SPARQL query performance

Pavel Klinov
@klinovp

STARDOG

Query optimiser vs query optimisation

this is not a talk about the internals of a SPARQL query optimiser

- it's not a database conference
- most of you are not SPARQL engine developers (iiuc)

but query optimisation is a broad term:

- optimisation inside the engine
- changing query's semantics for performance reasons
- performance debugging
- making query more amenable to optimisation

we will mostly talk about queries (and query plans), not about the optimiser's internals

Query optimisation vs data optimisation

sometimes query optimisation is just not the right subject!

we should talk (more) about optimising data for query performance

- using the right datatypes
- eliminating unnecessary joins or traversals
- precomputing facts
- partial pre-aggregation

cf. Julian Hyde (Apache Calcite): Don’t optimize my queries, optimize my data!

SPARQL: how does query evaluation work?

SPARQL specification defines what the answers should be

using the so-called evaluation semantics

```
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type :Article ;
           dc:creator ?person .
  ?person foaf:name ?name
  FILTER (contains(name, “Mary”))
}
```

SPARQL evaluation semantics: bottom-up

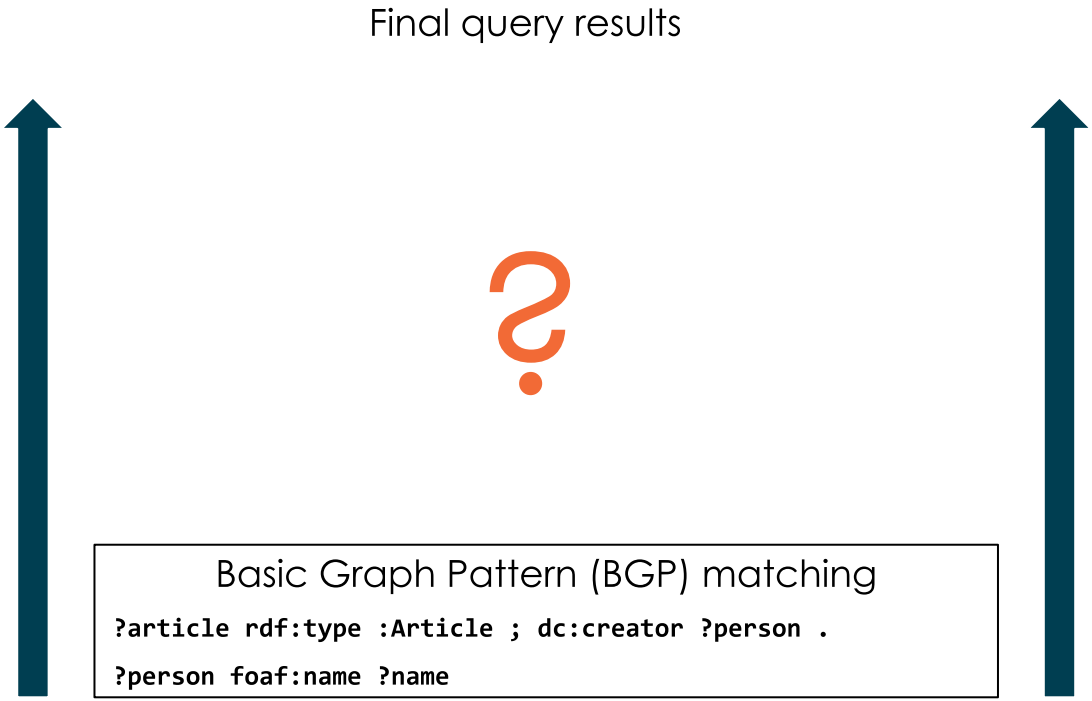
Final query results

```
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type :Article ;
           dc:creator ?person .
  ?person foaf:name ?name
  FILTER (contains(name, “Mary”))
}
```

Basic Graph Pattern (BGP) matching

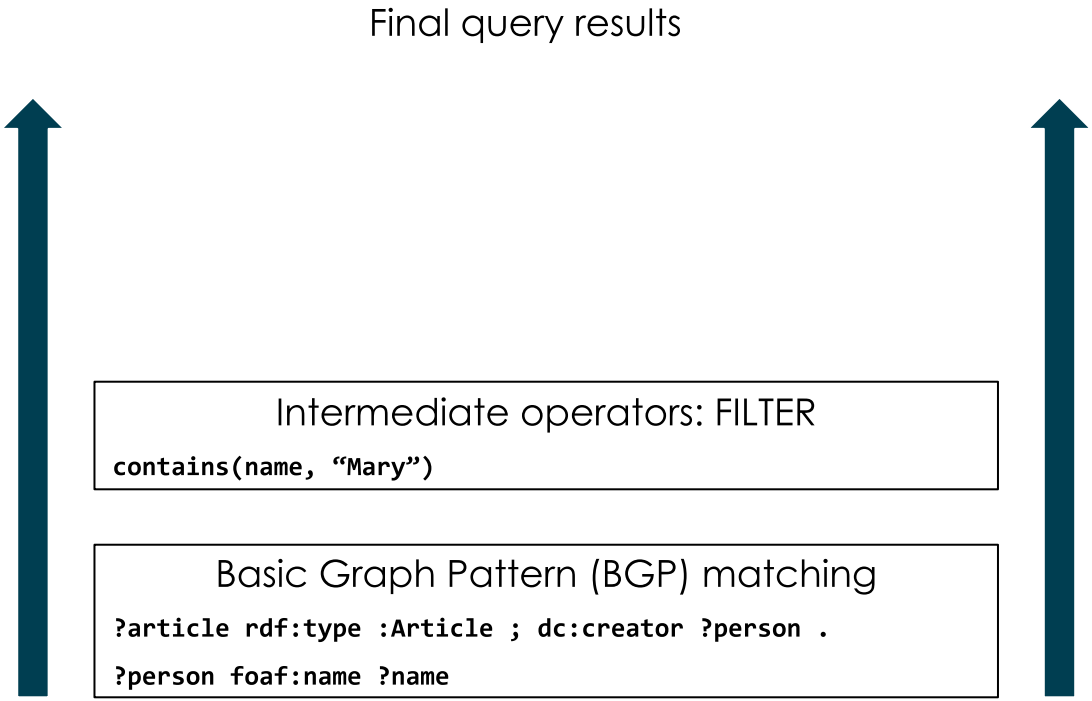
```
?article rdf:type :Article ; dc:creator ?person .
?person foaf:name ?name
```

SPARQL evaluation semantics: bottom-up



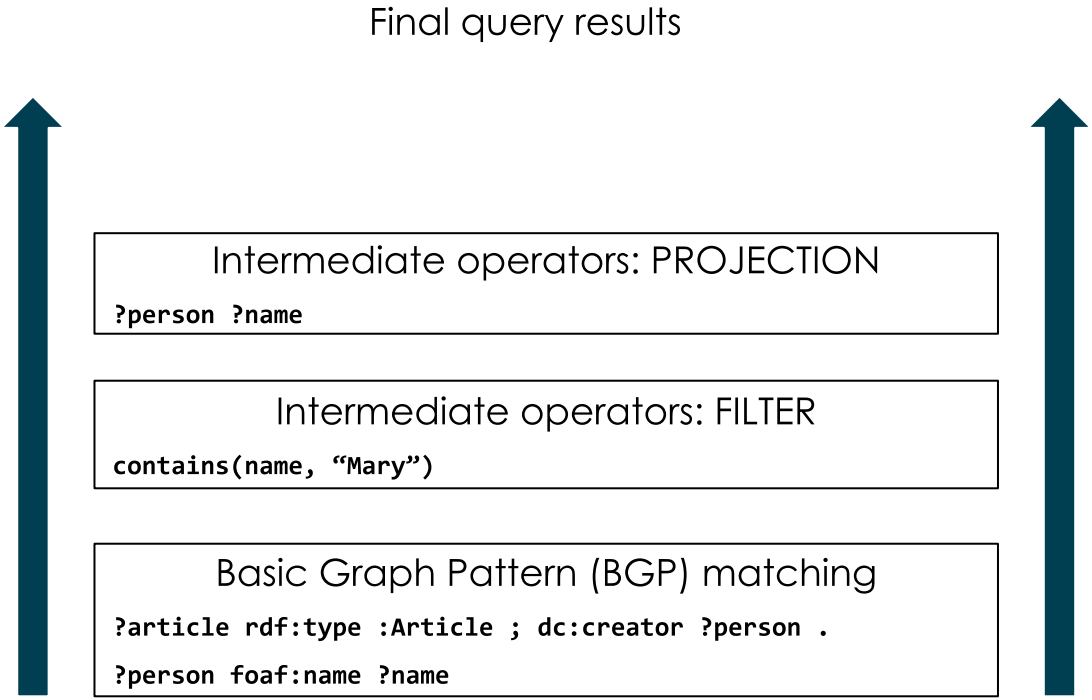

```
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type :Article ;
           dc:creator ?person .
  ?person foaf:name ?name
  FILTER (contains(name, “Mary”))
}
```

SPARQL evaluation semantics: bottom-up



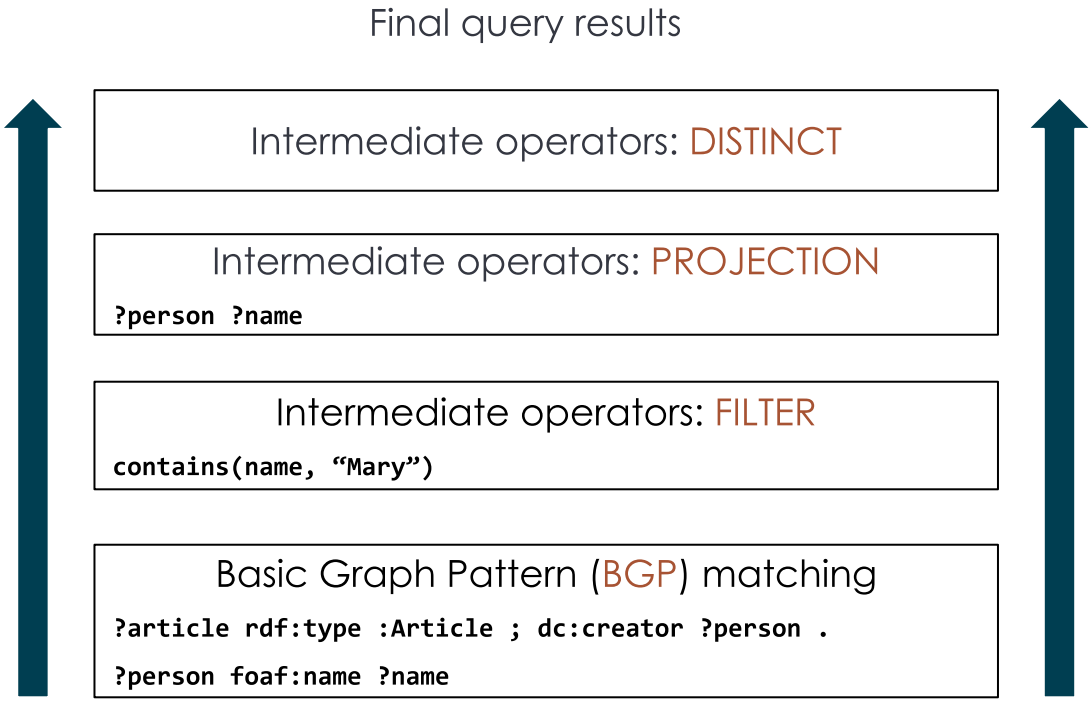

```
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type :Article ;
           dc:creator ?person .
  ?person foaf:name ?name
  FILTER (contains(name, “Mary”))
}
```

SPARQL evaluation semantics: bottom-up




```
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type :Article ;
           dc:creator ?person .
  ?person foaf:name ?name
  FILTER (contains(name, “Mary”))
}
```

SPARQL evaluation semantics: bottom-up




```
SELECT DISTINCT ?person ?name
WHERE {
  ?article rdf:type :Article ;
           dc:creator ?person .
  ?person foaf:name ?name
  FILTER (contains(name, “Mary”))
}
```

Wrong queries can be very expensive

Give me editors of all journals and all proceedings volumes

```
SELECT ?journal_editor ?inproc_editor
WHERE {
  ?journal rdf:type :Journal ;
           :editor  ?journal_editor .
  ?inProc  rdf:type :InProceedings ;
           :editor  ?inproc_editor
}
```

Wrong queries can be very expensive

Give me editors of all journals and all proceedings volumes

```
SELECT ?journal_editor ?inproc_editor
WHERE {
  ?journal rdf:type :Journal ;
           :editor  ?journal_editor .
  ?inProc  rdf:type :InProceedings ;
           :editor  ?inproc_editor
}
```

disconnected BGP

this computes all pairs of journal and volume editors!

full Cartesian Product

The right query: fast and correct

Give me editors of all journals and all proceedings volumes

```
SELECT ?journal_editor ?inproc_editor
WHERE {
  { ?journal rdf:type :Journal ;
             :editor  ?journal_editor . }
  UNION {
    ?inProc  rdf:type :InProceedings ;
             :editor  ?inproc_editor }
}
```

Wrong queries can be very expensive (2)

updating objects represented as nodes with outgoing edges

```
DELETE { ?doc dc:creator ?author . ?doc :booktitle ?book … }
INSERT { ... }
WHERE {
  ?doc dc:creator ?author .
  ?doc :booktitle ?book .
  ?doc rdfs:seeAlso ?seeAlso .
  ...
}
```

Wrong queries can be very expensive (2)

doc with 10 authors which appeared in 100 books → results explode!

```
DELETE { ?doc dc:creator ?author . ?doc :booktitle ?book … }
INSERT { ... }
WHERE {
  ?doc dc:creator ?author .
  ?doc :booktitle ?book .
  ?doc rdfs:seeAlso ?seeAlso .
  ...
}
```

A better DELETE query

matches only asserted edges, not combinations ⇒ less memory, faster

```
DELETE { ?doc ?p ?o }
INSERT { ... }
WHERE {
  ?doc a :Document .
  ?doc ?p ?o .
}
```

Tip: always run CONSTRUCT versions of UPDATE queries first

Joins

two SPARQL patterns in the same group are joined

for SQL users: all inner joins are natural equi-joins

easiest example: VALUES (easy to verify on empty dataset)

```
VALUES (?x ?y) { (:a :b) (:c :d) }
VALUES (?y ?z) { (:b :1) (:b :2) (:d :3) }
```

Joins

two SPARQL patterns in the same group are joined

for SQL users: all inner joins are natural equi-joins

easiest example: VALUES

```
VALUES (?x ?y) { (:a :b) (:c :d) }
VALUES (?y ?z) { (:b :1) (:b :2) (:d :3) }

Result: (?x->:a, ?y->:b, ?z->1)
```

Joins

two SPARQL patterns in the same group are joined

for SQL users: all inner joins are natural equi-joins

easiest example: VALUES

```
VALUES (?x ?y) { (:a :b) (:c :d) }
VALUES (?y ?z) { (:b :1) (:b :2) (:d :3) }

Result: (?x->:a, ?y->:b, ?z->1)
        (?x->:a, ?y->:b, ?z->2)
```

Joins

two SPARQL patterns in the same group are joined

for SQL users: all inner joins are natural equi-joins

easiest example: VALUES

```
VALUES (?x ?y) { (:a :b) (:c :d) }
VALUES (?y ?z) { (:b :1) (:b :2) (:d :3) }

Result: (?x->:a, ?y->:b, ?z->1)
        (?x->:a, ?y->:b, ?z->2)
        (?x->:c, ?y->:d, ?z->3)
```

Joins

two SPARQL patterns in the same group are joined

for SQL users: all inner joins are natural equi-joins

easiest example: VALUES

```
VALUES (?x ?y) { (:a :b) (:c :d) }
VALUES (?y ?z) { (:b :1) (:b :2) (:d :3) }

Result: (?x->:a, ?y->:b, ?z->1)
        (?x->:a, ?y->:b, ?z->2)
        (?x->:c, ?y->:d, ?z->3)
```

these tuples are called solutions

SPARQL query execution is basically processing bags of solutions (filters, joins, etc.)

Unbound join keys

what happens when a join key does not have a value?

(sorry but we must talk about nulls).

how many results will the following generate?

```
VALUES (?x ?y) { (:a :b) (:c UNDEF) }
VALUES (?y ?z) { (:b :1) (:b :2) (:d :3) }
```

Unbound join keys

what happens when a join key does not have a value?

(sorry but we must talk about nulls).

join condition is always satisfied when join variable is unbound (on either end)

```
VALUES (?x ?y) { (:a :b) (:c UNDEF) }
VALUES (?y ?z) { (:b :1) (:b :2) (:d :3) }
Result: (?x->:a, ?y->:b, ?z->1)
        (?x->:a, ?y->:b, ?z->2)
        (?x->:c, ?y->:d, ?z->3),
        (?x->:c, ?y->:b, ?z->1), (?x->:c, ?y->:b, ?z->2)
```

Unbound join keys

what happens when a join key does not have a value?

(sorry but we must talk about nulls).

join condition is always satisfied when join variable is unbound (on either end)

```
VALUES (?x ?y) { (:a :b) (:c UNDEF) }
VALUES (?y ?z) { (:b :1) (:b :2) (:d :3) }
Result: (?x->:a, ?y->:b, ?z->1)
        (?x->:a, ?y->:b, ?z->2)
        (?x->:c, ?y->:d, ?z->3),
        (?x->:c, ?y->:b, ?z->1), (?x->:c, ?y->:b, ?z->2)
```

OPTIONAL joins and unbound join keys

OPTIONALs are similar to left outer joins in SQL: deal with missing data

```
{
  ?person :livesIn ?city .
  ?city :locatedIn ?country .
  ?country :name ?name
}
```

both `:locatedIn` and `:name` triples can be missing

so let's use OPTIONAL?

OPTIONAL joins and unbound join keys

```
{
  ?person :livesIn ?city .
  OPTIONAL { ?city :locatedIn ?country }
  OPTIONAL { ?country :name ?name }
}
```

now we have a problem!

OPTIONAL joins and unbound join keys

```
{
  ?person :livesIn ?city .
  OPTIONAL { ?city :locatedIn ?country }
  OPTIONAL { ?country :name ?name }
}
```

now we have a problem! let's look at the SPARQL algebra (http://sparql.org/)

```
LeftJoin(?country)
  LeftJoin(?city)
    BGP(?person :livesIn ?city)
    BGP(?city :locatedIn ?country)
  BGP(?country :name ?name)
```

OPTIONAL joins and unbound join keys

```
{
  ?person :livesIn ?city .
  OPTIONAL { ?city :locatedIn ?country }
  OPTIONAL { ?country :name ?name }
}
```

now we have a problem! let's look at the SPARQL algebra

```
LeftJoin(?country)
  LeftJoin(?city)
    BGP(?person :livesIn ?city)
    BGP(?city :locatedIn ?country)
  BGP(?country :name ?name)
```

bottom-up: first join on `?city` (no nulls), second join on `?country` (nullable)

OPTIONAL joins and unbound join keys

```
{
  ?person :livesIn ?city .
  OPTIONAL { ?city :locatedIn ?country }
  OPTIONAL { ?country :name ?name }
}
```

for every person living in a city without a `:locatedIn` triple the query will return all countries.

even if data is perfect, dealing with nulls slows down joins by a lot
(e.g. standard hash joins won't work)

it often shows in the query plan

OPTIONAL joins and unbound join keys

```
{
  ?person :livesIn ?city .
  OPTIONAL { ?city :locatedIn ?country .
             OPTIONAL { ?country :name ?name }
  }
}
```

this avoids the correctness issue

still could be a performance issue but this query is optimisable

Joins on variable introduced in BIND

SPARQL allows for adding new variables to solutions

eg. `BIND(?x + ?y as ?z)`

these new variables can be join keys

SPARQL expressions can raise (type) errors which:

- make the target variable unbound
- typically not visible to the client

this may affect performance even when errors don’t happen

Joins on variable introduced in BIND

```
{
  ?company :employs ?person
  BIND(iri(concat(“urn:employee:”, strafter(?person, “:”))) as ?emp_iri)
  ?emp_iri a :Employee
}
```

this is a kind of data integration query

linking company employee data to instances of :Employee

Joins on variable introduced in BIND

```
{
  ?company :employs ?person
  BIND(iri(concat(“urn:employee:”, strafter(?person, “:”))) as ?emp_iri)
  ?emp_iri a :Employee
}
```

if ?person is not a string literal, ?emp_id won’t be bound

the query will return all employees for that person

this might be a good time to fix the data

(or need a more complex query with type checks or `bound(?emp_iri)` filters)

Joins vs equality filters

```
{
  ?person :livesIn ?city .
  ?city :locatedIn ?country
}
```

vs

```
{
  ?person :livesIn ?personCity .
  ?countryCity :locatedIn ?country
  FILTER (?personCity = ?countryCity)
}
```

these are often considered two ways of achieving the same thing.
that's true here but there's a catch...

Joins vs equality filters

```
?person :livesIn ?personCity .
?countryCity :locatedIn ?country
FILTER (?personCity = ?countryCity)
```

issue #1: the optimiser must figure out `?personCity` and `?countryCity` will bind to the same value in every solution of the BGP which passes the filter

that's not always trivial because:

- complex filter expressions
- nested graph patterns
- FILTERs in OPTIONALs have special semantics in SPARQL

```
?person :livesIn ?personCity .
OPTIONAL {
     ?countryCity :locatedIn ?country
     FILTER (?personCity = ?countryCity) }
```

Joins vs equality filters

```
?owner :owns ?company .
?employee :worksAt ?employer
FILTER (?company = ?employer)
```

issue #2: the optimiser cannot convert this to a join because

- `?company` or `?employer` may bind to different RDF literals
- which are still equal
- like 1.0 vs 1.00

unless your SPARQL engine knows that it's impossible for other reasons (SHACL? hints?)

Joins vs equality filters

```
?owner :owns ?company .
?employee :worksAt ?employer
FILTER (?company = ?employer)
```

issue #2: the optimiser cannot convert this to a join because

- `?company` or `?employer` may bind to different RDF literals
- which are still equal
- like 1.0 vs 1.00

unless your SPARQL engine knows that it's impossible for other reasons (SHACL? hints?)

advice: rename variables in your queries

Order of joins

SPARQL engines are pretty good at reordering inner joins

```
?person :livesIn ?city .
?city :locatedIn ?country .
?country :name ?name
```

can be evaluated in various ways:

- Join(:livesIn, Join(:locatedIn, :name))
- Join(Join(:livesIn, :locatedIn), :name), etc.

engines can fail to pick the optimal order but they will try

(inner join ordering is a classical query optimisation problem in databases)

OPTIONALs often not reordered

reordering OPTIONALs is hugely more difficult and error prone ⇒ often not done

```
?person :livesIn ?city .
OPTIONAL { ?city :locatedIn ?country }
?country :name ?name
```

often means it's executed as-is: Join(OuterJoin(:livesIn, :locatedIn), :name)

which may or may not be optimal

why is that?

Equivalent orders: inner joins vs OPTIONALs

join order optimisation (JOO) is a search problem where

- the search space is all equivalent join orders
- the goal function is based on cost

search space is easy for inner joins since all permutations are valid

Join(A, B) = Join(B, A), Join(A, Join(B, C)) = Join(Join(A, B), C)

most of that fails for outer joins in general

there's no straightforward procedure to enumerate all combinations

(the search space becomes hard to define)

OPTIONALs are hard

```
A OPTIONAL { B } ≠ B OPTIONAL { A }
```

OPTIONALs are hard

```
A OPTIONAL { B } ≠ B OPTIONAL { A }
```

how about

`A { B OPTIONAL { C } }` (or equivalently `{ B OPTIONAL { C } } A`)

vs

```
A
B
OPTIONAL { C }
```

can the optimiser freely move OPTIONAL patterns up and down?

Nope, not freely!

```
select ?x ?y {
 values (?x) { (:a) (:d) }
 optional { values (?x ?y) { (:a :c) } }
 values ?y { :e }
}
```

Results: (?x -> :d, ?y -> :e)

Nope, not freely!

```
select ?x ?y {
 values (?x) { (:a) (:d) }
 optional { values (?x ?y) { (:a :c) } }
 values ?y { :e }
}
```

Results: (?x -> :d, ?y -> :e)

```
select ?x ?y {      # now the inner join is evaluated first!
 values (?x) { (:a) (:d) }
 values ?y { :e }
 optional { values (?x ?y) { (:a :c) } }
}
```

Results: (?x -> :d, ?y -> :e), (?x -> :a, ?y -> :e)

OPTIONALs summary

some optimisations are possible e.g. Stardog will push selective patterns into OPTIONALs when it can detect that it won't change semantics

don't rely on it, place your OPTIONALs wisely

in many cases they should be pushed to the bottom (hint: OPTIONALs never decrease the number of results)

SQL engines often can rewrite outer joins into inner joins… not SPARQL engines (afaik)

this needs more work on bringing theory to practice, eg.

Rao et al. "Canonical Abstraction for Outerjoin Optimization" (for SQL)

SERVICE aka SPARQL Federation

SERVICE is just another kind of graph pattern, same evaluation semantics (bottom-up)

```
?person :worksAt :Stardog
SERVICE <https://query.wikidata.org/sparql> {
  ?person wdt:P31 wd:Q5;  # Any instance of a human.
          wdt:P19 wd:Q60  # Who was born in New York City.
}
```

SERVICE results are joined with the rest of the query (un-correlated!)

challenges

- no selectivity statistics (in general)
- unreliable endpoints
- data transmission and ingestion costs

SERVICE aka SPARQL Federation

most optimisers will try to constrain SERVICE invocation by local bindings

```
?person :worksAt :Stardog
SERVICE <https://query.wikidata.org/sparql> {
  SELECT * { ?person wdt:P31 wd:Q5;   # Any instance of a human
                     wdt:P19 wd:Q60 } # Who was born in New York City
  VALUES ?person { :mike :kendall :evren :pavel }
}
```

optimisers can fail at that for various reasons (pick wrong local pattern, etc.)

- check the plan
- place local patterns binding `?person` in the same scope as SERVICE

endpoints may throttle rapid requests (LIMITs on joined patterns could help)

Query plans

Understanding query performance

so your query is correct but slow, what's next?

Understanding query performance

so your query is correct but slow, what's next?

don't try to guess what happens based on intuition: learn to read the query plan

typical intuition failure:

"I have a query X which runs OK but then I added `?x rdfs:label ?y` *(somewhere) and it's slow. It should be just one extra join, right? Why is it slow?"*

Understanding query performance

so your query is correct but slow, what's next?

don't try to guess what happens based on intuition: learn to read the query plan

typical intuition failure:

"I have a query X which runs OK but then I added `?x rdfs:label ?y` *(somewhere) and it's slow. It should be just one extra join, right? Why is it slow?"*

local changes in the query do not mean local changes in the evaluation plan

it could be completely different!

Query plans in Stardog

Stardog implements the Volcano model where each algebraic expression corresponds to some executable operators (cf. Graefe work on Cascades framework)

- triple patterns → index scans
- BGPs → joins over scans
- joins → merge, hash, loop (etc.) join algorithms

very extensible

plans are easy to read

information (SPARQL solutions) flows bottom-up

Plan for the Cartesian product query

```
SELECT ?journal_editor ?inproc_editor
WHERE {
  ?journal rdf:type :Journal ;
           :editor  ?journal_editor .
  ?inProc  rdf:type :InProceedings ;
           :editor  ?inproc_editor
}
```

Plan for the Cartesian product query

```
Scan[POSC](?inProc, rdf:type, :InProceedings)
Scan[PSOC](?inProc, :editor, ?inproc_editor)

Scan[POSC](?journal, rdf:type, :Journal)
Scan[PSOC](?journal, :editor, ?journal_editor)
```

```
SELECT ?journal_editor ?inproc_editor
WHERE {
  ?journal rdf:type :Journal ;
           :editor  ?journal_editor .
  ?inProc  rdf:type :InProceedings ;
           :editor  ?inproc_editor
}
```

Plan for the Cartesian product query

```
MergeJoin(?inProc)
+— Scan[POSC](?inProc, rdf:type, :InProceedings)
`— Scan[PSOC](?inProc, :editor, ?inproc_editor)
MergeJoin(?journal)
+— Scan[POSC](?journal, rdf:type, :Journal)
`— Scan[PSOC](?journal, :editor, ?journal_editor)
```

```
SELECT ?journal_editor ?inproc_editor
WHERE {
  ?journal rdf:type :Journal ;
           :editor  ?journal_editor .
  ?inProc  rdf:type :InProceedings ;
           :editor  ?inproc_editor
}
```

Plan for the Cartesian product query

```
NestedLoopJoin(_)
+— MergeJoin(?inProc)
|   +— Scan[POSC](?inProc, rdf:type, :InProceedings)
|   `— Scan[PSOC](?inProc, :editor, ?inproc_editor)
`— MergeJoin(?journal)
    +— Scan[POSC](?journal, rdf:type, :Journal)
    `— Scan[PSOC](?journal, :editor, ?journal_editor)
```

```
SELECT ?journal_editor ?inproc_editor
WHERE {
  ?journal rdf:type :Journal ;
           :editor  ?journal_editor .
  ?inProc  rdf:type :InProceedings ;
           :editor  ?inproc_editor
}
```

Plan for the Cartesian product query

```
Projection(?journal_editor, ?inproc_editor)
`— NestedLoopJoin(_)
   +— MergeJoin(?inProc)
   |  +— Scan[POSC](?inProc, rdf:type, :InProceedings)
   |  `— Scan[PSOC](?inProc, :editor, ?inproc_editor)
   `— MergeJoin(?journal)
      +— Scan[POSC](?journal, rdf:type, :Journal)
      `— Scan[PSOC](?journal, :editor, ?journal_editor)
```

```
SELECT ?journal_editor ?inproc_editor
WHERE {
  ?journal rdf:type :Journal ;
           :editor  ?journal_editor .
  ?inProc  rdf:type :InProceedings ;
           :editor  ?inproc_editor
}
```

Plan for the Cartesian product query

```
Projection(?journal_editor, ?inproc_editor)
`— NestedLoopJoin(_)  ← Cartesian product here!
   +— MergeJoin(?inProc)
   |  +— Scan[POSC](?inProc, rdf:type, :InProceedings)
   |  `— Scan[PSOC](?inProc, :editor, ?inproc_editor)
   `— MergeJoin(?journal)
      +— Scan[POSC](?journal, rdf:type, :Journal)
      `— Scan[PSOC](?journal, :editor, ?journal_editor)
```

```
SELECT ?journal_editor ?inproc_editor
WHERE {
  ?journal rdf:type :Journal ;
           :editor  ?journal_editor .
  ?inProc  rdf:type :InProceedings ;
           :editor  ?inproc_editor
}
```

Query pipeline

most efficient query execution is streaming:

- index scans match some data, generate partial results
- they are immediately processed further (joined, filtered)
- shipped to the client

key: first results are processed before all results are generated

this is called the query execution pipeline

lazy, low-latency, min resource consumption

The Merge Join: streaming join algorithm

both inputs are sorted by the shared variable, the join key

```
?author  :published ?journal . ?journal :editor ?editor
```

:published

?author	?journal
Alice	:JoAIR
Bob	:JoCryp
...	
:Jim	:JoMLR

:editor

?journal	?editor
:JoAI	:Eve
:JoAIR	:Mary
:JoMLR	:Mark
:IEEECo	:Bryan

:published $\bowtie_{?journal}$:editor

?author	?journal	?editor

The Merge Join: streaming join algorithm

both inputs are sorted by the shared variable, the join key

```
?author  :published ?journal . ?journal :editor ?editor
```

:published

?author	?journal
Alice	:JoAIR
Bob	:JoCryp
...	
:Jim	:JoMLR

:editor

?journal	?editor
:JoAI	:Eve
:JoAIR	:Mary
:JoMLR	:Mark
:IEEECo	:Bryan

:published $\bowtie_{?journal}$:editor

?author	?journal	?editor
Alice	:JoAIR	:Mary
...		

The Merge Join: streaming join algorithm

both inputs are sorted by the shared variable, the join key

```
?author  :published ?journal . ?journal :editor ?editor
```

:published

?author	?journal
Alice	:JoAIR
Bob	:JoCryp
...	
:Jim	:JoMLR

:editor

?journal	?editor
:JoAI	:Eve
:JoAIR	:Mary
:JoMLR	:Mark
:IEEECo	:Bryan

:published $\bowtie_{?journal}$:editor

?author	?journal	?editor
Alice	:JoAIR	:Mary
Jim	:JoMLR	:Mark
...		

The Merge Join: streaming join algorithm

?author	?journal
Alice	:JoAIR
Bob	:JoCryp
…	
:Jim	:JoMLR

?journal	?editor
:JoAI	:Eve
:JoAIR	:Mary
:JoMLR	:Mark
:IEEECo	:Bryan

?author	?journal	?editor
Alice	:JoAIR	:Mary
Jim	:JoMLR	:Mark
…		

- results streaming out as inputs are coming in
- no memory pressure
- low disk IO overhead

When the pipeline breaks

not all SPARQL query processing can be done in streaming fashion

pipeline breaking: accumulating results for processing before sending them along

examples:

- hash joins: need to build the hashtable
- sort, order by
- aggregation: count, min/max, sum, avg, distinct

increases latency, memory pressure on the server

The Hash Join: pipeline breaker

There's a shared variable but no sortedness assumption

```
?author  :published ?journal . ?journal :editor ?editor
```

`:published`

?author	?journal
Alice	:JoAIR
Bob	:JoCryp
...	
:Jim	:JoMLR

`:editor`

?journal	?editor
:JoMLR	:Mark
:IEEECo	:Bryan
:JoAI	:Eve
:JoAIR	:Mary

The Hash Join: pipeline breaker

There's a shared variable but no sortedness assumption

```
?author  :published ?journal . ?journal :editor    ?editor
```

:published

?author	?journal
Alice	:JoAIR
Bob	:JoCryp
...	
:Jim	:JoMLR

hashtable (RAM/disk)

#journal	?journal	?editor
4657	:JoMLR	:Mark
3647	:IEEECo	:Bryan
3435	:JoAI	:Eve
9768	:JoAIR	:Mary

:editor

?journal	?editor
:JoMLR	:Mark
:IEEECo	:Bryan
:JoAI	:Eve
:JoAIR	:Mary

The Hash Join: pipeline breaker

There's a shared variable but no sortedness assumption

```
?author  :published ?journal . ?journal :editor    ?editor
```

:published

?author	?journal
Alice	:JoAIR
Bob	:JoCryp
...	
:Jim	:JoMLR

hashtable (RAM/disk)

#journal	?journal	?editor
4657	:JoMLR	:Mark
3647	:IEEECo	:Bryan
3435	:JoAI	:Eve
9768	:JoAIR	:Mary

:editor

?journal	?editor
:JoMLR	:Mark
:IEEECo	:Bryan
:JoAI	:Eve
:JoAIR	:Mary

The Hash Join: pipeline breaker

There's a shared variable but no sortedness assumption

```
?author  :published ?journal . ?journal :editor    ?editor
```

:published

?author	?journal
Alice	:JoAIR
Bob	:JoCryp
...	
:Jim	:JoMLR

hashtable (RAM/disk)

#journal	?journal	?editor
4657	:JoMLR	:Mark
3647	:IEEECo	:Bryan
3435	:JoAI	:Eve
9768	:JoAIR	:Mary

:editor

?journal	?editor
:JoMLR	:Mark
:IEEECo	:Bryan
:JoAI	:Eve
:JoAIR	:Mary

The Hash Join: pipeline breaker

performance-related issues:

- latency: hashtable is built before the 1st result is produced
- memory pressure, possible spilling to disk
- high disk IO (one relation is fully hashed, other fully scrolled)
- random memory access

these are typical for other pipeline breakers as well

- sort operators
- hash MINUS (anti-joins)
- GROUP BY, DISTINCT

a lot of performance analysis comes down to finding pipeline breakers in the plan or generally bad join orders (expensive joins before cheap joins)

How to “fix” a slow query

this is where knowing engine-specific tools really helps

mostly query hints: they tell the optimiser what to do (it may still ignore though)

but some general tricks also work

- subqueries
  - for defining join order
  - pushing `DISTINCT` down the plan
- move selective patterns around

Example: Erdoes coauthors

```
SELECT DISTINCT ?name
WHERE {
  ?erdoes foaf:name "Paul Erdoes"
  {
    ?document dc:creator ?erdoes, ?author .
    ?author foaf:name ?name
    FILTER (?author != ?erdoes)
  } UNION {
    ?document dc:creator ?erdoes, ?author .
    ?document2 dc:creator ?author, ?author2 .
    ?author2 foaf:name ?name
    FILTER (?author!=?erdoes &&
            ?document2!=?document &&
            ?author2!=?erdoes &&
            ?author2!=?author)
  }
}
```

find all 1- and 2-degree co-authors of Erdoes
trivial to do imperatively:

- iterate over his papers
- get other authors
- look at those authors other papers

but this is not how it's defined according to this query's SPARQL algebra

Query's algebra

```
SELECT DISTINCT ?name
WHERE {
  ?erdoes foaf:name "Paul Erdoes"
  {
    ?document dc:creator ?erdoes, ?author .
    ?author foaf:name ?name
    FILTER (?author != ?erdoes)
  } UNION {
    ?document dc:creator ?erdoes, ?author .
    ?document2 dc:creator ?author, ?author2 .
    ?author2 foaf:name ?name
    FILTER (?author!=?erdoes &&
            ?document2!=?document &&
            ?author2!=?erdoes &&
            ?author2!=?author)
  }
}
```

```
(join
  (bgp (triple ?erdoes foaf:name "Paul Erdoes"))
  (union
    (filter (!= ?author ?erdoes)
      (bgp
        (triple ?document dc:creator ?erdoes)
        (triple ?document dc:creator ?author)
        (triple ?author foaf:name ?name)
      ))
    (filter (&& (...))
      (bgp
        (triple ?document dc:creator ?erdoes)
        (triple ?document dc:creator ?author)
        (triple ?document2 dc:creator ?author)
        (triple ?document2 dc:creator ?author2)
        (triple ?author2 foaf:name ?name)
      ))))))))
```

Part of query plan

Stardog optimiser pushes the selective Erdoes pattern into the union (also splits & pushes filters)

```
`— Union [#1.6K]
   +— MergeJoin(?author) [#570]
   |  … 1-degree co-authors here …
   `— MergeJoin(?author2) [#1.0K]
      +— Scan[PSOC](?author2, foaf:name, ?name) [#433K]
      `— Sort(?author2) [#1.0K]
         `— Filter((?author2 != ?erdoes && ?author2 != ?author)) [#1.0K]
            `— MergeJoin(?document2) [#2.0K]
               +— Scan[PSOC](?document2, dc:creator, ?author2) [#898K]
               `— Sort(?document2) [#1.1K]
                  `— Filter(?document2 != ?document) [#1.1K]
                     `— MergeJoin(?author) [#2.1K]
                        +— Scan[POSC](?document2, dc:creator, ?author) [#898K]
                        `— Sort(?author) [#570]
                           `— Filter(?author != ?erdoes) [#570]
                              `— MergeJoin(?document) [#1.1K]
                                 +— Scan[PSOC](?document, dc:creator, ?author) [#898K]
                                 `— Sort(?document) [#591]
                                    `— MergeJoin(?erdoes) [#591]
                                       +— Scan[POSC](?erdoes, foaf:name, "Paul Erdoes") [#1]
                                       `— Scan[POSC](?document, dc:creator, ?erdoes) [#898K]
```

And if the plan is bad... rewrite manually

```
  {
    ?document dc:creator ?erdoes .
    ?erdoes foaf:name "Paul Erdoes" .
    ?document dc:creator ?author .
    ?author foaf:name ?name
    FILTER (?author != ?erdoes)
  } UNION {
    ?document dc:creator ?erdoes .
    ?erdoes foaf:name "Paul Erdoes" .
    ?document dc:creator ?author .
    ?document2 dc:creator ?author, ?author2 .
    ?author2 foaf:name ?name
    FILTER (?author!=?erdoes &&
            ?document2!=?document &&
            ?author2!=?erdoes &&
            ?author2!=?author)
}
```

If joins are still bad... subqueries!

```
  {
    { select * { ?document dc:creator ?erdoes .
                 ?erdoes foaf:name "Paul Erdoes" }
    ?document dc:creator ?author .
    ?author foaf:name ?name
    FILTER (?author != ?erdoes)
  } UNION {
    { select * { ?document dc:creator ?erdoes .
                 ?erdoes foaf:name "Paul Erdoes" }
    ?document dc:creator ?author .
    ?document2 dc:creator ?author, ?author2 .
    ?author2 foaf:name ?name
    FILTER (?author!=?erdoes &&
            ?document2!=?document &&
            ?author2!=?erdoes &&
            ?author2!=?author)
}
```

Other general tips

project only necessary variables (esp. from subqueries)

avoid ORDER BY in sub-queries (unless with LIMIT)

drop unnecessary DISTINCT (e.g in queries with GROUP BY)

be very careful with property paths with *

- match zero-length paths, `?c rdfs:subClassOf* ?sc`
- typically only useful in a sequence /, `?x rdf:type/rdfs:subClassOf* ?sc`
- often can be replaced with +

full-text search is often faster than FILTERs with regex

Takeaways

in the ideal world the engine always picks the best plan

- we do not live in the ideal world
- RDF's "flexible schema" is a double-edged sword
- join order optimisation alone is NP-hard
- optimisation algorithms operate under uncertainty
  - cost model gets poor inputs
  - *"every hard query optimisation problem is down to poor selectivity estimations"*

some query plans will be sub-optimal (particularly, the join tree)

- vendors daily work is to improve this
- you sometimes need to nudge the optimiser in the right direction

General advice

every decision to rewrite the query for performance should be based on evidence

- query plan
- profiler, etc.

General advice

every decision to rewrite the query for performance should be based on evidence

- query plan
- profiler, etc.

make theories why the query is slow, try to prove or refute them

- by running parts of the query (particularly with `count(*)`)

don’t assume the query plan is telling you what you think is happening

General advice

every decision to rewrite the query for performance should be based on evidence

- query plan
- profiler, etc.

make theories why the query is slow, try to prove or refute them

- by running parts of the query (particularly with `count(*)`)

don’t assume the query plan is telling you what you think is happening

and never make performance-oriented changes just because they seem to work

- you should understand why they work
- better to deal with suboptimal queries than queries you cannot understand

questions?

@klinovp, pavel@stardog.com
https://community.stardog.com/