SPARQLing Pig

Processing Linked Data with Pig Latin

Stefan Hagedorn, Katja Hose, Kai-Uwe Sattler

BTW 2015
Motivation

Pig

- data flow language, tuple oriented
- compiled to MapReduce
- process large datasets
- access data in HDFS

Linked Data / RDF

- connect information / datasets
- triple format
  (subject, predicate, object)
- query language: SPARQL
- federated query processing

SELECT ?x ?y WHERE {
  ?event <long> ?x.
  ?event <lat> ?y.
  ?event <artist> ?artist .
  ?artist <name> "Metallica" .
}
Motivation

- existing solutions: SPARQL to Pig, e.g.:
  
  Alexander Schätzle et. al.,

  **PigSPARQL: Übersetzung von SPARQL nach PigLatin**, BTW 2011
Motivation

Problems

- no BGPs in Pig
- self joins to reconstruct entities
  - load dataset twice (or more)
  - \textit{COGROUP}: combination of MapReduce jobs

Contribution

- Pig Latin language extension
  - data model
  - add SPARQL-like features
  - not only one dataset + access remote data
- efficient processing of Linked Data in Pig
- results as foundation for cost-based Pig compiler/rewriter
Outline

1. Data Model
2. Pig Extensions
   • conversion
   • load/access
   • BGP support
3. Extended Pig Rewriting
4. Evaluation
Data Model

- RDF very flexible model
  - represent arbitrary structures and graphs
  - requires self joins in Pig
- fixed schema not flexible enough: \((s, p_1, \ldots, p_n)\)

Our approach: for each subject: bag of predicate-object pairs

```
{ subject: bytearray,
  stmts: { (predicate: bytearray, object: bytearray) }
}
```

H. Kim, et al,

From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra, PVLDB 2011
Data Model

- RDF very flexible model
  - represent arbitrary structures and graphs
    - requires self joins in Pig
- fixed schema not flexible enough: \((s, p_1, \ldots, p_n)\)

Our approach: for each predicate: bag of subject-object pairs

\[
\begin{align*}
\{ & \text{predicate: bytearray}, \\
& \text{stmts: \{ (subject: bytearray, object: bytearray) \}} \\
\}
\end{align*}
\]

H. Kim, et al,
From SPARQL to MapReduce: The Journey Using a Nested TripleGroup Algebra, PVLDB 2011
Pig Extensions

TUPLIFY

- avoid self-joins
- convert plain triples to triple-bag format
  - using GROUP BY
  - on any component
- explicitly or implicitly (rewriting rules)

```
triple_groups = TUPLIFY triples BY subject

triple_groups = FOREACH (GROUP triples BY subject) 
   GENERATE group AS subject, 
   triples.(predicate, object) AS stmts;
```
Pig Extensions

LOAD - local

- load plain N3 files is supported natively by Pig
  - we use a UDF for tokenizing text lines to triples
- `RDFFileLoad` macro

```
DEFINE RDFFileLoad(file) RETURNS T {
    lines = LOAD '$file' AS (txt: chararray);
    $T = FOREACH lines
        GENERATE FLATTEN(pig.RDFize(txt))
    AS (subject, predicate, object);
}
triples = RDFFileLoad("hdfs:///rdf-data.nt");
```

- load TUPLIFIED dataset using BinStorage

```
rdf_tuples = LOAD "rdf-data.dat" USING BinStorage() AS
   (subject: bytearray,
    stmts: bag{t:(predicate: bytearray, object: bytearray)});
```
Pig Extensions

LOAD - remote
- run SPARQL query on endpoints
- filter remote data beforehand
  - depends on user query

```python
raw = LOAD "http://endpoint.org:8080/sparql"
    USING SPARQLLoader("SELECT * WHERE { ?s ?p ?o }")
    AS (subject, predicate, object);
```

```python
--> "hdfs:///rdf-data.nt"
```

```python
raw = RDFFileLoad("hdfs:///rdf-data.nt");
```

- materialize (remote) data
- share across queries
- could be used for frequent intermediate results
Pig Extensions

BGP Support

```
result = FILTER triples BY
{ ?s <geo:lat>  ?o1 .
  ?s <geo:long> ?o2 }
```

- extended `FILTER` operator
- hide internal details of BGP processing
- implementation depends on input schema

- implemented as language extension
  - internal operators stay unchanged
- rewriting step in Pig parser - transformation to native Pig code
- Pig compiler for optimization
Rewriting

Example - FILTER

\[
\text{out}(s, \{(p, o)\}) = \text{FILTER in}^{(s,\{(p,o)\})} \text{ BY } \{ 'value' ?p ?o \}; \\
\text{=>} \\
\text{out}^{(s,\{(p,o)\})} = \text{FILTER in}^{(s,\{(p,o)\})} \text{ BY } s == 'value';
\]

Example - FILTER (non-grouping component)

\[
\text{out}^{(s,\{(p,o)\})} = \text{FILTER in}^{(s,\{(p,o)\})} \text{ BY } \{ ?s ?p 'value' \}; \\
\text{=>} \\
\text{tmp}^{(s,\{(p,o),cnt\})} = \text{FOREACH in}^{(s,\{(p,o)\})} \{ \\
\hspace{1em} t = \text{FILTER stms BY } o == 'value'; \\
\hspace{1em} \text{GENERATE } *, \text{ COUNT}(t) \text{ AS cnt}; \}; \\
\text{out}^{(s,\{(p,o),cnt\})} = \text{FILTER tmp}^{(s,\{(p,o),cnt\})} \text{ BY } \text{cnt} > 0;
\]
Rewriting

Example - STAR JOIN (on grouping component)

\[ \text{out}^{(s,\{(p,o)\})} = \text{FILTER in}^{(s,\{(p,o)\})} \text{ BY } \{ TP_1. \ldots TP_N. \}; \]

\[ \Rightarrow \]

\[ \text{tmp}^{(s,\{(p,o)\},cnt1,\ldots,cntN)} = \text{FOREACH in}^{(s,\{(p,o)\})} \{ \]

\[ t1 = \text{FILTER stmts BY } p ==' p'_1; \]

\[ \ldots \]

\[ tN = \text{FILTER stmts BY } p ==' p'_N; \]

\[ \text{GENERATE } *, \text{ COUNT}(t1) \text{ AS cnt1}, \]

\[ \ldots, \text{ COUNT}(tN) \text{ AS cntN}; \} ; \]

\[ \text{out}^{(s,\{(p,o)\},cnt1,\ldots,cntN)} = \text{FILTER tmp}^{(s,\{(p,o)\},cnt1,\ldots,cntN)} \]

\[ \text{ BY cnt1} > 0 \text{ AND } \ldots \text{ AND cntN} > 0; \]

\[ \text{result} = \text{FILTER triples BY} \]

\[ \{ \ ?s <\text{geo:lat}> \ ?o1 . \]

\[ ?s <\text{geo:long}> \ ?o2 \}; \]

\[ \text{tmp} = \text{FOREACH triples} \{ \]

\[ t1 = \text{FILTER stmts BY predicate == "<\text{geo:lat}>";} \]

\[ t2 = \text{FILTER stmts BY predicate == "<\text{geo:long}>";} \]

\[ \text{GENERATE } *, \text{ COUNT}(t1) \text{ AS cnt1, COUNT}(t2) \text{ AS cnt2;} \]

\[ \text{result} = \text{FILTER tmp BY cnt1} > 0 \text{ AND cnt2} > 0; \]
Evaluation

Self-Join

- scripts manually rewritten
- Dataset: 8GB, 54 mio statements
- Hadoop Cluster: 8 Nodes, Pig 0.12

```
triples = RDFLoad("hdfs:///eventful.nt");
result = FILTER triples BY
    { ?s <geo:lat> ?o1 .
      ?s <geo:long> ?o2 }
```

![Graph showing time [seconds] vs partitions]
Evaluation
FILTER (non-grouping component)

```r
triples = RDFLoad("hdfs:///eventful.nt");
result = FILTER triples BY { ?s ?p "Metallica" };
```
Conclusion

- native Pig data model not suitable for RDF data
- combination of self-joins and filter needed
- support for BGP in Pig Latin
- join with remote data
- rewriter produces native Pig code
  - use Pig optimizer
- allows easier and faster linked data processing in Pig
- foundation for cost-based optimizer
  - materialized (intermediate) results