ?log .
                     OPTIONAL { ?url foaf:topic ?topic .
                                }
                     }
          OPTIONAL { ?someone foaf:publications ?log .
                     }
}
```

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

MANCHESTER 1824

The University of Manchester

Basic Graph Patterns into SQL: An Example

BGPtoSQL: Step 1

This allows us to distinguish between TPs in the SQL query.

- For each TP in a GP, assign it a unique table alias.
- Construct the FROM clause to with as many mentions of the Triples table as the aliases assigned above.
- Note that in the example in the next slide, the top, green-coloured TP is assigned the alias **t1** and the bottom, green-coloured TP is assigned the alias **t2**.
- Note also that we will not show the translation the other TPs (in the next slide) that are the right-hand side arguments of OPTIONAL clauses, but the approach is the same.

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

BGPtoSQL: Step 1 Example

```
SELECT ?url ?log ?topic
WHERE  { ?someone foaf:name "Artem Chebotko" .
         ?someone foaf:homepage ?url .
         OPTIONAL { ?someone foaf:weblog ?log .
                    OPTIONAL { ?url foaf:topic ?topic .
                               }
                  }
         OPTIONAL { ?someone foaf:publications ?log .
                    }
       }
```

The other TPs in blue, red and orange will give rise to separate subqueries too.

FROM Triples t1, Triples t2

BGPtoSQL: Step 2

This allows us to project the right variables in the SQL query.

- For each TP in the GP, collect the set of variables that occur in it with an index to which alias-qualified attribute represents it in the Triples table, then union this set with the set of variables in previous TPs.
- Construct the SELECT clause as follows:
  - for each pair of the form **(?x, A.i)** add a projection of the form **A.i AS x** where **A** is the alias of the table corresponding to the TP that **?x** originates from
- In the next slide, the top TP yields the set S = {(**?someone, t1.s**)} and the bottom the set S′ = {(**?url, t2.o**)}
- Therefore the result of the iteration is S ∪ S′ = {(**?someone, t1.s**), (**?url, t2.o**)}.

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

BGPtoSQL: Step 2 Example

```
SELECT ?url ?log ?topic
WHERE  { ?someone foaf:name "Artem Chebotko" .
         ?someone foaf:homepage ?url .
         OPTIONAL { ?someone foaf:weblog ?log .
                    OPTIONAL { ?url foaf:topic ?topic .
                               }
                  }
         OPTIONAL { ?someone foaf:publications ?log .
                    }
       }
```

The same process for creating SELECT clauses applies separately to the other TPs in blue, red and orange.

```
SELECT t1.subject AS someone, t2.object AS url
FROM Triples t1, Triples t2
```

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

BGPtoSQL: Step 3

This ensures that only the triples with the right grounded terms are selected.

- For each TP in the GP, collect the set of ground values that occur in it with an index to which alias-qualified attribute represents it in the Triples table, then union this set with the set of variables in previous TPs.
- Construct the WHERE clause as follows:
  - for each pair of the form **(gt, A.i)** add a conjunct of the form **A.i = gt** where **A** is the alias of the table corresponding to the TP that **gt** originates from
- In the next slide, the top TP yields the set S = {(**foaf:name, t1.p**), (**'Artem Chebotko', t1.o**)} and the bottom the set S′ = {(**foaf:homepage, t2.p**)}
- Therefore the result of the iteration is S ∪ S′ = {(**foaf:name, t1.p**), (**'Artem Chebotko', t1.o**),(**foaf:homepage, t2.p**)}.

BGPtoSQL: Step 3 Example

```
SELECT ?url ?log ?topic
WHERE  { ?someone foaf:name "Artem Chebotko" .
         ?someone foaf:homepage ?url .
         OPTIONAL { ?someone foaf:weblog ?log .
                    OPTIONAL { ?url foaf:topic ?topic .
                               }
                  }
         OPTIONAL { ?someone foaf:publications ?log .
                    }
       }
```

Again, the same process applies separately to the other TPs in blue, red and orange.

```
SELECT t1.subject AS someone, t2.object AS url
FROM Triples t1, Triples t2
WHERE t1.predicate = 'foaf:name'
    AND t1.object = 'Artem Chebotko'
    AND t2.predicate = 'foaf:homepage'
```

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

BGPtoSQL: Step 4

This ensures that multiple occurrences of the same variable lead to equijoins in the SQL query.

- Create an associative array **I**, as follows.
  - For each variable **?x** that occurs in the GP, let **A.i** be the alias-qualified attribute that represents it in the Triples table, then
    - if **?x** is not an existing key in **I**, then assign **I[?x]={A.i}** where **I[?x] = V** denotes that the value of **?x** in **I** is **V**
    - if **?x** is an existing key in **I** and the value of **?x** in **I** is **O**, then assign **I[?x] = O**∪**{A.i}**
- For each entry **I[?x]=V** such that |V|>1, i.e., ?x has multiple occurrences, then form all possible pairs with distinct first and second elements (ignoring order), and for each such pair, of the form (**A.i, A'.i'**), add to the WHERE clause a conjunct of the form **A.i=A'.i'**
- In the example in the next slide, this results in the following array:

variable	occurrence
?someone	t1.s, t2.s
?url	t2.o

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

BGPtoSQL: Step 4 Example

```
SELECT ?url ?log ?topic
WHERE  { ?someone foaf:name "Artem Chebotko" .
         ?someone foaf:homepage ?url .
         OPTIONAL { ?someone foaf:weblog ?log .
                    OPTIONAL { ?url foaf:topic ?topic .
                               }
                  }
       OPTIONAL { ?someone foaf:publications ?log .
                  }
       }
```

This step completes the translation of the AND of the two topmost TPs.

One needs now to apply Steps 1-4 separately to the other TPs in blue, red and orange.

Once that is done we'll have for SQL queries, call them Q1, Q2, Q3 and Q4.

```
-- Q1 =
SELECT t1.subject AS someone, t2.object AS url
FROM Triples t1, Triples t2
WHERE t1.predicate = 'foaf:name'
    AND t1.object = 'Artem Chebotko'
    AND t2.predicate = 'foaf:homepage'
    AND t1.subject = t2.subject
```

Here, (Chebotko, 2007) conjoins an equality predicate to express the join rather than use an explicit INNER JOIN as in (Chebotko et al., 2009).

(Chebotko, 2007) is also more relaxed than (Chebotko et al., 2009) in the treatment of NULLs, both in the WHERE clause and, due to the absent COALESCE, in the SELECT clause.

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

The University of Manchester

SPARQL into SQL:
An Example

SPARQLtoSQL: Step 1

- Step 1:
  - Translate all BGPs to SQL with BGPtoSQL.
  - This yields for subqueries **Q1**, **Q2**, **Q3** and **Q4**

```
SELECT ?url ?log ?topic
WHERE  { ?someone foaf:name "Artem Chebotko" .
         ?someone foaf:homepage ?url .
         OPTIONAL { ?someone foaf:weblog ?log .
                    OPTIONAL { ?url foaf:topic ?topic . } }
         OPTIONAL { ?someone foaf:publications ?log . } }
```

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

SPARQLtoSQL: Step 2

- Combine the inner queries (q1, q2, q3, q4) using INNER JOIN for AND, LEFT OUTER JOIN for OPTIONAL and UNION for UNION.
- In the running example, the only one of the above to occur is OPTIONAL, so we combine with LEFT OUTER JOIN.
- There are three occurrences of OPTIONAL, so Step 2 breaks down into three: Steps 2.1-2.3.

SPARQLtoSQL: Step 2.1 Example

```
SELECT ?url ?log ?topic
WHERE  { ?someone foaf:name "Artem Chebotko" .
         ?someone foaf:homepage ?url .
         OPTIONAL { ?someone foaf:weblog ?log .
                    OPTIONAL { ?url foaf:topic ?topic . } }
        OPTIONAL { ?someone foaf:publications ?log . } }
```

Q5 =

```
SELECT r1.someone AS someone,
       r1.url AS url,
       r2.log AS log
 FROM (Q1) r1 LEFT OUTER JOIN
      (Q2) r2 ON (r1.someone = r2.someone)
```

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

SPARQLtoSQL: Step 2.2 Example

```
SELECT ?url ?log ?topic
WHERE  { ?someone foaf:name "Artem Chebotko" .
         ?someone foaf:homepage ?url .
         OPTIONAL { ?someone foaf:weblog ?log .
                    OPTIONAL { ?url foaf:topic ?topic . } }
        OPTIONAL { ?someone foaf:publications ?log . } }
```

Q6 =

```
SELECT r5.someone AS someone,
       r5.url AS url,
       r5.log AS log,
       r3.topic AS topic
 FROM (Q5) r5 LEFT OUTER JOIN
      (Q3) r3 ON (r5.url = r3.url
            AND r5.log IS NOT NULL)
```

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

SPARQLtoSQL: Step 2.3 Example

```
SELECT ?url ?log ?topic
WHERE  { ?someone foaf:name "Artem Chebotko" .
         ?someone foaf:homepage ?url .
         OPTIONAL { ?someone foaf:weblog ?log .
                    OPTIONAL { ?url foaf:topic ?topic . } }
         OPTIONAL { ?someone foaf:publications ?log . } }
```

Q7 =

```
SELECT r6.someone AS someone,
       r6.url AS url,
       COALESCE(r6.log, r4.log) AS log,
       r6.topic AS topic
 FROM (Q6) r6 LEFT OUTER JOIN
      (Q4) r4 ON (r6.someone = r4.someone
                  AND (r6.log = r4.log OR r6.log IS NULL))
```

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

SPARQLtoSQL: Step 3

- This step constructs the outermost SELECT clause in the obvious way.

SPARQLtoSQL: Step 2.3 Example

```
SELECT ?url ?log ?topic
WHERE  { ?someone foaf:name "Artem Chebotko" .
         ?someone foaf:homepage ?url .
         OPTIONAL { ?someone foaf:weblog ?log .
                    OPTIONAL { ?url foaf:topic ?topic . } }
        OPTIONAL { ?someone foaf:publications ?log . } }
```

Q8 =

```
SELECT r7.url AS url, r7.log AS log, r7.topic AS topic
  FROM (Q7) r7
```

from: Semantics Preserving SPARQL-to-SQL Translation. A. Chebotko et al. (2009)

SPARQLtoSQL: (Almost) Final Result

Q8 =

SELECT r7.url AS url, r7.log AS log, r7.topic AS topic

FROM (**SELECT r6.someone AS someone, r6.url AS url,**

COALESCE(r6.log, r4.log) AS log, r6.topic AS topic

FROM (SELECT r5.someone AS someone, r5.url AS url, r5.log AS log, r3.topic AS topic

FROM (*SELECT r1.someone AS someone, r1.url AS url, r2.log AS log*

*FROM (***Q1***) r1 LEFT OUTER JOIN*

*(***Q2***) r2 ON (r1.someone = r2.someone)*

) r5 LEFT OUTER JOIN

(**Q3**) r3 ON (r5.url = r3.url

AND r5.log IS NOT NULL)

) r6 LEFT OUTER JOIN

(Q4) r4 ON (r6.someone = r4.someone

AND (r6.log = r4.log OR r6.log IS NULL))

) r7

Q7 is in bold.

Q6 is underlined.

Q8 is Q, the outermost query.

Q5 is in italics.

This is the result of Steps 1-3 with subqueries unfolded except (to save space) for Q1, Q2, Q3 and Q4.

The University of Manchester

SPARQL through SQL:
Empirical Evidence of Efficiency

Some Empirical Evidence on the Efficiency of the SPARQL-to-SQL Translation Algorithm

- The translation algorithm itself is efficient (in this case, runs in miliseconds or less) even for large queries (e.g., 50 OPTIONAL clauses with one TP each).
- The resulting queries, though, tend to be slow (in the order of seconds).
- The belief (confirmed by many empirical studies) is that the triple store approach is a performance drain even for top-breed relational DBMS.
- The next few slides briefly review some of this evidence from on empirical study.

Some Empirical Evidence on the Efficiency of Storage Strategies [1]

- (Schmidt et al., 2008) describe a benchmark whose purpose is to empirically investigate the efficiency of storage strategies for SPARQL query processing.
- They study the triple-store (call it TS) and the vertically-partitioned property-table (call it PT) strategies onto a relational backend.
- They also study Sesame, as an exemplar of a native-RDF backend, i.e., without SPARQL-to-SQL translation (call it NB) and a purely relational backend, i.e., a remodelling the RDF data as relational from scratch (call it RB).
- They use a dictionary table on terms.

Some Empirical Evidence on the Efficiency of Storage Strategies [2]

- For TS and PT, the authors translated SPARQL into SQL by hand, but they claim to have followed closely such translation procedures as the one we have described.
- One conclusion is that, in both TS and PT, sorting by *predicate* > *subject* > *object* allows efficient merge joins to be used.
- However, it is not guaranteed that a relational optimizer will always spot this opportunity and make the most of it.

Some Empirical Evidence on the Efficiency of Storage Strategies [3]

- Another conclusion is that neither TS nor PT is sufficiently efficient for the more challenging queries in the benchmark.
- This indicates that, perhaps as expected, the SPARQL-to-SQL translation does cause problems for relational optimizers, which are always unpredictable in the presence of many joins, particularly self-joins.

Some Empirical Evidence on the Efficiency of Storage Strategies [4]

- Another conclusion is that both TS and PT, nonetheless, outperform NB.

- This indicates that, perhaps as expected, more research is needed on storage backends that are designed to be specifically efficient for SPARQL query evaluation.

- We will come back to this observation in later slides.

Some Empirical Evidence on the Efficiency of Storage Strategies [5]

- A final conclusion is that RB, i.e., the remodelling of RDF data from scratch using a relational model, outperforms all other strategies by at least one order of magnitude.
- This indicates that, perhaps as expected, the flexibility in representation that RDF affords (i.e., the lack of regular schemas, the tolerance to missing data, etc.) comes with a performance price.

The University of Manchester

Index-Centric SPARQL Evaluation

RDF Storage in Secondary Memory

- While the SPARQL-through-SQL strategy is popular and efficient, many important and influential systems use native backends to store RDF data for SPARQL querying.
- In the next few slides, we briefly look into how using native RDF backends leads to an index-centric strategy.
- We follow closely the analytical work in (Wolff et al., 2015).

Alternatives in Native RDF Backends

- There are two basic storage alternatives for a native RDF backend:
  - a flat file of triples, one per line, sorted on some subset of positions (e.g., by subject values)
  - an index mapping a search key to a set of matching triples (e.g., a B+tree, a hash table).

from: An Extensible Framework for Query Optimization on TripleT-based RDF Stores. B. Wolff et al. (2015)

Indexes for Triples

- There are two complementary kinds of indexes that one could use:
  - value-based indexes, where given one or more values in the triple, we efficiently access the set of remaining values that share the given values
  - structure-based indexes, where we also group triples if they share some graph-theoretic property (e.g., they have topologically-similar neighbourhoods).
- Value-based indexes predominate, but we saw, in the cases of XML/XQuery, that structure-based indexes can be very useful.

from: An Extensible Framework for Query Optimization on TripleT-based RDF Stores. B. Wolff et al. (2015)

Value-Based Indexes for Triples

- Three influential approaches to value-based indexing for triples in a graph G, are:
  - the MAP approach
  - the HexTree approach
  - the TripleT approach
- (Wolff et al., 2015) provide references to the primary sources and related work.

The MAP Approach

- In the MAP approach, all permutations of S, P and O are indexed, i.e., taking # as a tuple-forming operator, in MAP we have:
  - SPO = {s#p#o | (s,p,o) ∈ G}
  - SOP = {s#o#p | (s,p,o) ∈ G}
  - PSO = {p#s#o | (s,p,o) ∈ G}
  - POS = {p#o#s | (s,p,o) ∈ G}
  - OSP = {o#s#p | (s,p,o) ∈ G}
  - OPS = {o#p#s | (s,p,o) ∈ G}

But every triple is stored six times.

This means that every triple in the index, i.e, there is no need for other files.

Used, e.g., by Virtuoso and by Sesame.

This is the most popular approach

from: An Extensible Framework for Query Optimization on TripleT-based RDF Stores. B. Wolff et al. (2015)

The HexTree Approach

- In the HexTree approach, all pairs of S, P and O are indexed, so we have:
  - SP = {s#p | (s,p,o) ∈ G}
  - SO = {s#o | (s,p,o) ∈ G}
  - PS = {p#s | (s,p,o) ∈ G}
  - PO = {p#o | (s,p,o) ∈ G}
  - OS = {o#s | (s,p,o) ∈ G}
  - OP = {o#p | (s,p,o) ∈ G}

Again, potential for six-time redundancy.

Now, given a pair of components, we fetch the the set of matching third components.

SP→ {o ∈ O(G) | (s,p,o) ∈ G}
SP→ {o ∈ O(G) | (s,p,o) ∈ G}

Here, partners can share payload.

from: An Extensible Framework for Query Optimization on TripleT-based RDF Stores. B. Wolff et al. (2015)

Strenghts and Weaknesses

- The MAP and HexTree approach allow the use of fast merge joins for all SPARQL queries whose WHERE clause requires a join.
- There are weaknesses, though:
  - High levels of redundacy leading to high consumption of persistent storage.
  - Joins require access to many data structures.
  - In general, weak data locality, i.e., when data is sought in persistent storage it is often scattered, which with single-head movable-arm devices leads to expensive seek and latency costs.

from: An Extensible Framework for Query Optimization on TripleT-based RDF Stores. B. Wolff et al. (2015)

The TripleT Approach

- For a graph G, define:
  - $S(G) = \{s \mid (s,p,o) \in G\}$
  - $P(G) = \{p \mid (s,p,o) \in G\}$
  - $O(G) = \{o \mid (s,p,o) \in G\}$
  - $A(G) = S(G) \cup P(G) \cup O(G)$
- In the TripleT (for Three-way Triple Tree), one single data structure indexes A(G).
- Given a subject, or predicate, or object, we fetch the set of pairs that have the given value.
- So, given a subject s, the payload is a set of (o,p) pairs; given a predicate p, the payload is a set of (s,o) pairs, and given an object o, the payloas is a set of (s,p) pairs.

from: An Extensible Framework for Query Optimization on TripleT-based RDF Stores. B. Wolff et al. (2015)

Benefits of the TripleT Approach

- The TripleT approach retains the stregnths of the MAP and HexTree approaches.
- However, it requires less storage space, and the reduced key size leads to shallower trees.
- Joins do not need to access more than one data structure.
- There is also, in general, better data locality.

from: An Extensible Framework for Query Optimization on TripleT-based RDF Stores. B. Wolff et al. (2015)

Using TripleT for Optimization

- (Wolff et al., 2015) investigate the contribution of heuristics, as well as metadata (in te form of dataset statistical information), to the generation of efficient plans over a native RDF store.
- They propose a generic, rule-based algorithm for generating physical plans from BGP queries.
- Their experiments have shown that:
  - a small number of relatively simple heuristics can consistently produce efficient evaluation plans for a wide variety of queries and datasets
  - while statistics do not add sufficient value to offset the costs involved in obtaining and maintaining them over massive graphs.

Take-Up

- The TripleT approach is used in 4store.
- It also occurs in columnar stores such as MonetDB and Vertica.
- Commercial DBMS vendors use it for their RDF solutions, along with MAP and HexTree.

from: An Extensible Framework for Query Optimization on TripleT-based RDF Stores. B. Wolff et al. (2015)

MANCHESTER 1824
The University of Manchester

Conclusions

Topics This Week

- We have taken a quick, example-driven tour of SPARQL, to remind ourselves of the basic characteristics of the language.
- We have introduced and briefly explored an algebraic view of SPARQL, and some equivalence laws that arise, on the basis of which one can conceive of a rewriting approach to logical optimization for the language.
- We have studied storage, optimization and evaluation of SPARQL, specially in the context of the SPARQL-through-SQL strategy.

Learning Objectives This Week

- We aimed to acquire a deeper understanding of some of the challenges arising in querying over large RDF collections using SPARQL.

Topics Next Week

- We will look into querying (live, distributed) data on the web, as opposed to querying (locally-materialized) data from/for the web.
- We will also briefly look into the NoSQL approach to data management.
- Finally, we will delve a little on map-reduce engines as a platform for query processing.

Learning Objectives Next Week

- We will aim to acquire a deeper understanding of distributed querying on the web.
- We will aim to acquire a deeper understanding of distributed stores under the NoSQL paradigm.
- We will aim to gain an insight into how map-reduce engines offer an opportunity for massively-parallelization of query execution.

Bibliography

References

1. Marcelo Arenas, Claudio Gutierrez ,Jorge Pérez. **SPARQL Formalization**. ESWC 2007 Tutorial. https://ai.wu.ac.at/~polleres/sparqltutorial/
2. Artem Chebotko **Semantic Web Query Processing with Relational Databases**. (2007) http://www.cs.wayne.edu/graduateseminars/gradsem_w07/slides/artem_talk.ppt
3. Artem Chebotko, Shiyong Lu, Farshad Fotouhi. **Semantics preserving SPARQL-to-SQL translation**. Data Knowl. Eng. 68(10): 973-1000 (2009) http://dx.doi.org/10.1016/j.datak.2009.04.001
4. Klitos Christodoulou, Norman W. Paton, Alvaro A. A. Fernandes. **Structure Inference for Linked Data Sources Using Clustering**. T. Large-Scale Data- and Knowledge-Centered Systems 19: 1-25 (2015) http://dx.doi.org/10.1007/978-3-662-46562-2_1
5. Lee Feigenbaum, Eric Prud'hommeaux. **SPARQL By Example: A Tutorial**.(2013) http://www.cambridgesemantics.com/semantic-university/sparql-by-example
6. Bart G. J. Wolff, George H. L. Fletcher, James J. Lu. **An Extensible Framework for Query Optimization on TripleT-based RDF Stores**. EDBT/ICDT Workshops 2015: 190-196 http://ceur-ws.org/Vol-1330/paper-31.pdf (slides) http://www.win.tue.nl/~gfletche/papers-final/lwdm2015.pdf
7. Olaf Hartig. **An Overview on Linked Data and SPARQL Querying**. ISSLOD 2011 Tutorial. http://www.slideshare.net/olafhartig/an-overview-on-linked-data-management-and-sparql-querying-isslod2011
8. Olaf Hartig. **An Overview on Execution Strategies for Linked Data Queries**. Datenbank-Spektrum 13(2): 89-99 (2013) http://dx.doi.org/10.1007/s13222-013-0122-1
9. Yongming Luo, François Picalausa, George H.L. Fletcher, Jan Hidders, Stijn Vansummeren. **Storing and Indexing Massive RDF Datasets**. (2012) Ch. 2 in *Semantic Search over the Web* (De Virgilio et al. (eds.)) Springer. http://dx.doi.org/10.1007/978-3-642-25008-8_2

References

11. Jorge Pérez, Marcelo Arenas, Claudio Gutierrez. **Semantics and Complexity of SPARQL**. ACM TODS 34(3), August 2009. http://doi.acm.org/10.1145/1567274.1567278

12. Andy Seaborne. **SPARQL**. ESWC 2007 Tutorial. https://ai.wu.ac.at/~polleres/sparqltutorial/ Andy Seaborne. **SPARQL Algebra**. ESWC 2007 Tutorial. https://ai.wu.ac.at/~polleres/sparqltutorial/

13. Michael Schmidt, Thomas Hornung, Norbert Küchlin, Georg Lausen, Christoph Pinkel. **An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario**. International Semantic Web Conference 2008: 82-97 http://dx.doi.org/10.1007/978-3-540-88564-1_6

14. Michael Schmidt, Michael Meier, Georg Lausen. **Foundations of SPARQL Query Optimization**. ICDT 2010: 4-33 http://doi.acm.org/10.1145/1804669.1804675

W3C Documents

I. **SPARQL 1.1 Query Language**. Steve Harris, Andy Seaborne. http://www.w3.org/TR/sparql11-query/

II. **SPARQL Query Language for RDF [*1.0*]**. Eric Prud'hommeaux, Andy Seaborne. http://www.w3.org/TR/rdf-sparql-query/

III. **RDF 1.1 Primer**. Guus Schreiber, Yves Raimond. http://www.w3.org/TR/rdf11-primer/

IV. **SPARQL Implementations**. W3C. http://www.w3.org/wiki/SparqlImplementations

V. **SPARQL Endpoints**. W3C. http://www.w3.org/wiki/SparqlEndpoints