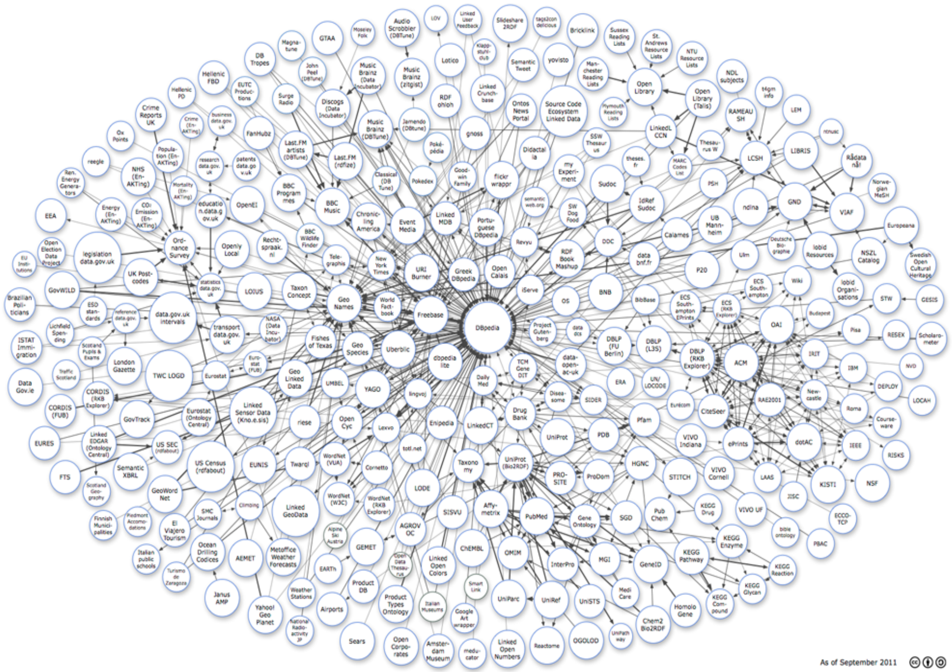
Query Optimization with RDF using SPARQL

By Lori London, Raul Cardenas and Rezwana Rimpi

Introduction

Semantic Web: push towards the creation of a web of data. It sets the standards for the web.



“Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch. http://lod-cloud.net/”

RDF (Resource Description Framework)

RDF databases contain data in a form known as **triples**, formatted as:

(Subject, Predicate, Object) or (S, P, O)

- Subject - a URI that is used as a Resource
- Object - a URI or literal that can be any primitive data type (i,e. float, String, integer)
- Predicate - a URI used as a relationship between the Subject and the Object in the triple

Prefix Lori: www.example.com/Lori

Ex. (Lori, property:age, 23)

RDF (Resource Description Framework) cont.

- In SQL, we are used to having tables and attributes define what data is in that particular table
- In RDF, we have the predicate column define what the triple is representing

SID	Title	Singer
id1	“rain”	pid1

Song

PIDI	Name	POB
pid1	John	Austin

Person

Relational Database

Subject	Predicate	Object
id1	hasType	“song”
id1	hastitle	“rain”
id1	sungBy	pid1
pid1	hasName	“John”
pid1	bornIn	“Austin”

RDF Store

Simple Protocol and RDF Query Language (SPARQL)

SPARQL is the W3C standard as query language to RDF.

Example SQL Select Statement

SELECT title from Books
WHERE book_id = “book1”

Example SPARQL Select Statement

SELECT ?title
WHERE { book1 hasTitle ?title }

A **triple pattern** is a predicate defined in the SPARQL where clause that indicates what kind of triple we are looking for

A **query variable** is a component with a “?” that will return every value of that component in the database

Simple Protocol and RDF Query Language (SPARQL) cont.

Examples Join Statements

```
SELECT  ?title ?price
WHERE   {  ?x ns:price ?price .
           ?x dc:title ?title . }
```

```
SELECT  ?title
WHERE   { ?x ns:isType ?book.
            ?book dc:title ?title.}
```

Joins can happen on query variables only

Motivation

What is the difference from SQL querying on regular relational databases?

- Movement from structured schemas to partially structured schemas
- Queries explore unknown data structures
- Enabling joining and obtaining information from multiple datasets with one simple query
- Processing hundreds and hundreds of datasets is expensive … so we optimize!

Topics for Today

- Efficient Indexing Techniques
- Optimization in Joins
- Optimization using Selectivity Estimations

Efficient Indexing Techniques

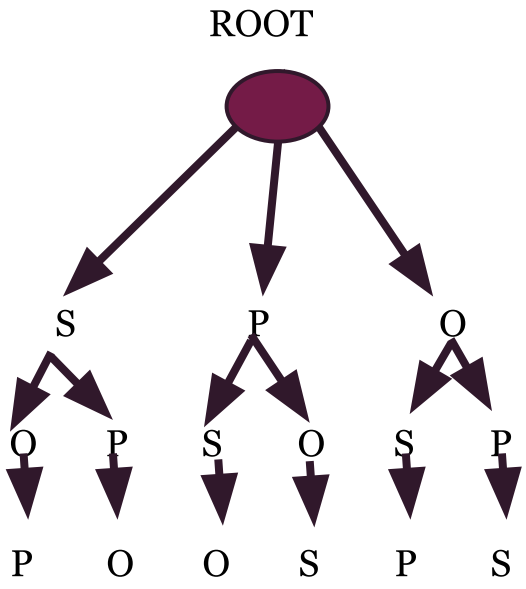
How do we Index Triples?

To index an RDF triple, we index through an **access pattern**. An **access pattern** is a combination of how each component in the triple is specified, be it a literal or a variable

1. ?x, ?y, ?z
2. s, ?y, ?z
3. ?x, p, ?z
4. ?x, ?y, o
5. s, p, ?z
6. s, ?y, o
7. ?x, p, o
8. s, p, o

Structure for Indexes

Multiple Access Pattern (MAP)



Example:
select ?x
where{ ?x property:student "UT" }

Triple Pattern - (?x, property:student, "UT")

POS - (property:student, "UT", ?x)
OR
OPS - ("UT", property:student, ?x)

The RDF-3X Engine for Scalable Management of RDF data

Thomas Neumann · Gerhard Weikum
Published 1 September 2009. VLDB Journal

Triple Store Implementation

Composed of three different data structures, but today we are only focused on two of the structures:

Mapping Dictionary - for each component in an RDF triple, the component is mapped to an object id (OID)

Ex. (Lori, major, “CS”)

OID	Dictionary
1	23
9	“CS”
4	Lori
15	“major”
24	“flower”

Triple Store Implementation (cont.)

Compressed Index - uses a MAP index pattern of a compressed RDF triple (a triple formed of its OIDs)

Ex.

insert data {Lori, major, “CS” }

(Lori, major, “CS”) ->(4, 15, 9)

1. **SPO-**(4,15,9)
2. **SOP-**(4,9,15)
3. **POS-**(15,9,4)
4. **PSO-**(15,4,9)
5. **OPS-**(9,15,4)
6. **OSP-**(9,4,15)

Query Processing

Query Processing and Translation for SPARQL is very similar to SQL with the exception of several nuances:

- Indexing on each Triple Pattern versus selecting one particular index
- Query Graph is based on Triple patterns versus relations
- Favors Bushy Join Trees versus Deep Left/Right Trees of R* Optimizer

Nuance 1: Index Access Pattern for each Triple Pattern

Example:

```
select ?u where{
?u <crime> .
?u <likes> "A.C. Doyle" .
?u <friend> ?f .
?f <romance> .
?f <likes> "J. Austen" .
}
```

Indexing Patterns

PS - (crime, ?u)

OPS - ("A.C. Doyle", likes, ?u)

POS - (friend, ?f, ?u)

PS - (romance, ?f)

OPS - ("J.Austen", likes, ?f)

Nuance 2: Triple Pattern Query Graph

SQL

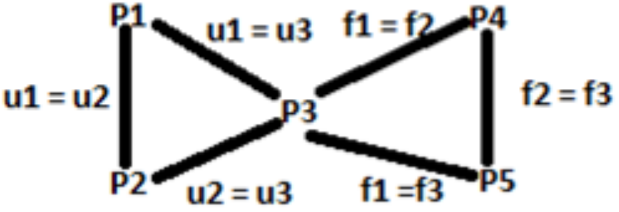
P1 - ?u1 <crime> .

P2 - ?u2 <likes> “A.C. Doyle” .

P3 - ?u3 <friend> ?f1 .

P4 - ?f2 <romance> .

P5 - ?f3 <likes> “J. Austen” .



SPARQL

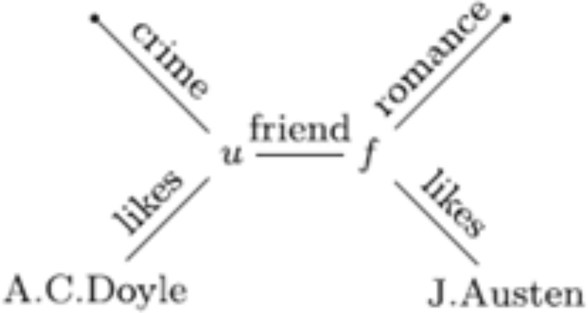
?u <crime> .

?u <likes> “A.C. Doyle” .

?u <friend> ?f .

?f <romance> .

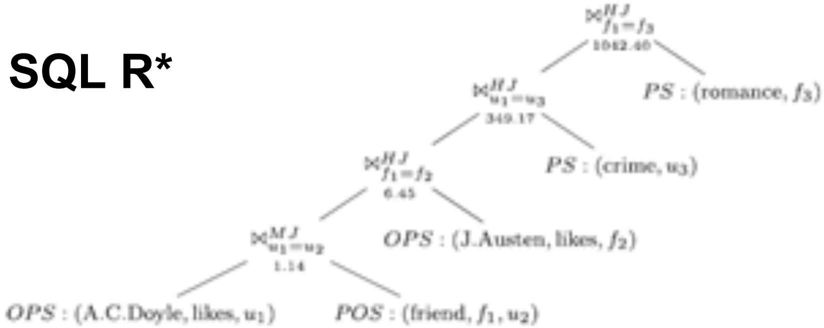
?f <likes> “J. Austen” .



Nuance 3: Bushy Join Trees versus Left/Right Deep Trees

- Attempts to use merge joins as much as possible
- Bottom-Top Dynamic Program is implemented to cache joins to increase efficiency

SQL R*



SPARQL RDF-3X

Optimization of Joins

Scalable Join Processing on Very Large RDF Graphs
Thomas Neumann and Gerhard Weikum

SQL & SPARQL

SPARQL queries over RDF map to SQL SELECT statements

- So why not query RDF graph with the performance of SQL
- Algorithms have been proven to be complete and sound
- Adaption of relational databases algorithms to RDF and SPARQL

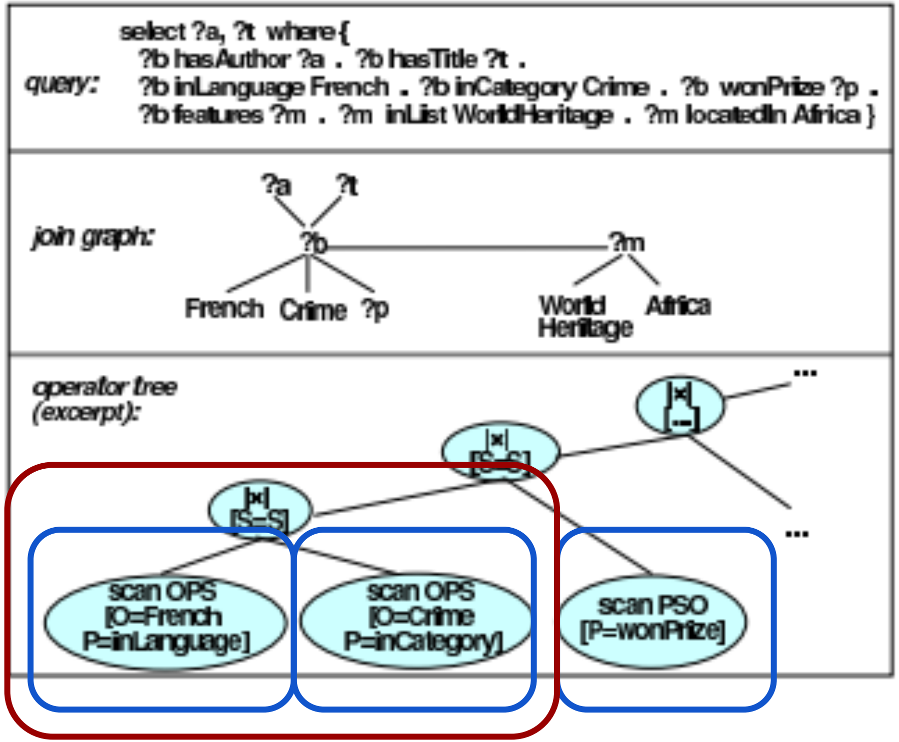
SQL & SPARQL

- SPARQL Basic Graph Pattern is the conjunction of triple patterns, where each is matches the given attributes
- Assumptions of relational operations:
  - Complete
  - Sound
  - Sequential
- Idea run sequential operations on parallel

```
SELECT  ?title ?price
WHERE { ?x ns:price ?price .T1
        ?x dc:title ?title    .T2}
```

T1 Joins T2 on ?x

SPARQL Joins



SQL Joins => SPARQL Joins

RDBMS

Scan = Simple predicates filter the relations

1. Merge Join

2. Hash Join

SPARQL & MapReduce

Scan = Each triple pattern filters the graph

1. Merge Join

2. Hash Join

SPARQL on Very Large RDF Graphs

Triples of the form : ?x <isType> ?y

is not really selective and will return a large data set.

?x ?y ?z is a really big problem. Hopefully, there are not many queries using this triple pattern.

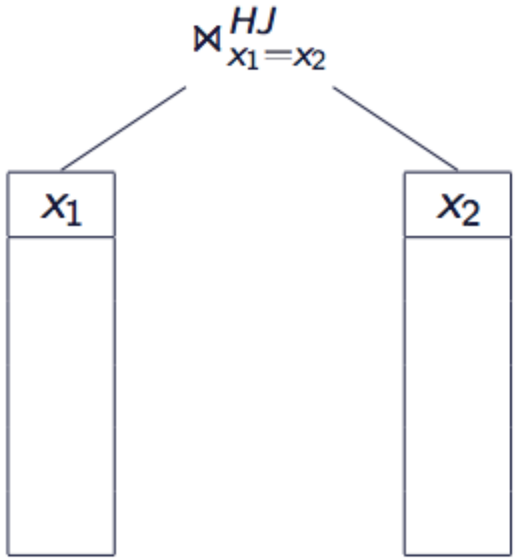
This is true because for some queries only the conjunction of triple patterns as whole is selective.

Execution Plan on SPARQL

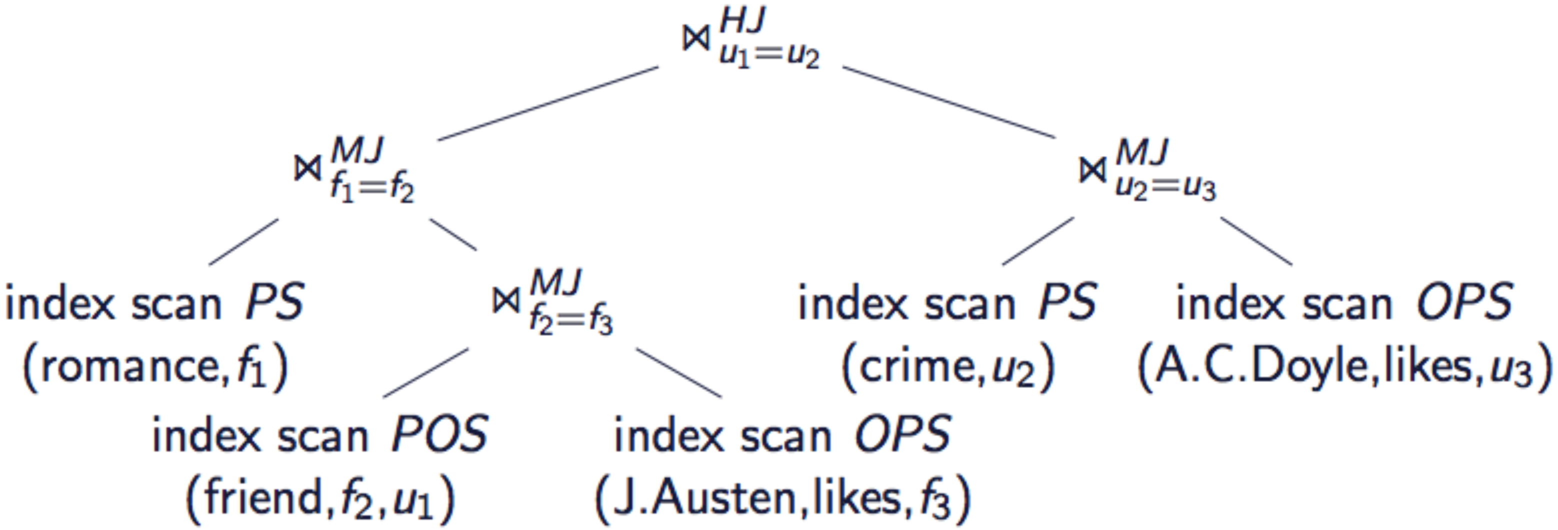
- A typical set of possible execution plans would include bushy trees!
- Bushy trees give more opportunity for parallelization
- Only 1 option for scan:
  - Index Scan
- Only 2 options for Joins:
  - Hash Join
  - Merge Join

Execution

- The scan operations are launched as the entry point of the pipeline
- Merge Joins are the next step in the pipeline
- A Hash Join would merge the two output streams into single pipeline
- Bushy trees implies multiple joins running in different jobs



Execution Tree Plan



Sideways-Information-Passing SIP

SIP: Pass relevant information between separate joins at query runtime

Goal: Highly effective filters on the input stream of joins

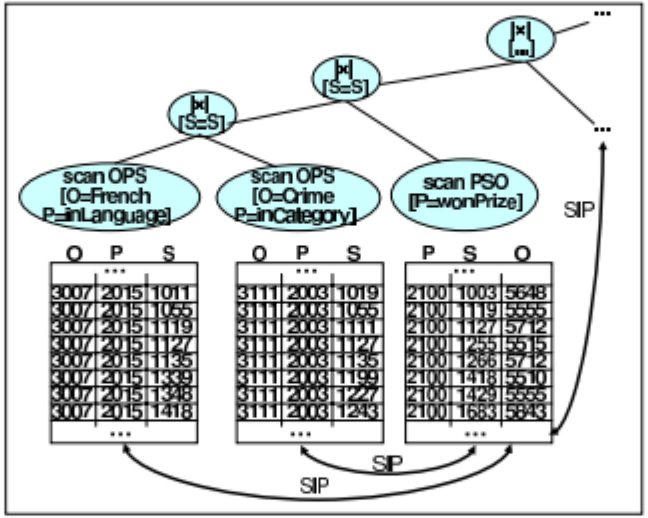
(Similar to magic sets)

This is a RDF-specific application of SIP. It enhances the filter on subject, predicate and object

Sideways-Information-Passing SIP

“Sideways”: Pass information across operators in a way that cuts through the execution tree

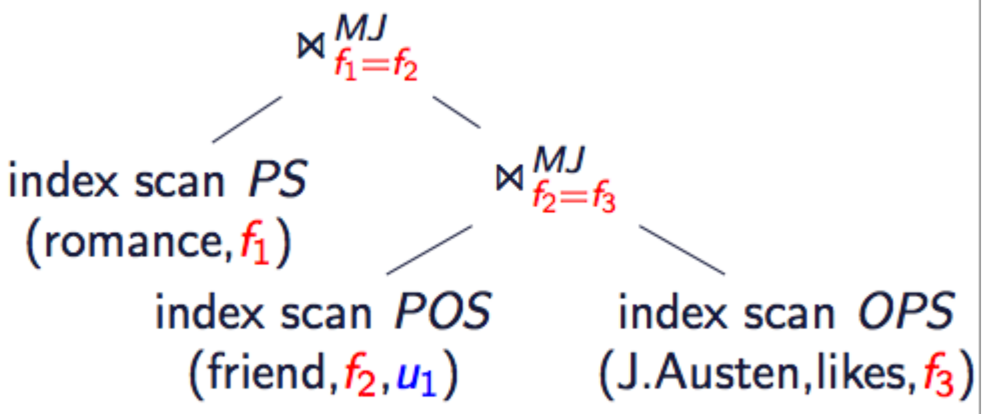
- Restrict scans
- Prune the input stream
- Holistic, there is no data flow



Sideways-Information-Passing SIP

Merge Join

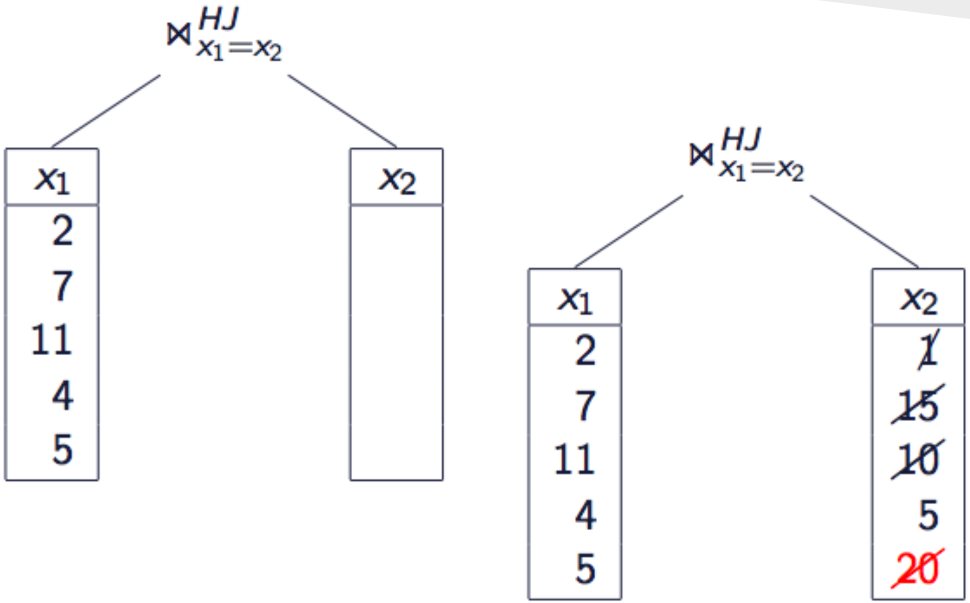
- Ascending order on the index value
- New constraint for each scan
  - f1>= f2
  - f2 >= max(f1,f3)
  - f3 >= max(f1,f2)
- The last values are recorded in the shared structure



Sideways-Information-Passing SIP

Hash Join

- There is not direct comparison index value
- Use of domain filter(min, max)
  - 2 domains
    - Observed Domain
    - Potential Domain
  - Intersection of both



Sideways-Information-Passing SIP

Index Scan

- It uses two previous techniques to skip and find “gaps” in the scan
- Index Scan are triple store in a B+tree

Sideways-Information-Passing SIP

- Results:
    - SIP aims to reduce the overhead of intermediate results
    - The higher in the tree the more accurate the domain filters become
    - SIP is still dependent on the execution order
        - Bad join order may to poor performance
        - Can we do better? Use selectivity and cardinality

Query Optimization using Selectivity Estimations

"SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation"

Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, Dave Reynolds

Basic Graph Pattern

- Basic Graph Pattern or BGP? - set of triple patterns

  ?x type Person .

  ?x hasSocialSecurityNumber “555–05–7880”

- Query Optimization Goal
  - To find an optimized execution plan
  - That means, to find the optimized order of executing the triple patterns

Triple Pattern Selectivity

- Def.: Fraction of *RDF data triples* satisfying the *triple pattern*.

- Selectivity of a triple pattern t = (s, p, o),
  - ***sel(t) = sel(s) * sel(p) * sel(o)***
  - Assumption: sel(s), sel(p), sel(o) are statistically independent.

Triple Pattern Selectivity

- Selectivity of Predicate
  - sel(p) = T_P/T, when p is bound

    here, T_p = number of triples matches P

    T = Total number of triples in RDF
  - sel(p) = 1, when p is a variable

Joined Triple Pattern

- Joined Triple pattern
  - A *pair* of triple patterns that share a variable

    Return the name of person who have SocialSecurityNumber = “555-05-7880”.

    select ?x where{

    ?x type Person .

    ?x hasSocialSecurityNumber “555–05–7880”}
  - Size - the size of the result set satisfying the two patterns

Joined Triple Pattern Selectivity

- Let P represents a Joined Triple pattern

$$sel(P) = S_p/T^2$$,where

Sp = upper bound size Joined Triple pattern P

T= total number of triples in RDF dataset

Basic Graph Pattern Optimization

BGP

1 ?X rdf : type ub : GraduateStudent .

2 ?Y rdf : name ub : University .

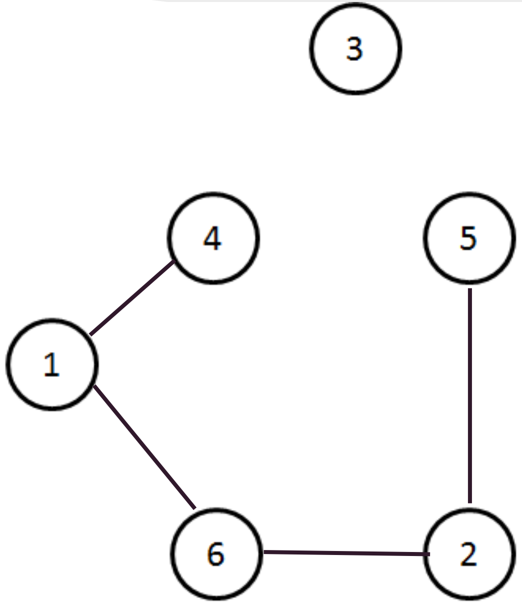
3 ?Z rdf : dept ub : Department .

4 ?X ub :memberOf ?Z .

5 ?Z ub : subOrganizationOf ?Y .

6 ?X ub : undergraduateDegreeFrom ?Y .

Graph



G

node: a triple pattern
edge: joined triple pattern

Basic Graph Pattern Optimization

BGP

1 ?X rdf : type ub : GraduateStudent .

2 ?Y rdf : name ub : University .

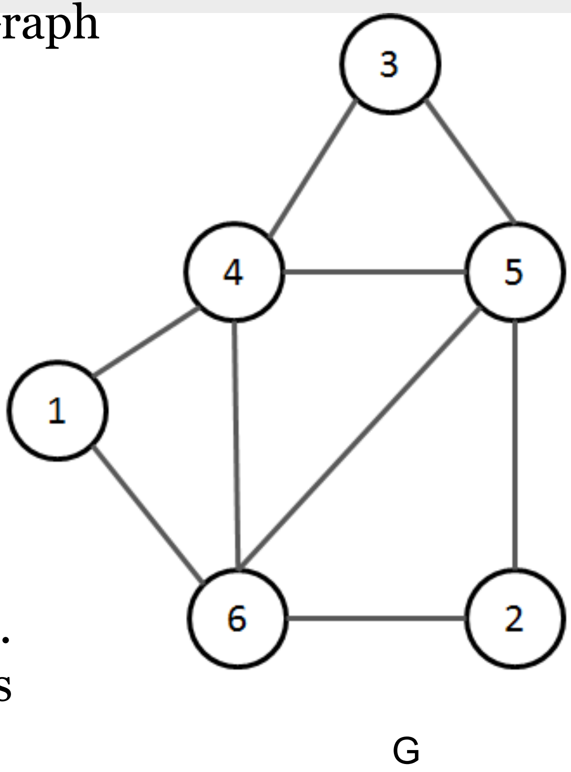
3 ?Z rdf : dept ub : Department .

4 ?X ub :memberOf ?Z .

5 ?Z ub : subOrganizationOf ?Y .

6 ?X ub : undergraduateDegreeFrom ?Y
.

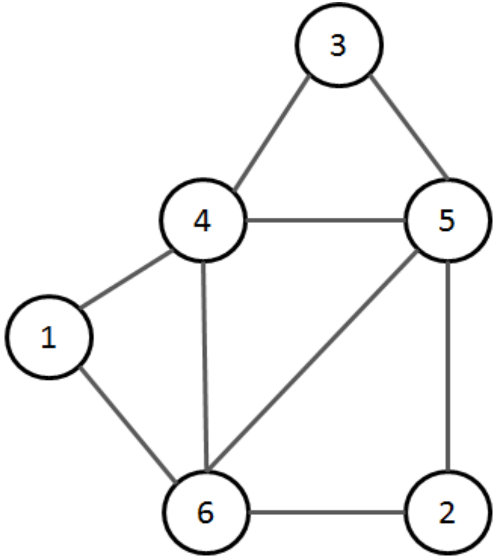
Graph



Execution plan: an order of nodes.
An order to join the triple patterns
Ex. 1, 2, 4, 3, 5, 6

Deterministic Execution Plan Generation

Input



+

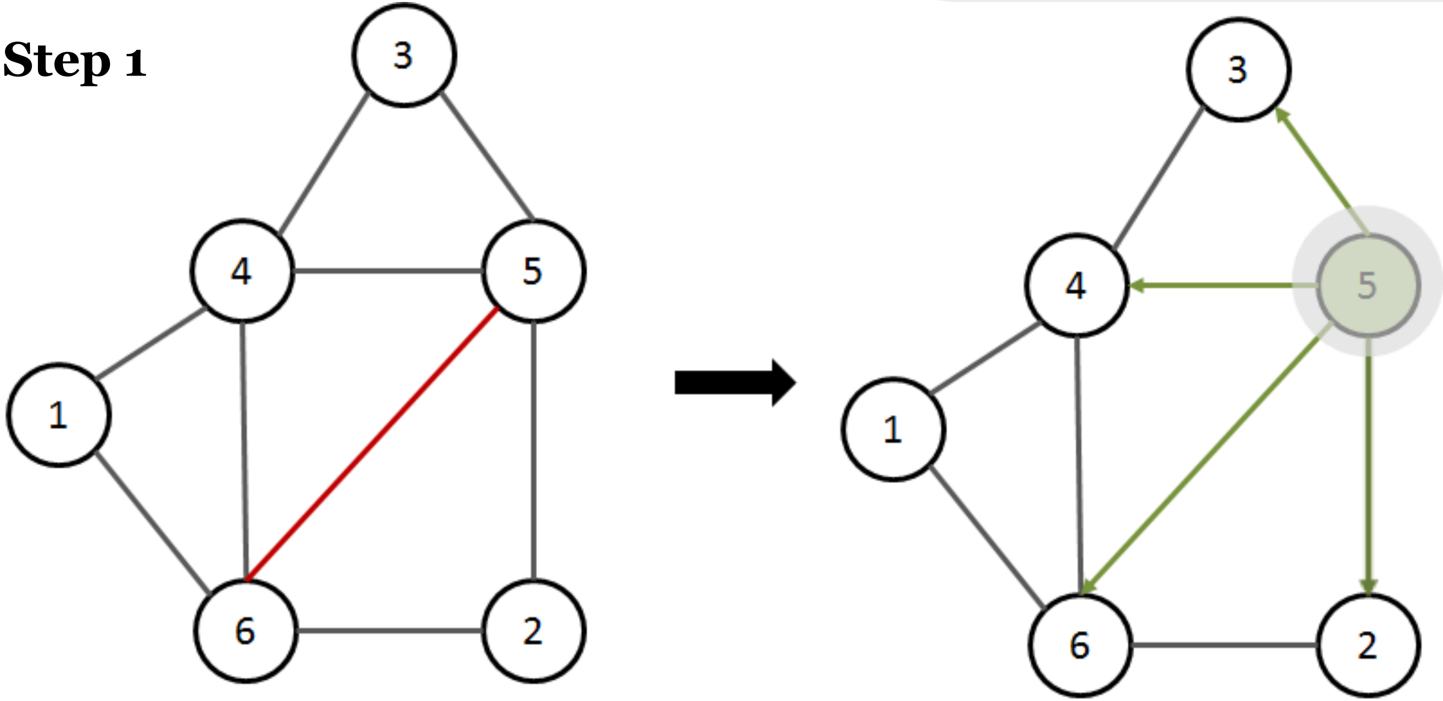
Node selectivity is *Triple Pattern Selectivity*

Edge selectivity is *Joined Triple Pattern Selectivity*

Output ***Execution plan***

Execution Plan Generation(contd.)

Step 1



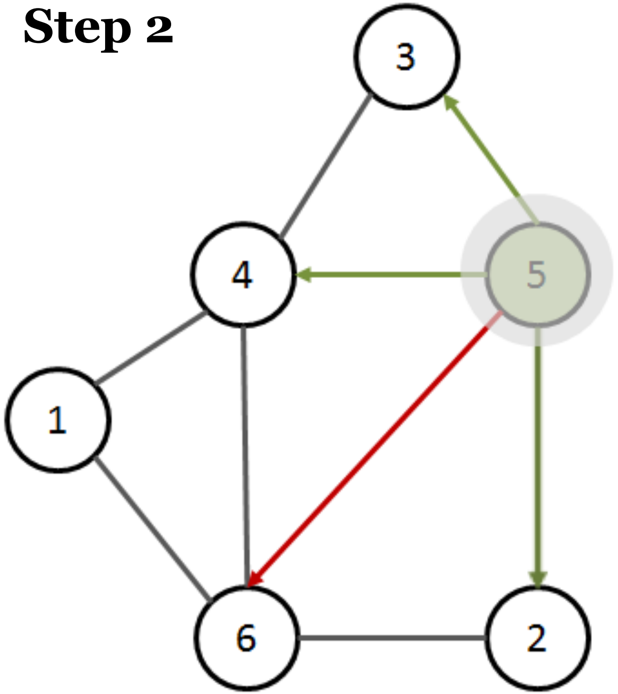
Edges is ascending order of selectivity

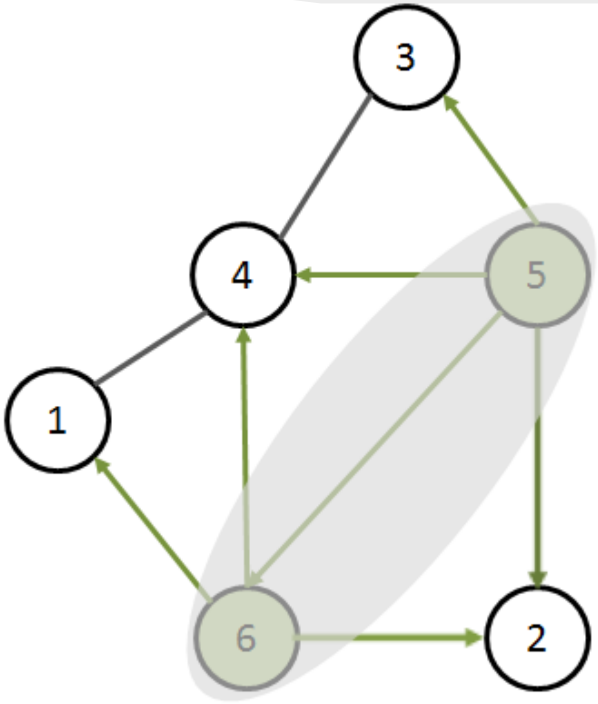
(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Sink: 5

Execution Plan Generation(contd.)

Step 2





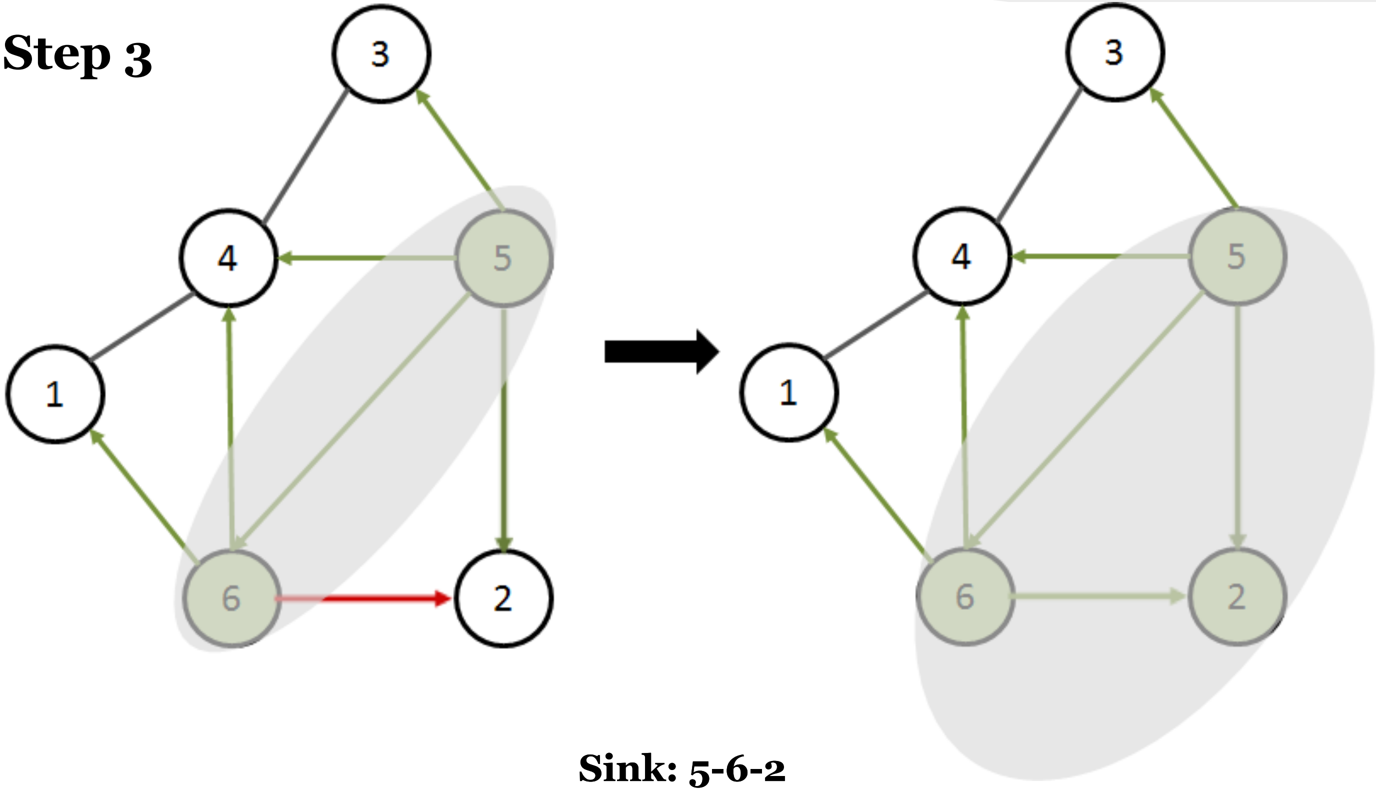
Edges is ascending order of selectivity

(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Sink: 5-6

Execution Plan Generation(contd.)

Step 3



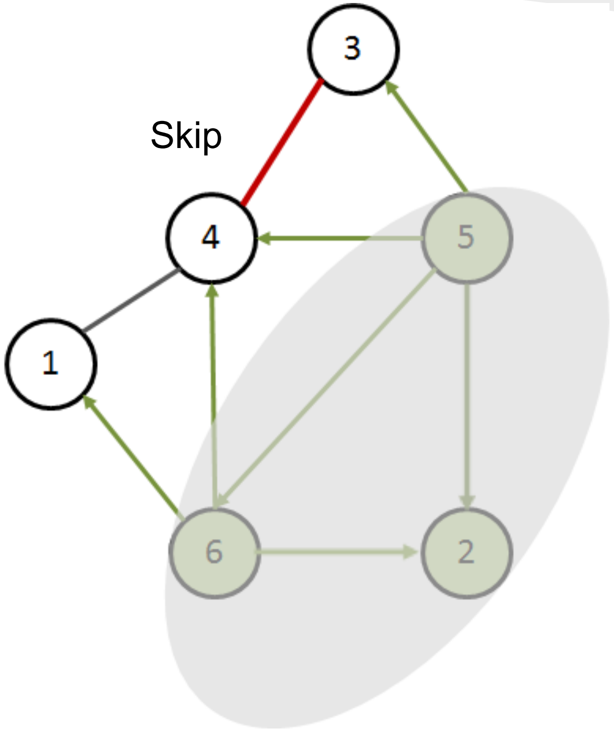
Sink: 5-6-2

Edges is ascending order of selectivity

(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Execution Plan Generation(contd.)

Step 4



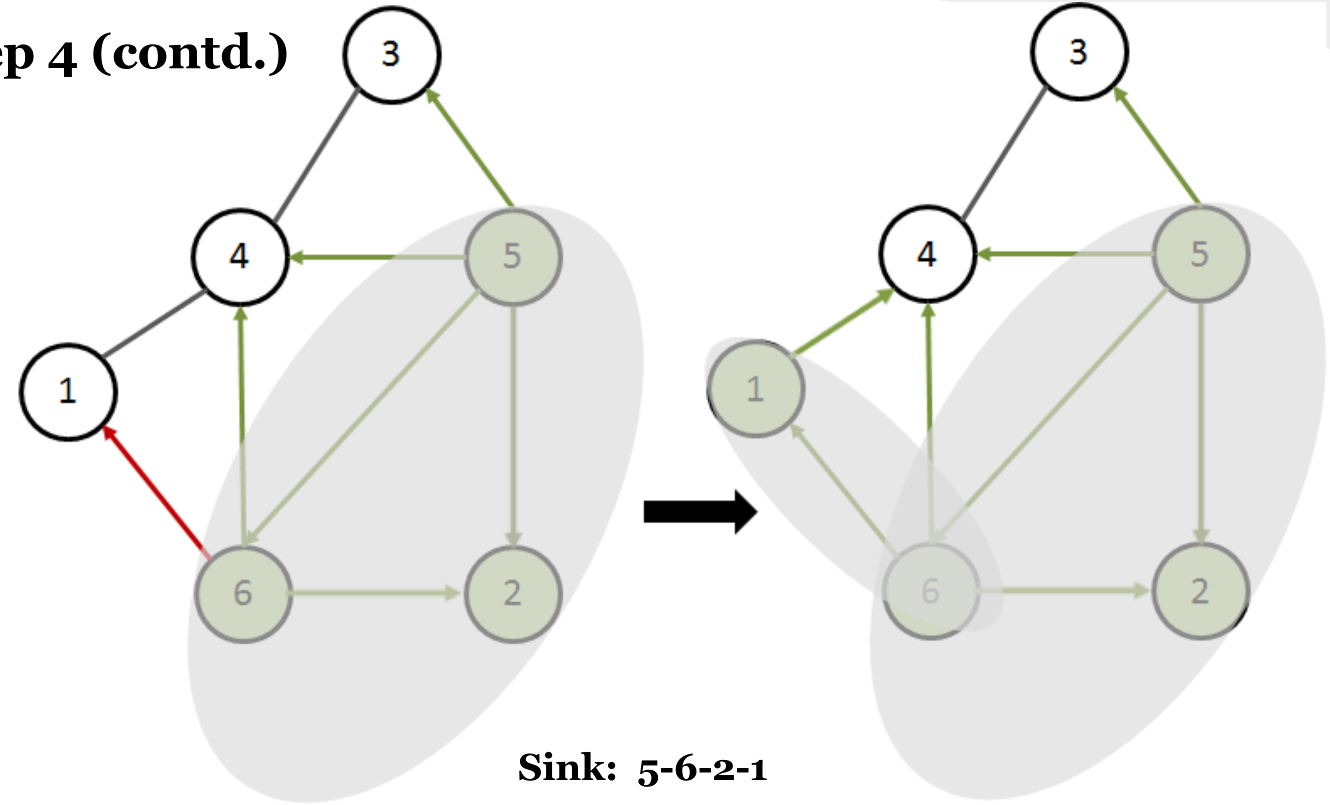
Sink: 5-6-2

Edges is ascending order of selectivity

(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Execution Plan Generation(contd.)

Step 4 (contd.)



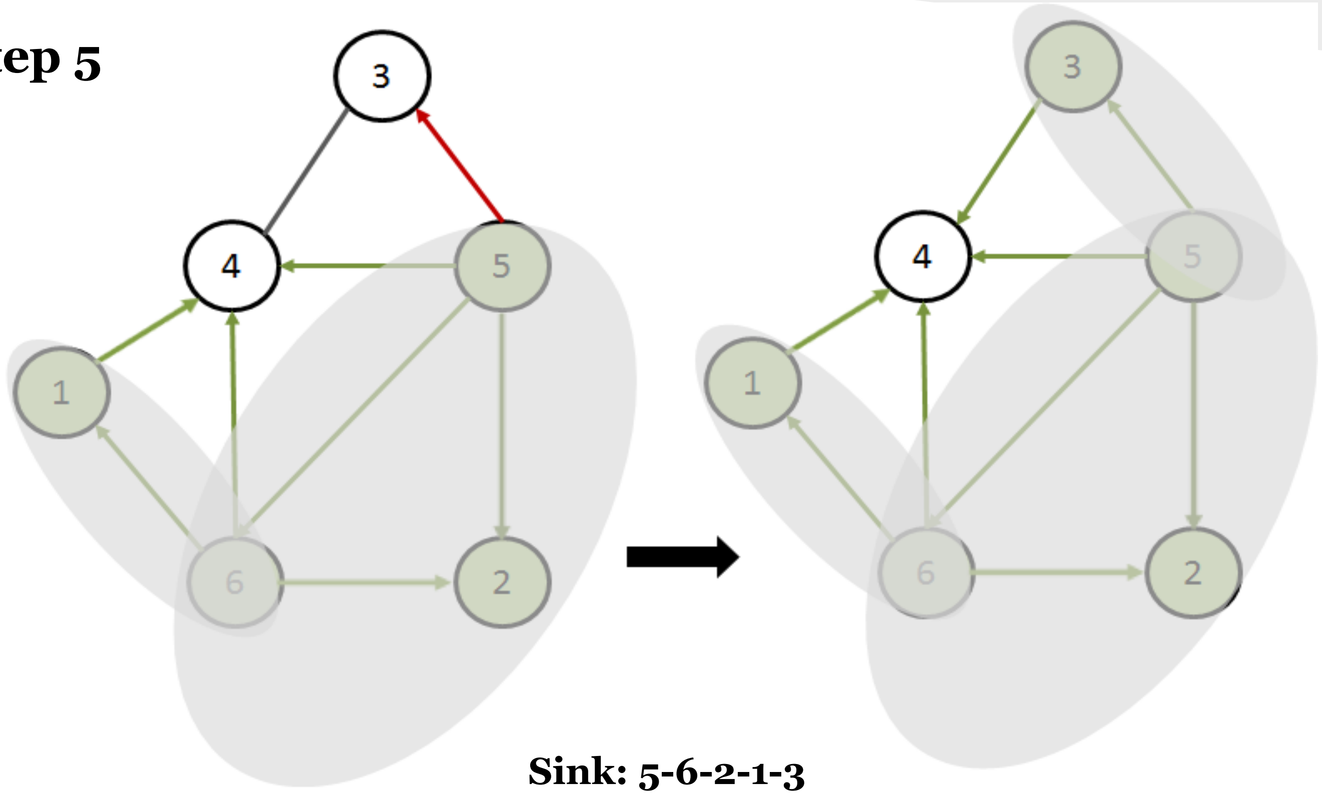
Sink: 5-6-2-1

Edges is ascending order of selectivity

(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Execution Plan Generation(contd.)

Step 5



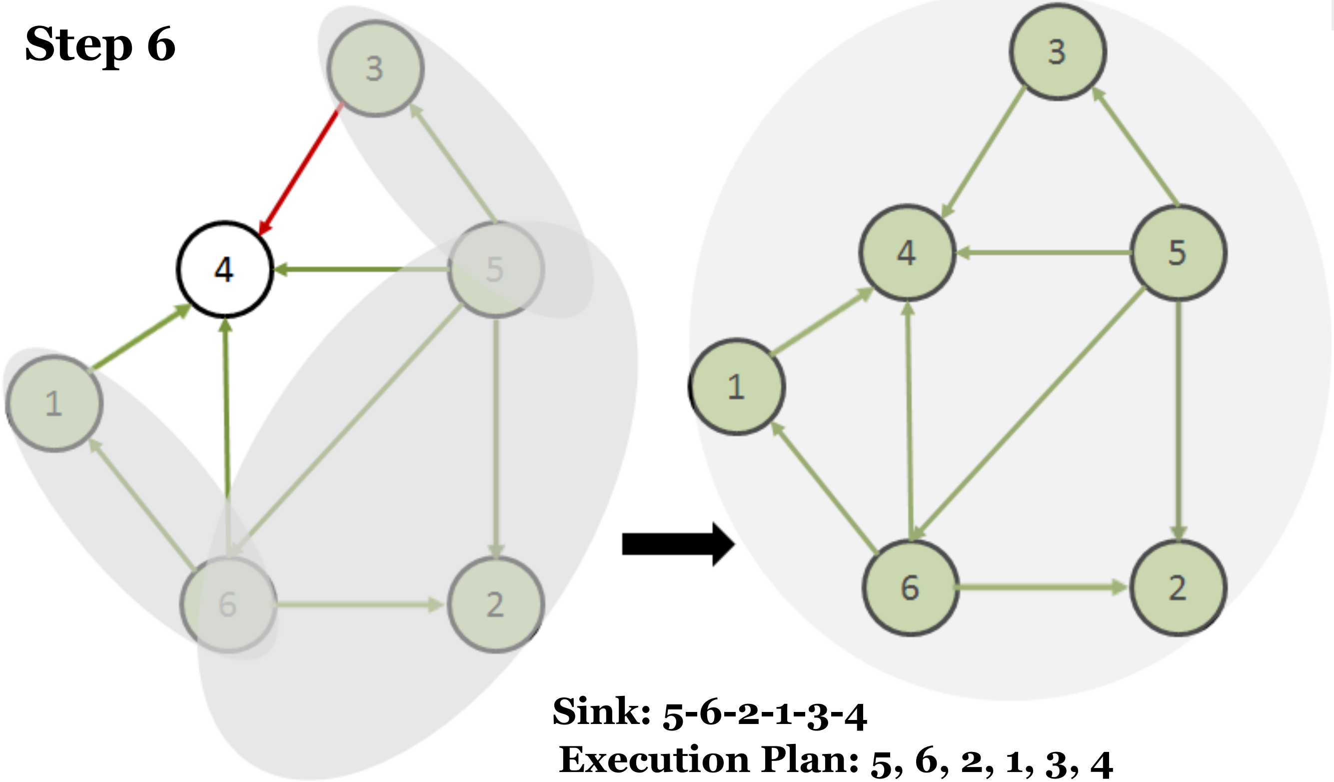
Edges is ascending order of selectivity

(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Sink: 5-6-2-1-3

Execution Plan Generation(contd.)

Step 6



Edges is ascending order of selectivity

(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Sink: 5-6-2-1-3-4

Execution Plan: 5, 6, 2, 1, 3, 4

Deterministic Algorithm

Algorithm 1 Find optimized execution plan EP for $g \in \mathcal{G}$

$N \leftarrow Nodes(g)$
$E \leftarrow Edges(g)$
$EP[size(N)]$
$e \leftarrow SelectEdgeMinSel(E)$
$EP \leftarrow OrderNodesBySel(e)$
while $size(EP) \leq size(N)$ **do**
  $e \leftarrow SelectEdgeMinSelVisitedNode(EP, E)$
  $EP \leftarrow SelectNotVisitedNode(EP, e)$
end while
return EP

Select Sink (*Deterministically*):
select the **minimum selectivity edge** xy
if sel(x) <= sel(y) then
sink = x
else sink = y

Main Loop: While there is a *non-visited* node
xy <- **Next minimum selectivity edge**
if **one of its endpoint is visited** (say x is visited), then
add y to the execution plan
make y visited

What about disconnected graph?

- Graph G may have more than one component
- Like System-R algorithm, take cross product of result sets of components.

Properties

- *Deterministic execution plan* based on selectivity estimations.
- Size of intermediate result set is reduced.
- Cartesian product of intermediate results is avoided within a component.

Summary

You now know about basic Query Optimization in RDF with SPARQL!

SPARQL optimizer will have all of three fundamentals that we spoke about today:

- Due to the simplicity of the RDF model, we are allowed to index on every component in an RDF triple
- SPARQL involves many joins in their queries, and thus we must be aware of only executing the most optimal of query plans
- With SPARQL having deterministic solutions, we do not have to exhaust the entire search space

Thank You

Questions?