Caching Intermediate Result of SPARQL Queries

Mengdong Yang
Southeast University
Nanjing, China
mdyang@seu.edu.cn

Gang Wu
Northeastern University
Shenyang, China
wugang@ise.neu.edu.cn

ABSTRACT
The complexity and growing scale of RDF data has made data management back end the performance bottleneck of Semantic Web applications. Caching is one of the ways that could solve this problem. However, few existing research projects focus on caching in RDF data processing. We present an adaptive caching scheme that caches intermediate result of basic graph pattern SPARQL queries. Benchmark test results are provided to illustrate the effectiveness of our caching scheme.

Categories and Subject Descriptors
H.2.4 [**Systems**]: *Query processing*

General Terms
Management, Performance.

Keywords
Intermediate Result, SPARQL, Cache

1. INTRODUCTION
SPARQL is a standard RDF query language for RDF. The directed, labeled data model of RDF makes the query processing of SPARQL more complex. Although diverse measures including indexing, query optimizations, parallelization, etc., are employed to speed up query processing in mainstream RDF data management approaches, few of them concentrate on caching. Caching scheme is usually designed based on spatial/temporal locality principle to improve system performance with reasonable space expense. Caching techniques have been widely developed and applied in both RDBMS and RDBMS-based web 2.0 applications. Server-side caching systems based on application object caching like Memcached are widely applied. Other systems employ a client-side semantic data caching approach [1].

As for Semantic Web applications, [2] builds a proxy cache layer between web application and RDF repository where both query caching and application object caching are employed. There is some research on a similar topic: [3], [4] and [5] automatically materialize frequent join paths based on statistics information. [6] allows manual build of materialized views according to application context. However, materialized view isn't a caching scheme after all because it doesn't have a replacement principle and has to be built before issuing query on the database.

2. APPROACH
Suppose a scenario that two SPARQL queries Q_1 and Q_2 are issued sequentially, and they have their Algebra Expression Trees (AETs) for query execution as shown in Figure 1. Take the AET of Q_1 for example, the query engine first tries to get the output of N_1, finding that the output of N_1 depends on N_2 and N_3. So it evaluates N_2 and N_3, finding the output of N_2 further depends on N_4 and N_5. Hence, the query engine evaluates N_4 and N_5, and then evaluates N_2 with their outputs. After evaluating N_3, N_1 is evaluated together with the output of N_2 and N_3, and eventually Q_1 is answered. The answering of Q_2 has a similar process. Note that the AETs of Q_1 and Q_2 have common sub AET annotated by dash line. Observing that adjacent SPARQL queries may have such common structure, we cache the result of sub AETs for future reuse. If the result of P_1 is cached, the execution of P_2 can be omitted, thus improves query answering performance.

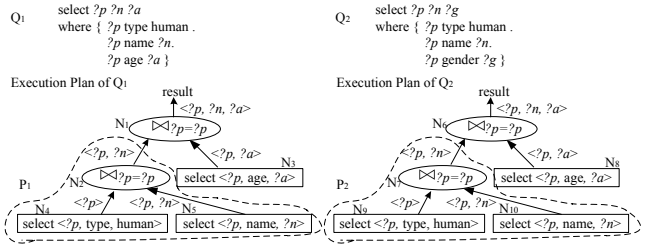

Figure 1. Two SPARQL Queries with Common Part

2.1 AET Normalization and Identification
The first thing we need to do in caching the result of an AET is normalization. Literally different SPARQL queries may have the same query logic due to variable naming (e.g. use *?person* in a query for a person entity but use *?p* in another one). Therefore, AET normalization needs to be performed to convert those logically equal AETs to a uniform one. In our normalization approach, a pre-order traverse in the AET is performed. During this process, variables are labeled with sequential integers i.e. 1, 2, 3… A normalized AET can identify a specific query execution plan with unique query logic regardless of variable naming. To specify the equality of two existing AETs, we present a simple recursive definition of equality of two AETs:

Definition 1. Given two AETs T_A and T_B with their corresponding root node R_A and R_B, then T_A is equal to T_B if and only if the left sub AET of R_A is equal to the left sub AET of R_B and the right sub AET of R_A is equal to the right sub AET of R_B.

Base on the above definition, we can serialize a normalized AET to generate an identifier, which can identify the result of the AET. A normalized AET is converted into a function-call style expression. Take the AET of Q_1 in Figure 1 for example, its sub AET P_1 would have a serialized expression like

WWW 2011, March 28–April 1, 2011, Hyderabad, India.
ACM 978-1-4503-0637-9/11/03.

Table 1. Space Consumption of the Caching System

Original=Original Sesame Repository Size, Cache=Cache Repository Size, both in MB

	LUBM(10)	LUBM(100)	LUBM(1000)	SP²Bench 2.5M	SP²Bench 10M	SP²Bench 40M
Original	112.60	1.18GB	11.77GB	282.89	1.08GB	4.30GB
Cache	21.77KB	1.77	17.24	10.11	47.62	60.66

$\bowtie_{?p=?p}$(<?1, type, human>, <?1, name, ?2>).

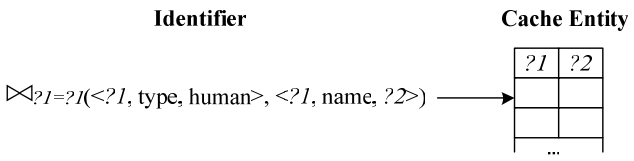

Figure 2. Cache Entity Storage in the Cache Repository

2.2 Cache Build and Utilization

With the identifier, we can do our cache work. We perform the caching by adding some extra work steps to the query engine during query answering. Before evaluating the output of a node N in the AET, the query engine first computes the serialized expression S of the sub AET, whose root node is N, and looks up S in the cache repository. If a cache entity is hit, the query engine directly accesses it avoiding the evaluation work from N downward. Otherwise the query engine evaluates the output O of N in normal way and stores S→O in the cache repository. Cache entity storage in the cache repository is shown in Figure 2.

Note that this process can be performed multiple times during the execution of one SPARQL query. For complex basic graph pattern SPARQL queries, its corresponding AET can have complex structure as well. Such AETs have different sub AETs at different tree level. Cache hit may happen less probably in a node N_H at a higher level because the sub AET which takes N_H as the root node takes a larger portion of the global AET. But the output size of such an AET is usually small. Plus the fact that a cache hit at a higher level contributes more to query answering performance, so caching such AETs is worthwhile. Oppositely, a small, simpler sub AET may have a big output, which makes the caching of such AETs more space-consuming. But simple sub AETs stand for simple graph patterns, and they are more likely to appear in different query contexts.

3. EXPERIMENTAL EVALUATION

We implement our caching scheme on Sesame. The system has a two-level (memory-disk) cache architecture. LRU cache replacement strategy is employed. Two mainstream benchmarks LUBM and SP²Bench are applied to test the performance of our caching system. The following metrics are measured to evaluate the performance of the system:

1. Temporal Improvement. Three values are compared in this metric: 1) time consumption of query evaluation with conventional means without our caching scheme; 2) time consumption of query evaluation with our cache scheme at cache warm-up phase, when few cache hits happen; 3) time consumption of query evaluation with our cache scheme at warm cache phase, when cache hit happens relatively more frequently. Temporal improvement is shown in Figure 3.

2. Space Consumption. Space Consumption evaluates how much disk space our caching scheme consumes, shown in Table 1.

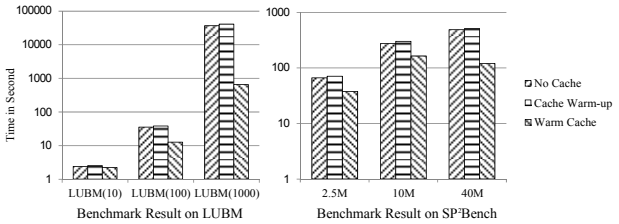

Figure 3. Benchmark Result on LUBM and SP²Bench

3. Cache Hit. As the engine evaluates the AET top-down, cache hit may happen at any level of it. We here define that a Cache Hit happens at the evaluation of a query whenever a cache item is hit at any level of the corresponding AET. For LUBM, no cache hit happen at cache warm-up phase because there are few associations between LUBM test queries. SP²Bench test queries have more similarities, so some cache hits happen at cache warm-up time on SP²Bench. At warm cache phase the system has a relatively good cache hit rate.

4. FUTURE WORK

Future work includes: 1) Considering AET heterogeneity. 2) Supporting more SPARQL operations. Only join is supported at present (basic graph pattern queries). 3) Considering context information and mining association triples.

5. ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant No. 60903010, the Natural Science Foundation of Jiangsu Province under Grant No. BK2009268, and the Key Laboratory of Advanced Information Science and Network Technology of Beijing under Grant No. XDXX1011.

6. REFERENCES

[1] Shaul Dar, Michael J. Franklin, Bjorn T. Jonsson, Divesh Srivastava, and Michael Tan. Semantic Data Caching and Replacement. In Proc of VLDB 1996.

[2] Michael Martin, Jorg Unbehauen, and Soren Auer. Improving the performance of Semantic Web Applications with SPARQL Query Caching. In Proc of ESWC 2010.

[3] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In VLDB, 2007.

[4] E. I. Chong et al. An Efficient SQL-based RDF Querying Scheme. In VLDB, 2005.

[5] Thomas Neumann and Gerhard Weikum. RDF-3X: A RISC-Style Engine for RDF. Proc. VLDB Endow., 2008.

[6] Roger Castillo, Christian Rothe, and Ulf Leser. RDFMatView: Indexing RDF Data for SPARQL Queries. Technical Report