JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 5, No. 2, June 2011, pp. 111-120

Parallel Multithreaded Processing for Data Set Summarization on Multicore CPUs

Carlos Ordonez*, Mario Navas, and Carlos Garcia-Alvarado
Department of Computer Science, University of Houston, Houston, TX, USA
ordonez@cs.uh.edu, marionv@cs.uh.edu, cgarcia@cs.uh.edu

Abstract
Data mining algorithms should exploit new hardware technologies to accelerate computations. Such goal is difficult to achieve in database management system (DBMS) due to its complex internal subsystems and because data mining numeric computations of large data sets are difficult to optimize. This paper explores taking advantage of existing multithreaded capabilities of multicore CPUs as well as caching in RAM memory to efficiently compute summaries of a large data set, a fundamental data mining problem. We introduce parallel algorithms working on multiple threads, which overcome the row aggregation processing bottleneck of accessing secondary storage, while maintaining linear time complexity with respect to data set size. Our proposal is based on a combination of table scans and parallel multithreaded processing among multiple cores in the CPU. We introduce several database-style and hardware-level optimizations: caching row blocks of the input table, managing available RAM memory, interleaving I/O and CPU processing, as well as tuning the number of working threads. We experimentally benchmark our algorithms with large data sets on a DBMS running on a computer with a multicore CPU. We show that our algorithms outperform existing DBMS mechanisms in computing aggregations of multidimensional data summaries, especially as dimensionality grows. Furthermore, we show that local memory allocation (RAM block size) does not have a significant impact when the thread management algorithm distributes the workload among a fixed number of threads. Our proposal is unique in the sense that we do not modify or require access to the DBMS source code, but instead, we extend the DBMS with analytic functionality by developing User-Defined Functions.

Category: Embedded computing

Keywords: Algorithms; Data mining; Multicore CPU; Table scan; Thread; DBMS

I. INTRODUCTION

Computer hardware is constantly evolving with faster CPUs and larger disks every year. Currently, the number of cores per CPU is expected to double every two years. Thus, database management systems (DBMSs) face new challenges in exploiting existing cores, available for parallel multithreaded processing [1, 2]. However, developing multithreaded algorithms is a complex problem due to the difficulty of managing and balancing the workload among a large number of threads, as well as synchronizing them.

Parallel multithreaded processing is especially valuable for data mining and statistical computations due to the fact that most data processing is translated into a large number of mathematical CPU operations that generally can be processed in parallel [3-6]. Moreover, data mining algorithms require efficient I/O mechanisms when processing large data sets, in which it is preferable to interleave mathematical processing with a full table scan [7, 8]. Unfortunately, most hardware improvements do not accelerate access to secondary storage (hard disk technology) and therefore I/O processing remains a performance bottleneck. This is particularly important when aggregate operations are performed on large input tables. As a result, for practical purposes, aggregations become a bottleneck for data mining algorithms [7-9]. Therefore, improving the I/O mechanisms for aggregates in DBMSs represent an important research issue.

Open Access 10.5626/JCSE.2011.5.2.111 http://jcse.kiise.org

Received 26 October 2010, **Revised** 06 December 2011, **Accepted** 16 March 2011
*Corresponding Author

DBMS extensibility allows taking advantage of programming mechanisms that can extend its analytic functionality. However, most of the time, such approach is generally ignored by researchers in database systems, data mining and statistics. Despite the fact that a DBMS can indeed be extended with data mining processing [4, 7, 8], the majority of data mining processing is done outside the DBMS by exporting data samples to small flat files that are analyzed by efficient programs in languages such as C++ or Java [4], statistical tools such as Mat-lab, SAS, or R [5, 8], or more recently MapReduce [4, 6]. User-defined functions (UDFs) are a powerful extensibility mechanism [5, 10, 11]. Furthermore, some UDF implementations include application programming interfaces (APIs) [12] that enable the control of multithreaded processing. Aggregate user-defined functions (also called user-defined aggregates) are developed by implementing a well-defined sequence of processing steps; such steps provide enough information to manage the working threads in charge of processing database aggregations [13]. However, the user is often oblivious to parallel execution, and does not have control over optimizations that can be applied at the hardware level. With such issues in mind, our contributions are mostly focused on accelerating data mining processing in a DBMS exploiting UDFs. However, our ideas can be applied on any database algorithm that requires efficient processing on large data sets. In this paper, we work on extending the DBMS to compute sufficient statistics that are fundamental for many data mining models: principal component analysis (PCA), linear regression, clustering, variable selection, among others [8, 14]. In addition, we study caching, efficient memory management and multithreaded processing in order to exploit multiple cores and large RAM memory. It is important to emphasize that our research studies efficiently interleave data set processing with a full table scan on a modern DBMS. This research is applied, for instance, in one-pass data mining algorithms, such as Naïve Bayes [6], or dimensionality reduction with PCA [5, 14], that can take advantage of sufficient statistics to compute the model, instead of reading the data set multiple times. These models can now be obtained more efficiently with a middleware layer (UDFs in our case) that maximizes the utilization of all cores in the CPU.

This paper is organized as follows. Section II compares our research with previous works. Section III introduces definitions, presents an overview of sufficient statistics and explains how they can be computed with UDFs. Section IV presents our main contributions. This section introduces algorithms and optimizations to compute aggregations exploiting multithreaded processing. Section V contains an experimental evaluation on a computer with a multicore CPU, comparing performance of different aggregation algorithms, integrated in a DBMS. Finally, Section VI presents the conclusions and directions of our future work.

II. RELATED WORK

There is a wide range of related works regarding efficiently using memory and parallel processing for database operations and data mining. Previous research has been done on the hardware level to exploit operations in main memory for fast database processing [1, 15]. Adibi et al. [1] evaluate the link discovery algorithms in a processing-in-memory (PIM) architecture. In this research, experiments on multithreading and in-memory processing are presented. Unlike our work, the latter algorithms were specifically coded and tested for their hardware architecture and cannot be extended to any configuration of hardware. Manegold et al. [15] propose data structure partitioning algorithms for query joins that optimize cache performance by memory access. As in our research, the authors seek to speed up the execution of a critical database operator by optimizing memory management. However, their modifications are in the database core modules that will require modifying the DBMS engine. In our work, we decided to extend the DBMS capabilities by exploiting the existing framework for multithreading and memory access, which can be incorporated to any current database system.

In a similar manner, aggregate operations have been previously approached with the use of multicore technology. Cieslewicz and Ross [9] analyze several factors of multithreading and caching. In this work, an adaptive aggregation algorithm is proposed to optimize access to L1 and L2 cache memory in order to minimize cache misses. The latter algorithm is successful, even with skewed data. Unfortunately, despite the fact that the latter algorithm can be extended to work on different CPU architectures, it requires a complete rewrite of low-level algorithms for managing multithreaded processing and aggregation operations. In contrast, our algorithm can be extended to perform more complex processing than just sufficient statistics. Cache performance of in-memory and block oriented aggregate operations is studied by Cieslewicz et al. [16]. The main difference between our proposal and previous research is that joins and aggregations are evaluated together to avoid cache misses by modifying the size of the buffer. Notice that we avoid join operations in our aggregation process. This is an important assumption because any required join operations for obtaining sufficient statistics is assumed to be performed in a pre-processing step. We also evaluate the performance of our algorithm by modifying the block size. Using a separate context per thread for data mining algorithms is proposed by Ghoting et al. [2]. Although there has been considerable work on exploiting the current hardware technology for optimizing database performance, our work goes further by optimizing the aggregation bottleneck.

Integrating data mining and statistical techniques into a DBMS has received little attention by the research community. However, aggregate functions have been used not only in probabilistic databases [17], but also to construct patterns in multi-relational data mining and online analytical processing (OLAP) [18, 19]. Aggregate UDFs are shown to be useful when implementing database algorithms [8, 20]. The main functionality needed to define an aggregate UDF for multithreading is identified in [21]. Furthermore, it has been pointed out that traditional cost-based optimizations cannot be applied when working with UDFs because they represent non-traditional database systems processing [22]. As a consequence, memory and core usage have to be managed by the user. Recently, there has been research on optimizing the computation of sufficient statistics by exploiting caching in RAM and sampling [23]. This work takes a step further by proposing specific changes to the DBMS aggregation algorithms and accelerating performance with asymmetric multithreaded processing.

Table 1. Example of an input data set X with $d = 3$; $n = 10$

i	X_1	X_2	X_3
1	18.58	20.39	13.70
2	31.69	91.18	21.51
3	45.00	18.81	49.83
4	47.93	88.34	93.59
5	40.36	12.82	9.33
6	37.14	14.85	87.20
7	30.94	61.83	37.66
8	36.03	9.04	46.66
9	88.57	83.79	80.27
10	66.99	10.49	46.90

III. DEFINITIONS AND PROCESSING IN A DBMS

A. Data Set Summary

Assume we have an input data set X, with d dimensions and n data points $X = \{x_1,..., x_n\}$. Table 1 gives an example. There are three data summaries that are essential for several statistical linear models [5, 8]: n, L and Q, given in Equation 1 and Equation 2. L is the linear sum of the d dimensions in X and is stored on a vector with d values. Since matrix Q is the quadratic sum of dimension cross-products of each point, it is $d \times d$ and it is symmetric (i.e., it suffices to compute its lower triangular submatrix).

The data set X is stored in a table inside the DBMS, which has a column for each dimension X_1, and one row for every data point. Therefore, the table to store X has schema $X(i, X_1,..., X_d)$, where i represents its primary key. Previous research has shown that only one table scan over the input data is needed to obtain the data summarization with sufficient statistics for several models [8]. More importantly, from a performance standpoint, sufficient statistics n, L and Q, are distributive [7]. Thus, they can be computed in parallel over different partitions of the data set, where the global sufficient statistics are given by the addition of sufficient statistics on each partition.

$$L = \sum_{i=1}^{n} x_i \tag{1}$$

$$Q = XX^T = \sum_{i=1}^{n} x_i \cdot x_i^T \tag{2}$$

B. SQL Queries and User-Defined Functions

The computation of n, L and Q can be expressed in terms of the SUM aggregation. A single SQL statement is used to calculate all values of the summary matrices. Furthermore, notice that Q is a symmetric matrix, so it is enough to compute only the upper or lower triangular elements. An efficient SQL query for obtaining sufficient statistics, requiring a single table scan, is shown on Fig. 1. Notice this SQL query computes only one half of Q because Q is symmetric. Further details on how to compute n, L and Q for horizontal and vertical layouts of the data set can be found in [8].

```
SELECT SUM(1.0)                              /* n */
      ,SUM(X1),SUM(X2),...,SUM(Xd)   /* L */
      ,SUM(X1 * X1)                          /* Q */
      ,SUM(X2 * X1),SUM(X2 * X2)
      ...
      ,SUM(Xd * X1),SUM(Xd * X2),...,SUM(Xd * Xd)
FROM X;
```

Fig. 1. Efficient SQL query for data set summarization.

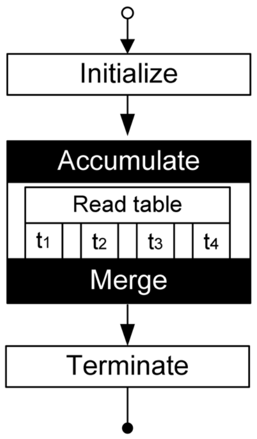

Fig. 2. Aggregate User-Defined Function (UDF) steps.

Aggregate user-defined functions (aggregate UDFs) give users the capability to extend the functionality of DBMS. The set of steps (see Fig. 2) that must be implemented by the user to program the aggregation are [5]:

1) *Initialize:* Data structures and variables for the aggregation are initialized.
2) *Accumulate:* This step is the most important. In this step, each row of the data set is processed, one at a time. An accumulation in a local variable is performed by every thread. Notice that while the table scan is being processed, the threads are fed the rows to accumulate.
3) *Merge:* This step merges the accumulated values of independent threads into the main result. This thread is responsible for merging both local variables and local data structures into a global aggregation result.
4) *Terminate:* In this final step, after all threads partial results have been merged, the function return value is computed. Once this last step is finalized, the final aggregation result is returned to the user.

It is important to point out that due to the fact that user-defined functions are compiled fragments of C code, the input arguments for aggregate functions must be fixed in order to allocate the memory space and allow argument value passing to each thread [8]. Therefore, to allow a dynamic d dimensional vector as argument, the values of the d attributes of a data point have to be packed as a single object: either a string or a binary object. Hence, to implement data summarization, two basic UDFs are

```
SELECT Agg(CAST(CAST(X1 AS VARCHAR)+,+
  CAST(X2 AS VARCHAR)+,+
  .
  .
  .
  CAST(Xd AS VARCHAR) AS Type))
FROM X;
```

(a) String (STR-UDF).

```
SELECT Agg(obj)
FROM X;
```

(b) Binary object (BIN-UDF).

Fig. 3. User-defined functions (UDFs) calls.

developed: (1) a parsing function (STR-UDF); (2) a serializable function for reading and writing binary objects (BIN-UDF) (under the .NET programming environment [12]). The parsing function receives one string value, where all dimension values of the input vector are concatenated. Individual dimension values are parsed and assigned to local variables at run-time. On the other hand, the aggregate UDF for summarization receives all values for the d dimensions of a data point packed as a binary object, and returns all the elements in n, L, and Q also packed as a user-defined type (see Fig. 3b).

Execution performance of the aggregate UDF can be improved by materializing a table with a single column storing the packed dimensions with the user-defined type. Even though the aggregation can be efficiently computed when data is already in binary format, creating such a table is a pre-step that can be time-consuming, especially for large input tables.

Table-valued functions (TVFs) are a type of user-defined function that, unlike aggregate UDFs, is able to return a table as the final result of the function. TVFs read an input data set as a single data stream and do not implicitly manage parallelism. Despite the lack of “out-of-the-box” parallelism, it is common that database systems allow the user to implement routines that support parallelism. Without loss of generality, in this work, a TVF will be used to compute multidimensional aggregations, with internal thread management algorithms returning a result table with just one row.

IV. PARALLEL MULTITHREADED COMPUTATION

We now present our main contributions. We start with an overview on how we optimize the processing of UDFs. We then go into more technical detail, explaining how to manage memory and how to guarantee correct results under concurrent processing by multiple threads. We introduce three alternatives to manage workload among threads. Such workload involves disk I/O and CPU operations. We conclude with a brief time complexity and I/O cost analysis.

Our basic UDF-based algorithm exploits parallel processing to distribute the workload among all threads, while ensuring the hard disk access is accessed with full table scans, to achieve maximum performance. More importantly, processing with concurrent threads needs to guarantee correct results without race conditions, deadlocks or process starvation.

A. Processing Aggregate UDFs

In developing aggregate functions for data summarization, we incorporated several changes that increase the speed of multi-core CPU computers where computing the aggregations is faster than reading records from secondary storage. To obtain sequential reading, one thread is the only process in charge of reading records from the input table, caching blocks of records in main memory, and calling a monitor to dispatch the job to another thread that actually performs the calculations. All threads share memory to update the global aggregate computation. Moreover, we define techniques to control the number of threads executing simultaneously, and the amount of memory used by the aggregation process.

Even though algorithms for aggregation are explained in a general manner, we target the specific problem of computing sufficient statistics. Some key aspects have been modified from the UDF API. For example, the accumulate step receives a complete row from the input table without the need to pack its values as a user-defined type. Also, the initialization includes multithreaded execution parameterization, which would be specific for the hardware configuration. Since the computing of sufficient statistics requires a set of matrices and vectors, results are returned as tables using the common connectivity features of database programmability. Finally, our algorithms integrate into the modern DBMS without modifying any of the primitives for access data.

B. Memory Management for Caching and Concurrent Processing

The purpose of caching part of the input table in main memory is to have quick access to its data records. We address the problem of obtaining a sequential reading by introducing a thread to cache the data blocks of the input data set in main memory. Since each worker thread is assigned the task of computing the aggregation of one block, portions of the data are cached throughout the execution time. Once the computation of a thread is concluded, the memory space occupied by the block is sent to the garbage collector. The reading process is oblivious to multithreaded execution since its only task is to allocate memory space to fit a fixed number of rows and fill the current block with records from the input table. As soon as a block is full or there are no more records to retrieve, a pointer to the block is sent to the monitor process, and a new block is started once the monitor is done. The characteristic difference between such processing approach and a standard parallel aggregation is the way threads access the input table. Instead of having the threads request data blocks, threads are assigned a block as workload by the “monitor” process. Both reading and monitoring are done by the main thread. Thus, in addition to the cost of sequentially reading the input and allocating blocks in main memory, we must consider the overhead of dispatching the worker threads. There is little overhead caused by the monitor calls, due to the difference in speed between reading the rows from disk and computing the flops by CPU.

Correct concurrent processing is solved by defining different types of memory access for working threads. Since each block will be accessed only by one thread after its creation, it will be immutable, and the memory space can then be disposed once the thread is done. Each working thread has a private memory space to compute the local aggregation of its data block and public access to update the global aggregation computed by all threads. Since the results of global aggregation must be the same regardless of how individual operations are interleaved, access to the memory space storing the global aggregation is granted only after the thread acquires a lock on the shared resource. In other words, only one thread is allowed to update the global aggregation at some point in time. In a multithreaded processing environment, there will be several threads working on the private memory space of their current task. Since each thread computes the aggregation on a private memory space and each data block is accessed only by one thread, working threads do not interfere with each other during the aggregation processing. We redefine the merging step of aggregate UDFs to update the global result every time a thread completes a task. All private memory space (cached data and local aggregation) is sent to the garbage collector, so that it can be released by the DBMS once a task is finished. It is important to notice that there are no deadlocks because there is only one lock granting access to all values in the global computation. In order to minimize concurrency control overhead, the data block size should be large enough to have a small number of threads simultaneously attempting to update the global aggregation data structure, thereby reducing contention.

C. Monitoring Multithreaded Processing

In order to manage multiple worker threads, it is necessary to create a monitor process, similar to those processes used by the operating system (OS). Such monitor process executes as part of the main thread; it is in charge of dispatching the workload and terminating execution when all worker threads have finished their individual computations. We propose three monitor alternatives (Fig. 4):

1) Creating a new thread (i.e., there are multiple threads) for every upcoming task (MT-UDF).
2) Using a fixed number of threads (FT-TVF).
3) Exploiting a pool of threads; also called thread pool (TP-TVF).

The first, simplest approach of the monitor is to create a new thread for every request to dispatch a workload (MT-UDF) and to then add the thread to a list. When the reading process is finished and the monitor requests to join the threads, the monitor uses the list to wait until all threads complete their execution. At this point, the aggregation is complete. In this configuration, the reading process allocates blocks in RAM memory regardless of whether or not the processing power is sufficient for completing the tasks prior to causing stack or memory overflow. Moreover, the scheduling policy of the operating system assigns CPU time slices to the threads. A higher priority is not necessarily given to tasks closer to being finished and incomplete tasks will retain memory space until completed.

We now discuss the FT-TVF approach. The maximum amount of memory used for caching can be controlled by the monitor process. Even though this approach does have a circular list to keep a fixed number of working threads, the need for a queue is eliminated since there can be at most one task waiting to be executed. Whenever a new task is created by the monitor, it initially locates the first available slot on the list. A slot is considered available either when its thread is done or when there is no thread assigned to it. Although the circular list decreases the amount of RAM memory used for caching, the reading process has to be stopped every time the list becomes full. Finally, stopping a sequential read for a long period of time can severely impact the algorithm performance.

We now explain the third monitor approach (TP-TVF). To control the number of threads executed in the system and to have a first-in-first-out (FIFO) policy for the upcoming workload, we include a thread pool managed by the monitor process. With such configuration, all tasks created by the monitor are added to the thread pool. When the thread pool is initialized, it creates a fixed number of threads and a FIFO list. As such, whenever a thread finishes its current task, it is assigned the next task in the queue. Even though completed tasks free up memory space, each task in the thread pool queue has a data block associated with it. Moreover, if the waiting queue grows large enough, then it could cause memory overflow.

D. Time Complexity and I/O Analysis

Time complexity and the number of I/O operations for each

Table 2. Time complexity of *n*, *L* and *Q*

	n	L	Q
Elements	1	d	$((d+1)\times d)/2$
Flops per accumulate step	1	d	$(d+1)\times d$
Flops per merging	1	d	$((d+1)\times d)/2$
Overall time complexity	$O(n)$	$O(nd)$	$O(nd^2)$

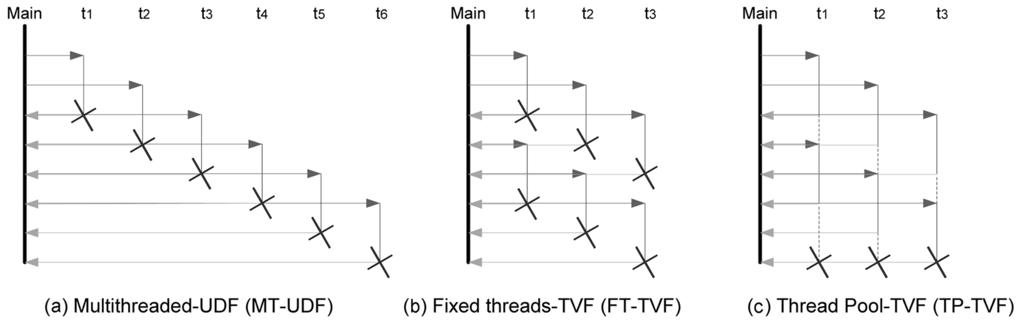
Main
t1
t2
t3
t4
t5
t6
(a) Multithreaded-UDF (MT-UDF)
Main
t1
t2
t3
(b) Fixed threads-TVF (FT-TVF)
Main
t1
t2
t3
(c) Thread Pool-TVF (TP-TVF)

Fig. 4. Thread monitor policies.

of the data summarization matrices are given in Table 2. Since the expected input consists of large data sets ($d \ll n$), one thread is dedicated to the task of performing a sequential read. Little or no overhead is expected from the working threads in charge of the aggregation (only when creating and destroying threads). Even though the overall time complexity of the summarization process is $O(nd^2)$, with efficient multithreading, processing time is dominated by the table scan, in time $O(nd)$. On the other hand, the space complexity of the aggregation process has the global aggregation invariant, and it fluctuates depending on the number of threads working in the system at any time. Thus, the amount of memory used is given by:

$$Mem = ((t + u)((bd) + e) + e)f \qquad (3)$$

where t is the number of threads executed in the system, u is the number of tasks created and waiting for a thread, b is the blocking factor (number of records per block), d is the number of dimensions, e is the number of elements in the aggregation and f is the number of bytes used to store a floating point number.

V. EXPERIMENTAL EVALUATION

We conducted our experiments on a DBMS installed on a server with an Intel Core 2 Quad CPU with four cores at 2.83 GHz each, and 3.2 GB of RAM. The hard disk had 320 GB of capacity, with a SATA interface running at 3 GB/s, and 7,200 RPM. The operating system was Windows XP and the DBMS was Microsoft SQL Server. UDFs were programmed in C#.

In the following sections, we evaluate different alternatives to compute sufficient statistics. The code was compiled and added to the DBMS server using aggregate UDFs and TVFs. The main experimental parameters were the chosen aggregate UDF agg, the blocking factor b (number of records per block) and the number of threads t. All data sets had multivariate normal distributions. Values were stored as double precision floating point numbers. Finally, all experiments were repeated five times and their result was averaged.

A. Concurrency Control

In this section, we analyze the performance impact of all different alternatives to manage threads in multithreaded processing:

1) Creating a thread for every task or block to aggregate (MT-UDF).
2) Using a maximum fixed number of threads with at most one task waiting to start; no queue (FT-TVF).
3) Exploiting a pool of threads with a fixed number of threads and a queue of waiting tasks (TP-TVF).

All control alternatives are compared against the execution time of performing a full table scan. No aggregation is performed during the full table scan: this query is used as a benchmark baseline. The full table scan is performed with a sequential data access given by the UDF API of the DBMS.

The results in Fig. 5 show time performance of the three alternatives to compute Q at different d values, where $n = 1$ M. We can see that the monitoring process of multithreading adds little or marginal overhead to the table scan. For the cases when $d =$

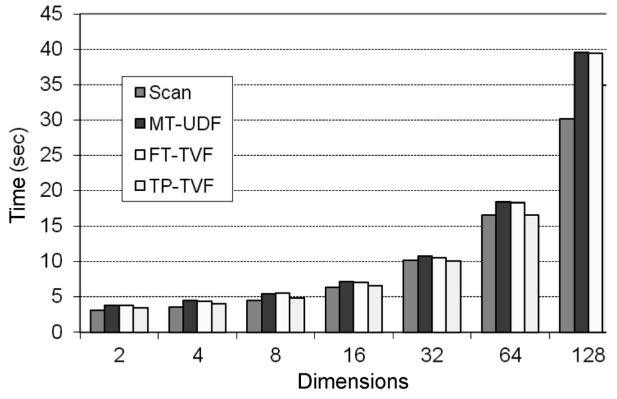

Fig. 5. Concurrency control comparison (task = Q, $n = 1$ M, $d = 64$, $t = 4$). MT-UDF: multiple thread user-defined function, FT-TVF: fixed threads table-valued function, TP-TVF: thread pool table-valued function.

32, the impact of performing the aggregation while reading the table is minimal. On the other hand, the trend becomes evident when $d = 64$; there is a difference in performance given by the monitoring policy used for the working threads. Nevertheless, the time impact of any of the policies is not enough to consider a higher complexity than a table scan. The biggest overhead between the policies is caused when the OS is left to manage all threads. Such results were expected, because there is no memory control, and the policy to assign workload is suboptimal for aggregations. The scheduler will attempt to assign equal time slices to the existing threads in the system, adding context switching (thus increasing overhead), and letting the DBMS saturate with an unfinished workload. The best performance is acquired by TP-TVF because there is little saturation of the OS components by the number of threads managed. As for FT-TVF, controlling the maximum memory used by the aggregation does not significantly affect execution time, and the performance is comparable to a table scan while being faster than MT-TVF.

B. RAM Memory Management and CPU Usage

Caching data blocks of the input table, regardless of the amount of memory used for this matter, could cause memory overflow. While being aware of this potential problem, we have proposed methods to limit memory usage, and implemented

Table 3. UDF processing time varying the number of threads t (time in seconds and $n = 10$ M)

d	Threads	FT-TVF	TP-TVF
64	1	173	155
64	2	173	155
64	3	173	155
64	4	173	156
128	1	386	324
128	2	381	324
128	3	379	326
128	4	380	330

UFD: user-defined function, FT-TVF: fixed threads table-valued function, TP-TVF: thread pool table-valued function.

them in FT-TVF. Since our experimental study focuses on $d \ll n$, such limitation is never reached by varying the values of $d=128$. Furthermore, we found that in all our experimental cases, a reduced number of threads is enough to efficiently compute sufficient statistics.

Table 3 displays the performance of our monitor policies, when varying the number of working threads. For $d=64$, execution time is made almost invariant by increasing the number of threads assigned to the accumulated part of the aggregation.

Such behavior is caused because the time to read a record of size d from the input table is greater than the time it takes the process to calculate the $O(d^2)$ flops in the accumulate step. Nevertheless, when d increases, we need more threads to catch up with the reading speed. This is the case for $d=128$; the performance of FT-TVF reaches a peak when $t=3$. When $t<3$, the sequential read has to be stopped every time the limit for the number of working threads (t) is reached. In contrast, TP-TVF does not stop whenever all working threads are busy; instead, it places all waiting workload in a queue. Finally, the cases when $d=128$ and $t>3$ for FT-TVF, and $t>1$ for TP-TVF show the tradeoff caused by using more threads than those required by the aggregation.

The blocking factor is directly related to two aspects of the execution: the amount of memory used for caching and contention for concurrently updating the global aggregation.

The performance of different policies computing Q of a data

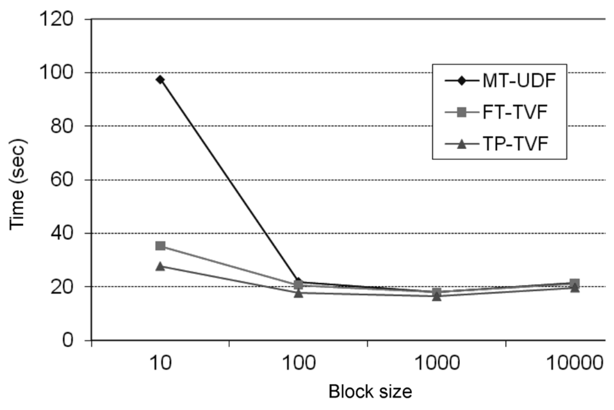

Fig. 6. Impact of the block size (agg = Q; n = 1 M; d = 64; t = 4). MT-UDF: multiple thread user-defined function, FT-TVF: fixed threads table-valued functions, TP: thread pool table-valued functions.

set of $n=1$ M and $d=64$, while varying the block size, is presented in Fig. 6. When the blocking factor is relatively large ($b=1000$), there is less concurrency for updating the global aggregation, and the tradeoff is the amount of memory used for caching. In contrast, the performance of MT-TVF is severely affected by the unmonitored number of threads when $b=100$. Nevertheless, the multithreading policies of FT-UDF and TP-TVF have a maximum of $t=4$ threads attempting to acquire the lock simultaneously, so the effect of the block size is less evident. In data summarization, flops of the accumulate function are always greater or equal to the flops in the merge step. Therefore, the complexity of the algorithm centers on acquiring the lock, waiting for the lock, and the overhead of managing the lock.

C. Comparison with SQL Queries

For comparison purposes, we also tested plain SQL aggregations (Plain SQL) and aggregate user-defined functions: binary object input parameter (BIN-UDF) and string input parameter (STR-UDF). We include a comparison with a TVF that does not use multithreading to distribute workload (Regular TVF). Table 4 shows the execution performance when solving the Q aggregation for $n=1$ M. FT-TVF, and TP-TVF have the same number of threads, $t=4$. The three TVFs with multithreading also have a blocking factor $b=1000$.

The execution performance of Regular TVF is severely affected by d; only for the cases when $d=8$ is the execution time comparable with the multithreaded TVFs. The impact of multithreading is shown for $d=128$ where the time is reduced by less than half. On the other hand, BIN-UDF aggregates rows packed in binary structures that follow the user-defined type definition. It has more efficient performance than the multithreaded TVFs for $d=32$. However, BIN-UDF is faster only if we do not consider the pre-step of physically materializing the table inside the DBMS.

Nevertheless, the BIN-UDF step alone remains as a good comparison example for multithreading. The second implementation of aggregate UDFs (STR-UDF) moves the CAST function, for packing rows into a user-defined type object, into the SQL statement of the aggregation. Yet, STR-UDF still has a higher execution time than any of our algorithms because of the overhead of the parsing function in the user-defined.

It can be seen in Fig. 7 that tendencies hold when increasing the size of n. Although SQL is the fastest way to compute the aggregates in all cases when $d=16$, for $d=32$ the performance decreases slightly. Unfortunately, limitations of the number of

Table 4. Comparison with aggregate UDFs, with/without multithreaded processing (time in seconds and agg = Q; n = 1 M; t = 4; b = 1000)

d	Plain SQL	REG-TVF	BIN-UDF pre-step	BIN-UDF	STR-UDF	MT-TVF	FT-TVF	TP-TVF
2	0.81	3.10	18.14	6.00	8.01	3.68	3.29	3.19
4	0.74	3.42	21.29	6.01	8.57	3.88	3.68	3.67
8	1.36	4.51	28.07	6.11	10.55	4.76	4.83	4.48
16	4.87	7.00	41.07	6.41	14.21	6.38	6.31	6.29
32	86.30	13.55	68.45	7.45	21.23	9.78	9.60	9.44
64	*	31.80	126.73	10.66	38.09	17.42	17.55	16.09
128	*	89.71	244.90	22.89	78.28	38.00	37.89	33.13

UDF: user-defined function, TVF: table-valued functions, REG: regular, BIN: binary, STR: string, MT: multiple thread, FT: fixed threads, TP: thread pool.

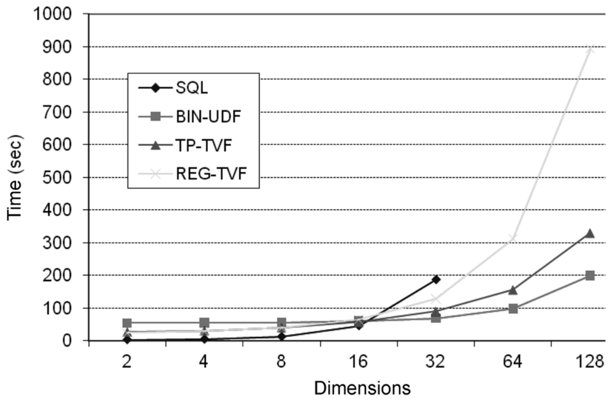

Fig. 7. Aggregate comparison when varying d (agg = *Q*; *n* = 10 *M*; *t* = 4; *b* = 1000). BIN-UDF: binary user-defined function, TP-TVF: thread pool table-valued functions, REG-TVF: regular table-valued functions.

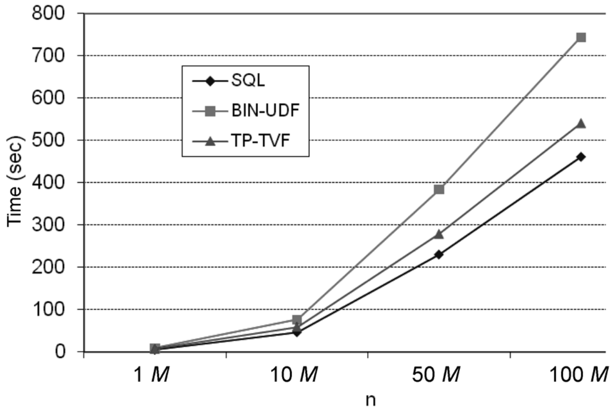

Fig. 8. Aggregate comparison when varying *n* (agg = *Q*; *d* = 16; *t* = 4; *b* = 1000). BIN-UDF: binary user-defined function, TP-TVF: thread pool table-valued functions.

rows in a query prevent experimenting with $d = 64$. Since the complexity of Regular TVF is $O(nd^2)$, the performance of TP-TVF demonstrates its efficiency to exploit primitives for accessing data in the DBMS. Consequently, the time difference with REG-TVF increases with *d*, while TP-TVF remains comparable with the aggregate step of BIN-UDF. The final set of experiments verifies the linear complexity with respect to the size of *n*. The execution results for *Q*, with a number of dimensions $d=16$, are presented in Fig. 8. Our multithreaded monitoring algorithm with a tread pool (TP-TVF) is parameterized with threads $t = 4$, and a block size of $b = 1000$. The three methods (SQL, BIN-UDF, and TP-TVF) show linear scalability with respect to data size *n*. Finally, the experimental results have provided evidence that our algorithms for multithreading scale linearly in both *n* and *d*. Such achievement is obtained by efficiently taking advantage of DBMS primitives, and integrating them with multithreaded aggregates.

VI. CONCLUSIONS

This paper studied the efficient computation of data set summaries on a large data set, exploiting parallel processing with multiple threads. We especially focused on UDFs as the programming mechanism to extend a DBMS with data mining capabilities. More specifically, we proposed algorithms that can be embedded into a DBMS as UDFs (Table UDFs) to interleave table scans and CPU processing. We presented three "monitor" algorithms to manage threads and to evenly distribute the workload. Such algorithms are particularly useful in cases of multiple core CPUs. The first algorithm (MT-UDF) has a master thread that reads the table and dynamically assigns data blocks to new threads. The second algorithm (FT-TVF) has a master thread that dynamically allocates a workload to new threads until a maximum number of threads has been reached; threads are destroyed once they finish processing the aggregation. The third algorithm (TP-TVF) has a pool of threads that reuses idle threads. Hardware optimization is achieved by controlling the CPU usage and managing RAM memory for data caching. We performed a careful experimental evaluation on a DBMS working on a multicore CPU computer. Our algorithms performance was compared against plain SQL queries and aggregate UDFs. We show our algorithms generally outperform the aggregate UDFs provided by the DBMS. Our algorithms showed better performance at high dimensionality ($d > 32$) but are slower than SQL queries for small dimensionality. Even though for most cases an efficient summarization can be done using few threads, there is a significant difference between this and the summarization performance without multithreading. Our algorithms exhibit linear scalability on both the number of points in the data set n and number of dimensions *d*. Our experiments found that the block size plays an important role in avoiding a large number of concurrent updates. The MT-UDF algorithm is more sensitive to the block size than FT-TVF and TP-TVF algorithms.

Although our optimizations are specific to computing data summarization for linear Gaussian models, we believe that future research on UDFs should also consider thread management for fast parallel processing. Research issues include synchronization policies for threads with complex data structures and memory allocation to avoid table misses and data overflow. In addition, user-defined functions can be extended to allow a low-level of data access in order to speed up the execution of critical parallel processes (e.g., a thread retrieving a data page directly from disk). Finally, more research is needed on exploiting DBMS extensibility mechanisms like UDFs to perform data mining, instead of processing large data sets on flat files.

ACKNOWLEDGMENTS

This work was supported by National Science Foundation grants CCF 0937562 and IIS 0914861. We would like to thank the help from Zhibo Chen to understand the basics of multithreaded processing with UDFs and TVFs.

REFERENCES

1. J. Adibi, T. Barrett, S. Bhatt, H. Chalupsky, J. Chame, and M. Hall, "Processing-in-memory technology for knowledge discovery algorithms," *2nd International Workshop on Data Management on New*

Hardware (DaMon 2006), Chicago, IL, 2006.

2. A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey, "A characterization of data mining algorithms on a modern processor," *Proceedings of the 1th International Workshop on Data Management on New Hardware*, Baltimore, MD, 2005.
3. S. Chaudhuri, U. Fayyad, and J. Bernhardt, "Scalable classification over SQL databases," *Proceedings of the 15th International Conference on Data Engineering*, NSW, Australia, 1999, pp. 470-479.
4. C. Ordonez and J. García-García, "Database systems research on data mining," *SIGMOD '10 Proceedings of the 2010 International International Conference on Management of Data*, Indianapolis, IN, 2010, pp. 1253-1254.
5. C. Ordonez, "Building statistical models and scoring with UDFs," *ACM SIGMOD International Conference on Management of Data*, Beijing, China, 2007, pp. 1005-1016.
6. S. K. Pitchaimalai, C. Ordonez, and C. Garcia-Alvarado, "Comparing SQL and MapReduce to compute Naive Bayes in a single table scan," *Proceedings of the Second International Workshop on Cloud Data Management (CloudDB)*, Toronto, ON, 2010, pp. 9-16.
7. C. Ordonez and S. K. Pitchaimalai, "Bayesian classifiers programmed in SQL," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 139-144, Jan. 2010.
8. C. Ordonez, "Statistical model computation with UDFs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 12, pp. 1752-1765, Dec. 2010.
9. J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," *Proceedings of the 33rd International Conference on Very Large Data Bases*, Vienna, Austria, 2009, pp. 339-350.
10. S. Sarawagi, S. Thomas, and R. Agrawal, "Integrating association rule mining with relational database systems: alternatives and implications," *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, Seattle, WA, 1998, pp. 343-354.
11. S. Cohen, "User-defined aggregate functions: bridging theory and practice," *ACM SIGMOD International Conference on Management of Data*, Chicago, IL, 2006, pp. 49-60.
12. J. A. Blakeley, M. Henaire, C. Kleinerman, I. Kunen, A. Prout, and V. Rao, ".NET database programmability and extensibility in microsoft SQL server," *ACM SIGMOD International Conference on Management of Data*, Vancouver, BC, 2008, pp. 1087-1097.
13. M. Jaedicke and B. Mitschang, "On parallel processing of aggregate and scalar functions in object-relational DBMS," *SIGMOD Record*, vol. 27, no. 2, pp. 379-389, Jun. 1998.
14. M. Navas and C. Ordonez, "Efficient computation of PCA with SVD in SQL," *Workshop on Data Mining using Matrices and Tensors in Conjunction with the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (DMMT)*, Paris, France, 2009.
15. S. Manegold, P. Boncz, and M. Kersten, "Optimizing main-memory join on modern hardware," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 4, pp. 709-730, Jul. 2002.
16. J. Cieslewicz, W. Mee, and K. A. Ross, "Cache-conscious buffering for database operators with state," *Proceedings of the 5th International Workshop on Data Management on New Hardware*, Providence, RI, 2009, pp. 43-51.
17. R. Ross, V. S. Subrahmanian, and J. Grant, "Aggregate operators in probabilistic databases," *Journal of ACM*, vol. 52, no. 1, pp. 54-101, Jan. 2005.
18. A. Knobbe, A. Siebes, and B. Marseille, "Involving aggregate functions in multi-relational Search," *Principles of Data Mining and Knowledge Discovery. Lecture Notes in Computer Science vol. 2431*, Heidelberg: Springer Berlin, 2002, pp. 145-168.
19. C. Garcia-Alvarado, Z. Chen, and C. Ordonez, "OLAP with UDFs in digital libraries," *ACM 18th International Conference on Information and Knowledge Management*, Hong Kong, 2009, pp. 2073-2074.
20. H. Wang and C. Zaniolo, "User defined aggregates in object-relational systems," *Proceedings of the 16th International Conference on Data Engineering*, San Diego, CA, 2000, pp. 135-144.
21. C. Luo, H. Thakkar, H. Wang, and C. Zaniolo, "A native extension of SQL for mining data streams," *ACM SIGMOD International Conference on Management of Data*, Baltimore, MD, 2005, pp. 873-875.
22. Z. He, B. S. Lee, and R. Snapp, "Self-tuning cost modeling of user-defined functions in an object-relational DBMS," *ACM Transactions on Database Systems*, vol. 30, no. 3, pp. 812-853, Sep. 2005.
23. C. Ordonez and S. K. Pitchaimalai, "Fast UDFs to compute sufficient statistics on large data sets exploiting caching and sampling," *Data & Knowledge Engineering*, vol. 69, no. 4, pp. 383-398, Apr. 2010.

Carlos Ordonez

Carlos Ordonez got his Ph.D. degree in Computer Science from the Georgia Institute of Technology, USA, in 2000. He worked six years extending the Teradata DBMS with advanced data mining techniques to analyze large databases. He is currently an Assistant Professor at the University of Houston. His research is centered on the integration of machine learning and statistical techniques into database systems to analyze large data sets as well as their application to scientific problems. His research has produced over 70 papers, over 800 citations and has been funded by NSF.

Mario Navas

Mario Navas got his B.S. in Computer Science from PUCE University, Ecuador, in 2006. Then he got an M.S. degree in Computer Science from the University of Houston, USA, in 2009. His research focuses on how to integrate dimensionality reduction and regression models with a DBMS.

Carlos Garcia-Alvarado

Carlos Garcia-Alvarado got a B.E. degree in Computer Engineering from Universidad de las Americas, Puebla, and an M.S. degree in Industrial Engineering from Instituto Tecnologico de Estudios Superiores de Monterrey. He came to the University of Houston where he received an M.S. degree from University of Houston in 2008. His research focuses on combining database systems and information retrieval technologies.