An Efficient Sheet Partition Technique for Very Large Relational Tables in OLAP

Sung-Hyun Shin[1], Hun-Young Choi[2], Jinho Kim[2], Yang-Sae Moon[2], and Sang-Wook Kim[1]

[1] College of Information & Communications, Hanyang University, Korea
[2] Department of Computer Science, Kangwon National University, Korea
{shshin, hychoi, jhkim, ysmoon}@kangwon.ac.kr, wook@hanyang.ac.kr

1 Introduction

Spreadsheets such as Microsoft Excel are OLAP (On-Line Analytical Processing) [2] applications to easily analyze complex multidimensional data. In general, spreadsheets provide grid-like graphical interfaces together with various chart tools [4,5]. However, previous work on OLAP spreadsheets adopts a naive approach that directly retrieves, transmits, and presents all the resulting data at once. Thus, it is difficult to use the previous work for very large relational tables with millions of rows or columns due to the communication and space overhead.

In this paper we propose an efficient spreadsheet-based interface to incrementally browse the query result on very large relational tables. The proposed interface exploits the *sheet partition technique* that selectively browses small parts of the resulting table. Our sheet partition technique first divides a large resulting table into many small-sized sheets, called *partitions*, and then browses the partitions one by one according to the user's request. More precisely, the technique works as follows: (1) the client, i.e., the user requests a query with a specific column (or row); (2) the server stores the query result on the given column (or row) as the temporary data; (3) the server provides an initial partition, which are constructed by using the initial column (or row) range of temporary data; and (4) the client repeatedly interacts with the server to browse more partitions. Since the sheet partition technique enables us to use a few small-sized partitions instead of a single large-sized sheet, we can reduce the communication and space overhead. Also, we can easily analyze large relational tables by exploiting the concept of *divide and conquer* in spreadsheet applications.

2 The Proposed Sheet Partition Technique

There have been many efforts to provide spreadsheet-like views for easy analysis on multidimensional data. In [5], Witkowski et al. defined spreadsheet-typed tables in relational databases by extending standard SQL statements. In [6], Witkowski et al. also proposed an Excel-based analysis method to exploit various powerful Excel functions in handling the original relational data. These works, however, do not consider very large relational tables with millions of rows or

R. Cooper and J. Kennedy (Eds.): BNCOD 2007, LNCS 4587, pp. 176–179, 2007.

columns [1] in presenting the tables as the spreadsheets. Therefore, in this paper we focus on the spreadsheet interface for the large relational tables.

The *sheet partition technique* enables us to analyze a few small-sized sheet partitioned from a single large-sized sheet. Figure 1 shows an overall working framework for our sheet partition technique. In Step (1) a user indicates column-based or row-based partitions by providing a specific column or row together with a query. In Step (2), the server evaluates the query and stores the result as temporary data. In Step (3), the server provides an initial partition to the user as the form of a spreadsheet. Finally, in Steps (4) and (5) the client repeatedly interacts with the server to get more partitioned sheets for the user's additional request. Likewise, by using the sheet partition technique, we do not need to handle a large-sized table at once, but we can incrementally process the table with a few small-sized partitions that are selected by the continuous user interactions.

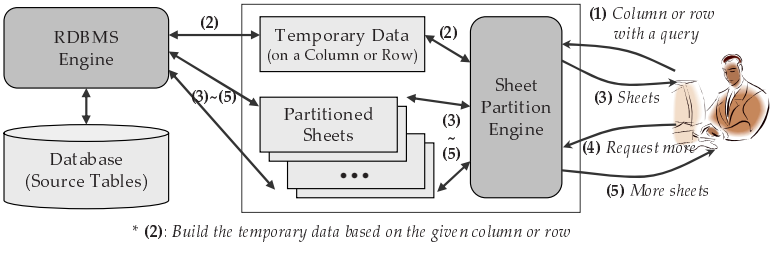

Fig. 1. An overall query processing framework using the sheet partition technique

We first explain the *column-based* sheet partition. This method uses a user-specified column to partition a large table into small sheets, and provides a few selected sheets as the resulting views. Figure 2 shows an algorithm *StoreAColumn* that retrieves the distinct values of the user-specified column and stores them in a temporary array, which will be used to construct the partitions based on the column values. In Line 1, we first declare a temporary array *ColValues*[1..*count*] to store the values of the given column. Here, *count* is the total number of distinct values of the column. In Line 2, we then declare *Cursor* as the `select` statement that retrieves values of the specific column. Since the column may have duplicate values, we explicitly specify the quantifier `distinct`. In Lines 3 to 6, we finally store the resulting values obtained by *Cursur* in *ColValues*[]. Eventually, the array *ColValues*[] contains the distinct values of the given column.

Figure 3 shows an algorithm *ColumnBasedSheet* that retrieves tuples whose column values in the user-specified range. We note that the algorithm uses the array *ColValues*[] obtained by *StoreAColumn* in Figure 2. The inputs to the algorithm are *from* and *to* of a sheet (Line 1). Using them as indexes of *ColValues*[] we select the tuples from the table (Line 2). We then store the tuples in `result` (Line 3), and return them as the partitioned sheet (Line 5). Therefore, using *ColumnBasedSheet* we can interactively and repeatedly access the partitioned sheets by changing the input range (i.e., *from* and *to*).