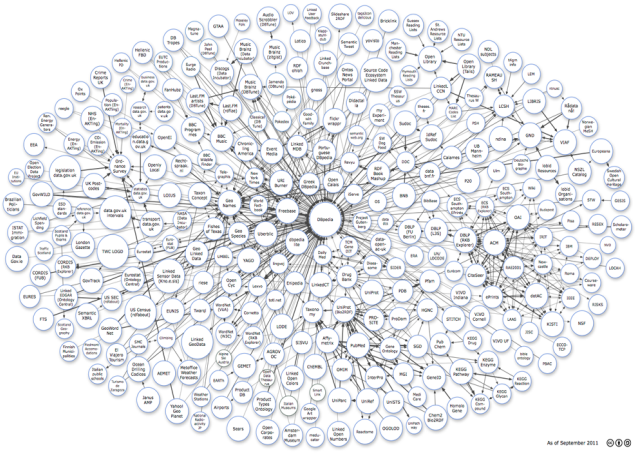
Query Optimization with RDF using SPARQL

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/"

What is the semantic web? Currently, lots of data is found on the web and everything is connected in some way or form, thus forcing us to think of data in a graph-like format. Since there is a push to view data in this way, it is in our best interest to think/model data in this graph-like format.

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)

An example of thinking of data in a graph-like structure, is to see data in a database known as RDF (Resource Description Framework). An RDF database is composed of data in the form known as triples. Triples are simply tuples with three attributes: Subject, Predicate, and Object, in that specific order.

Subjects are composed of only unique resource identifier (URI) which helps identify the source of the triple. It could be a website, a document; anything to help identify the triple. It is very common for URI to be very long since they can take in long names such as a URL. Thus, it is very common to have a variable to shorten the URI name in the triple. An Object can be a URI or a literal, where a literal could be String, float, long, etc. A Predicate is a URI that used to relationship between the Subject and the Object in the triple.Keep in mind that a URI and a literal are not the same thing. A URI essentially acts as a primary key, but the database can have multiple primary keys for that component.

In the example, Notice how the variable Lori holds the website www.example.com/Lori, a URI, defined as a Prefix. Lori represents the Subject, property:age represents the Predicate, and 23 represents the Object.

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

In SQL, we are used to setting up our tables by giving it a name, telling it what column names will be in that table, and then we insert tables based on that table’s criteria.

In RDF databases, there are no sense of tables, and data is usually in one database. RDF is trying to move away from a traditional structured database, to a partially structured database. In the example, we are used to seeing relational database where we have tables and attributes, as we see the contents of the relation Song and Person. Notice how the table Song has the columns SID, Title, and Singer; and Person has the columns PID, Name, and Place Of Birth (POB). In RDF, we have the predicate column, as defined in the previous slide, is used to help describe how the subject and object relate, hence why RDF is a “description” framework.

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

In the following example is a SQL statement with an equivalent SPARQL statement.

What the SPARQL statement is doing is that it is finding all triples in the database where the Subject = "book1" and the predicate = "hasTitle". Analyzing this Where-Clause predicate is called a **triple pattern**, which search for triples that have the specified values in the triple pattern. If a component has a "? in front of it, called a **query variable**, will return every value for that component.
Thus this query will return every triple that has a "book1" for the Subject component and "hasTitle" for the Predicate component.

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

6

One aspect about SPARQL is that having multiple triple patterns in the where clauses causes many self joins to happen in RDF. Joins are taken place where the components in any of the triple patterns are the same, whether it be a variable or a specified component.

Joins can happen on query variables only. It is a semantic that has been formed in SPARQL.

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!

No relational Schema - In normal relation databases, we are used to having very rigorous table structures that could have many attributes, foreign keys, primary keys, etc., and tables vary from one another. In SQL, you cannot query on a database unless you know something about the data inside of the database, meaning what tables, what attributes are in that table, etc. RDF databases are composed of only three attributes that are consistent throughout every RDF database, letting us query on the database without prior knowledge of what is inside of it.

Queries explore unknown data structures - In SQL, queries calls as well as query results are based on the tables that we want to look at in our query. Since there are no table structures in RDF, we query based on what data we want to retrieve, thus is more data-driven in the query calls than queries on Relational Databases.

Enabling joining and obtaining information from multiple datasets with one simple query - In SQL, in order to retrieve data from datasets, you must specify the tables you want to retrieve and multiple calls to the database must be called if you want data from different tables. RDF can simply use one call in order to get data from many different datasets, quickly and more efficiently than SQL since it will only make one call to the database rather than multiple calls

And of course, the last bullet is the motivation of this project: to see the optimizations of RDF databases in order to query commands quickly and efficiently

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

In SQL, how do we normally index records? We look at a relation and we index attributes individually, creating a B+ tree for each attribute we wish to index. In SQL, we prioritize what attributes we wish to index, in case the table itself has many attributes to index on.

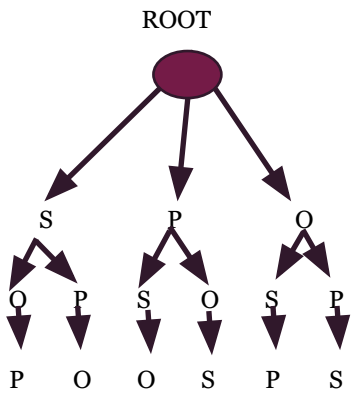
Since we are guaranteed at most three defined components in an RDF triple, we can index on an access pattern, where the specified components define a key for us to index on.

We assume that that the letters beginning with a "?" are variables, and letters not beginning with a "?" are specified components. Since there are three components in a triple, and each component either be a variable or specified, we can have up to 2^3 = 8 access patterns to index an RDF database

To help us figure out what kind of access pattern we would like to index on, we figure out how many components of the triple do we wish to index on. In the Access Pattern 1, notice how all the components are variables. Since there are all variables, we would have no index using that access pattern, since that would index every unique triple, which would defeat the purpose of indexing in general. However on Access Pattern 8, we would index on all three components and create an index structure based on that. For Access Patterns 5-7, their access pattern are on only 2 of the components, that's why those 3 access patterns have every combination of indexing on 2 components of a triple.

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)

11

For B+ tree structures in RDF, it chooses an access pattern by deciding how many components of the triple the programmer chooses to index on, and creates a B+ tree based on that access pattern.

For Multiple Access Pattern (MAP) index structure, the B+ tree takes on indexing all three components, thus it takes on the access pattern of S, P, O and makes an index for every permutation of S, P, O of a triple for a total of six different indexes. We keep in mind that all access patterns are all pointing to the same set of data (except permuted); it is only matter of a given query is it faster to go through the S bucket first, versus the P bucket. The leaf nodes at the end of bucket gives us the triples that match the specified S, P, O based on the specified values in the query.

For the example query above, we retrieve indexes based on the triple patterns that are in the where clause. Since you cannot index on variables, you can rearrange the following triple pattern in one of two ways: By either starting at the P bucket, then going to the O bucket, or you can go through the O bucket first, then the P bucket in the index tree. Notice how we would never go through the S bucket first. This is because the Subject component of the triple pattern is a variable, which means it would look at all possible Subjects loaded in the RDF. We essentially rearrange the triple pattern in a way that we can access the index through specified components.

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

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

What most people thing is that “oh no, we are replicating the data 6 times. Isn’t that expensive?” Well, based on implementation, we can help reduce the storage that store a triple 6 times may make.

This paper explains a way of how to create and maintain a data structure that they call a “triple store” in order to have excellent performance with query commands in SPARQL. They call their implementation the RDF-3X engine.

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"

The mapping dictionary is kept in alphabetical order, and as each new component is found in a triple, it is given an index and put into the dictionary.

Keep in mind that this mapping dictionary and the triple (Lori, major, "CS") will be used for the next two slides as well.

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)

Compressed Indexes uses a MAP access pattern and creates 6 copies of the the compressed RDF triple inserted into the database, one copy for each of the 6 indexes. Each index in the MAP access pattern is in alphabetically order, and the triples themselves are also sorted in alphabetical order based on the access pattern they follow.

Thus in the following example, the triple (Lori, major, "CS") would be compressed into just a triple with OIDs based on the mapping dictionary, and then we permute the compressed triple in order to store the 6 copies of the compressed triple into the index structure. The paper mentions that not only can they afford the storage of a triple in six different ways, but also since the triples are store in a "compressed form" which are essentially just numbers, the storage is not as bad. Storing numbers are much less memory intensive than storing strings or floats.

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

How is query processing different in SPARQL than in SQL? Both methods are pretty much the same. You find what you want to index on, develop a logical access plan, create a physical access plan, a join tree if necessary and find the minimal plan. However, there are three nuances that makes query processing slightly different in SPARQL than in SQL.

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)

This example will be used throughout the next 2 slides as well. It is simple a query to figure out who are the suspects that committed a crime based on the following criteria defined in the Where clause.

In SQL we are used to defining a particular attribute to index on and querying for that index. In most cases, we don’t index on every attribute in a table because it is very memory intensive if a table has many attributes.

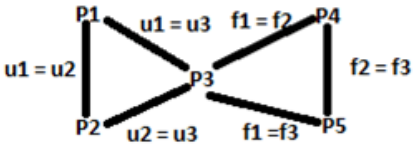
SPARQL, since we have the luxury of having an index on each component of an RDF triple, we never have to make a choice in what attributes to index on, because we can index easily on all three components.

Notice how each triple pattern was rearranged. The index structure of RDF-3x wants the triple patterns to index on the specified components, or prefixes since we cannot index on a variable.Luckily in SPARQL, we don’t have to worry about what attributes of the triple pattern we want to index on, as in SQL. Since we have an index on everything, we always utilize indexes to get what we want.

Nuance 2: Triple Pattern Query Graph

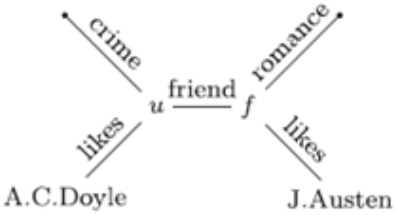
SQL

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” .



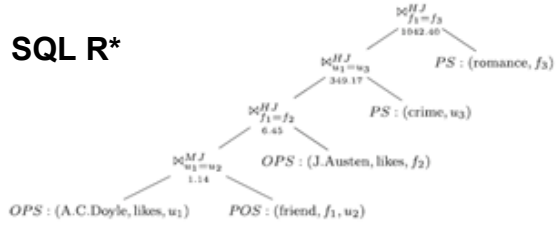
In SQL, we are used to seeing the nodes for our query graph as a relation, and they connect to relations through joining predicates together. In SQL, each one of those where clause triple patterns could be treated as five separate tables and joined together by their connecting predicates which is show in the query graph on the left portion of the slide.

In SPARQL, rather than creating a Query Graph of relations, the triple patterns involve in query ARE the query graph, meaning that the components that constitute as variables are used in the joining process.Thus, in the example, the first three triple patterns are joined based on the query variable ?u, and the last three triple patterns are joined on the query variable ?v. Majority of triple queries will for these “star-graphs” that has many triple patterns connected to only a few query variable nodes.

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

18

How are we used to finding optimal execution plans in SQL? In the R* optimizer by choosing relations that we join based on what join predicates would produce the most selective join, R* would pick a sink node and then join tables in based on a sink node, thus creating a left/right deep tree join execution.

The other aspect of SQL is that since we do not index on everything, after apply local predicates first, we usually revert to using some form of a Hash Join in order to join relations because the records may not be order to effectively use merge joins. Hash Joins require pre-processing of tuples and small nested loops in order to join tuples accordingly for each Hash Join that is processed in a query.

Since all the triples are indexed on each component, the query optimizer will attempt to prioritize merge joins as possible. Since the indexes are in sorted order, the processor will try to capitalize on merge joins as much as possible before resulting to Hash joins, thus the optimizer favors bushier trees versus deep trees. Another reason that merge joins perform well in this implementation is that our index structure is based on numbers. We can easily do merge join very quickly, versus having to do merge joins on Strings. In addition, merge joins don't require any preprocessing because the indexes are already sorted and ready to be linearly searched through in order to do the merge join.

One additional aspect the engine does is that it implemented a bottom-top dynamic program algorithm to cache very big join predicates that are passed in to queries so to not do the computation again. Will this implementation always produce the most

optimal execution plan? This leads us to our next topic...

Optimization of Joins

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

I will be talking about Join Optimization in SPARQL. RDF-3X was one of the early SPARQL engines and therefore it still had a lot work ahead of it. In the case of the Join processing they implemented the naive Join algorithms and then they proceeded to design some optimizations targeting RDF and SPARQL. This is one of the proposed Join Optimizations.

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

It seems that SPARQL and SQL have a lot of common. Actually, there is a direct mapping from SPARQL SELECT queries over RDF to SQL Select statements. So it would be nice to perform SPARQL queries with the SQL enhanced performance. Relational Database Management Systems have been around long enough to optimize and design algorithms that have been proven to be complete and sound. Why not adapt them to SPARQL and RDF and save some of the work. Therefore, a lot of the optimizations on SPARQL and RDF are mainly adapting some old idea to the RDF world. A perfect example of this is indexing paper, presented by Lori.

.

SQL & SPARQL

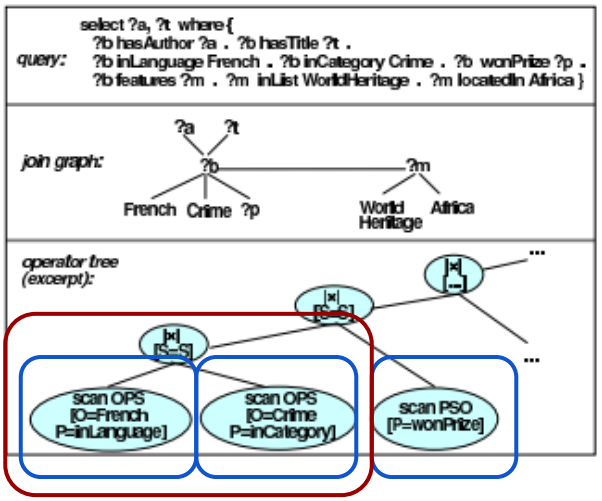
- SPARQL Basic Graph Pattern is the conjunction of triple patterns, where each is matches the given attributes
- Assumptions of relational operations:
  - Complete
  - Sound
  - Sequential
- Idea run sequential operations on parallel

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

T1 Joins T2 on ?x

The graph structure that we try to match over RDF is called a Basic Graph Pattern. BGP is nothing more than the conjunction of triple pattern. The main idea of SQL adaption to SPARQL is that we map the SPARQL statement to computable mapping on SQL. In the case of BGP it entails star joins and chain joins, we saw in class that joins in SQL were optimized by using index techniques, so we actually port even the optimizations. Finally, since we are working on a parallel environment we can run distinct algorithms.

SPARQL Joins



As we saw for System R, SPARQL engines follow the same query processing. The processing of the basic query processing is enclosed by parsing the query, join graph and operator plan. If we relate this image to class, our query graph is presented as a join graph and the operator tree is a logical plan. Therefore, we can assume that SPARQL engines have a similar optimizer as we saw in class for relational databases. However, since we are running on a parallel environment we have different decisions one of them is to run parallel algorithms or run multiple sequential algorithms. The approach of this paper chooses to do the latter. For this example the operations enclosed by a blue box at time 1, and the red box is executed at time 2.

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

RDF-3x and most of the SPARQL engines have implemented the following algorithms. As we mentioned before SQL provided a lot of useful techniques for SPARQL processing. Its worth mentioning that there are more implementation, but this optimizer just consider this 2 and its variants. Joins are pipelined together with index scans. The main difference in the scans is that the predicate will be matched to a relation, in RDF the triple pattern is compared to the whole graph, therefore it could be slow. This is the reason why RDF-3X implements all the index permutations of SPO

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.

However, even if we implement the optimizations from SQL joins into SPARQL and RDF there will bigger bottleneck. This bottleneck occurs when the triple pattern is not really selective and so it returns a large data set. For example the triple ?x <isType> ? y will return a large data set because isType is one of the most common predicates on the RDF world. And if we think about ?x ?y ?z, it really returns the whole RDF graph. This triples patterns are completely valid, the selectivity is a result of the constructed subgraph.

So as we did on relational databases, we want to compute just the relevant data. Remember magic sets lecture?

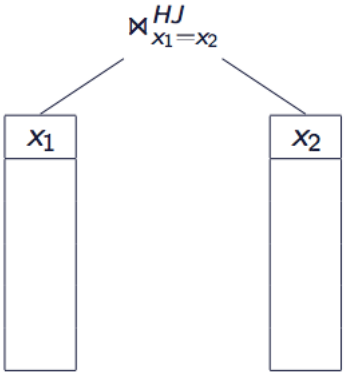
Execution Plan on SPARQL

- 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

A major difference from what we saw during class. Is that RDF-3X optimizer has a dynamic programming algorithm that generates optimal bushy join trees.
The paper does not specify details, but points to the paper where is explained: “Analysis of two existing and one new dynamic programming algorithm for the generation of optimal busy trees without cross products”.
The optimizer in this case will have mainly 2 algorithms to choose from for joining triples and 1 for index scan.

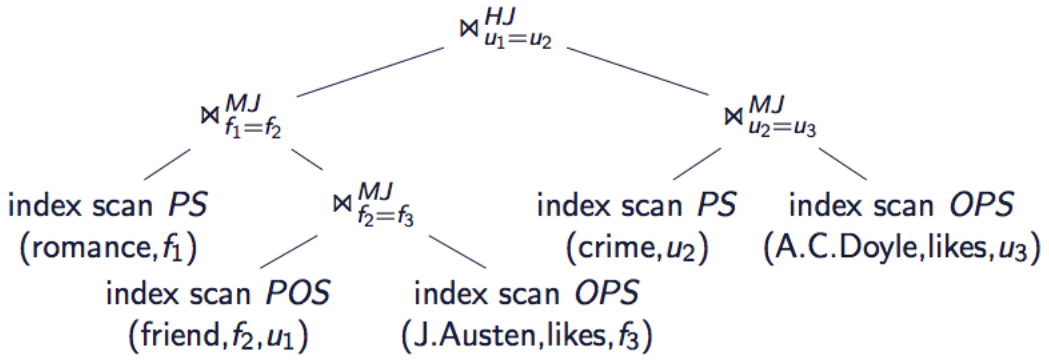
Execution

- 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



This implementation has the goal of creating multiple pipelines to process a query. In other words, this algorithm promotes the uses of bushy trees rather left-deep or right-deep trees. In general the entry point of the pipelines are the index scan, and the merge joins are the next in line. The hash joins will become the pipeline breakers, meaning that will join 2 pipelines together.

Execution Tree Plan



There are some cases where a pipeline just consist on one scan index. However, as we can see the bushy trees give more opportunity to parallelize pipelines compared to the right-deep tree.

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)

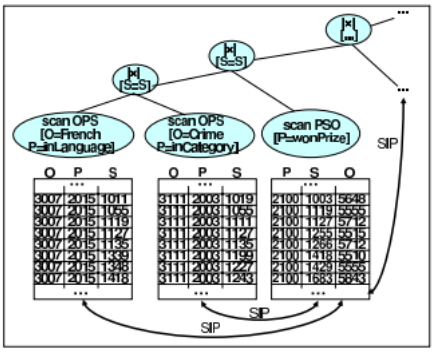
This is a RDF-specific application of SIP. It enhances the filter on subject, predicate and object

The main idea of Sideways information passing is that the parallel processes have access to a shared-memory structure. The purpose of using this technique with SPARQL and RDF is to prune and filter irrelevant triples from operation to operation. The critical point is that RDF is one big table. THis helps because we know that the operations are affected by parallel operation on the same data set.
This is similar to magic sets, however the main difference is that Magic sets are constructed at compile-time and SID is on run-time. The rewrite will happen at run time by creating explicit variables, for each attribute.

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



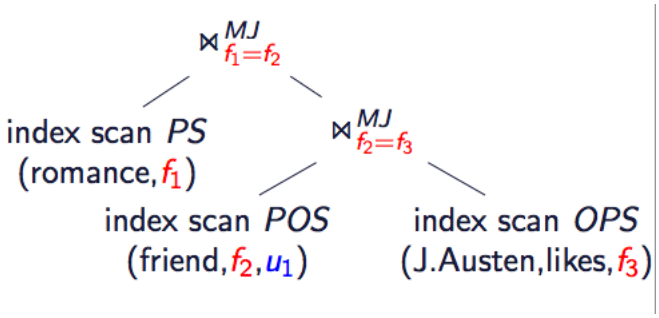
They constructed a light-weight bookkeeper, which keeps track of the max identifier that complied with the triple pattern. It is important to note that their approach is holistic, which means that there is no direction on the data flow. Therefore, the data will be pruned everywhere, even if the operators are not in the same subtree.

The main idea is that during index scans, the scanners will restrict their search from only one predicate to multiple, therefore pruning the resulting set.

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



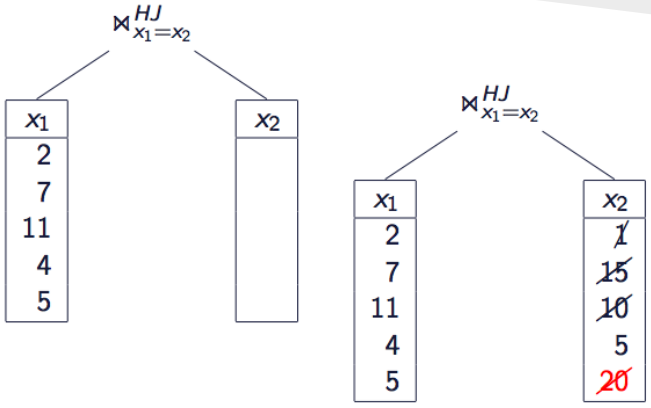
30

A merge join has 2 data streams as input, which is an index scan. The merger join will consume the triples from each stream on ascending order. Therefore we can constrain the consumption by constraining the tuples. This is done by comparing the equivalent attributes in join conditions of the ancestor. In this case, we expect the scan of f1 to be faster and therefore it constraints the join f2=f3, by adding f2 >= max (f1,f3) and f3 >= max(f1,f2). This will prune the resulting data set from the children join to the parent.

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



31

Due to the unorganized nature of hashing, we cannot skip gaps. Therefore, in this implementation our hash join will be not compared to from value to value but to domain. for each attribute the hash will be separated into domains. One domain per variable. The domain captures the whole set values and the hash join will be mainly doing the intersection of all domains.

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

The index scan uses the constraints and domain filters in order to find “gaps” in the triples and skip them. We could compare one by one but this will lead to a high CPU cost. Therefore, thanks to the B+tree structure of the index we are able to optimize. This is because the B+trees contain the actual triples so if the next value after skip, is outside the working memory page, we can easily jump to the next page.This technique reduces the amortized CPU cost to nearly zero, note there are many thousands of triples on a page.

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

I am going to talk about the query optimization approach that uses selectivity estimation of triple patterns to generate query execution plan. Optimizers use selectivity estimations of triple patterns in different ways. I will discuss an interesting and easy technique of query optimization with selectivity estimation presented in the paper "*SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation*".

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

What is Basic Graph Pattern?
Recall that SPARQL query consists of triple patterns. This set of triple patterns is also known as Basic Graph Pattern or BGP. (The triple patterns that make up a query is known as BGP)

So..given a BGP, the query optimization goal is to find an optimized execution plan which is expected to return the result set fastest. That means, to find an order of executing the triple patterns such that the size of the intermediate result set is minimized.

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.

Selectivity of a triple pattern is defined as the fraction of RDF data triples that satisfy the triple pattern.

Selectivity of a triple pattern t is computed with the formula in bold. That means selectivity of subject, selectivity of predicates, selectivity of objects are multiplied together. The assumption here is, selectivities of subject, predicate, object 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

When predicate is a variable, then selectivity is 1, as it can be mapped to any Data triple. Otherwise sel(p) is the ratio of the number of data triples having predicate P to the total number of triples is RDF.
Subject and object selectivities are calculated in the same way.

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

In a set of triple patterns, multiple patterns may share one or more components. When two patterns join on a variable, that pair is called a joined triple pattern. Suppose we have the query to return the name of person who have SSN=“555-05-7880”. The SPARQL query for this consists these 2 patterns. They join on variable x. This pair is known a joined triple pattern.
Size or cardinality of a joined triple pattern is basically the cardinality of result of a query consisting these 2 triples only. To estimate selectivity, we have to take an estimated size. Because to get the actual size, we need to execute the query.

Joined Triple Pattern Selectivity

- Let P represents a Joined Triple pattern

$$sel(P) = S_p / T^2 \text{ ,where}$$

Sp = upper bound size Joined Triple pattern P

T= total number of triples in RDF dataset

Let capital P denotes a joined triple pattern. The selectivity is measured by the ratio of size of the result set of P to the square of the total number of RDF triples.

Basic Graph Pattern Optimization

BGP

1 ?X rdf : type ub : GraduateStudent .

2 ?Y rdf : name ub : University .

3 ?Z rdf : dept ub : Department .

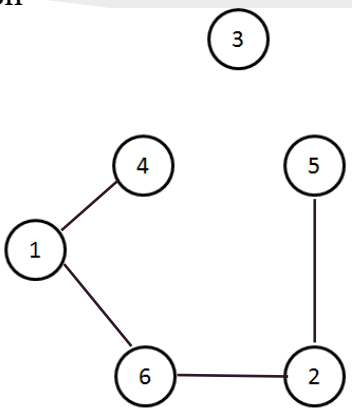
4 ?X ub :memberOf ?Z .

5 ?Z ub : subOrganizationOf ?Y .

6 ?X ub : undergraduateDegreeFrom ?Y .

node: a triple pattern
edge: joined triple pattern

Graph



G

Suppose we have a set of 6 triples. We can build a graph from that. For each triple pattern, we have a node

In this graph, each node corresponds to a triple pattern. There is an edge between two nodes if the corresponding triples share a variable.

Look at pattern 1, it has a variable in common with pattern 4 and 6. So we connect node 1 & 4 with an edge and node 1 & 6 with an edge. In this way, we add the remaining edges.

Basic Graph Pattern Optimization

BGP

1 ?X rdf : type ub : GraduateStudent .

2 ?Y rdf : name ub : University .

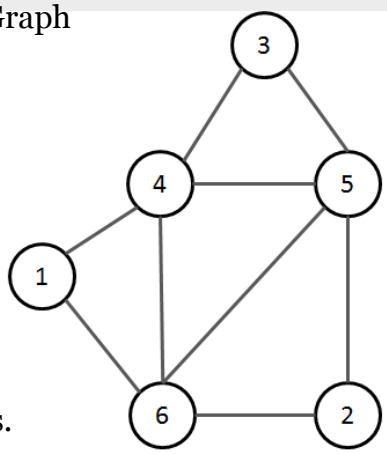
3 ?Z rdf : dept ub : Department .

4 ?X ub :memberOf ?Z .

5 ?Z ub : subOrganizationOf ?Y .

6 ?X ub : undergraduateDegreeFrom ?Y .

Graph



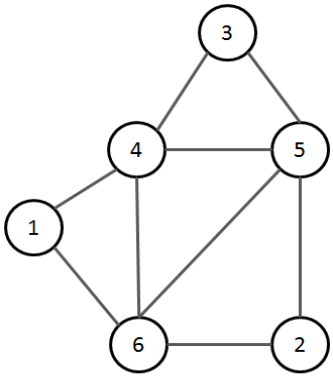
G

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

Finally we get this graph from this set of triple patterns.
An execution order in which triples are to be joined.

Deterministic Execution Plan Generation

Input



Node selectivity is *Triple Pattern Selectivity*

Edge selectivity is *Joined Triple Pattern Selectivity*

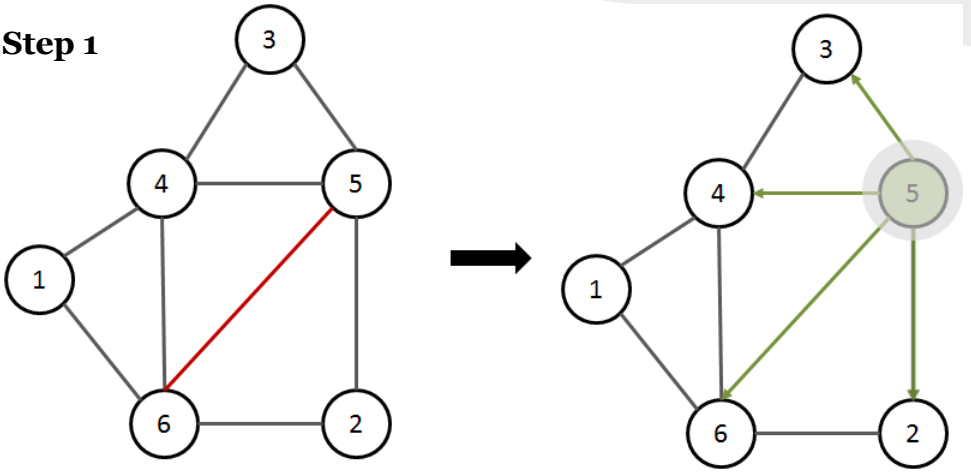
Output ***Execution plan***

42

We have the graph. We also have the edge selectivities listed in ascending order. With edge selectivity I will refer to joined triple pattern selectivity by edge selectivity and triple pattern selectivity by node selectivity.
Now let's derive a deterministic execution plan.

Execution Plan Generation(contd.)

Step 1



Edges is ascending order of selectivity

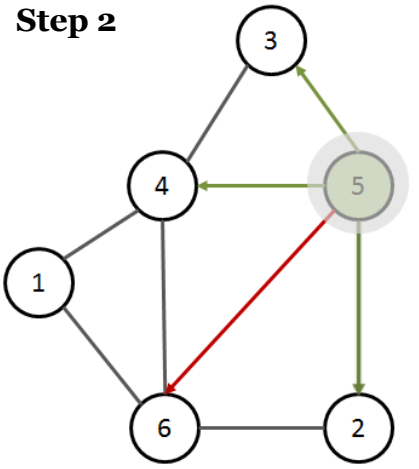
(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

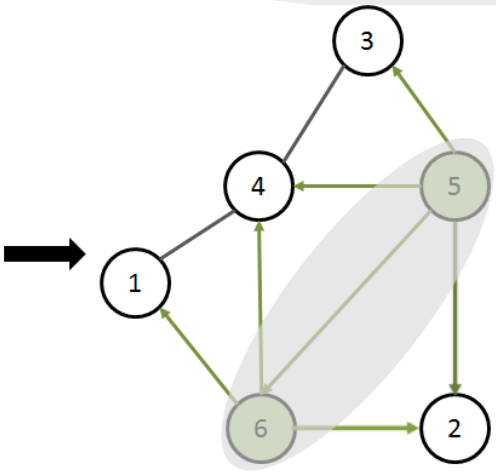
Sink: 5

The algorithm is same as system- R algorithm except that at each step we have a deterministic choice of node. First we select the sink node.
The possible joins to consider are represented with outgoing edges from node 5.

Execution Plan Generation(contd.)

Step 2





Edges is ascending order of selectivity

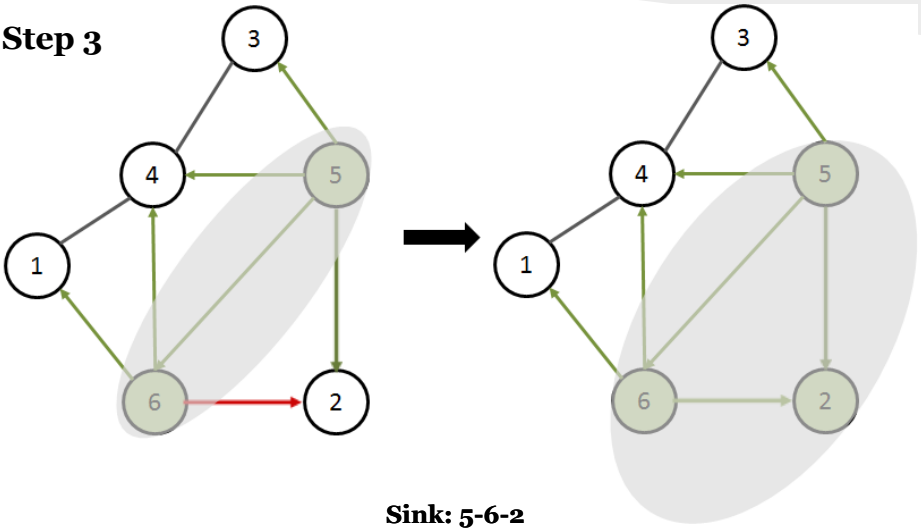
(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Sink: 5-6

Since join of 5 & 6 has lowest selectivity, so we choose 6. Now the sink is 5-6. Again we mark possible edges to consider with outgoing edges from the sink.

Execution Plan Generation(contd.)

Step 3



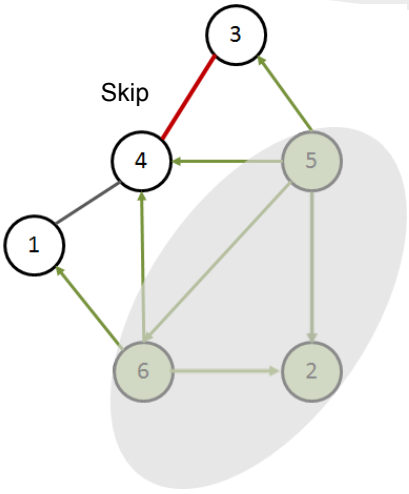
Edges is ascending order of selectivity

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

Sink: 5-6-2

Execution Plan Generation(contd.)

Step 4



Sink: 5-6-2

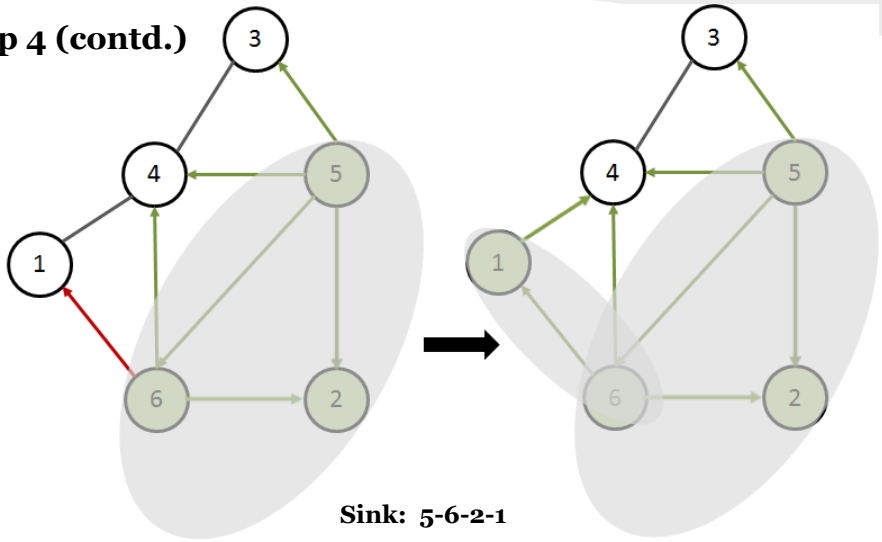
Edges is ascending order of selectivity

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

The next minimum selectivity edge is (3,4). But it is not within set of possible edges to consider. So we skip this edge. By doing that, we avoid cartesian product.

Execution Plan Generation(contd.)

Step 4 (contd.)



Edges is ascending order of selectivity

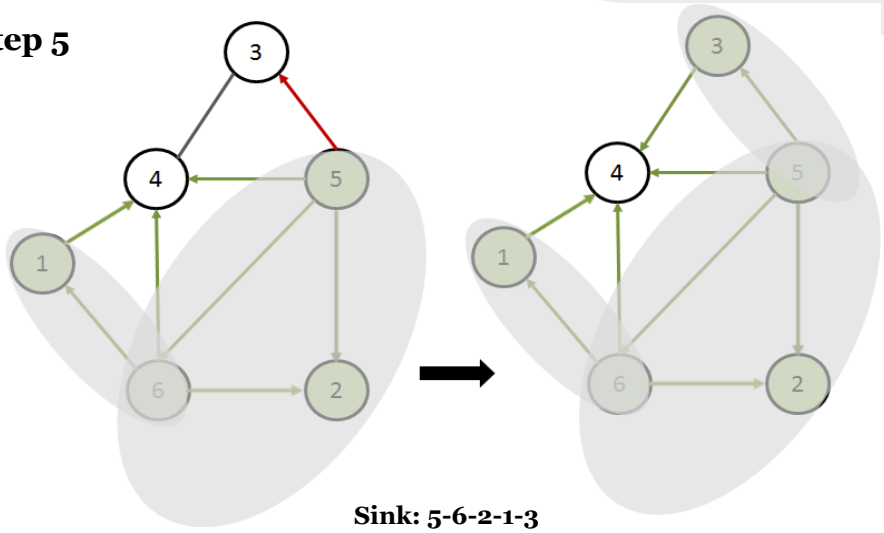
(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

Sink: 5-6-2-1

The minimum selectivity edge is (1,6) and it is in possible edges to consider. So merge node 1 to the sink.

Execution Plan Generation(contd.)

Step 5



Edges is ascending order of selectivity

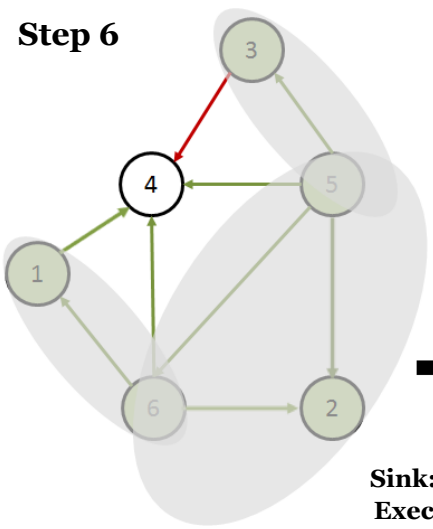
(6,5)
(6,2)
(3,4)
(1,6)
(3,5)
(4,5)
(4,6)
(5,2)
(1,4)

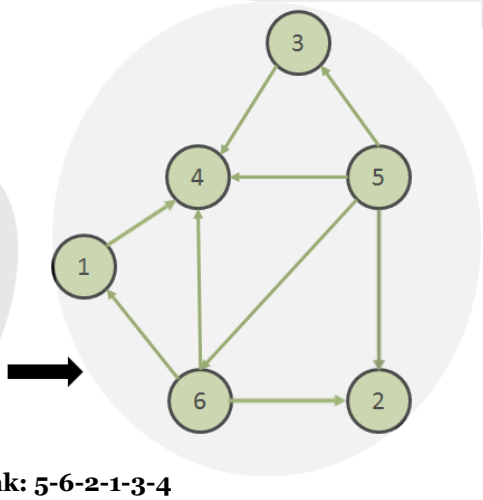
Sink: 5-6-2-1-3

The next minimum selectivity edge is (3,5). This is in possible next joins to consider. So we agg node 3 to the sink.

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

Now among the outgoing edges from sink, edge (3,4) is minimum selectivity edge, which we skipped earlier. So we add node 4 now. Now we have sink with all nodes. So we have got our execution plan.

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

Here's the algorithm that uses node and edge selectivity information to generate an execution plan, EP.
At first the edge with minimum edge selectivity is selected. Nodes of this edge are added to EP by ascending order of their selectivities..
Next edge (x, y) is selected based on 2 criteria: minimum edge selectivity and the at least one visited node.

What about disconnected graph?

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

In the example, G is connected. But G is not necessarily connected. It may have two or more components. Then the execution plan is the the unordered set of execution plans of the components. The order of processing nodes within a component matters, but the order of processing components doesn't matter.
So like System-R algorithm we need to take the cross product of the 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?