Data Windows: A Data-Centric Approach for Query Execution in Memory-Resident Databases *

Jayaprakash Pisharath, Alok Choudhary
Dept. of Electrical and Computer Engineering
Northwestern University
Evanston IL - 60208 USA
{jay, choudhar}@ece.northwestern.edu

Mahmut Kandemir
Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park PA - 16802 USA
kandemir@cse.psu.edu

Abstract

Structured embedded databases are currently becoming an integrated part of embedded systems, thus, enabling higher standards in system automation. These embedded databases are typically memory resident. In this paper, we present a data-centric approach called data windowing that optimizes multiple queries issued to an embedded database. Traditional approaches improve the performance by optimizing the control flow of operations, whereas we target performance improvements based on the data that is brought into the system.

1. Introduction

As more and more functionalities of embedded systems are being implemented in software, there is a clear need for system software support. While most of prior software-based studies on embedded systems targeted operating systems, compilers, and applications, a growing segment of the industry is now considering embedded database support [4, 6]. This is because, with the growing complexities of real-time embedded systems, use of a commercially developed structured embedded database is becoming a must [2]. To meet the performance and reliability demands, the embedded database engine must be small, high-performance, flexible, and maintenance free.

Since most embedded environments do not employ a disk subsystem, database support generally comes in the form of an in-memory (also called memory-resident) database [7]. Memory-resident databases have been studied since a long time back by various researchers. In the past, memory technology bottlenecks have prevented the actual implementation of these databases. Currently, BerkeleyDB [8], BirdStep RDM [2], TimesTen [9], DataBlitz [5], and Monet [3] are a few known memory-resident databases.

Obviously, one of the key issues in attaining decent performance from such an embedded database is to exploit data locality (cache behavior). In fact, it is known that data cache misses can form as high as 90% of overall memory stalls [1]. Consequently, optimizing database queries to improve their data access patterns can be extremely useful in practice. This can be achieved by maximizing the reuse of the data that resides in the data cache. With data reuse as the primary goal, we propose a technique called data windowing that maximally utilizes every block of data brought into the cache. The following section elaborates our data windowing strategy.

2. Data Windowing

A database usually consists of multiple tables. Each table has multiple columns (fields). A query views the table as a collection of data blocks. Depending on the implementation, these blocks can either be file blocks, database page blocks or just user-defined blocks. Based on this, we define data window as a block of data from the table being accessed by a query. We propose an optimization strategy called data windowing that works on data windows. A query uses a set of data windows from each table that it accesses. We begin by presenting a scheme for a single table case and then extend it to the multiple table case. Data windowing consists of the following steps.

1. Consider a single table T. The table T is divided into N data windows such that each window ($i \in N$) fits into the cache.
2. For each data window i in N, identify a set of queries Q to be performed when i is loaded to the processor. The goal is to reuse the data in a data window to the maximum possible extent.

* This work was supported by National Science Foundation (NSF) grant CCR-0325207

3. The number of queries to look at, and the order in which the selected queries are executed is not fixed. That is, the order of accessing the data in a data window is not pre-determined. One way to do it is to just follow the actual execution order as done without this optimization. Given a data window, there is lot of room usually for reusing the data when the execution order is rearranged. A query queue contains the queries that can be handled at a given instance (maximum look-ahead of multiple queries).
4. The queries are scheduled based on the window accesses. For each window that is accessed, instances of multiple queries that require data from the particular window are scheduled sequentially. That is, queries in the query queue are shuffled in a safe way to reuse the data that is being processed at a given instance. Thus, a schedule queue is built based on the data accessed by queries in the query queue.
5. The first query finishes executing the particular code section that requires data from the window and passes control to the section of the second query that requires the same set of data. The control goes from one query to another until all queries in the schedule queue are executed.
6. Now control is shifted to the next data window of the table. Steps 2 to 5 are repeated for this data window. The algorithm continues until all data windows are completed.

As an example, consider a single table T and three queries, Q1, Q2, and Q3. The table T is first divided into data windows (say, T1, T2, T3, and T4) such that each Ti fits in the cache. Let Qij refer to the part of Qi that access Tj. A classical query execution would be in this order: Q11, Q12, Q13, Q14, Q21, Q22, Q23, Q24, Q31, Q32, Q33, Q34. Our approach, on the other hand, executes this: Q11, Q21, Q31, Q12, Q22, Q32, Q13, Q23, Q33, Q14, Q24, Q34. That is, we restructure query parts (Qij) around the data and make use of the best cache locality. A graphical schematic of this example when applied for M queries and N windows is shown in Figure 1. An extension to the algorithm is the presence of multiple tables. When multiple tables are accessed by a query, each table has its set of data windows. The queries are scheduled based on multiple data windows from multiple tables. The main goal of our technique is to exploit spatial locality in database queries. Since data windowing changes cursor control based on the data windows, care should be taken to ensure that the data consistency of the database and data dependencies in the original query execution plan does not change. When queries are reordered, they might corrupt the database or just follow a wrong execution order violating the dependencies.

Data windowing uses the data and not the query execution flow to improve performance. Our technique achieves

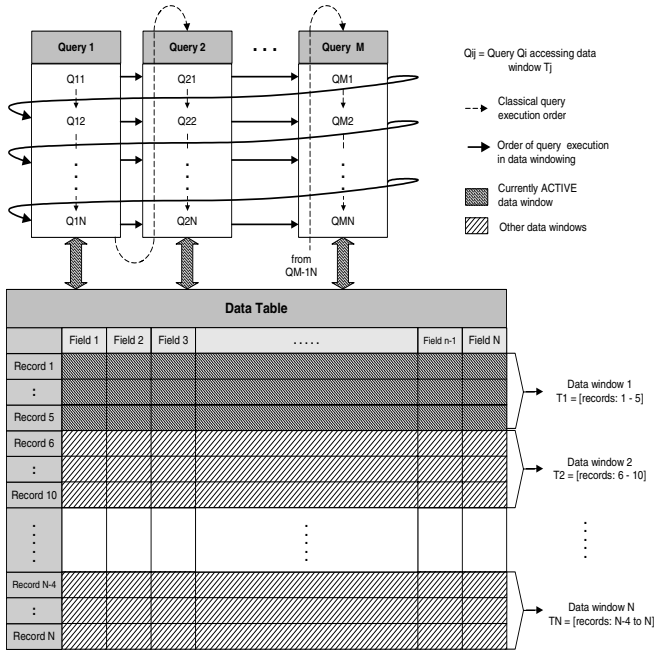

Figure 1. Data windowing applied to M queries that work on data from a single table with N data windows.

significant improvements in the latency of memory hierarchy (results not presented here). We also studied the impact of our technique on the overall system. Our results showed that data windowing improves the overall execution times as well.

References

[1] A. Ailamaki, D. DeWitt, M. Hill, and D. Wood. Dbmss on a modern processor: Where does time go? *The VLDB Journal*, 1999.

[2] Birdstep Technology. *Database Management In Real-time and Embedded Systems - Technical White Paper*, 2003.

[3] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, The Netherlands, May 2002.

[4] Database Trends and Applications. *Embedded Databases Drive New Computing Model*, July 2002.

[5] J. B. et al. Datablitz storage manager: Main memory database performance for critical applications. In *ACM SIGMOD - Industrial Session: Database Storage Management*, 2000.

[6] Information Week. *Embedded Databases Reveal Gems*, October 7, 2002.

[7] McObject LLC. *Main Memory vs. RAM-Disk Databases*, 2003.

[8] Sleepycat Software. *Berkeley DB V4.1.25*, 2003.

[9] TimesTen Inc. *TimesTen Architectural Overview*, 2003.