SPARQL Optimization 101

YarcData
A DIVISION OF CRAY INC.

About Me

- **Software Engineer at YarcData, part of Cray Inc**
  - One of my responsibilities is the SPARQL Optimizer
  - Have developed several database specific optimizations and run internal training sessions on SPARQL optimization
- **PMC Member and Committer on Apache Jena project**
  - Joined project in January 2012
  - Have contributed implementations of several common SPARQL optimizations from the literature
  - Have also contributed some entirely new optimizations
- **Lead developer on the dotNetRDF Project**
  - Developed two SPARQL engines over the past 5 years

Procedural Notes

- **Feel free to ask questions as we go along**
  - If I repeat the questions before I answer it's for the benefit of the audio recording
- **USB sticks with resources relevant to the tutorial are available**
  - You may need to share depending on number of attendees
- **Resources also available for download at TBC**
- **Or you can download just the tools at http://jena.apache.org/download/**
  - Get both Apache Jena (the main distribution) and Jena Fuseki
- **Slides will be up on Slideshare at http://www.slideshare.net/RobVesse**

Tutorial Schedule

Topic	Time Slot
Key Concepts	13:30 - 13:45
Tooling	13:45 - 14:15
BGP Optimization	14:15 - 14:30
Algebra Optimization	14:30 - 15:30
Writing Better Queries	15:30 - 16:00
Customizing the Optimizer	16:00 - 16:30

Key Concepts

Section Overview

- **Key Concepts**
  - SPARQL
  - SPARQL Algebra
  - Basic Graph Patterns (BGP)
  - What is SPARQL Optimization?

SPARQL

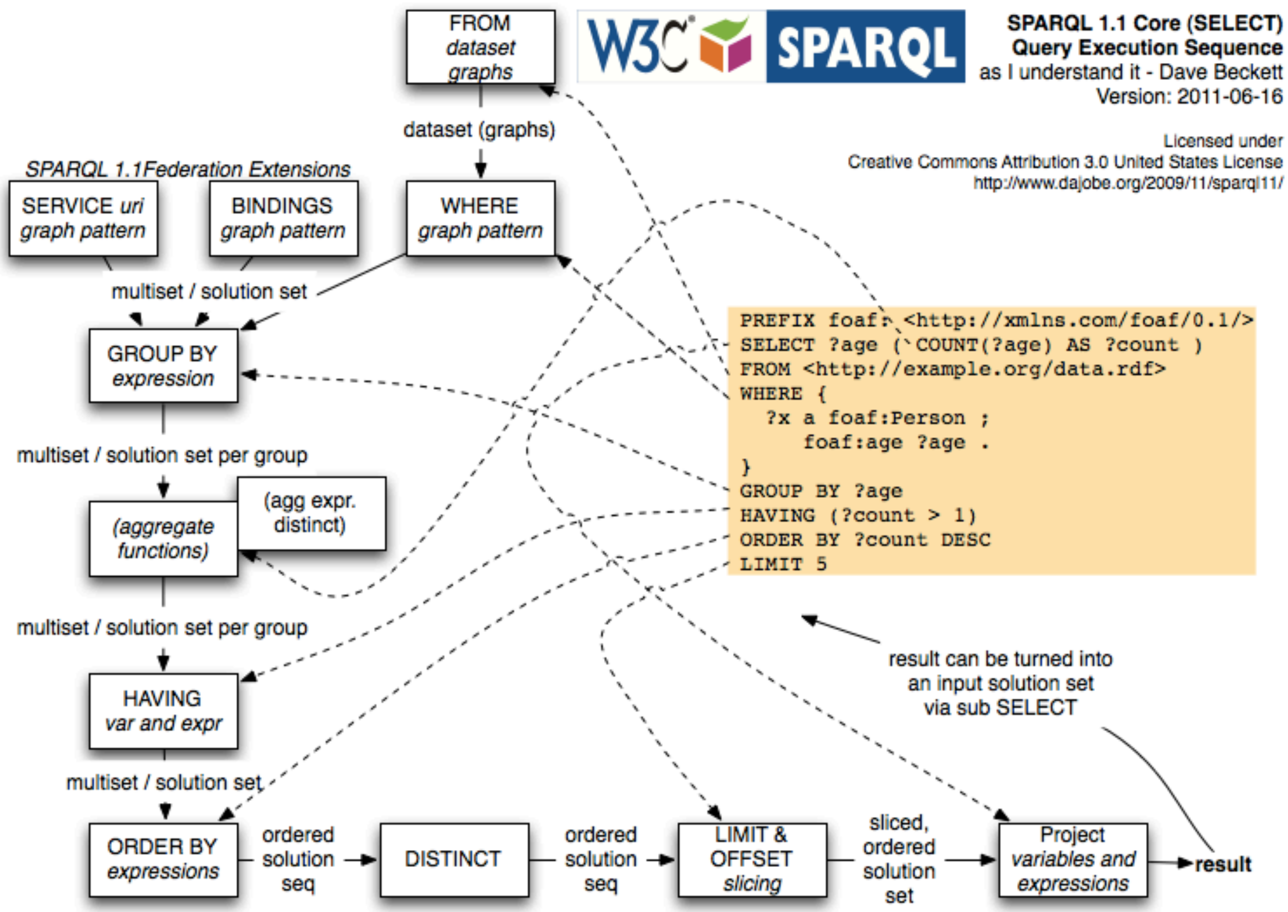
- **Declarative graph pattern matching query language for RDF data**
- **Two versions:**
  - SPARQL 1.0 (Jan 2008) - http://www.w3.org/TR/rdf-sparql-query/
  - SPARQL 1.1 (March 2013) - http://www.w3.org/TR/sparql11-overview/
- **SPARQL 1.1 added many new features:**
  - Grouping and Aggregation
  - Federated Query
  - Simpler negation constructs
  - Sub-queries
  - Update commands
- **SPARQL is widely supported by APIs, tools and RDF databases**
  - SPARQL 1.1 is fairly universally supported since it adds so many valuable new features
  - http://www.w3.org/2009/sparql/implementations/

SPARQL - Example Query

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

SELECT ?age (COUNT(?age) AS ?count)
FROM <http://example.org/data.rdf>
WHERE
{
  ?x a foaf:Person ;
     foaf:age ?age .
}
GROUP BY ?age
HAVING (COUNT(?age) > 1)
ORDER BY DESC(?count)
LIMIT 5
```

SPARQL - Execution Sequence



SPARQL - Learning More

- **SPARQL by Example - Leigh Feigenbaum and Eric Prud'hommeaux**
  - https://www.cambridgesemantics.com/en_GB/semantic-university/sparql-by-example
- **Learning SPARQL - Bob DuCharme**
  - http://learningsparql.com
  - **Disclaimer** - I was a Technical Reviewer for the 2nd Edition

SPARQL Algebra

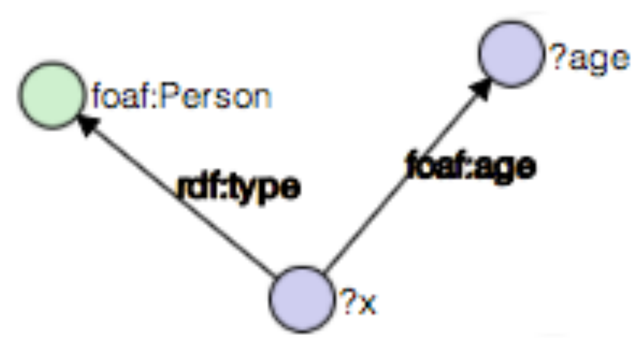
- Defined as part of the SPARQL Query specification
  - http://www.w3.org/TR/sparql11-query/#sparqlDefinition
- A formal semantics for how to evaluate SPARQL queries
  - Specification defines how to translate a query into an algebra
- In relational terms think of the SPARQL Algebra as being the logical query plan
- Most high level optimization happens on the algebra

SPARQL Algebra - SPARQL Set Expressions (SSE)

```
(slice _ 5
  (project (?age ?count)
    (order ((desc ?count))
      (filter (> ?.0 1)
        (extend ((?count ?.0))
          (group (?age) ((?.0 (count ?age)))
            (bgp
              (triple ?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://xmlns.com/foaf/0.1/Person>)
              (triple ?x <http://xmlns.com/foaf/0.1/age> ?age)
            )))))))
```

- **SPARQL Set Expressions (SSE) is a way of serializing SPARQL Algebra defined as part of the Apache Jena project**
- **Lisp style nested set expression syntax**

Basic Graph Patterns (BGPs)



- Basic Graph Patterns (BGPs) are the lowest level unit of a SPARQL query
- Comprised of some combination of individual triple patterns
- Expresses some pattern to be found in the graph
- Above example was visualized with http://graves.cl/visualSparql/

BGPs - Why are they relevant?

- **BGPs translate into database scans over your backend database**
- **SPARQL engines are free to implement the scans however they wish**
  - Depends on underlying data storage, use of indices etc
- **However most SPARQL engines treat each triple pattern as an individual scan**
  - Therefore query engines need to be smart in how they order the scans
  - e.g. feeding bindings from one scan to the next to give more specific scans

What is SPARQL Optimization?

- **Term can be used to mean several different things:**
  1. Rewriting the raw SPARQL Query
  2. Rewriting the SPARQL Algebra
  3. Low level query engine execution optimization
- **We're going to cover all three today in varying levels of details**

Tooling

Section Overview

- **Apache Jena**
  - ARQ
  - TDB
  - Fuseki
- **Command line tools**
  - qparse
  - arq
  - tdbloader/tdbloader2
  - tdbquery
- **Online services:**
  - sparql.org

Apache Jena

- **ASF project providing a RDF, SPARQL and Semantic Web stack written in Java**
  - http://jena.apache.org
- **Key components for us:**
  - Jena ARQ - SPARQL engine implementation
  - Jena TDB - RDF triple store that uses ARQ as its SPARQL engine
  - Jena Fuseki - A database server that can encapsulate TDB

Apache Jena - ARQ

- ARQ is the module that provides the SPARQL engine
  - SPARQL Parsing
  - SPARQL Algebra
  - SPARQL Optimization
  - SPARQL Query and Update Evaluation
- For this talk we're primarily interested in its API
  - We'll reference various interfaces and concrete classes as we go along
- Javadoc at http://jena.apache.org/documentation/javadoc/arq/index.html

Apache Jena - TDB

- Persistent disk based RDF store
- Uses memory mapped files to maximize database access and query speeds
- If you're using the provided resources there are some pre-built databases in the dbs/ directory
  - We'll use these later in the tutorial
- Documentation at http://jena.apache.org/documentation/tdb/index.html

Apache Jena - Fuseki

- **A database server that can be backed by TDB**
- **Provides a bare bones web UI**
  - New UI currently in the work
- **Lets us easily launch a server connected to a TDB dataset for testing**
- **E.g.**

```
$> java -jar fuseki-server.jar --loc=path/to/db --update /ds
```

- --loc argument provides path to a TDB database directory
- --update argument enables writes
- /ds is the path used for the database in the web UI and HTTP interface
- Documentation at http://jena.apache.org/documentation/serving_data/index.html

Command Line Tools

- These are all part of the Apache Jena convenience binary distribution
- Found in the tutorial resources under the apache-jena/bin folder
- You'll need to first set the environment variable JENA_HOME to the apache-jena directory
  - `*Nix - export JENA_HOME=/path/to/apache-jena`
  - `Windows - set JENA_HOME=C:\path\to\apache-jena`
- You can then run any of the scripts while in the bin/ folder
- Optionally you can add this to your PATH to be able to run them from anywhere
  - `*Nix - export PATH=$JENA_HOME/bin:$PATH`
  - `Windows - set PATH=%JENA_HOME%/bin;%PATH%`

Command Line Tools - qparse

Get algebra for a query

```
$> ./qparse --print op "SELECT * WHERE { FILTER(?unbound = <http://
constant>) }"
```

Get optimized algebra for a query

```
$> ./qparse --print opt "SELECT * WHERE { FILTER(?unbound = <http://
constant>) }"
```

- **Supports various values for --print that allow you to inspect the query in different ways**
  - op - Basic algebra
  - opt - Optimized algebra
  - plan - ARQ's execution plan
  - query - Query with reformatting where applicable
  - Can repeat --print multiple times to ask for multiple formats
- **Or use --explain option to print query and optimized algebra**

Command Line Tools - arq

Run a query

```
$> ./arq --data=TODO.ttl "SELECT * WHERE { ?s ?p ?o }"
```

- **Provides a CLI for running queries**
- **--data argument provides data file to be queried**
- **Then simply provide the query to be executed**
- **--results can be used to choose result format**
  - text - ASCII table
  - xml - SPARQL Results XML
  - json - SPARQL Results JSON

Command Line Tools - tdbloader/tdbloader2

tdbloader

```
$> ./tdbloader --loc /path/to/database data.ttl
```

tdbloader2

```
$> ./tdbloader2 --loc /path/to/new-database data.ttl
```

- **tdbloader is a bulk loader for TDB**
- **tdbloader2 is an alternative bulk loader**
  - *Nix only
  - Create only, can't be used to append to existing databases
- **Once created we can expose it via Fuseki or query it with other command line tools like tdbquery**

Command Line Tools - tdbquery

tdbquery

```
$> ./tdbquery --loc /path/to/database "SELECT * WHERE { ?s ?p ?o }"
```

- Similar to arq command except it queries a pre-built TDB database
- Useful if you want to query without standing up a Fuseki server/writing code
- Downside is you pay startup costs on every query and get minimal cache benefits

Online Tools - sparql.org

- Service provided by the Jena project
  - http://sparql.org
- Installation of Fuseki with a couple of toy in-memory databases to play with
- Provides web based interfaces that have similar functions to the CLI tools already seen
  - e.g. Query Validator lets you see raw and optimized algebra

BGP Optimization

Section Overview

- What is it?
- Reordering Strategies
- Configuring with Jena

What is it?

- As already discussed BGPs are a low level operation in SPARQL essentially representing a DB scan
- Therefore the order in which scans are performed and whether results from scans inform subsequent scans are important
- Three main reordering strategies:
  - None
  - Heuristics
  - Statistics
- Often a configuration option in a SPARQL engine

Reordering Strategies - None

- **No pattern reordering is done**
- **How is this an optimization?**
  - Useful in the case where the user wants to manually control the order of operation
  - You may have a query that exhibits pathological execution behavior unless the exact execution order is followed

Reordering Strategies - Heuristic

- Sometimes known as fixed
- Uses heuristic rules about the approximate selectivity of triple patterns
- Typically favors putting patterns with more constants first
  - Exact heuristics vary by implementation
  - The **ReorderFixed** class encodes Jena's implementation of this
- Tends to do a good job most of the time
  - Very data dependent
  - Datasets that don't match the assumptions of the rules may see poor performance e.g. DBPedia (http://dbpedia.org)

Reordering Strategies - Statistics

- **Using statistics about the data either directly or indirectly to decide how to order the triple patterns**
  - May use statistics directly
  - May generate rules based on the statistics
- **Like the Heuristic strategy the aim is to put patterns deemed more selective first**
- **Assuming the data does not change then this is often the most effective strategy**
  - For data that changes you either need to keep the statistics up to date
  - Or use the derived rules approach as accuracy of rules is typically less directly affected by changes to the data
- **Jena's implementation is encoded in the ReorderWeighted and StatsMatcher classes**
- **Statistic based strategy may get worse as complexity of BGP increases:**
  - http://www.csd.uoc.gr/~hy561/papers/storageaccess/optimization/Characteristic%20Sets.pdf

Reordering Strategies - Example

```
$> ./tdbquery --time --loc dbs/none --query queries/sp2b_2.rq --repeat 5

$> ./tdbquery --time --loc dbs/fixed --query queries/sp2b_2.rq --repeat 5

$> ./tdbquery --time --loc dbs/stats --query queries/sp2b_2.rq --repeat 5
```

- Try running the above commands
- The dataset is very small (250,000) triples so difference is negligible but you will see a small difference
- Small differences add up as you scale upwards

Configuring with Jena

- **For Jena TDB:**
  - Place a **.opt** file in in the database directory
  - Use **none.opt** for no reordering or **fixed.opt** for heuristic reordering
  - Use the **tdbstats** tool to generate a statistics file **stats.opt** for statistics based reordering
  - See http://jena.apache.org/documentation/tdb/optimizer.html for more information
- **For stock Jena:**
  - The None strategy is used by default
  - A custom optimizer must be written to introduce an alternative strategy - more on this later

YarcData
A DIVISION OF CRAY INC.

Algebra Optimization

Section Overview

- **Formal Algebra vs ARQ Algebra**
  - Operators
  - Quick Reference
- **Algebra Optimization in ARQ**
  - Limitations
  - Optimization vs Performance Trade Offs
  - Rewrite interface
  - Transformer and ExprTransform interfaces
- **Algebra Optimizations**
  - Importance of ordering
  - ARQ Standard Optimizer
  - Examples and Discussion of Optimizations

Formal Algebra vs ARQ Algebra

- **ARQ has its own API for representing SPARQL Algebra**
- **Mostly 1-1 relationship to formal algebra**
  - Some differences for extensions, optimizations etc.
  - See later Quick Reference section for an overview of mapping from SPARQL operations to algebra
- **Also has a string serialization of the algebra using a syntax called SPARQL Syntax Expressions (SSE)**
  - http://jena.apache.org/documentation/notes/sse.html
- **We often use the string serialization as output for debugging and discussion**
- **Examples later in these slides will use SSE to show the algebra**

ARQ Algebra - Operators

- **Algebra elements are referred to in ARQ as operators**
  - Top level interface is the **Op** interface
  - Op0, Op1, Op2 and OpN are the more specific interfaces
- **Several classes of operators:**
  - Terminals – Match some data in the store
  - Unary – Apply some operation to the results of an inner operator
  - Binary – Apply some operation to the results of two inner operators, first inner operator (LHS) is always evaluated first
  - Nary - Apply some operation to the results of N inner operators evaluated in order
- **Algebra is evaluated bottom up**
  - If you think of it as a tree the left most leaf node gets evaluated first

ARQ Algebra - Quick Reference 1/2

SPARQL Operator/Clause	ARQ Algebra Class	SSE Form
SELECT ?var	OpProject	project
DISTINCT	OpDistinct	distinct
REDUCED	OpReduced	reduced
Project Expression/BIND	OpExtend/OpAssign	extend/assign
Empty BGP	OpTable	table unit
BGP	OpBgp/OpQuadPattern	bgp/quadpattern
FILTER/HAVING	OpFilter	filter
Joins	OpJoin	join
GRAPH	OpGraph	graph
UNION	OpUnion	union
OPTIONAL	OpLeftJoin	leftjoin

ARQ Algebra - Quick Reference 2/2

SPARQL Operator/Clause	ARQ Algebra Class	SSE Form
MINUS	OpMinus	minus
LIMIT and/or OFFSET	OpSlice	slice
GROUP BY and Aggregates	OpGroupBy	group
ORDER BY	OpOrderBy	order
VALUES	OpTable	table
	ARQisms	
	OpPropFunc	propfunc
	OpTable	table empty
	OpExt	
	OpSequence	sequence
	OpConditional	conditional
	OpDisjunction	disjunction

Limitations of Algebra Optimization

- In ARQ at least algebra optimization is done with zero knowledge of the data
- Therefore all optimizations are static transformations on the raw algebra generated from the query
- Must preserve evaluation semantics
  - Executing the optimized the algebra **must** result in the same results as executing the raw algebra
- Typically conservative
  - If it cannot decide whether an optimization preserves semantics it won't apply it

Optimization vs Performance Trade Off

- There is a trade off between the amount of time you spend analyzing & transforming the query and the performance gains of those transformations
- If an optimization is too specialized then the cost to the system of testing for that situation on every query will outweigh the benefit of applying the optimization
- XKCD - CC-BY-NC 2.5 - http://xkcd.com/303/

Rewrite interface

- **The Rewrite interface is a trivial interface used for optimizers**
  - Single rewrite(Op op) method
- **Related RewriterFactory interface is used to select which optimizer to use**
  - You can substitute your own custom optimizer by setting a RewriterFactory with the Optimize.setFactory() method
- **ARQ's standard optimizer is the Optimize class**
  - Individual parts can be turned off through global configuration settings e.g.

```
// In Java Code
ARQ.getContext().set(ARQ.optFilterPlacement, false);

// With command line tools
--set http://jena.hpl.hp.com/ARQ#optFilterPlacement=false
```

Transformer and ExprTransform interface

- **Transformer interface is used to implement specific transformations on algebra**
  - Similarly ExprTransform does the equivalent for expressions
- **Both applied as bottom up transformations**
- **Each method receives the original operator/expression plus the results of transforming any inner operator(s)/ expression(s)**
  - Means transformations are applied potentially multiple times in complex algebras
  - In principle a transformer can throw out inner transformations in favour of alternative transformations at a higher level
- **TransformCopy provides a standard base implementation**
  - Implements all methods as simple copy operations
  - Means we only need to implement specific methods we want to override
  - Similarly ExprTransformCopy for expressions
- **Lots of nice example implementations in ARQ**
  - Let's take a look at a couple of examples

Importance of ordering

- The order in which optimizations are applied matters
- For example there are some optimizations which enable other optimizations
- Sometimes there is a specific and general version of an optimization
  - The specific version gives bigger benefits but applies in fewer cases
  - The general version yields smaller benefits but applies in more cases

ARQ Standard Optimizer

1. Variable Scope Renaming
2. Constant Folding
3. Property Functions
4. Filter Conjunction (&&)
5. Filter Expand One Of
6. Filter Implicit Join
7. Implicit Left Join
8. Filter Disjunction (||)
9. Top N Sorting
10. ORDER BY + DISTINCT
11. DISTINCT to REDUCED
12. Path Flattening
13. Index Join Strategy
14. Filter Placement
15. Filter Equality
16. Filter Inequality
17. Table Empty promotion
18. Merge BGPs

Variable Scope Renaming

- **Renames variables in the algebra to ensure that any potential scope clashes are avoided**
  - Particularly relevant for sub-queries
  - Scope clashes can also be introduced by complex nested queries
- **Done early so that later optimization steps don't perform semantically invalid optimizations**
- **See TransformVarScopeRename for implementation**

Constant Folding

Original Algebra

```
(project (?value)
  (extend ((?value (* 2 2)))
    (table unit)))
```

Optimized Algebra

```
(project (?value)
  (extend ((?value 4))
    (table unit)))
```

- **Where possible pre-evaluate all/part of some expressions**
  - Similar to what compilers do with code
- **Avoids making the engine repeat simple calculations**
  - Important to remember we're working in RDF Nodes not native Java data types i.e. type casting involved
- **See ExprTransformConstantFolding for implementation**

Property Functions

- Property Functions are a SPARQL extension supported by ARQ
- Property Functions are expressed as some number of triple patterns in a single BGP
- TransformPropertyFunctions contains the relevant implementations
  - Finds relevant triple patterns and transforms them into the relevant OpPropFunc algebra

Filter Conjunction (&&)

- Combines filters that use && expressions into flat expression lists
- Makes it easier to extract and optimize specific conditions in later optimization steps
- This is primarily an ARQism
- See TransformFilterConjunction for implementation

Filter Expand One Of

- **Turns IN expressions into the equivalent || expression**
- **Allows for later optimization steps to better optimize the individual filter conditions**
  - e.g. Filter Placement and Filter Disjunction (||)
- **This is actually specification motivated**
  - See http://www.w3.org/TR/sparql11-query/#func-in
- **See TransformExpandOneOf for implementation**

Filter Expand One Of - Example

Original Algebra

```
(filter (in ?s <http://x> <http://y>)
  (bgp
    (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)
  ))
```

Optimized Algebra

```
(filter (|| (= ?s <http://x>) (= ?s <http://y>))
  (bgp
    (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)
  ))
```

Filter Implicit Join

- **Applies where a filter denotes an implicit join**
  - e.g. FILTER (?x = ?y)
  - e.g. FILTER (SAMETERM(?x, ?y))
- **Requires that we can guarantee that at least one of the variables cannot be a literal**
- **Substitutes one variable for the other**
- **Introduces an extend operator to ensure the other variable remains visible outside of the filtered operation**
  - The other variable may be used elsewhere in the algebra for a larger query
- **Can yield huge performance improvements**
  - Where implicit joins are present there is often a cross product
  - Much more efficient to do a constrained join than to do an unconstrained cross product and filter over it
- **See TransformImplicitJoin for implementation**

Filter Implicit Join - Example

Original Algebra

```
(filter (= ?s ?t)
  (bgp
    (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)
    (triple ?t <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)
  ))
```

Optimized Algebra

```
(extend ((?s ?t))
  (bgp
    (triple ?t <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)
    (triple ?t <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)
  ))
```

Implicit Left Join

- Applies where the filter in a left join denotes an implicit join
- Requires that we can guarantee that at least one of the variables cannot be a literal
- Substitutes one variable for another
- Essentially a variation on Filter Implicit Join specific to Left Joins (i.e. OPTIONAL)
- Uses an extend to ensure the other variable remains visible outside the RHS
  - The other variable may be used elsewhere in the algebra for a larger query
- Can yield huge performance improvements
  - Where implicit joins are present there is often a cross product
  - Much more efficient to do a constrained join than to do an unconstrained cross product and filter over it
- See TransformImplicitLeftJoin for implementation

Implicit Left Join - Example

Original Algebra

```
(leftjoin
  (bgp
    (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
type>))
  (bgp
    (triple ?t <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
anotherType>))
  (= ?s ?t))
```

Optimized Algebra

```
(leftjoin
  (bgp
    (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://
type>))
  (extend ((?t ?s))
    (bgp
      (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://anotherType>)
    )))
```

Filter Disjunction (||)

- **Applies where there are multiple equality constraints combined with ||**
- **Substitute each constant into the inner algebra separately and uses union to combine the results**
- **Uses an extend to ensure the eliminated variable remains visible outside of the filtered operation**
  - The other variable may be used elsewhere in the algebra for a larger query
- **Can yield good performance improvements**
  - Equality constraints can cause too much data to be retrieved by the inner algebra
  - Often much more efficient to do several more specific scan than to do a single more generic scan and filter over it
- **See TransformFilterDisjunction for implementation**

Filter Disjunction (||) - Example

Original Algebra

```
(filter (|| (= ?s <http://x>) (= ?s <http://y>))
  (bgp
    (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)
  ))
```

Optimized Algebra

```
(union
  (extend ((?s <http://x>))
    (bgp
      (triple <http://x> <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> ?type)
    ))
  (extend ((?s <http://y>))
    (bgp
      (triple <http://y> <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> ?type)
    )))
```

Top N Sorting

- Used when there is a LIMIT/OFFSET and an ORDER BY
- Stores only the N top intermediate results seen
  - Avoids a full sort of all intermediate results
  - Reduces memory usage during query execution
- Can optionally also include a DISTINCT/REDUCED condition
  - Again reduces memory usage during query execution
- See TransformTopN for implementation

Top N Sorting - Example

Original Algebra

```
(slice _ 10
  (order (?type)
    (bgp (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?
type))))
```

Optimized Algebra

```
(top (10 ?type)
  (bgp (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?
type)))
```

YarcData
A DIVISION OF CRAY INC.

ORDER BY + DISTINCT

- SPARQL states that DISTINCT happens after ORDER BY
- However where there are a large number of non-distinct results doing that ordering first can harm performance
- In some cases it is safe to move the DISTINCT before the ORDER BY and this can yield big performance gains
  - Requires that the query selects specific variables and that the ORDER BY does not use any variables that are not projected
- This is broadly equivalent to wrapping everything except the ORDER BY in a sub-query with SELECT DISTINCT applied to it
- See TransformOrderByDistinctApplication for implementation

ORDER BY + DISTINCT - Example

Original Algebra

```
(distinct
  (project (?s)
    (order (?s)
      (bgp
        (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?
type)
       ))))
```

Optimized Algebra

```
(order (?s)
  (distinct
    (project (?s)
      (bgp
        (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?
type)
      ))))
```

DISTINCT to REDUCED

- Replaces DISTINCT with REDUCED
- Often this gives the same effect as a DISTINCT
  - In ARQ REDUCED eliminates only adjacent duplicates - very memory efficient
  - When ORDER BY is used as well almost certainly identical behaviour to DISTINCT
- See TransformDistinctToReduced for implementation

Path Flattening

- **Some simple property paths can be flattened into simpler and more efficient algebra**
- **Flattens simple property paths**
  - Sequence
  - Inverse
- **Primarily only flattens property path syntax that can be considered convenience syntax**
  - i.e. where they could be written as standard graph patterns
- **See TransformPathFlattern for implementation**

Path Flattening - Example

Original Algebra

```
(graph <urn:x-arq:DefaultGraphNode>
  (path ?s (seq <http://predicate> <http://label>) ?subItemLabel))
```

Optimized Algebra

```
(bgp
  (triple ?s <http://predicate> ??P0)
  (triple ??P0 <http://label> ?subItemLabel)
)
```

Index Join Strategy

- ARQ heavily relies on the use of indexed joins to improve performance
  - Flows intermediate results from one part of the query to the next
- Analyses the algebra looking for portions of the query where indexed joins can be safely applied
  - i.e. where variable scoping rules permit a linearization of the join
- Three forms depending on the type of join operation involved
  - OpSequence for standard joins
  - OpConditional for left joins
  - OpDisjunction for unions
- See TransformJoinStrategy for implementation

Index Join Strategy - Example

Original Algebra

```
(leftjoin
  (bgp (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?
type))
  (bgp (triple ?s <http://p> ?o)))
```

Optimized Algebra

```
(conditional
  (bgp (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?
type))
  (bgp (triple ?s <http://p> ?o)))
```

Filter Placement

- **Takes filter conditions and tries to push them deeper into the query**
  - i.e. aims to make filters be applied as early as possible and so limit the intermediate results earlier in the query execution
- **May place individual conditions in different places in the query**
- **Sometimes this can have adverse effects because it can split BGPs**
  - This may introduce cross products which is undesirable for some systems
  - Can be configured to place filters without splitting BGPs
- **See TransformFilterPlacement for implementation**

Transform Filter Placement - Example

Original Algebra

```
(filter (> ?o 10)
  (union
    (bgp (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type))
    (bgp (triple ?x <http://p> ?o))))
```

Optimized Algebra

```
(filter (> ?o 10)
  (union
    (bgp (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type))
    (filter (> ?o 10)
      (bgp (triple ?x <http://p> ?o)))))
```

Filter Equality

- Applies where a FILTER compares a variable against a constant
  - e.g. FILTER(?x = <http://constant>)
  - e.g. FILTER(SAMETERM(?x, <http://constant>)
- Substitutes the constant for the variable
- Uses an extend to ensure the substituted variable remains visible outside of the filtered operation
  - The other variable may be used elsewhere in the algebra for a larger query
- Can yield huge performance improvements
  - Equality constraints can cause too much data to be retrieved by the inner algebra
  - Often much more efficient to do a more specific scan than to do a more generic scan and filter over it
- See TransformFilterEquality for implementation

Filter Equality - Example

Original Algebra

```
(filter (= ?s <http://constant>)
  (bgp
    (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)
  ))
```

Optimized Algebra

```
(extend ((?s <http://constant>))
  (bgp
    (triple <http://constant> <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> ?type)
  ))
```

Filter Inequality

- **Experimental optimization for where a FILTER compares a variable for inequality against a constant**
  - e.g. FILTER(?x != <http://constant>)
- **Constant must be non-literal**
- **Currently off by default**
  - Use optFilterInequality key to enable
- **Transforms the query to use MINUS and VALUES to subtract the solutions the user is not interested in**
  - Takes advantage of the fact that joins are typically more performant than filters
- **Testing shows limited performance gains depending on the number of variables involved**
- **See TransformFilterInequality for implementation**

Filter Inequality - Example

Original Algebra

```
(filter (!= ?type <http://type>)
  (bgp (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)))
```

Optimized Algebra

```
(minus
  (bgp (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type))
  (table (vars ?type)
    (row [?type <http://type>])
  ))
```

Table Empty Promotion

- Some optimizations can produce the special (table empty) operator
- This denotes that the optimizer has determined that part of the query will evaluate to no results
  - Allows the engine to skip evaluating it entirely
- In many cases this may mean a larger portion of the query than initially identified actually returns no results
  - Due to SPARQL evaluation semantics e.g. join
- Promotes the operator up the tree to skip the largest portion of evaluation possible

Merge BGPs

- Applies where there are adjacent basic graph patterns joined together
- Combines them into a single graph pattern i.e. eliminates the join
- Allows the database layer to have more control over the order in which it does the scans
- Can sometimes eliminate unintentional cross products

Merge BGPs - Example

Original Algebra

```
(join
  (bgp
    (triple ?s <http://predicate> ?value))
  (bgp
    (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)
  ))
```

Optimized Algebra

```
(bgp
  (triple ?s <http://predicate> ?value)
  (triple ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?type)
)
```

Writing Better Queries

Value Equality vs Term Equality

- The = operator in SPARQL is value equality, while the SameTerm() function provides term equality
- Value equality looks at the value of the term
  - e.g. 1 = 1.0 => true
  - So things that don't have the same RDF representation can be considered equal like the integer and decimal in the above example
  - Requires the database to inspect the value of the term
- Term equality looks at the precise term only
  - e.g. SameTerm(1, 1.0) => false
  - Means the database can do the comparison purely on the internal identifiers
- Term equality is only different from value equality for literals
- Always use term equality for URIs or in situations where you expect literal values to be clean and consistent
  - e.g.. where you known 1 is always stored as 1 and never as 1.0

Simplifying Expressions

- If you have expressions that contain operations on constants try to simplify the expressions to use as few as constants as possible
- Otherwise you may be requiring the database engine to do a trivial part of the calculation for you many times which you could do once yourself
- For example:
  - BIND (2 * 2 AS ?TwoSquared) => BIND(4 AS ?TwoSquared)
- Particularly relevant if you are encoding complex conditions into expressions
  - Any simplification you can do can improve the queries performance
- ARQ will try and do this anyway but may fail in complex cases

REGEX vs String Functions

- REGEX() is always an expensive function for any database
- SPARQL 1.1 added many useful string functions that can be used to carry out a lot of tasks that could only be done with REGEX() in SPARQL 1.0
  - http://www.w3.org/TR/sparql11-query/#func-strings
- CONTAINS() for finding strings containing a search string
- STRSTARTS() and STRENDS() for finding strings with a given prefix/suffix
- LCASE() and UCASE() can be used to help simulate case insensitivity when not using REGEX()

REGEX vs String Functions – CONTAINS()

REGEX Query

```
SELECT * WHERE {
  # Some Patterns
  FILTER(REGEX(?value, "search", "i"))
}
```

CONTAINS Query

```
SELECT * WHERE {
  # Some Patterns
  FILTER(CONTAINS(LCASE(?value), "search"))
}
```

- CONTAINS() can be used to filter for values that contain a given string
- LCASE() can be used if case insensitive search is needed

REGEX vs String Functions – STRSTARTS()

REGEX Query

```
SELECT * WHERE {
  # Some Patterns
  FILTER(REGEX(?value, "^http://"))
}
```

STRSTARTS Query

```
SELECT * WHERE {
  # Some Patterns
  FILTER(STRSTARTS(?value, "http://"))
}
```

- STRSTARTS() can be used to filter on values that start with a specific string

Avoid overly broad FILTERs

- **Apply the FILTER as deeply in the query as you can**
  - i.e. as soon as all the relevant data needed to calculate your FILTER condition is available
  - If you have multiple filter conditions combined with && split the filter up into separate filters where you can
  - Doing so may allow you to apply the separate conditions deeper in the query
- **Avoid using FILTER to do things that can be done other ways**
  - Especially true when that FILTER applies over a large portion of the query
  - e.g. FILTER(?var = <http://constant>) which can be done by using a constant and a BIND instead – see Filter Equality earlier in these slides
- **While the optimizer tries to improve some queries with overly broad filters these optimization cannot be safely applied to all queries**
  - You as the query writer have more information about your intentions and the data being queried
  - e.g. Filter Equality

Avoid SELECT * where unnecessary

- While SELECT * is useful during query development and debugging once a query is in production using it can reduce performance
- When you SELECT specific variables there are more optimizations that can be applied to your query
- Also less data for the database to transfer back to you when your query completes

Avoid DISTINCT where unnecessary

- **DISTINCT can be quite costly to compute in terms of memory**
  - The system has to build a hash table/similar to detect and eliminate duplicates
- **Unless you actually need to eliminate duplicate rows it is better to avoid usage**
- **In some cases if you only need part of the results to be distinct it may be better to push the DISTINCT down into a sub-query**
  - Only applies over the portion of the query you require to give distinct results
  - Avoids the DISTINCT being over the entire intermediate results
- **Try using REDUCED instead**

Use LIMIT and OFFSET

- **Use LIMIT and OFFSET wherever possible**
- **In many systems this can cause them to do less work**
  - Especially true when ORDER BY is also used
- **As you are asking for less data from the database there is less IO required to get your answers back**

Using VALUES to assign constants

- Using BIND to assign constants to several variables may hurt performance
- Better to use VALUES to add in the constants you desire via joining which may be more efficient
- Particularly useful if you are introducing multiple constants

Customizing the Optimizer

Section Overview

- Configuring the Standard Optimizer
- Writing a specific optimization
- Writing your own optimizer

Configuring the Standard Optimizer

- **As alluded to earlier the various parts of the Standard Optimizer can be turned on/off by configuration keys**
- **Which context should you change?**
  - Use ARQ.getContext() to change the global context - applies to all subsequent queries
  - Use getContext() on QueryExecution objects to change the query context - applies to only the execution of that query
  - Query context is populated from global context so you can set global options on the global context and per-query options on the per-query context
- **See the javadoc for the optimizer configuration keys**
  - http://jena.apache.org/documentation/javadoc/arq/com/hp/hpl/jena/query/ARQ.html#field_summary
  - The fields beginning with **opt** are the relevant keys

Writing a specific optimization

- **Assuming an algebra optimization we'll start by extending TransformCopy**
- **We then need to override all the relevant methods for algebra we want to consider for optimization**
- **Example - Trivially true/false filters**
  - Optimize filters that can be evaluated in full without executing the query
  - i.e. those that only use constants

Writing your own Optimizer

- As already noted we want to implement the Rewrite interface
- In the rewrite() method want to apply a sequence of Transformer's that implement the optimizations we care about
  - Ideally you should wrap each application with a check as to whether the specific optimization is enabled
  - Depends on whether your optimizer will be used outside of your organization
- Finally register as the default optimizer by calling Optimize.setFactory()
  - Actually takes a RewriteFactory rather than a Rewrite but we can use a trivial anonymous implementation to return our Rewrite implementation

Questions?

Email: rvesse@apache.org

References & Resources

Topic	URL
Apache Jena Downloads	http://jena.apache.org/download/
SPARQL 1.0 Query Specification	http://www.w3.org/TR/rdf-sparql-query/
SPARQL 1.1 Overview	http://www.w3.org/TR/sparql11-overview/
SPARQL 1.1 Implementation Report	http://www.w3.org/2009/sparql/implementations/
SPARQL by Example	https://www.cambridgesemantics.com/en_GB/semantic-university/sparql-by-example
Learning SPARQL	http://learningsparql.com
Visual SPARQL	http://graves.cl/visualSparql/
RDF Characteristic Sets Paper	http://www.csd.uoc.gr/~hy561/papers/storageaccess/optimization/Characteristic%20Sets.pdf

Acknowledgments

Resource	Rightsholder	License	URL
SPARQL 1.1 Execution Sequence	Dave Beckett	CC-BY 3.0	http://www.dajobe.org/2009/11/sparql11/
XKCD Compiling Comic	XKCD	CC-BY-NC 2.5	http://xkcd.com/303/