Performance

Data Loading and Querying Performance for Stardog

Taught by:

Al Baker
VP, Enterprise Solutions

Learning Objectives

- Understand capacity and data inputs to provide Stardog with the right resources for optimal performance
- Enable self-diagnosis of a performance issue leveraging SPARQL semantics and Joins, Filters, and Optional operations
- Learn to read Query Plans and use it to identify query performance issues
- Review Stardog’s tools and examples for debugging performance issues

Capacity and Data

Stardog and System Resources

- Stardog uses memory aggressively, and total system memory often the **most critical factor** in optimizing performance
- Disk space is also used for both storage of data, incremental transaction metadata, and to handle memory overflow to answer a query

Memory Guidelines

- Heap memory should not be less than 2GB and setting it higher than 100GB
- JVM is set to compressOOPS, 32GB and larger sees benefits starting around 50-60GB
- Direct memory should be set higher than heap memory except for very small scales to prevent the heap size going below the recommended 2GB limit
- Sum of heap and direct memory settings should be around 90% of the total system memory
- Do not run other memory intensive applications on the same machine as Stardog

Capacity Planning

- Follow Stardog docs and analyze metrics to find optimal memory allocation

Number of Triples	JVM Heap Memory	Direct Memory	Total System Memory
100 million	3G	4G	8G
1 billion	8G	20G	32G
10 billion	30G	80G	128G
25 billion	60G	160G	256G
50 billion	80G	380G	512G

Disk Usage

- Industry standard disk guidance: prefer SSD, avoid NFS
- In general, a million triples require 70 MB to 100 MB
- Actual disk usage for a database may be different
- Disk space needed at creation time for bulk loading data is higher as temporary files will be created
    - This should be 2x of the final database size
- Per database quotas can be summed to find total disk requirements

Bulk Load Options

- Database creation time is the most optimized for large scale data loading
    - Prefer compressed data
    - File loads happen in parallel
    - Multicore machines provide benefits on loading and index creation
    - Database strict parsing option can be turned off
    - Use the “bulk_load” memory option for very large databases
    - “--copy-server-side” for copying files to remote machine

Other Data Load Options

- ETL, e.g. the Stardog ETL, can be done with parallel processing steps
- Cache node creation can be done out of band for maximizing operational changes to running clusters
- Stardog data add can use "--server-side" for uploaded files

Understanding SPARQL Evaluation

STARDOG ACADEMY

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

Final query results

?

Basic Graph Pattern (BGP) matching

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

```
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

Intermediate operators: FILTER

```
contains(name, “Mary”)
```

Basic Graph Pattern (BGP) matching

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

```
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

Intermediate operators: PROJECTION

```
?person ?name
```

Intermediate operators: FILTER

```
contains(name, “Mary”)
```

Basic Graph Pattern (BGP) matching

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

```
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

Intermediate operators: DISTINCT

Intermediate operators: PROJECTION

```
?person ?name
```

Intermediate operators: FILTER

```
contains(name, “Mary”)
```

Basic Graph Pattern (BGP) matching

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

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

Understanding Problems in Queries

- Select 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
}
```

Example: Cartesian Product Result Sets

- Select 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
}
```

Note: Disconnected BGP

This computes all pairs of journal and volume editors!

Result: Full Cartesian Product

How to Correct Cartesian Products

- Select 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 }
}
```

Example: Selectivity in Variables

- Doc with 10 authors which appeared in 100 books → the number of results grows combinatorially

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

Example: Improving Selectivity

- 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

Operation Considerations

Scan

- Reads data from an index
- Index orders sort triples in different ways
- Index orders are selected based on constants in the query and join variables

Index Orders

- Single letter indicates sorting order
  - S(ubject), P(redicate), O(bject), C(ontext)
- Index order is identified by a series of letters
- Example index orders
  - SPOC: Sorted by first S, then P, then O, then C
  - PSO: Sorted by first C, then P, then S, then O

Scan Types

```
SELECT ?type {?x a foaf:Person }
Projection(?x)
  Scan[POS](?x, rdf:type, foaf:Person)
```

Scan all triples with a specific predicate and object value

```
SELECT ?cls {?x a ?cls}
Projection(?cls)
  Scan[POC](_, rdf:type, ?cls)
```

Scan binary count index for a specific predicate

```
SELECT ?s {?s ?p ?o}
Projection(?s)
  Scan[SC](?s, _, _)
```

Scan unary count index

Node Types

- Scan
- Join
- Union
- Filter
- Bind
- Values
- Minus
- Slice
- Order
- Distinct
- Reduced
- Projection
- Group
- Singleton
- Empty
- Sort
- PropertyPath
- SERVICE

Node Characteristics

- Nodes have basic pieces of information:
  - Cardinality: How many solutions will be generated
  - Sort: Variable by which solutions will be sorted
- Nodes might have 0, 1, or 2 children based on the node type

Join

- Joins results from two other nodes
- Solution from both nodes should agree on the value assigned to shared variables
  - For outer joins, right solution may contain null value
- Different kind of join types used based on the children nodes

Join Types

Merge	Requires both nodes to be sorted by the same variable, skips unrelated solutions
Hash	Materializes right operand in a hashtable, iterates over left
DirectHash	Iterates over left, computes right operand on-demand
Loop	Iterates over right operand for each solution on the left

Commonly Used Plan Types

Union	Returns all solutions generated by children
Filter	Filter child solutions based on an expression
Bind	Adds another mapping to the child solution
Projection	Keeps selected variables in child solution
Sort	Retrieves all the solution from child and sort them by a specific variable

Joins: Example Walkthrough

- Two SPARQL patterns in the same group are joined
- For SQL users: all inner joins are natural equi-joins

Note: easiest example: VALUES

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

Joins: Example Walkthrough

- Two SPARQL patterns in the same group are joined
- For SQL users: all inner joins are natural equi-joins
- Follow :b to the first match

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

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

Joins: Example Walkthrough

- Two SPARQL patterns in the same group are joined
- For SQL users: all inner joins are natural equi-joins
- Follow :b to the first match
- Follow :b to the second match

```
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: Example Walkthrough

- Two SPARQL patterns in the same group are joined
- For SQL users: all inner joins are natural equi-joins
- Follow :b to the first match
- Follow :b to the second match
- Follow :d

```
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: Example Walkthrough

- These tuples are called solutions
- SPARQL query execution is basically processing bags of solutions (filters, joins, etc.)

```
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)
```

Performance Issue: Unbound Join Keys

- When a join key does not have a value: 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)
```

Note: may cause an unintended large number of results

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
- **Question**: Can OPTIONAL be used?

OPTIONAL Joins and Unbound Join Keys

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

- OPTIONAL Introduces an unbounded variable
- Inspect 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 }
}
```

- Inspect the SPARQL Algebra

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

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

Inspecting the Unbound Predicate

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

- **First:** for every person living in a city without a `:locatedIn` triple the query will return all countries.
- **Second**: even if data is perfect, dealing with nulls slows down joins by a lot (e.g. standard hash joins won't work)
- **Third**: it often shows in the query plan

OPTIONAL Joins and Unbound Join Keys

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

FIX: this avoids the correctness issue

Note: still could be a performance issue but this query is optimizable

Joins on Variable Introduced in BIND

- SPARQL allows for adding new variables to solutions
  - eg. `BIND(?x + ?y as ?z)`
  - Note: new variables can be join keys
- SPARQL expressions can raise (type) errors which:
  - make the target variable unbound
  - typically not visible to the client
- Note: 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
}
```

- Evaluating this expression:
  - Observation: this is a kind of data integration query
  - Results in: 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
}
```

- Observation: if ?person is not a string literal, ?emp_id won't be bound
- Therefore: the query will return all employees for that person
- Analysis: this might be a good time to fix the data (or need a more complex query with type checks or `bound(?emp_iri)` filters)

Comparing Joins vs. Equality Filters

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

vs

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

- Observation: often considered two ways of achieving the same thing

Analysis: Joins vs. Equality filters

- 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:
  - Nested filters or graph patterns
  - FILTERs in OPTIONALs have special semantics in SPARQL

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

Analysis: 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
  - For example: 1.0 vs 1.00
- Note: May be optimized if the SPARQL engine knows that it's impossible for other reasons (SHACL? hints?)
- Recommendation: 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
```

- This can be evaluated in various ways:
  - Join(:livesIn, Join(:locatedIn, :name))
  - Join(Join(:livesIn, :locatedIn), :name), etc.
- Note: engines can fail to pick the optimal order but they will try (inner join ordering is a classical query optimisation problem in databases)

Everything Changes for OPTIONALs

- Observation: reordering OPTIONALs is more difficult and error prone → often not done

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

- Result: often it's executed as-is i.e. Join(OuterJoin(:livesIn, :locatedIn), :name) which may or may not be optimal

Everything Changes for 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
  - Example: Join(A, B) = Join(B, A), Join(A, Join(B, C)) = Join(Join(A, B), C)
- Note: 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

Everything Changes for OPTIONALs

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

Everything Changes for OPTIONALs

`A OPTIONAL { B } ≠ B OPTIONAL { A }`

How about

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

vs

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

- **Question**: can the optimiser freely move OPTIONAL patterns up and down?

Analysis: Moving Optionals

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

- Results: `Cannot move optionals freely (?x -> :d, ?y -> :e)`

  `However..`

Analysis: Where is the Inner Join Evaluated?

```
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 on OPTIONAL are possible
  - E.g. Stardog will push selective patterns into OPTIONALs when it can detect that it won't change semantics
- Risky in practice.
  - Place your OPTIONALs wisely.
  - In many cases they should be pushed to the bottom (hint: OPTIONALs never decrease the number of results)
- Note: SQL engines often can rewrite outer joins into inner joins
  - This needs more work on bringing theory to practice (references)
  - "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!)
- SERVICE queries challenges:
  - No selectivity statistics (in general)
  - Unreliable endpoints
  - Data transmission and ingestion costs

SERVICE aka SPARQL Federation

- Observation: 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 :pavel }
}
```

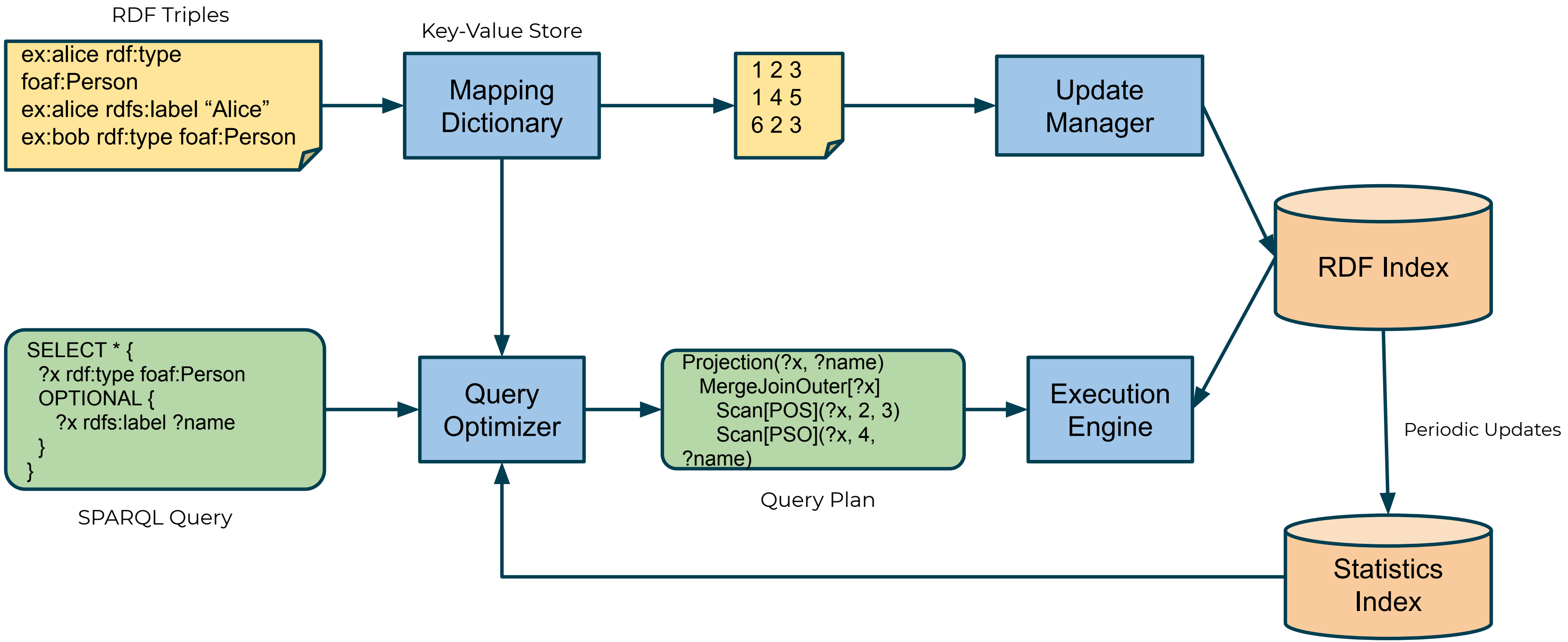
- Optimisers can fail at that for various reasons (pick wrong local pattern, etc.)
    - Check the plan
    - Place local patterns binding ?person in the same scope as SERVICE
- Endpoints may throttle rapid requests (LIMITs on joined patterns could help)

Query Plans

Query Plans

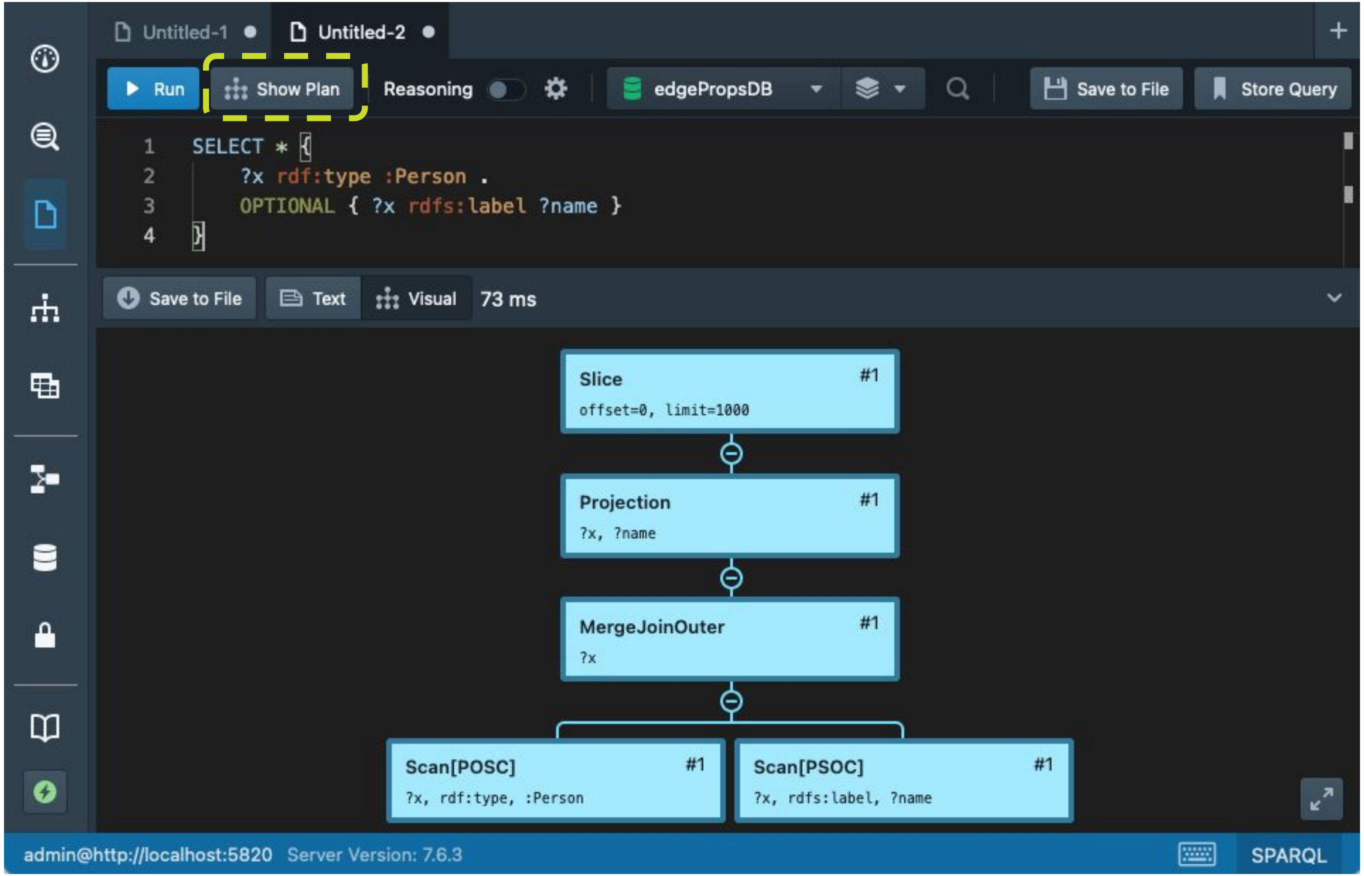
- Tree of plan nodes
- Each node generates zero or more solutions
  - Solution is a mapping from variables to values
- Execution is bottom-up
  - Solutions generated by a node go to the parent node
- Solutions are generated while the client is consuming results

Query Engine Internals



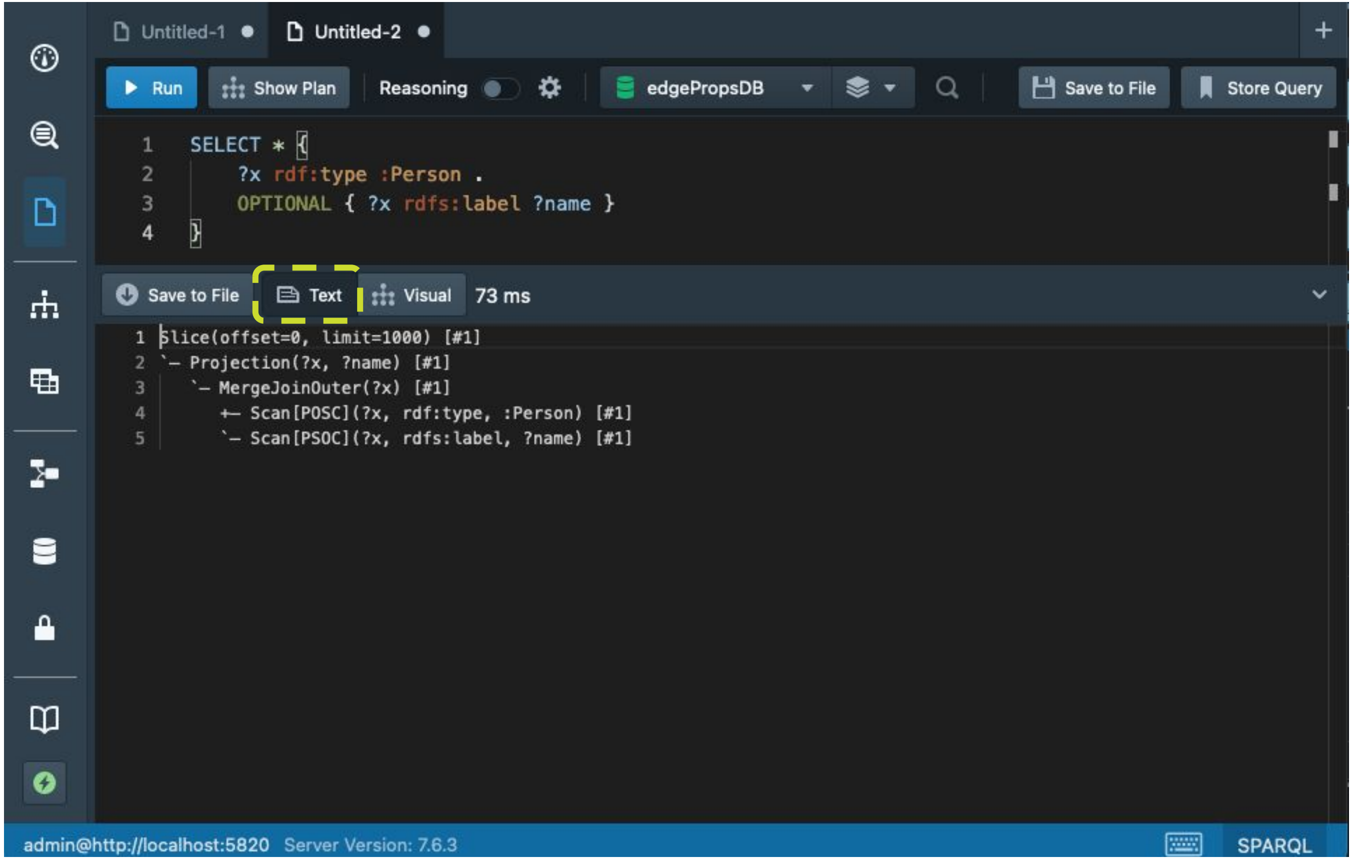
Query Plan

Example



Query Plan

Example



Query Algebra vs. Query Plan

- SPARQL spec defines algebra expressions for SPARQL constructs
  - Basic graph patterns (BGP)
  - Joins
  - UNIONs
  - FILTER, etc.
- Algebra is useful for understanding query's semantics
  - Independent of the actual implementation
- Query Plan: how the engine evaluates the query

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
- Benefits:
    - 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

- The most efficient query execution is streaming:
  - Index scans match some data, generate partial results
  - Immediately processed further (joined, filtered)
  - Results returned to the client
- Key: first results are processed before all results are generated
- This is called the query execution pipeline
- Benefits: lazy, low-latency, min resource consumption

When the Pipeline Breaks

- Observation: 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
- Results: increases latency, memory pressure on the server

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 are streamed as inputs are coming in
- No memory pressure
- Low disk IO overhead

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 selective joins)

How to “Fix” a Slow Query

- Note: engine specific tools help the analysis and correction of a slow query
- Mostly query hints: they tell the optimiser what to do (it may still ignore though)
- General tactics also work to improve performance:
  - Subqueries
    - For defining join order
    - Pushing `DISTINCT` down the plan
  - Move selective patterns around

Example: Erdoes Query from SP2B

```
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
- 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]
```

Tactic: 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)
}
```

Tactic: 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
- 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 *
  - includes 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
  - Queries do not live in the ideal world
  - RDF's "flexible schema" is a double-edged sword
  - Join Order optimisation alone is NP-hard
- optimisation algos operate under uncertainty
  - Cost estimation
  - "Every 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
- Query developers sometimes need to give hints to the optimizer

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
- Never make performance-oriented changes just because they seem to work
  - Understand why they work
  - Better to deal with suboptimal queries than those you cannot understand

Tools and Examples

Explain Example

```
$ stardog query explain myDb query.sparql
```

```
SELECT ?article {
  ?article rdf:type bench:Article .
  ?article ?property ?value
  FILTER (?property=swrc:pages)
}
```

```
Projection(?article) [cardinality=16K]
  Bind((swrc:pages AS ?property)) [cardinality=16K]
    MergeJoin[?article] [cardinality=16K]
      Scan[POSC](?article, rdf:type, bench:Article) [cardinality=17K]
      Scan[PSC](?article, swrc:pages, _) [cardinality=24K]
```

Detect Slowness in Queries

```
SELECT *
WHERE {
  ?article1 swrc:journal ?journal1 .
  ?article2 swrc:journal ?journal2 .
  FILTER (?journal1=?journal2)
}
```

```
Projection(?article1, ?journal1, ?article2, ?journal2) [cardinality=146.7M]
  Filter(?journal2 = ?journal1) [cardinality=146.7M]
    LoopJoin[_] [cardinality=293.4M]
      Scan[PSOC](?article1, swrc:journal, ?journal1) [cardinality=17K]
      Scan[PSOC](?article2, swrc:journal, ?journal2) [cardinality=17K]
```

Rewrite Slow Queries

```
SELECT * {
  ?article1 swrc:journal ?journal .
  ?article2 swrc:journal ?journal
}
```

```
Projection(?article1, ?journal, ?article2) [cardinality=704K]
  MergeJoin[?journal] [cardinality=704K]
    Scan[POSC](?article1, swrc:journal, ?journal) [cardinality=17K]
    Scan[POSC](?article2, swrc:journal, ?journal) [cardinality=17K]
```

Common Signs of Problem

1. Very large cardinalities, especially higher in the tree
2. Loop joins
3. Empty plan node
4. Large cardinalities for non-streaming nodes

    Sort, HashJoin, Distinct, Aggregate

Bugs in Stardog

```
ASK {
  SELECT(COUNT(?s) AS ?count) WHERE {
    ?s ?p ?o .
  } HAVING(?count > 1)
}
```

```
Slice(offset=0, limit=1) [cardinality=0]
  Projection(?count) [cardinality=0]
    Group(aggregates=[(COUNT(?s) AS ?count)]) [cardinality=0]
      Empty [cardinality=0]
```


(This bug was fixed in 4.1.1)

Simple Ways to Speed Up Queries

1. Add LIMIT to query
2. Avoid DISTINCT and/or minimize SELECT vars
3. Add more constants to the query
4. Split into multiple queries

Query Plans for Reasoning

1. Query time reasoning rewrites queries
2. Axioms in the ontology encoded into the query
3. Rewritten query typically has many UNIONs
4. Plan may contain special reasoning plan nodes

Reasoning Node Types

Type	Returns inferred types of an individual
Property	Returns inferred properties of an individual
Top	Returns all individuals
Schema	Returns results for schema queries

Explain with Reasoning

```
$ stardog query explain --reasoning lubm student.sparql
```

```
SELECT ?x WHERE { ?x a lubm:Student }
```

```
Distinct [cardinality=1.7M]
  Projection(?x) [cardinality=1.7M]
    Union [cardinality=1.7M]
      Union [cardinality=1.3M]
        Scan[PSC](?x, lubm:takesCourse, _) [cardinality=1.1M]
        Scan[POSC](?x, rdf:type, lubm:GraduateStudent>) [cardinality=126K]
      Union [cardinality=430K]
        Scan[POSC](?x, rdf:type, lubm:ResearchAssistant>) [cardinality=36K]
        Scan[POSC](?x, rdf:type, lubm:UndergraduateStudent>) [cardinality=394K]
```

Learning Objectives

Learning Objectives

- Understand capacity and data inputs to provide Stardog with the right resources for optimal performance
- Enable self-diagnosis of a performance issue leveraging SPARQL semantics and Joins, Filters, and Optional operations
- Learn to read Query Plans and use it to identify query performance issues
- Review Stardog's tools and examples for debugging performance issues

Thank you