Information Systems 55 (2016) 54–72

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Efficient processing of enumerative set-based queries

Guoping Wang [a], Chee-Yong Chan [b,*]

[a] Shannon Lab, Huawei, No. 360, Jiangshu Road, Binjiang District, Hangzhou City, Zhejiang Province 310051, China
[b] Department of Computer Science, School of Computing, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417, Singapore

ARTICLE INFO

Article history:
Received 22 September 2014
Received in revised form
7 July 2015
Accepted 11 August 2015
Recommended by: J. Van den Bussche
Available online 4 September 2015

Keywords:
Set-based queries
Multi-query optimization
Relational database systems

ABSTRACT

Many applications often require finding sets of entities of interest that meet certain constraints. Such set-based queries (SQs) can be broadly classified into two types: *optimization SQs* that involve some optimization constraint and *enumerative SQs* that do not have any optimization constraint. While there has been much research on the evaluation of optimization SQs, there is very little work on the evaluation of enumerative SQs, which represent the most fundamental fragment of set-based queries. In this paper, we address the problem of evaluating enumerative SQs using RDBMS. While enumerative SQs can be expressed using SQL, existing relational engines, unfortunately, are not able to efficiently evaluate such queries due to their complexity. In this paper, we propose a novel evaluation approach for enumerative SQs. Our experimental results on PostgreSQL demonstrate that our proposed approach outperforms the conventional approach by up to three orders of magnitude.

1. Introduction

Many applications often require finding sets of entities of interest that meet certain constraints. Such set-based queries (SQs) can be broadly classified into two types: *optimization SQs* that involve some optimization constraint and *enumerative SQs* that do not have any optimization constraint. For example, consider a relation *R(id ,type,city, price,duration,rating)* shown in Table 1 that stores information about various places of interest (POI), where *type* refers to the category of the POI (e.g., museum, park), *duration* refers to the recommended duration to spend at the POI and *rating* refers to the average visitors' rating of the POI. Suppose that a tourist is interested to find all tour trips near Shanghai consisting of POIs that meet the following constraints: the trip must include both Shanghai (S.H.) and Suzhou (S.Z.) cities, the trip must include POIs of type museum and park, and the total duration of the trip should be between 6 and 10 h. There are three packages that satisfy the above query: $\{t_1, t_2\}$, $\{t_1, t_2, t_3\}$ and $\{t_1, t_2, t_5\}$. The above is an example of an enumerative SQ to find all sets of POIs that satisfy the given constraints. If the query had an additional constraint to minimize the total cost of the tour package, it would become an optimization SQ.

As another example, suppose that an employer is looking to hire a team of language translators for a project that meet the following constraints: each team member must know English; the team collectively must be knowledgeable in French, Russian, and Spanish; the team consists of at least two translators; and the total monthly salary of the team is no more than $50K. Consider a relation *Translator (id,location,salary,english,french,russian,spanish)* that stores information about language translators available for hire, where the four binary valued attributes *english*, *french*, *russian*, and *spanish* indicate whether a translator is knowledgeable in

* Corresponding author. Tel.: +65 65166736.
E-mail addresses: wang.guoping@huawei.com (G. Wang), chancy@comp.nus.edu.sg (C.-Y. Chan).

http://dx.doi.org/10.1016/j.is.2015.08.005
0306-4379/

Table 1
An example relation *R*.

id	type	city	price	duration	rating
t_1	Museum	S.H.	50	4	7
t_2	Park	S.Z.	70	3	5
t_3	Museum	S.Z.	60	3	8
t_4	Shopping	S.H.	80	5	7
t_5	Shopping	H.Z.	90	2	9

the specific languages, *location* represents the translator's living place, and *salary* represents the translator's expected monthly salary. To browse through all the possible teams for hiring, the employer executes an enumerative SQ on the *Translator* relation.

Another application of enumerative SQs is in the area of set preference queries [1–3], which computes all sets of entities of interest that satisfy some preference function. Consider again our example on hiring translators. In addition to the previously discussed constraints, the employer could prefer to hire a team where (a) the team members are located close to one another and (b) their total salary is low. Thus, this set preference query is essentially a skyline [4] set-query to retrieve non-dominated teams where the members have close proximity and low total salary. The most general approach to evaluate skyline set-queries is to first enumerate all the candidate sets followed by pruning away the dominated sets. Although there has been recent work to integrate these two steps [3], such optimization is applicable only for restricted cases (e.g., when the sets are of fixed cardinality and the preference function satisfies certain properties); and is not applicable for queries such as our example query. Therefore, efficient algorithms to evaluate enumerative SQs are essential for the efficient processing of set preference queries.

There has been much research on evaluating optimization SQs where the focus is on heuristic techniques to compute approximately optimal or incomplete query results (e.g., [3,5–10]). However, to the best of our knowledge, there has not been any prior work on the evaluation of enumerative SQs. Enumerative SQs are essentially a generalization of conventional selection queries to retrieve a collection of sets of tuples (instead of a collection of tuples), and they represent the most fundamental fragment of set-based queries.

In this paper, we address the problem of evaluating enumerative SQs using RDBMS. For convenience, we refer to enumerative SQs as simply SQs in the rest of this paper.

While SQs can be expressed using SQL, existing relational engines, unfortunately, are not able to efficiently optimize and evaluate such queries due to their complexity involving multiple self-joins and/or view expressions. In this paper, we propose a novel evaluation approach for SQs. The key idea is to first partition the input relation into disjoint blocks based on the different combinations of constraints satisfied by the tuples and then compute the answer sets by appropriate combinations of the blocks. In this way, a SQ is evaluated as a collection of cross-product queries (CPQs). However, applying existing multiple query optimization (MQO) techniques for this evaluation problem is not effective for two reasons. First, the scale of the problem could be very large involving hundreds of CPQ evaluations. Existing MQO heuristics, which are mainly designed for optimizing a handful of queries, are not scalable for our problem. Second, as the queries here are CPQs (and not join queries), existing MQO techniques, which are based on materializing and reusing common subexpressions, is not effective as the cost of materialization exceeds the cost of recomputation.

In this paper, we make three key contributions to the study of SQs. First, we experimentally show that conventional RDBMS are unable to efficiently evaluate SQs. Second, we propose a novel approach to evaluate SQs in terms of a collection of CPQs. Our approach includes both effective MQO heuristics designed to optimize a large collection of CPQs and efficient evaluation techniques that exploit the properties of set predicates in the SQs. Third, we demonstrate the effectiveness of our approach with a comprehensive experimental evaluation on PostgreSQL which shows that our approach outperforms the conventional SQL-based solution by up to three orders of magnitude.

The rest of this paper is organized as follows. In Section 2, we formally introduce set-based queries (SQs) and a fragment of SQs referred to as basic SQs (BSQs). Section 3 presents some preliminaries. Section 4 presents a baseline SQL-based solution to evaluate SQs. Section 5 presents our main-memory based approach to evaluate BSQs, and Section 6 extends the approach to evaluate BSQs on disk-based data. In Section 7, we extend our approach to evaluate general SQs beyond BSQs. Section 8 presents an experimental performance evaluation of the proposed techniques. Section 9 presents related work, and we conclude our paper in Section 10.

2. Set-based queries

In the simplest form, a *set-based query* (*SQ*) *Q* is defined by an input relation *R*, which represents a collection of entities of interest, and an input set of predicates *P* on *R*. The query's result is a collection of all the subsets of *R* such that each subset satisfies the predicates in *P*.

For convenience, we introduce an extended SQL syntax to express SQs more explicitly. The example SQ in Section 1 can be expressed by the following extended SQL query.

Q_{poi}: SELECT *
FROM SET(*R*) S
WHERE v_1 in S AND v_2 in S
AND v_3 in S AND v_4 in S
AND v_1. city = S. H. AND v_2. city = S. Z.
AND v_3. type = museum AND v_4. type = park
AND $6 \leq$ SUM(S. duration) ≤ 10

The "SET(R) S" in the from-clause specifies *S* as a *set variable* whose value is a subset of tuples in relation *R*. Each of the predicates of the form "v_i in S" specifies v_i as a *member variable* representing a member of the set variable *S*. Note that the values of the member variables are not necessarily distinct. Each of the next four predicates