EaseDB: A Cache-Oblivious In-Memory Query Processor

Bingsheng He[†] Yinan Li[‡] Qiong Luo[†] Dongqing Yang[‡]
[†]Hong Kong Univ. of Science and Technology
{saven,luo}@cse.ust.hk
[‡]Peking University
{liyinan,dqyang}@pku.edu.cn

ABSTRACT

We propose to demonstrate EaseDB, the first cache-oblivious query processor for in-memory relational query processing. The cache-oblivious notion from the theory community refers to the property that no parameters in an algorithm or a data structure need to be tuned for a specific memory hierarchy for optimality. As a result, EaseDB automatically optimizes the cache performance as well as the overall performance of query processing on any memory hierarchy. We have developed a visualization interface to show the detailed performance of EaseDB in comparison with its cache-conscious counterpart, with both the parameter values in the cache-conscious algorithms and the hardware platforms varied.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing, Relational Databases*

General Terms

Algorithms, Measurement, Performance

Keywords

Cache-oblivious, In-memory query processing

1. INTRODUCTION

As the gap between the processor speed and the memory speed increases, the memory performance has become an important factor for the overall performance of relational query processing. However, it is a challenging task to optimize the memory performance for relational query processing, due to the diversity, complexity and runtime dynamics of database systems as well as memory hierarchies. Thus, it is imperative to investigate self-optimizing query processing techniques for the memory hierarchy.

EaseDB [5] is a new *cache-oblivious* in-memory relational query processor that can automatically optimize the query processing performance on any multi-level memory hierarchy. In EaseDB, data structures and algorithms for query processing are cache-oblivious, since they do not depend on any cache parameters of a specific memory hierarchy, e.g., the cache capacity and block size. They automatically achieve a high performance comparable to the fine-tuned *cache-conscious* algorithms on various platforms. In contrast, cache-conscious algorithms (see Ailamaki's survey [1]) explicitly take cache parameters as input and have a high performance with suitable parameter values.

EaseDB automatically optimizes the performance of relational query processing on all levels of a memory hierarchy. This automaticity is especially desirable for the levels above the main memory, because caches at these levels, e.g., L1 and L2 caches, are managed by the hardware. As a result, the accurate state information of these caches are difficult to obtain due to the system runtime dynamics and the hardware complexity. Thus, we consider cache-oblivious algorithms that can automatically improve the in-memory performance of query processing.

The system demonstration exposes the cache behavior of cache-oblivious query processing on a multi-level memory hierarchy in comparison with cache-conscious query processing, and also provides an intuitive way of visualizing the cache performance as well as the overall performance of the query processor. Through our visualization tool, we can examine the cache performance together with the status of the query execution at runtime.

We organize our demo into two parts:

- Visualizing the cache behavior of a cache-conscious algorithm with different parameter values.This part is to simulate the performance tuning process for cache-conscious algorithms. Initially, the parameter value is set to be a small one. Subsequently, it doubles the one used in the previous execution. The execution stops when the parameter value is larger than the L2 cache capacity. Through comparing these executions, we determine the suitable parameter value for the cache-conscious algorithm.

- Visualizing the cache behavior of the cache-oblivious algorithm in comparison with its cache-conscious counterpart. The cache-conscious algorithm under comparison uses the suitable parameter value obtained in the first part.

Through the demonstration, we show that (1) cache-conscious algorithms typically optimize only the target level of the memory hierarchy, and (2) cache-oblivious algorithms in EaseDB have a comparable cache performance as well as overall performance to their fine-tuned cache-conscious counterparts.

2. SYSTEM OVERVIEW

We first give a brief overview of EaseDB in this demonstration. Next, we describe our visualization tool for the cache performance.

2.1 EaseDB

Figure 1 shows the system architecture of EaseDB. There are three major components, namely the SQL parser, the query optimizer, and the plan executor. The query optimizer in turn consists

Copyright is held by the author/owner(s).
SIGMOD'07, June 12–14, 2007, Beijing, China.
ACM 978-1-59593-686-8/07/0006.

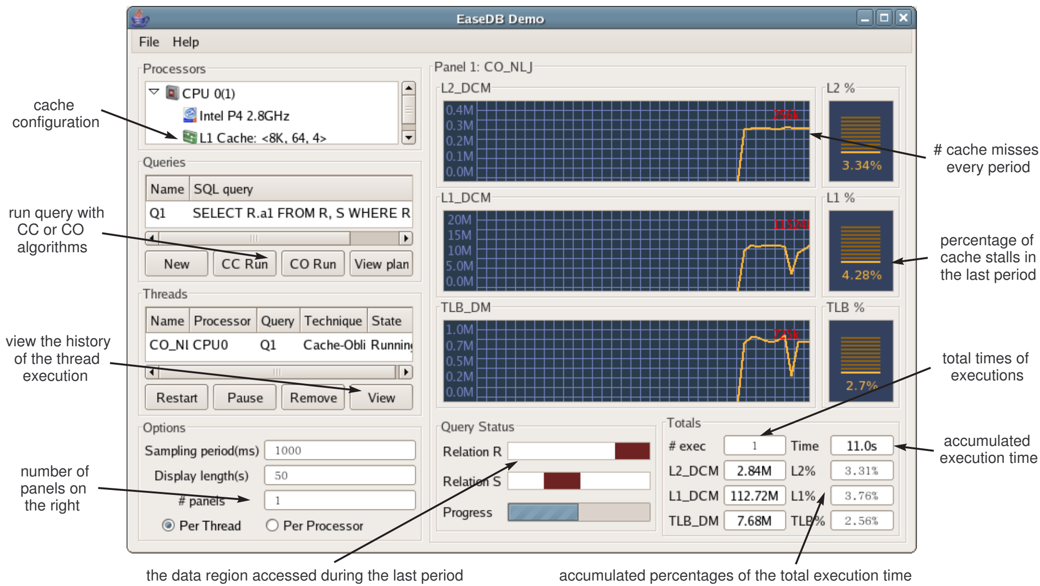


Figure 2: The main execution window

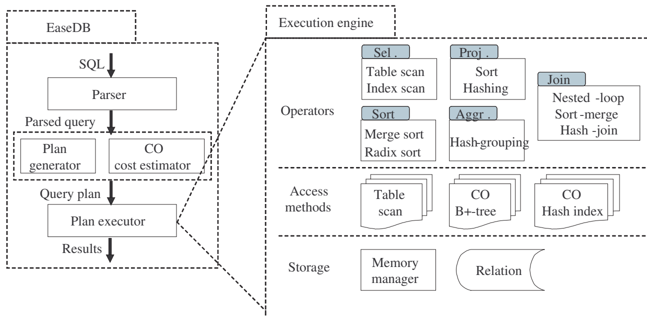


Figure 1: The system architecture of EaseDB

of a query plan generator and a cache-oblivious cost estimator. The execution engine is divided into three layers, including the storage, the access methods and the query operators [7]. All these components and algorithms are designed to be cache-oblivious.

We use two main methodologies, divide-and-conquer and amortization, to develop a cache-oblivious algorithm in EaseDB. For example, the cache-oblivious non-indexed nested-loop join (CO_NLJ) [4] applies *recursive partitioning* on both relations. The algorithm first divides each of the inner and outer relations into two equal-sized sub-relations. Next, it performs joins on the pairs of inner and outer sub-relations. This partitioning and joining process goes on recursively. At some point of the recursion, the join on two partitions fits into the L2 data cache. As the recursion goes on, the join fits into the L1 data cache. The base case is evaluated using the tuple-based nested-loop join algorithm. More details about the design and implementation of EaseDB can be found in our system overview paper [5].

2.2 Visualizing the cache performance

The visualization tool is a graphical user interface (GUI) that dynamically displays the cache performance and the state of all queries in the system. The GUI is implemented in Java SWT (Standard Widget Toolkit), since Java SWT is widely applicable on various systems. The performance on CPU caches including the numbers of cache hits and misses are typically measured using a profiling tool. In our demonstration, we use PCL (Performance Counter Library) [3] to periodically obtain the cache performance from the lower-level hardware counters. In addition, PCL allows user-level code to access the performance counters.

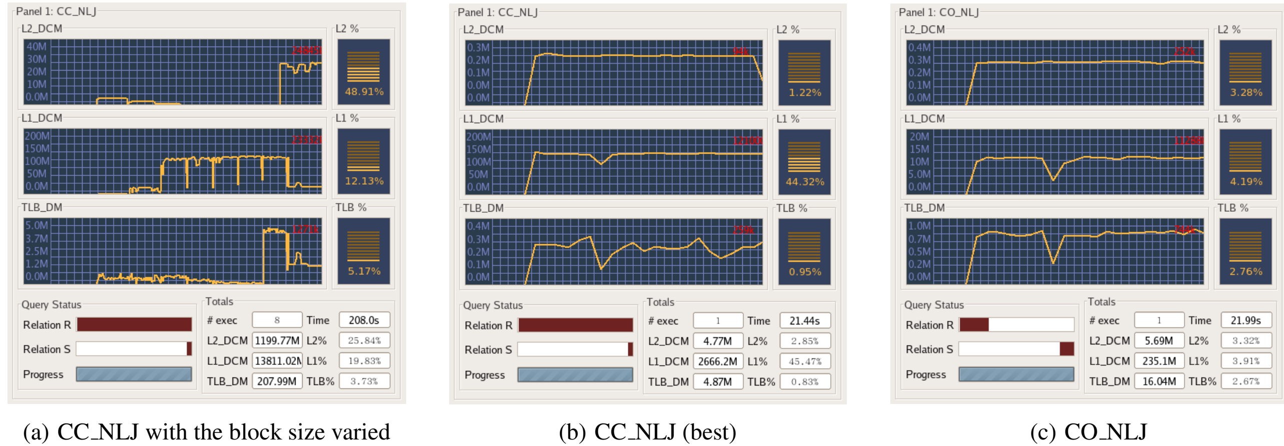
Figure 2 shows the main execution window when a simple query is running. There are four panels on the left of the window, namely `Processors`, `Queries`, `Threads` and `Options`. On the right of the window, there are a number of panels visualizing the cache performance dynamically. The information shown in the `Processors` panel is automatically obtained from the calibrator [6]. We define the *cache configuration* as a three-element tuple $<C, B, A>$, where C is the cache capacity in bytes, B the cache line size in bytes and A the set associativity.

In the `Queries` panel, the user can submit custom queries on the in-memory data. The user can choose cache-conscious or cache-oblivious algorithms to execute the query. When the cache-conscious algorithm is chosen, its parameters can be individually configured. More specifically, when the `CC Run` button is clicked, a popup window is shown for the user to input the parameter value for the cache-conscious algorithm. The default parameter value for the cache-conscious algorithm is L2 cache capacity or cache line size according to the algorithmic characteristics, since the L2 cache is typically a major bottleneck for memory accesses [2]. The user can also click on the `View plan` button and view the query plans generated from both the cache-conscious optimizer used in traditional relational databases and the cache-oblivious one used in EaseDB.

In the `Threads` panel, we display the basic information of all threads that are running or ran previously in the system. The state of a thread can be *Running* (the thread is running), *Stop* (the user clicks the `Pause` button in the panel, and stops the execution) and *Done* (the execution ends).

In the `Options` panel, the user can specify the time period of reading the cache performance metrics from hardware counters. The main performance metrics shown in our demo include the number of L1 and L2 data cache misses (*L1_DCM* and *L2_DCM* respectively) and the number of translation lookaside buffer (TLB) misses (*TLB_DM*). These performance metrics can be shown on a per thread or per processor basis. The default setting is per thread.

In each panel on the right of the main execution window, we display the main performance metrics graphically (from top down: L2_DCM, L1_DCM and TLB_DM), query status and totals. The



(a) CC_NLJ with the block size varied

(b) CC_NLJ (best)

(c) CO_NLJ

Figure 3: Screen shots: CO_NLJ and CC_NLJ on P4

query status includes the relation regions that are accessed in the last period and the query progress. Based on this information, we can determine the *hot* region during the process of evaluating the query.

3. DEMONSTRATION

We use the visual tool to study cache-oblivious algorithms in EaseDB in comparison with cache-conscious algorithms.

3.1 Demo setup

We will run our demo on two different platforms, namely P4 and AMD. Some features of these machines are listed in Table 1.

Table 1: Machine characteristics

Name	*P4*	*AMD*
OS	Linux 2.4.18	Linux 2.6.15
Processor	Intel P4 2.8GHz	AMD Opteron 1.8GHz
L1 D-Cache	<8K, 64, 4>	<64K, 64, 2>
L2 cache	<512K, 128, 8>	<1M, 128, 16>
DTLB	64	1024
Memory (bytes)	2.0G	2.0G

In our demonstration, we will submit custom queries to our synthetic data sets. An example custom query shown in Figure 2 is a join "SELECT R.a_1 FROM R, S WHERE R.a_1<S.a_1 and R.a_2<S.a_2... and R.a_n<S.a_n". Each of relations R and S consists of n four-byte integer attributes, a_1, a_2, ..., and a_n. All fields of each table are involved in the non-equijoin predicate so that an entire tuple is brought into the cache for the evaluation of the predicate. This query is evaluated with either the blocked nested-loop join (CC_NLJ) [8] or the cache-oblivious nested-loop join (CO_NLJ) [4].

Figure 2 shows the screen shot of evaluating the exapmle query with CO_NLJ when the query progress was 50%. Each relation has 32 attributes. The tuple size is 128 bytes (the L2 cache line size of P4). The size of each relation is $8M$ bytes, which is larger than the L2 data cache capacity. We have varied the relation size and the tuple size, and obtained similar results [4].

3.2 Runtime performance results

As an example, Figure 3 shows some screen shots of the runtime performance results running our demonstration system on P4.

Figure 3 (a) shows the screen shot of evaluating the query with CC_NLJ with a changing block size of the inner relation. The block size is ranged from $4K$ to $512K$ bytes. The performance variance of CC_NLJ with different block sizes is large. This large performance variance quantifies the potential performance loss with ineffective tuning or without any tuning. Moreover, the three kinds of cache misses reach their minimum at different block sizes. This is evidence of a disadvantage of cache-conscious algorithms, i.e., they often optimize the performance for a specific level in the memory hierarchy, but do not optimize for all levels of the memory hierarchy. Note, there are a very small number of outliers in the measurements. A possible reason is the casual error of the profiling tool [3]. Nevertheless, the accumulated results are stable in our experiments.

Figures 3 (b, c) show the screen shots of evaluating the query with the best CC_NLJ and CO_NLJ, respectively. The best CC_NLJ has a block size of $128K$ bytes. Compared with the best CC_NLJ, CO_NLJ has a consistently good performance on the L1, the L2 and TLB. This performance advantage shows the power of automatic optimization for all levels of the memory hierarchy achieved by cache-oblivious techniques. The execution time of CO_NLJ is similar to that of the best CC_NLJ.

In addition to the runtime performance results, we will demonstrate the working mechanisms of our cache-oblivious algorithms.

4. REFERENCES

[1] A. Ailamaki. Database Architectures for New Hardware. VLDB, 2004.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a Modern Processor: Where Does Time Go? VLDB, 1999.

[3] R. Berrendorf, H. Ziegler, and B. Mohr. PCL: Performance Counter Library. http://www.fz-juelich.de/zam/PCL/.

[4] B. He and Q. Luo. Cache-Oblivious Nested-Loop Joins. CIKM, 2006.

[5] B. He and Q. Luo. Cache-Oblivious Query Processing. CIDR, 2007.

[6] S. Manegold. The Calibrator (v0.9e), a Cache-Memory and TLB Calibration Tool. http://www.cwi.nl/∼manegold/Calibrator/.

[7] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3 edition, 2003.

[8] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. VLDB, 1994.