For the DISTINCT Clause of SPARQL Queries

Medha Atre
Dept. of Computer Science and Engineering
Indian Institute of Technology, Kanpur, India
medha.atre@gmail.com

ABSTRACT

Evaluating SPARQL queries with the DISTINCT clause may become memory intensive due to the requirement of additional auxiliary data structures, like hash-maps, to discard the duplicates. DISTINCT queries make up to 16% of all the queries (e.g., DBPedia), and thus are non-negligible. In this poster we propose a novel method for such queries, by just manipulating the compressed bit-vector indexes called *BitMats*, for *acyclic* basic graph pattern (BGP) queries.

1. PRELIMINARIES

SPARQL[1], the standard query language for RDF[2], provides various query constructs. The DISTINCT clause eliminates duplicates from the results. SPARQL *basic graph pattern* (BGP) queries with the DISTINCT clause make up to 16% of the DBPedia logs, and hence are non-negligible [4]. Consider the following BGP query over an RDFized version of a movie database like IMDB, which is asking for all the *distinct* pairs of the actors (?a) and their directors (?d).

```
SELECT DISTINCT ?a ?d WHERE {
    ?m rdf:type :Movie .
    ?m :hasActor ?a .
    ?m :hasDirector ?d .}
```

:UmaThurman has acted in three movies directed by *:QuentinTarantino*, they are *:PulpFiction*, *:KillBillVol1*, and *:KillBillVol2*. Without the DISTINCT clause, we would get three copies of (*:UmaThurman*, *:QuentinTarantino*) as the variable bindings for (?a, ?d) in the results, but the DISTINCT clause ensures that we get only one copy.

SPARQL algebra allows an arbitrary number of variables in the DISTINCT clause. For multiple variables in the DISTINCT clause (like in our example), the distinct values are *composite* of bindings of those variables, e.g., (*:UmaThurman*, *:QuentinTarantino*) is distinct from (*:UmaThurman*, *:WoodyAllen*), although they both share *:UmaThurman*. If the variables in the DISTINCT clause appear in different triple patterns (TPs), like in our example, we have to generate intermediate variable bindings of the other variables not in the DISTINCT clause, and discard them later, thus creating more memory overhead. E.g., we first have to generate bindings of all three variables (?m, ?a, ?d), project out only bindings of (?a, ?d), and then pass the (?a, ?d) pairs through the DISTINCT filter to remove duplicates. Hence for an arbitrary number of variables in the WHERE and DISTINCT clauses, evaluation of a query becomes memory intensive as the number of variables grows.

Our technique, proposed for *acyclic* BGP queries with an arbitrary number of variables in the WHERE and DISTINCT clauses, avoids the use of any auxiliary data structures like hash-maps to remove duplicates. It is built on two main concepts: (1) *acyclicity* of BGP queries and (2) *minimality* of triples, introduced earlier in [1, 3, 5].

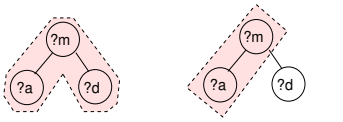

Figure 1: GoV and MCS

Acyclic queries: Borrowing the concepts introduced in [1], we first build a graph of variables (GoV) in a BGP query as follows. Every variable is a node, and two variable nodes have an undirected edge between them if they appear together in a TP in the query. The undirected edge, in a way, represents a TP[3]. If this GoV is acyclic, the BGP query is said to be *acyclic*. This concept of acyclicity is similar to the concept of *acyclic SQL join queries* [3, 5].

Minimality: A TP in a BGP query is said to have *minimal* triples, if every triple creates one or more variable bindings in the final results, and no triple gets eliminated as a result of a join. For an *acyclic* query, we can *prune* the initial set of triples associated with each TP in the query to *minimal* using *semi-joins*. We have described this pruning procedure for acyclic BGP queries in [1, 2]. It is equivalent to the process described in [3, 5] for the acyclic SQL inner-joins.

2. OUR TECHNIQUE

Before presenting our technique, we review some important properties of Boolean matrix multiplication (BMM) of a graph's adjacency matrices. Consider the adjacency matrices of only predicates (edge labels) *:hasActor* and *:hasDirector* in IMDB's RDF graph. If we do a BMM of the adjacency matrices of *transpose* of *:hasActor* with *:hasDirector*, the resultant matrix gives all the *distinct* pairs of nodes that have *at least one* 2-length *undirected* path with edge labels *:hasActor–:hasDirector* between them.

If we remove the DISTINCT clause from our SPARQL query in Section 1, we get three copies of (*:UmaThurman,*

[1] http://www.w3.org/TR/rdf-sparql-query/

[2] http://www.w3.org/TR/2014/REC-n-triples-20140225/

[3] For simplicity and space constraints, we defer the discussion of TPs with three variables for the future work, because such TPs are rare.

WWW'16 Companion, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4144-8/16/04.
http://dx.doi.org/10.1145/2872518.2889390.

:QuentinTarantino) for the (?a, ?d) variables pair. This means there are three undirected 2-length paths between *:UmaThurman* and *:QuentinTarantino*, with edge labels *:hasActor–:hasDirector*, and they go through *:KillBillVol1*, *:KillBillVol2*, and *:PulpFiction* each, in the RDF graph. However, the DISTINCT clause selects just one copy because there is *at least one* such 2-length path with edge labels *:hasActor–:hasDirector*. We observe an important property from this as follows.

PROPERTY 1. *For any two non-adjacent variables ?x, ?y, connected via an undirected path* $?x, ?m_1, ?m_2 ... ?m_k, ?y$ *in a GoV, DISTINCT values of (?x, ?y) are those bindings of (?x, ?y) which have at least one undirected path between them in the RDF graph, that goes through the bindings of the intermediate variables* $?m_i$, $1 \leqslant i \leqslant k$ *on the path.*

We make use of this equivalence between Boolean matrix multiplication (BMM) and the DISTINCT clause to present our method. We use *BitMats*, which are same as the *compressed* adjacency matrices of each predicate (edge label) of an RDF graph [2]. Thus each TP in the query has a BitMat associated with it. Given an *acyclic* BGP query, we first run Algorithms 3.1 and 3.2 in [1], which *prune* the triples in the BitMats to *minimal* (ref. Lemma 3.3 in [1]). Next, we identify a *minimal covering subgraph* (MCS) of GoV, which *covers* all the paths between the variables in the DISTINCT clause[4]. For the query in Section 1, MCS is the *entire* GoV itself, because ?a and ?d are connected via ?m. But if the query was "SELECT DISTINCT ?m ?a", the MCS would have been just two nodes ?m and ?a (see Figure 1). Note that an MCS may have nodes that do not appear in the DISTINCT clause, like ?m in our example. Our aim is to remove all such *non-required* ?m variables from MCS without affecting the correctness of the results.

Eliminate non-required variables: We assume an MCS with an arbitrary number of required variables (those appearing in the DISTINCT clause), and an arbitrary number of non-required variables (those not in the DISTINCT clause). Denoting the BitMat associated with a TP of type "*?x :someEdgeLabel ?y*" as $BM(?x, ?y)$, and $BM(?y, ?x) = transpose(BM(?x, ?y))$, we remove the non-required variables methodically using the following algorithm.

(1) Generalizing the variable names, choose any 2-length path ?x–?y–?z in the MCS, such that ?x or ?z or both $\in$ DISTINCT, but ?y $\notin$ DISTINCT. Do a BMM, $BM(?x, ?y) \times BM(?y, ?z) = BM(?x, ?z)$. Add edge (?x, ?z).

(2) If ?y has only ?x and ?z as its neighbors, remove edges (?x, ?y), (?y, ?z), BitMats $BM(?x, ?y)$, $BM(?y, ?z)$, and node ?y. Else remove edge (?x, ?y) or (?y, ?z), but not both, and remove the respective $BM(?x, ?y)$ or $BM(?y, ?z)$, but not both. If the MCS has $\geqslant$1 non-required variables go to Step 1. Else terminate.

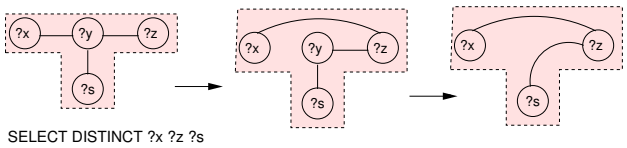

Figure 2: Remove ?y from MCS using the algorithm

In Figure 2, we have shown a sample MCS with three required variables, ?x, ?z, ?s and a non-required variable ?y. The figure also shows an evolution of this MCS to eliminate ?y using the above algorithm. Intuitively, we eliminate all the intermediate *non-required* variables, by establishing direct correlations between the bindings of the required variables, that were maintained through the non-required variable bindings in the original MCS. E.g., when ?x and ?z have a path through ?y in MCS, bindings of ?x and ?z are correlated through ?y. When we do a BMM, $BM(?x, ?y) \times BM(?y, ?z) = BM(?x, ?z)$, we establish a direct correlation between the bindings of the (?x, ?z) pair, and eliminate the need of having ?y as an intermediary.

This algorithm is *monotonic* – at the end of one iteration of Steps 1 and 2, the edges, nodes, and BitMats in an MCS remain the same or become fewer than before. We gradually reduce the degree of the non-required variables, and eventually eliminate them when their degree is 2. Thus this algorithm always converges when all the non-required variables are eliminated from the MCS. Also note that the total BitMats at the end of the algorithm are always *fewer* than the original BitMats in the query – note that in Step 2, we *remove two* BitMats while creating a *new one* when the degree of the non-required variable is 2, and when the degree of the non-required variable is >2, we create one new BitMat and remove one. Hence eventually we are left with *fewer* BitMats – thus reducing the memory requirements.

We join these BitMats with each other using the *multiway-pipelined-join* procedure (Algorithm 5.4) in [1]. Note that we can *carve* out an MCS from the original GoV, because the query is *acyclic*, and each TP in the query has *minimal* triples after the pruning process (Algorithms 3.1, 3.2 in [1]).

Space and time complexity: The BitMat indices (a.k.a. adjacency matrices) formed over an RDF graph are typically sparse and are kept compressed (ref. [2]). Hence in practice, the space complexity of BitMats is much lesser than the worst case $O(n^2)$ bound. We performed our experiments presented in [1, 2] over complex BGPs (not with the DISTINCT clause) involving up to 13 BitMats over an RDF graph of 1.33 billion triples, on commodity machines of 4–8 GB memory. Also we use methods like *fold–unfold* (ref. [2]) to manipulate the compressed BitMats without uncompressing them. Thus we conjecture that in practice, the time complexity of a BMM would be much lesser than the worst case bound of $O(n^3)$.

Cyclic queries: For the cyclic BGP queries, although we can use the same pruning procedure (Algorithms 3.1, 3.2 in [1]), the minimality of triples cannot be guaranteed. Hence for cyclic BGP queries with the DISTINCT clause, we cannot identify an MCS from GoV, and cannot use this memory optimization technique. For cyclic queries, we have to resort to using additional auxiliary data structures, such as hash-maps, to remove duplicates.

[4]Since GoV is acyclic any two nodes in it have a unique undirected path between them.

3. REFERENCES

[1] M. Atre. Left Bit Right: For SPARQL Join Queries with OPTIONAL Patterns (Left-outer-joins). In *SIGMOD*, 2015.

[2] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit"loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *WWW*, 2010.

[3] P. A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM Journal of Computing*, 10(4), 1981.

[4] M. A. Gallego, J. D. Fenández, M. A. Martínez-Prieto, and P. de la Fuente. An Empirical Study of Real-World SPARQL Queries. In *USEWOD workshop at WWW*, 2011.

[5] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.