Cache-Oblivious Databases: Limitations and Opportunities

BINGSHENG HE and QIONG LUO
Hong Kong University of Science and Technology

8

Cache-oblivious techniques, proposed in the theory community, have optimal asymptotic bounds on the amount of data transferred between any two adjacent levels of an arbitrary memory hierarchy. Moreover, this optimal performance is achieved without any hardware platform specific tuning. These properties are highly attractive to autonomous databases, especially because the hardware architectures are becoming increasingly complex and diverse.

In this article, we present our design, implementation, and evaluation of the first cache-oblivious in-memory query processor, EaseDB. Moreover, we discuss the inherent limitations of the cache-oblivious approach as well as the opportunities given by the upcoming hardware architectures. Specifically, a cache-oblivious technique usually requires sophisticated algorithm design to achieve a comparable performance to its cache-conscious counterpart. Nevertheless, this development-time effort is compensated by the automaticity of performance achievement and the reduced ownership cost. Furthermore, this automaticity enables cache-oblivious techniques to outperform their cache-conscious counterparts in multi-threading processors.

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

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Cache-oblivious, cache-conscious, data caches, chip multiprocessors, simultaneous multithreading

ACM Reference Format:

He, B. and Luo, Q. 2008. Cache-oblivious databases: Limitations and opportunities. ACM Trans. Datab. Syst. 33, 2, Article 8 (June 2008), 42 pages. DOI = 10.1145/1366102.1366105 http://doi.acm.org/10.1145/1366102.1366105

1. INTRODUCTION

For the last two decades, processor speeds have been increasing at a much faster rate (60% per year) than memory speeds (10% per year) [Ailamaki 2005]. Due

This work was supported by grants HRUST6263/04E and 617206, all from the Hong Kong Research Grants Council.

Authors' address: Department of Computer Science and Engineering, Hong Kong University of Science and Technology; email: {saven, luo}@cse.ust.hk.

to this increasing speed gap, the memory hierarchy has become an important factor for the overall performance of relational query processing [Ailamaki et al. 1999; Boncz et al. 1999]. However, it is a challenging task to consistently achieve a good query processing performance across various memory hierarchies, because both relational database systems and hardware platforms are becoming increasingly complex and diverse. This problem is even more severe for in-memory query processing, because the upper levels of a memory hierarchy, specifically the CPU caches, are hard, if feasible at all, to manage by software.

Cache-conscious techniques are a leading approach to improving the in-memory query processing performance [Bohannon et al. 2001; Boncz et al. 1999; Chen et al. 2004; He et al. 2006; Shatdal et al. 1994; Zhou and Ross 2003]. In this approach, the capacity and block size of a target level in a specific memory hierarchy, for example, the L2 cache, are taken as explicit parameters for data layout and query processing. As a result, cache-conscious techniques can achieve a high performance with suitable parameter values and fine tuning. Nevertheless, it is difficult to determine these suitable parameter values at all times and across platforms [Kim et al. 2004; Samuel et al. 2005; Bender et al. 2006; He and Luo 2006, 2007], since they are affected by the characteristics of the memory hierarchy as well as the system runtime dynamics, such as the number of concurrent threads sharing the cache.

Considering the difficulties faced by the cache-conscious approach, we propose another alternative, namely cache-oblivious query processing, to achieve the goal of improving in-memory performance automatically. Our approach is based on the cache-oblivious model [Frigo et al. 1999], on which an algorithm is aware of the existence of a multilevel memory hierarchy, but assumes no knowledge about the parameter values of the memory hierarchy, such as the number of levels in the hierarchy, the cache block size, and the cache capacity of each level. By eliminating the dependency on the memory parameters, cache-oblivious data structures and algorithms [Frigo et al. 1999; Bender et al. 2000; Brodal and Fagerberg 2002; Brodal et al. 2004; Bender et al. 2006] usually have provable upper bounds on the number of block transfers between any two adjacent levels of an arbitrary memory hierarchy. Furthermore, this memory efficiency asymptotically matches those more knowledgeable external memory algorithms in many cases. With such nice theoretical properties, the cache-oblivious approach can, in principle, achieve a consistently good performance on an arbitrary machine.

In this article, we present in detail our design and implementation of EaseDB, the first cache-oblivious query processor for memory-resident relational databases. Without the knowledge of cache parameters, we employ two methodologies, namely divide-and-conquer and buffering, to develop cache-oblivious algorithms. With these two methodologies, all of our query processing data structures and algorithms asymptotically match the cache efficiency of their cache-conscious counterparts.

We evaluate the performance of EaseDB using both existing microbenchmarks and our homegrown workloads. We also consider the emerging processor techniques, for example, hardware prefetching [Intel Corp. 2004], multicore and multithreading chip designs [Marr et al. 2002; Hardavellas et al. 2007]. We ran

EaseDB on four different platforms without modification or tuning. Through evaluating the query processing algorithms of EaseDB in comparison with their cache-conscious counterparts, we show that our algorithm achieves a performance comparable to the best performance of the fine-tuned cache-conscious algorithm for each platform. Moreover, our cache-oblivious algorithms are up to 28% faster than cache-conscious algorithms on a multithreading processor.

The contributions of this article are as follows.

—We develop the first cache-oblivious query processor, EaseDB, for in-memory relational databases. It includes a full set of relational operators in their cache-oblivious implementation forms. Additionally, we propose a cache-oblivious cost model for estimating the expected cache cost of an algorithm on an arbitrary memory hierarchy. With this cost estimation, we choose the query plan with the minimum cost among multiple candidate plans. We also use this cost model to determine the suitable base case size in the divide-and-conquer methodology in order to reduce the recursion overhead.

—We provide an extensive experimental study on EaseDB in comparison with its cache-conscious counterpart on four different architectures. With our homegrown workloads and existing microbenchmarks, we demonstrate that EaseDB, without any manual tuning, achieves a comparable performance to its cache-conscious counterpart with platform-specific tuning.

—Based on our hands-on experience on developing an efficient cache-oblivious query processor, we discuss the inherent limitations of the cache-oblivious approach as well as the opportunities given by the upcoming hardware architectures.

The remainder of this article is organized as follows. In Section 2, we briefly review the background on the memory hierarchy, and discuss related work on main memory databases as well as cache-conscious and cache-oblivious techniques. In Section 3, we give a system overview of EaseDB. In Section 4, we present our cache-oblivious techniques and use them to develop cache-oblivious algorithms for two core operators in EaseDB, the sort and the join. We experimentally evaluate EaseDB in Section 5. We discuss the strengths and the weaknesses of the cache-oblivious approach in Section 6. Finally, we conclude in Section 7.

2. BACKGROUND AND RELATED WORK

In this section, we first review the background on the memory hierarchy. We next survey related work on main memory databases and cache-centric techniques including both cache-conscious and cache-oblivious techniques.

2.1 Memory Hierarchy

The memory hierarchy in modern computers typically contains multiple levels of memory from top down [Hennessy and Patterson 2002]:

—Processor registers. They provide the fastest data access, usually in one CPU cycle. The total size is hundreds of bytes.

—Level 1 (L1) cache. The access latency is a few cycles and the size is usually tens of kilobytes (KB).
—Level 2 (L2) cache. It is an order of magnitude slower than the L1 cache. Its size ranges from 256 KB to a few megabytes (MB).
—Level 3 (L3) cache. It has a higher latency than the L2 cache. Its size is often several to dozens of megabytes.
—Main memory. The access latency is typically hundreds of cycles and the size can be several gigabytes (GB).
—Disks. The access latency is hundreds of thousands of cycles. The capacity of a single disk can be up to several hundred gigabytes.

Each level in the memory hierarchy has a larger capacity and a slower access speed than its higher levels. A higher level serves as a cache for its immediate lower level and all data in the higher level can be found in its immediate lower level (known as cache-inclusion [Hennessy and Patterson 2002]). In this paper, we use the cache and the memory to represent any two adjacent levels in the memory hierarchy whenever appropriate.

We define a *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 degree of set-associativity. The number of cache lines in the cache is $\frac{C}{B}$. $A = 1$ is a direct-mapped cache, $A = \frac{C}{B}$ a fully associative cache and $A = n$ an n-associative cache ($1 < n < \frac{C}{B}$).

In addition to these static characteristics, we consider *dynamic* characteristics, for example, the number of concurrent threads or tasks using the cache. Compared with static characteristics, dynamic characteristics are more difficult to capture but are important in multi-task systems, such as databases.

Dynamic characteristics become more complex due to the thread-level parallelism in modern architectures, where multiple threads or processes can run simultaneously on a single chip via multiple on-chip processor cores (chip multiprocessors, or CMP [Kim et al. 2004; Hardavellas et al. 2007]) and/ or multiple simultaneous threads per processor core (simultaneous multithreading, or SMT [Marr et al. 2002]). Since concurrent threads share cache resources such as the L2 cache, the overall performance of each thread is affected by the runtime dynamics of the cache.

The notations used throughout this article are summarized in Table I. In the remainder of this article, we assume $||R|| \leq ||S||$.

2.2 Main Memory Databases

Main memory databases have been an active research field since the 1980's [DeWitt et al. 1984; Garcia-Molina and Salem 1992; Lehman and Carey 1986a, 1986b]. Recently, they have attracted an even more significant amount of attention due to the great increase of the memory capacities [Ailamaki et al. 1999; Baulier et al. 1999; Boncz et al. 1999; MonetDB 2006; TimesTen 2006; He et al. 2007]. With large memory capacities, most frequently accessed data items in relational databases are likely to fit into the main memory, and the CPU caches become an important performance factor.

Table I. Notations Used in this Article

Parameter	Description
C	Cache capacity (bytes)
B	Cache line size (bytes)
R, S	Outer and inner relations of the join
r, s	Tuple sizes of R and S (bytes)
$\|R\|,\|S\|$	Cardinalities of R and S
$\|\|R\|\|,\|\|S\|\|$	Sizes of R and S (bytes)
C_S	The base case size in number of tuples
b	The minimum size of buffers in our buffer hierarchy in number of tuples
$Bits$	The total number of bits used in the radix sort

Table II. Main Memory Relational Query Processors

	FastDB	*TimesTen*	*MonetDB*	*EaseDB*
Cache awareness	—	—	cache-conscious	cache-oblivious
Optimizer	rule-based	instruction cost-based	cache cost-based	cache cost-based
Indexes	T-tree, hash index	T-tree, hash index	B+-Tree, T-tree, hash index	B+-Tree, hash index
Database file	virtual memory	main memory	virtual memory	main memory
Storage model	row-based	row-based	column-based	row-based, column-based

We study cache-oblivious algorithms to optimize the CPU cache performance for main memory databases. These algorithms are especially attractive to CPU caches, for example, L1 and L2 caches, because CPU 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. Moreover, even with the knowledge of the cache parameters, the performance of cache-conscious algorithms needs to be tuned carefully. Therefore, we investigate whether and how cache-oblivious algorithms can automatically optimize CPU caches for main memory databases.

Table II summarizes the main features of EaseDB in comparison with three existing main memory relational query processors, including FastDB [FastDB 2002], TimesTen [TimesTen 2006], and MonetDB [MonetDB 2006]. FastDB and TimesTen are among the first-generation main memory databases. Their processing is unaware of CPU caches. In contrast, both MonetDB and EaseDB are cache-optimized. Similar to MonetDB, EaseDB uses cache cost as the cost metric in the query optimizer. The cache cost in EaseDB is an expected value on an arbitrary memory hierarchy as opposed to an estimated value on a specific memory hierarchy in MonetDB. Additionally, recognizing the pros and cons of row-based versus column-based storage, we support both storage models in a cache-oblivious format in EaseDB.

2.3 Cache-Conscious Techniques

Cache-conscious techniques [Shatdal et al. 1994; Boncz et al. 1999; Bohannon et al. 2001; Zhou and Ross 2003; Chen et al. 2004; He et al. 2006] have been the leading approach to optimizing the cache performance of in-memory

Table III. Cache Complexity of Cache-Conscious Algorithms

Algorithm	Complexity
Single search on the B+-tree built on R	$O(\log_B \lvert R\rvert)$ [Comer 1979]
Sorting R	$O(\frac{\lvert R\rvert}{B} \log_C \lvert R\rvert)$ (merge sort) [Knuth 1998] $O(\frac{\lvert R\rvert}{B} \log_C 2^{Bits})$ (radix sort) [Knuth 1998]
Non-indexed nested-loop join (NLJ) on R and S	$O(\frac{\lvert R\rvert \cdot \lvert S\rvert}{C \cdot B})$ [Shatdal et al. 1994]
Indexed NLJ on R and S (with a B+-tree on S)	$O(\lvert R\rvert \cdot \log_B \lvert S\rvert)$ [Ramakrishnan and Gehrke 2003]
Sort-merge join on R and S	sorting R+sorting S+merging [Ramakrishnan and Gehrke 2003]
Hash join on R and S	$O(\frac{\lvert R\rvert + \lvert S\rvert}{B} \log_C \lvert S\rvert)$ [Boncz et al. 1999]

relational query processing. Representatives of cache-conscious techniques include blocking [Shatdal et al. 1994], buffering [Zhou and Ross 2003; He et al. 2006], partitioning [Shatdal et al. 1994; Boncz et al. 1999] and merging [LaMarca and Ladner 1997; Ramakrishnan and Gehrke 2003] for temporal locality, and compression [Bohannon et al. 2001] and clustering [Chilimbi et al. 1999] for spatial locality. We list the *cache complexity* of the state-of-the-art cache-conscious query processing algorithms in Table III. The cache complexity of an algorithm is defined to be the asymptotical number of block transfers between the cache and the memory incurred by the algorithm. In the following, we discuss the cache-conscious techniques for B+-trees, hash joins and storage models, which are most relevant to our work.

Cache-conscious optimizations for the B+-tree typically set the node size to be several cache blocks [Rao and Ross 1999, 2000; Bohannon et al. 2001]. Given a fixed tree node size, both Cache-Sensitive Search Trees (CSS-trees) [Rao and Ross 1999] and Cache-Sensitive B+-Trees (CSB+-trees) [Rao and Ross 2000] increase the tree fanout by eliminating the pointers in each tree node, whereas the partial key technique [Bohannon et al. 2001] increases the tree fanout through compression. The CSS-tree eliminates all pointers in the tree node. Thus, the nodes are fully packed with keys and are laid out contiguously, level by level. The CSB+-tree stores groups of sibling nodes in consecutive memory areas. One node stores one pointer only. The partial key technique reduces the key size by storing only part of the key. All these techniques require the knowledge of cache block size. However, this static block size is often different from the effective block size due to the advanced memory techniques, such as hardware prefetching data from the main memory to the CPU cache [Intel Corp. 2004].

Partitioning schemes have been proposed to improve the in-memory performance of hash joins [Boncz et al. 1999; Shatdal et al. 1994]. Shatdal et al. [1994] showed that cache partitioning, in which the partitioning granularity is smaller than the cache capacity, can improve the cache performance of hash joins. This partitioning was performed in a single pass. The major problem for the single-pass partitioning is that a large relation may result in a large partitioning fanout, which in turn causes cache thrashing and TLB (Translation Lookaside Buffer) thrashing. To solve this problem, Boncz et al. [1999] proposed the radix join, which uses a multi-pass partitioning method. For each

pass, the partitioning fanout is tuned according to the cache or TLB parameters. More recently, Zhou et al. [2005] applied staging to restructure the hash join, and proposed a multi-threaded hash join using a worker thread to preload the cache lines required in the join on a SMT processor. Garcia et al. [Garcia and Korth 2006] demonstrated software prefetching could further improve the multi-threaded hash join on CMP/SMT processors. These two studies mainly exploit the parallelism and the memory optimization within a single join. In addition to the optimizations within a single operator, we also investigate the cache behavior among concurrent operators on the CMP/SMT processors.

Two common storage models are NSM (N-ary Storage Model) [Ramakrishnan and Gehrke 2003] and DSM (Decomposition Storage Model) [Copeland and Khoshafian 1985], which store the relation into a cache block in rows and columns, respectively. Alternatives to NSM and DSM were PAX (Partition Attributes Across) [Ailamaki et al. 2001] and data morphing [Hankins and Patel 2003]. They are essentially a cache block level decomposition model where each attribute or each group of attributes is stored within a cache block. They are found to have a good performance for both the disk and the CPU caches. More recently, a column-based variant called C-Store [Stonebraker et al. 2005] was developed to optimize the read performance for database applications.

The major drawback of cache-conscious algorithms is that they explicitly take cache parameters as input and require tuning on these parameter values. A common way of obtaining these cache parameters is to use calibration tools [Manegold 2004]. One problem of calibration is that some calibration results may be inaccurate or missing, especially as the memory system becomes more complex and diverse. For example, the calibrator [Manegold 2004] did not give the characteristics of TLB on the P4 or Ultra-Sparc machines used in our experiments. Even with the knowledge of the cache configuration, the suitable parameter values for a cache-conscious algorithm need further tuning. First, the best parameter values for the cache-conscious algorithm may be none of the cache parameters. Second, these values are affected by the cache dynamics on CMP/SMT processors. In contrast, cache-oblivious algorithms are an automatic approach to performance optimization for the entire memory hierarchy without any tuning.

2.4 Cache-Oblivious Techniques

The cache-oblivious model [Frigo et al. 1999] assumes a two-level memory hierarchy, the cache and the memory. The cache is assumed to be tall ($C \gg B^2$), fully associative, and has an optimal replacement policy. Frigo et al. [1999] showed that, if an algorithm has an optimal cache complexity in the cache-oblivious model, this optimality holds on all levels of a memory hierarchy.

Two main methodologies of designing a cache-oblivious algorithm are divide-and-conquer and amortization [Demaine 2002]. The divide-and-conquer methodology is widely used in general cache-oblivious algorithms, because it usually results in a good cache performance. In this methodology, a problem is

recursively divided into a number of subproblems. At some point of the recursion, the subproblem can fit into the cache, even though the cache capacity is unknown. The amortization methodology is used to reduce the average cost per operation for a set of operations, even though the cost of a single operation may be high.

Existing work on cache-oblivious algorithms has focused on basic computation tasks, for example, sorting and matrix operations [Frigo et al. 1999; Brodal and Fagerberg 2002], and basic data structures, for example, cache-oblivious B-trees [Bender et al. 2000; Bender et al. 2006]. These algorithms have the same cache complexity as their cache-conscious counterparts. Moreover, previous studies [Brodal et al. 2004; Bender et al. 2006] have shown that cache-oblivious algorithms can outperform the best cache-conscious algorithms in the main memory or the disk. In comparison, we develop cache-oblivious techniques for relational query processing, and pay attention to both the cache complexity of our algorithms and their performance on real systems. In particular, we implement the merge sort using a buffer hierarchy similar to the funnel sort [Frigo et al. 1999; Brodal et al. 2004] with two major differences. One is that the previous work defines the buffer size in theoretical bounds, whereas we define exact buffer sizes considering the cache reuse within the hierarchy. The other difference is that we propose a buffer reuse technique to reduce the memory consumption. Yoon et al. [Yoon and Lindstrom 2006] proposed a cache-oblivious cost model for the mesh layout. The cost model estimates the expected number of cache misses caused by the accesses to a mesh. In contrast, our cost model estimates the expected cache cost between the cache and the memory for general query processing algorithms.

Our previous work [He and Luo 2006] designed cache-oblivious algorithms for nested-loop joins (NLJs). In our position paper [He and Luo 2007], we briefly discussed our initial efforts in developing EaseDB. Additionally, we gave a system demonstration of the internal working mechanisms and the end-to-end performance of EaseDB in comparison with cache-conscious algorithms [He et al. 2007]. In contrast, this paper provides a detailed description of the architectural design and implementation issues of EaseDB with a focus on discussing the limitations and the opportunities for cache-oblivious query processing. Moreover, we investigate the cache behavior of cache-oblivious algorithms on emerging CMP/SMT processors.

3. SYSTEM OVERVIEW

In this section, we describe the architectural design and implementation of EaseDB. The goal of EaseDB is to automatically and consistently achieve a good performance on various platforms without any platform-specific tuning.

3.1 Components

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 of a query plan generator and a cache-oblivious

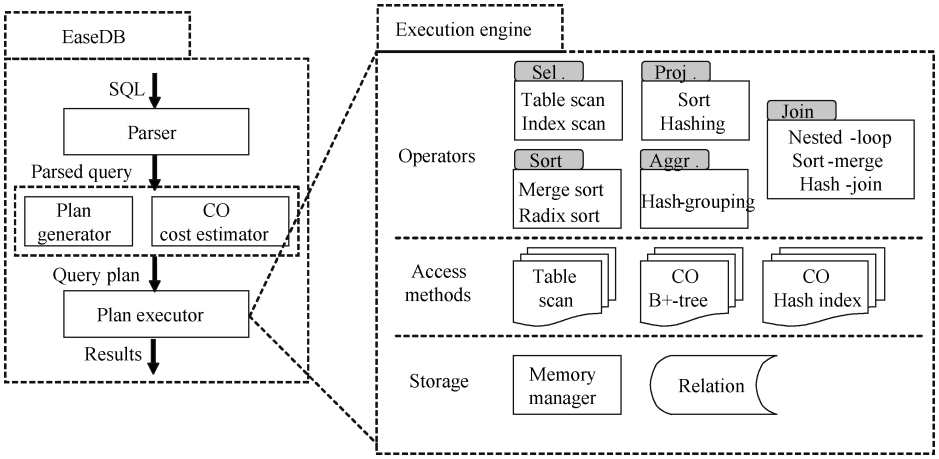


Fig. 1. The system architecture of EaseDB.

cost estimator. We use a Selinger-style optimizer [Selinger et al. 1979] for plan generation.

EaseDB employs a multi-threading mechanism at the query level. It creates a pool of worker threads when it starts up. When a query is issued to the system, EaseDB schedules a free worker thread to handle the query.

In the following, we focus our discussion on the plan executor and the cost estimator.

3.2 Execution Engine

We divide the execution engine into three layers, including the storage, access methods, and query operators.

3.2.1 *Storage.* As in previous work [MonetDB 2006; Boncz et al. 1999], we focus on the read-only queries. Thus, EaseDB uses arrays to represent relations either in rows (as NSM [Ramakrishnan and Gehrke 2003]) or in columns (as DSM [Copeland and Khoshafian 1985]). EaseDB does not support on-line updates. Instead, it rebuilds a relation for batch updates. Since recent work [Ailamaki et al. 2001; Stonebraker et al. 2005] has shown that column-based and row-based storage models have their own pros and cons, we support both of them. Specifically, we allow a table to be stored in one format and to have a full table replica or a different table stored in the other format at one point in time. We let the user explicitly specify which storage model(s) to use for each table. It will be an interesting extension to develop methods to choose the suitable storage model for a given workload.

EaseDB supports variable-sized columns using two arrays, *values* and *start*. Array *values* stores the values of the column. Array *start* stores the pair of the record identifier and the start position in the *values* array.

EaseDB has a memory manager to handle memory allocation and deallocation. Similar to other main memory databases [FastDB 2002; MonetDB 2006; TimesTen 2006], EaseDB does not have a global buffer manager. When EaseDB

starts up, it pre-allocates a large memory pool. The memory space required for query processing is allocated from the pool on demand.

3.2.2 *Access Methods and Query Operators.* EaseDB supports three common access methods, including the table scan, the B+-tree and the hash index.

—*Table scan.* The relation is sequentially scanned. If the relation is sorted, a binary search is performed on the relation according to the sort key.

—*B+-trees.* We have implemented two versions of COB+-trees, with and without pointers (explicit vs. implicit addressing). In the latter version, we use an existing implicit addressing scheme [Brodal et al. 2002]. We compared these two versions of COB+-trees, and found that COB+-trees without pointers outperformed those with pointers when the COB+-tree with pointers can not fit into the L2 cache. Thus, we used the COB+-tree without pointers in our experiments.

—*Hash indexes.* The hash index for R consists of $|R|$ buckets so that each bucket contains one tuple on average. We implement the hash index using two arrays. One is the array of hash headers, each of which maintains a pointer to its corresponding bucket. The other is the array of buckets. Each bucket stores the key values of the records that have the same hash value, and their record identifiers. Due to the larger number of hash headers, our hash index has more space overhead than the cache-conscious hash index. The memory efficiency bound of a probe on the hash index is $O(1)$, which matches the memory efficiency bound of a probe on the cache-conscious hash index [Ramakrishnan and Gehrke 2003].

EaseDB supports the following common query processing operators.

—*Selection.* In the absence of indexes, a sequential scan or binary search on the relation is used. In the presence of indexes, if the selectivity is high, a data scan on the relation is performed. Otherwise, the B+-tree index or the hash index can be used.

—*Projection.* If duplicate elimination is required, we use either sorting or hashing to eliminate the duplicates for the projection.

—*Sorting.* We consider two sorting algorithms, the cache-oblivious radix sort and the cache-oblivious merge sort. We determine which sorting algorithm to use in a cost-based way (Details in Section 4.2).

—*Grouping and aggregation.* We use the build phase of the hash join to perform grouping and aggregation.

—*Joins.* We consider cache-oblivious nested-loop joins with or without indexes, sort-merge joins, and hash joins.

3.3 Cost Estimator

When multiple query plans are available, the optimizer chooses the one of the minimum cost. In the optimizer of EaseDB, the cost estimator estimates the cache cost of a query plan in terms of the data volume transferred between the cache and the memory in the cache-oblivious model. Compared with the

cache-conscious cost model [Boncz et al. 1999], our estimation assumes no knowledge of the cache parameters of each level or the number of levels in a specific memory hierarchy.

Our cost estimator is based on a two-level memory hierarchy with the following characteristics for simplicity of cost estimation. First, the cache block size is a power of two. Second, C is no less than B^2 according to the tall cache assumption ($C \geq B^2$) [Frigo et al. 1999]. Third, the cache is fully associative and uses an optimal cache replacement policy. The latter two characteristics are in the original cache-oblivious model, and the first one is our own simplifying assumption. These assumptions are based on the fundamental properties of the memory hierarchy rather than specific parameter values. Consequently, there is no platform-specific tuning or calibration involved.

To compute the expected volume of data transferred between the cache and the memory caused by an algorithm, we need a cost function for the algorithm. This cost function estimates the number of cache misses caused by the algorithm for a given cache capacity and cache block size. The cache misses include compulsory and capacity misses, since the number of conflict misses is zero in the fully associative cache of our model. We denote this cost function to be $F(C, B)$.

Suppose a query plan has a working set size ws bytes, which is the total size of the data (e.g., relations and indexes) involved in the query plan. We consider all possible combinations of the cache capacity and the cache block size to compute the expected volume of data transferred between the cache and the memory on an arbitrary memory hierarchy. If $C \geq ws$, this working set can fit into the cache. In such cases, once data in the working set are brought into the cache, they stay in the cache for further processing. That is, further processing does not increase the volume of data transferred. Therefore, we estimate the volume of data transferred to be zero when the cache capacity is larger than the working set of the query plan. Thus, given a certain cache block size, B_x, we consider all possible C values, that is, B_x^2, $B_x^2 + B^x$, $B_x^2 + 2B^x, \ldots,$ $\lceil \frac{ws}{B_x} \rceil B_x$, and compute the expected volume of data transferred to be $\frac{1}{\lceil \frac{ws}{B_x} \rceil - B_x + 1}(B_x \cdot F(B_x^2, B_x) + B_x \cdot F(B_x^2 + B^x, B_x) + B_x \cdot F(B_x^2 + 2B^x, B_x) + \cdots + B_x \cdot F(\lceil \frac{ws}{B_x} \rceil B_x, B_x)) = \frac{B_x}{\lceil \frac{ws}{B_x} \rceil - B_x + 1} \sum_{C_x = B_x^2}^{\lceil \frac{ws}{B_x} \rceil B_x} F(C_x, B_x)$.

Subsequently, we estimate the expected volume of data transferred to be $\mathcal{Q}(F)$, as shown in Equation (1).

$$\mathcal{Q}(F) = \frac{1}{\lceil \log_2 \sqrt{ws} \rceil + 1} \times \left(\sum_{k=0, B_x = 2^k}^{k = \lceil \log_2 \sqrt{ws} \rceil} \frac{B_x}{\lceil \frac{ws}{B_x} \rceil - B_x + 1} \sum_{C_x = B_x^2}^{\lceil \frac{ws}{B_x} \rceil B_x} F(C_x, B_x) \right). \quad (1)$$

Our model includes the estimations of some unrealistic cache capacities and cache block sizes. In contrast, the cache-conscious cost model [Boncz et al. 1999] includes the estimation with the exact cache parameter values of a specific machine. Therefore, we consider a hybrid approach by adding the ranges on the cache parameters to our cost model as a tradeoff between the knowledge of the exact cache parameter values and the estimations on the unrealistic cache parameter values using our cache-oblivious cost model.

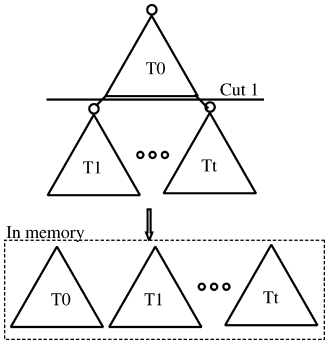


Fig. 2. The cache-oblivious B+-tree.

4. ALGORITHMS

In this section, we first develop our cache-oblivious techniques applying the divide-and-conquer methodology and buffering. Next, we use these techniques to develop our cache-oblivious algorithms for two core operators, the sort and the join, in relational databases. Additionally, we present the results on the cache complexity of our cache-oblivious algorithms. Finally, we describe a cost-based way of estimating the suitable base case size for a divide-and-conquer algorithm.

4.1 Techniques

We develop our cache-oblivious techniques considering the divide-and-conquer methodology and buffering, as well as considering that query processing algorithms are inherently data centric.

4.1.1 *Applying the Divide-and-Conquer Methodology*. Following the divide-and-conquer methodology, we propose to use three cache-oblivious techniques, *recursive clustering* for the spatial locality, and *recursive partitioning* as well as *recursive merging* for the temporal locality.

As its name suggests, recursive clustering recursively places related data together so that a cluster fits into one cache block at some level of the recursion. An example of recursive clustering is the cache-oblivious B+-tree (COB+-tree) [Bender et al. 2000, 2004]. A COB+-tree is obtained by storing a complete binary tree according to the van Emde Boas (VEB) layout [van Emde Boas et al. 1977], as illustrated in Figure 2. Suppose the tree consists of N nodes and has a height of $h = \log_2(N+1)$. The VEB layout proceeds as follows. It first cuts the tree at the middle level, i.e., the edges below the nodes of height $h/2$. This cut divides the tree into a *top* subtree, T_0, and approximately $\sqrt{N}$ *bottom* subtrees below the cut, $T_1, T_2, \ldots$, and T_t. Each subtree contains around $\sqrt{N}$ nodes. Next, it recursively stores these subtrees in the order of $T_0, T_1, T_2, \ldots$, and T_t. With the VEB layout, an index node and its child nodes are stored close to each other. Thus, the spatial locality of the tree index is improved. Bender et al. showed that a search on the COB+-tree has a cache complexity matching that on the cache-conscious B+-tree (CCB+-tree) [Bender et al. 2000].

Recursive partitioning recursively divides data into two subpartitions so that at some level of the recursion a subpartition is fully contained in some level of the memory hierarchy. An example of recursive partitioning is the quick sort, one of the most efficient sorting algorithms in the main memory.

The cache complexity of recursive partitioning is higher than that of cache-conscious partitioning. Suppose we divide relation R into N partitions. The cache-conscious partitioning algorithm (such as the radix cluster [Boncz et al. 1999]) has a partitioning fanout of C/B. This multi-pass partitioning algorithm achieves a cache complexity of $O(\frac{|R|}{B} \log_C N)$. In contrast, the cache complexity of recursive partitioning is $O(\frac{|R|}{B} \log_2 N)$.

Recursive merging can be considered as a dual technique to recursive partitioning. It recursively combines two partitions into one. An example for recursive merging is the merging phase of the binary merge sort. At the beginning, the relation is divided into multiple partitions. It then sorts each partition and recursively merges every two partitions into one. This recursive merging process will not end until all data are merged into one partition. Suppose we merge N partitions of size $O(1)$. The cache complexity of recursive merging is $O(\frac{N}{B} \log_2 N)$, which is higher than that of its cache-conscious counterpart, $O(\frac{N}{B} \log_C N)$ [Knuth 1998].

Note that recursive partitioning and recursive merging by themselves have no constraints on the partitioing/merging fanout. We set the fanout to be a small constant, two, due to the cache-obliviousness.

4.1.2 *Applying the Buffering Technique.* We use the buffering technique to improve the cache efficiency of recursive partitioning and recursive merging. In the following, we first introduce the buffering technique for recursive partitioning in detail and then briefly discuss the technique for recursive merging.

The high cache complexity of recursive partitioning is because the cost of processing each tuple in the partitioning is high in the worst case. Since each tuple requires to be processed multiple times and there are a larger number of tuples in recursive partitioning, we consider using buffers to improve the cache locality and the overall performance.

To match the cache complexity of the cache-conscious partitioning, we design a cache-oblivious buffer hierarchy, as shown in Figure 3(a). We call this hierarchy a *partitioner tree*. Each node of a partitioner tree is a partitioner, which decomposes a relation into two sub-relations. The partitioner can use either range partitioning or hash partitioning. Each non-leaf partitioner has two output buffers for temporarily storing tuples having been processed by the partitioner. The input buffer of a nonroot partitioner is the output buffer of its parent partitioner. The tuples having been processed by a leaf node are inserted into the result partitions. If we divide relation R into N partitions, the partitioner tree consists of $N/2$ leaf partitioners.

We use the following representations for a partitioner: *level* is its level in the partitioner tree, *buf* [0, 1] are the two output buffers for a nonleaf node or the two result partitions for a leaf node, and *child*[0, 1] are the pointers to its two child partitioners (if any). The partitioner tree has an attribute *root* pointing to the root partitioner. The level of the root node is one. We define two operations

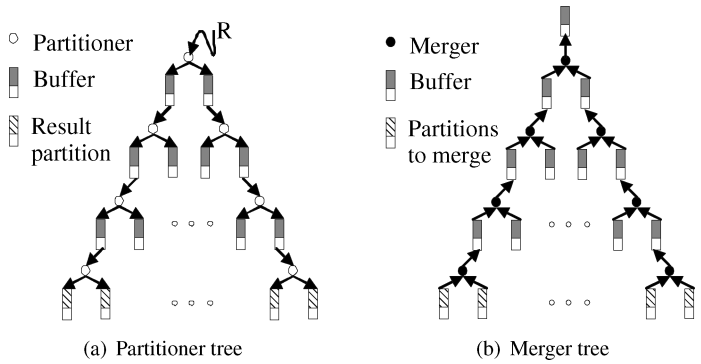


Fig. 3. Cache-oblivious buffer hierarchies: (left) partitioner tree, (right) merger tree. The size of the buffers in the hierarchy is recursively defined.

on the buffer, *fill* and *flush*. *Fill* inserts a tuple at the tail of the buffer. *Flush* removes all tuples from the buffer, and fills them into the buffers at lower levels or result partitions.

Algorithm 1 uses a hash partitioning scheme for recursive partitioning with buffers. The set of hash functions for the partitioners are $oid = H(x, level)$, where x is the value of the hashing attribute of the tuple and $oid \in \{0, 1\}$. For a partitioner at level $level$, the result of the hash function is the ID of its output buffer or the result partition that the tuple should go to. For instance, the radix sort uses $H(x, level) = (x >> (Bits - level))\&1$. Procedures *fillBuf* and *fillResult* implement the fill operation on a buffer and the insertion of a tuple into a result partition, respectively. Procedure *flush* implements the flush operation on a buffer. Procedure *flushTree* flushes all the buffers in the tree in the depth-first order.

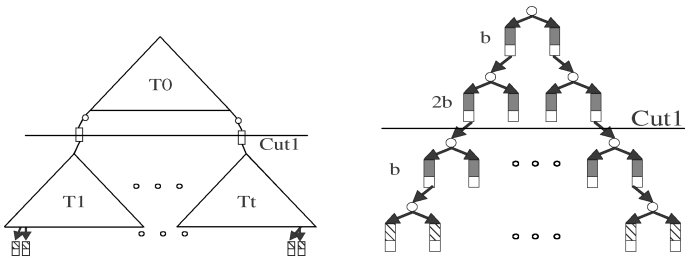
Having defined the data structure and the partitioning algorithm, we now describe a recursive scheme to determine the size of each buffer. The basic idea of our definition is that at each level of the recursion, the size of the input buffer for the root node of a subtree is set to be the total size of the buffers in the subtree. Thus, the input buffer size of a node is defined as the total buffer size of the nodes in the subtree resulting from the recursion, not an arbitrary subtree in the tree structure. We use this design because the temporal locality of the subtree will be high, if the total size of all buffers in the subtree and the input buffer for the root node is smaller than the cache capacity.

We define the size of each buffer following the VEB recursion [van Emde Boas et al. 1977], which is widely used in cache-oblivious algorithms [Bender et al. 2000; Brodal and Fagerberg 2002; Brodal et al. 2004; He and Luo 2006]. We follow the VEB recursion and cut the tree at the middle level. This cut results in a top tree as the subtree above the cut and multiple bottom trees below the cut, as shown in Figure 4(a). Next, we set the size for the buffers at the middle level.

At each recursion level, if a bottom tree contains k buffers, we set the input buffer size for its root node to be $(b \cdot k \log_2 k)$ in number of tuples, where b is the minimum buffer size in number of tuples. Denote $S(N)$ to be the total size (in

Algorithm 1. Partitioning R into N_p partitions: recursive partitioning with a partitioner tree

```
/*part_buf: divide R into N_p partitions with buffers*/
Procedure: part_buf(R, N_p)

1: Construct a partitioner tree, T, of height L = log_2 N_p;
2: baseLevel = 0; /*a global variable representing the adjustment for the level of the
   partitioner*/
3: for each tuple r in R do
4:         fillBuf(T.root, r);
5: flushTree(T.root);

/*fillBuf: fill tuple into node's output buffer*/
Procedure: fillBuf(node, tuple)

1: aL = node.level + baseLevel;
2: bufID = H(tuple.key, aL); /*compute the buffer ID with adjusted level*/
3: let oBuf be node.buf[bufID];
4: if oBuf.full() then
5:         flush(node, bufID);
6: oBuf.fill(tuple);

/*fillResult: fill tuple into node's result partition*/
Procedure: fillResult(node, tuple)

1: aL = node.level + baseLevel;
2: partID = H(tuple.key, aL); /*compute the result partition ID with adjusted level*/
3: let oPart be node.buf[partID];
4: oPart.fill(tuple);

/*flush: flush node's buf[bufID]*/
Procedure: flush(node, bufID)

1: if node.level < L then
2:         for each tuple t in node.buf[bufID] do
3:             fillBuf(node.child[bufID], t); /*insert tuples to the child buffers*/
4: else
5:         for each tuple t in node.buf[bufID] do
6:             fillResult(node.child[bufID], t); /*insert tuples to the result partitions*/

/*flushTree: flush the buffers in the subtree rooted at node recursively*/
Procedure: flushTree(node)

1: for bufID = 0, 1 do
2:         flush(node, bufID);
3:         flushTree(node.child[bufID]);
```



(a) VEB recursion (b) An example

Fig. 4. The VEB buffer size definition of the partitioner tree.

number of tuples) of buffers in the partitioner tree consisting of N buffers. Note, the size of the input buffer for the root node is not included in $S(N)$. At the first cut of the recursion, each bottom tree has around $N^{1/2}$ buffers. According to our definition, the input buffer size for the root node of each bottom tree equals the total size of the buffers in the tree, $S(N^{1/2})$. According to the VEB recursion, we have the formula, $S(N) = N^{1/2}S(N^{1/2}) + (N^{1/2} + 1) \cdot S(N^{1/2})$. We solve this formula, eliminate lower-order terms, and obtain $S(N) = b \cdot N \log_2 N$. Thus, we set the input buffer size for the root node of the bottom tree to be around $b \cdot N^{\frac{1}{2}} \log_2 N^{\frac{1}{2}}$. The sizes of buffers in the top tree and the bottom trees are recursively defined following the VEB recursion. This process ends when the tree contains only one level. We then set the size of each of its output buffers to be b.

An example of our VEB buffer size definition on a partitioner tree of four levels is shown in Figure 4(b). The first cut of the VEB recursion on this tree is between levels two and three. This results in one top tree and four bottom trees. Since each bottom tree has two buffers, the size of the input buffer for its root node is $b \cdot 2 \log_2 2 = 2b$. Further cuts on each subtree complete the setting of the buffer sizes in the tree.

Proposition 1 gives the total buffer size for a partitioner tree.

PROPOSITION 1. *Given the number of partitioners in a partitioner tree, N, the total buffer size for the partitioner tree is $S(N) = b \cdot N \log_2 N$.*

PROOF. This proposition is directly derived from our buffer size definition. □

While examining our buffer scheme for the partitioner tree on sorting R using the radix sort, we find that the total buffer size is more than linear to $|R|$. In the basic implementation of the radix sort using the partitioner tree, the number of partitioners is $O(|R|)$. According to Proposition 1, the total buffer size is superlinear to the relation size. To improve the scalability of recursive partitioning, we need to reduce the total size of the buffers used in the partitioner tree.

One observation on the first cut of the VEB recursion is that the top tree is similar to a bottom tree. Specifically, the height of each of these subtrees is around one half of the height of the original tree. Based on this observation, we propose a *buffer reuse* technique to reduce the total buffer size, as illustrated in Algorithm 2. First, we construct a partitioner tree, T, whose height is one half

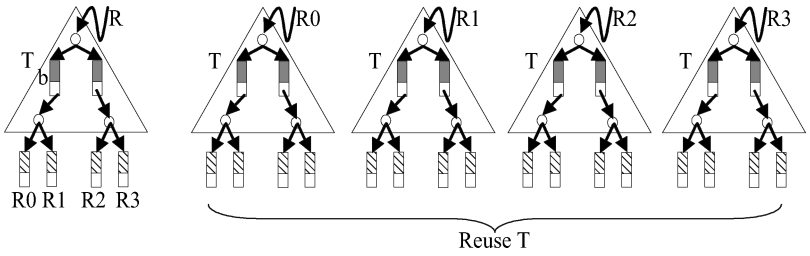


Fig. 5. Buffer reuse. Relation R is first decomposed into four partitions, $R_0, \ldots, R_3$, with T. Next, each partition is decomposed by reusing T.

of the original partitioner tree. This tree is similar to the subtrees obtained at the first level of the recursion in the original tree. Next, we use T to decompose R into multiple partitions, and reuse T to further decompose each partition. The total buffer size with buffer reuse becomes $S(N^{1/2}) = b \cdot N^{1/2} \log_2 N^{1/2}$, which is sublinear to $|R|$. Take the partitioner tree in Figure 4(b) as an example. The partitioning process with buffer reuse constructs a tree of height two, as illustrated in Figure 5. This process essentially performs four-pass binary partitioning on R and generates 16 result partitions.

Algorithm 2. Partitioning R into N_p partitions with buffer reuse

/*$part\_reuse$: divide R into N_p partitions with buffer reuse*/
Procedure: $part\_reuse(R, N_p)$

1: Construct a partitioner tree, T, of height $L = \frac{1}{2} \log_2 N_p$;
2: $baseLevel = 0$;
3: **for** each tuple r in R **do**
4: $\quad$ *fillBuf*$(T.root, r)$;
5: $\quad$ *flushTree*$(T.root)$;
6: $baseLevel = L$; /*adjust the global variable.*/
7: **for** $pid = 0; pid < 2^L; pid++$ **do**
8: $\quad$ **for** each tuple r in R_{pid} **do**
9: $\quad\quad$ *fillBuf*$(T.root, r)$;
10: $\quad$ *flushTree*$(T.root)$;

Having discussed the partitioner tree structure for recursive partitioning, we present our buffering technique for recursive merging. We develop a buffer hierarchy called a *merger tree* to improve the cache performance of recursive merging. As shown in Figure 3(b), the merger tree has a similar buffer hierarchy to the partitioner tree. The buffer size definition of the merger tree is the same as the partitioner tree. The major difference is the direction of data flow. Specifically, data flow from top down in the partitioner tree so that the data from the buffer at a higher level are partitioned and temporarily stored in the buffers at a lower level. In contrast, data move from bottom up in the merger tree so that the data from the buffers at a lower level are merged and temporarily stored in the buffer at a higher level. To reduce their total buffer

size of the merger tree, we develop the buffer reuse technique for the merger tree in a similar way to the partitioner tree.

4.2 Sort

We have developed two cache-oblivious sorting algorithms, namely the cache-oblivious radix sort and the cache-oblivious merge sort.

We choose to implement a most significant digit (MSD) radix sort for its divide-and-conquer nature [Knuth 1998]. We use recursive partitioning to develop the cache-oblivious radix sort algorithm. The recursive partitioning in the radix sort is to recursively partition the relation into two subrelations according to the bits of the sorting key. At the ith pass ($i \geq 1$) of the partitioning, we use the $(Bits - i + 1)$th bit from the right, where $Bits$ is the total number of bits required in the radix sort. This recursion ends when the number of tuples in the relation is less than C_S (the base case size) or the sort reaches the least significant bit. We use the insertion sort to sort the base case, because the base case is small or is nearly sorted. The recursive partitioning is implemented with our partitioner tree and the buffer reuse technique. The height of the partitioner tree is the minimum value of $\frac{Bits}{2}$ and $\frac{1}{2} \cdot \log_2 \frac{|R|}{C_S}$.

After presenting the cache-oblivious radix sort, we describe the cache-oblivious merge sort in brief. The cache-oblivious merge sort is a dual algorithm to the cache-oblivious radix sort. It starts with dividing the relation into $\frac{|R|}{C_S}$ partitions. Each partition consists of C_S tuples. We sort each partition using the insertion sort. Next, we construct a merger tree of height $L = \frac{1}{2} \log_2 \frac{|R|}{C_S}$. We use the merger tree to merge every 2^L consecutive partitions. Thus, the number of intermediate result partitions is 2^L. Finally, we use the merger tree to merge these intermediate result partitions. The final result partition is the sorted relation.

We develop a cost-based method to compare the performance of the radix sort and the merge sort. The cache complexity of the first ith pass ($1 \leq i \leq Bits$) in the partitioning process of the radix sort matches that of the last ith pass (if any) in the merging process of the merge sort. Thus, the performance comparison of the radix sort and the merge sort can be determined by the comparison of the number of partitioning passes in the radix sort and the number of merging passes in the merge sort. If the number of merging passes in the merge sort is larger than the number of partitioning passes in the radix sort, we choose the radix sort. Otherwise, we choose the merge sort. Formally, given relation R, we choose the merge sort when $\log_2 \frac{|R|}{C_S} \leq Bits$, i.e., $\frac{|R|}{C_S} \leq 2^{Bits}$.

4.3 Joins

We have implemented cache-oblivious algorithms for the four most common join implementations in relational databases [Ramakrishnan and Gehrke 2003], including the nested-loop join with or without indexes, the sort-merge join, and the hash join.

4.3.1 *Nested-Loop Joins.* Our previous work applies recursive partitioning to implement the non-indexed NLJ and recursive clustering to implement the

indexed NLJ [He and Luo 2006]. To make our presentation self-contained, we briefly review these cache-oblivious nested-loop join algorithms.

The non-indexed NLJ (denoted as CO_NLJ) performs recursive partitioning on both relations. With recursive partitioning, both relations are recursively divided into two equal sized subrelations, and the join is decomposed into four smaller joins on the subrelation pairs. This recursive process goes on until the base case is reached; that is, the number of tuples in the inner relation is no larger than C_S. The base case is evaluated using the simple tuple-based NLJ algorithm.

The indexed NLJ is similar to the traditional one [Ramakrishnan and Gehrke 2003] except that the tree index is the cache-oblivious B+-tree [Bender et al. 2000; He and Luo 2006]. Our previous work [He and Luo 2006] proposed a buffer hierarchy, similar to that in a partitioner tree, for the temporal locality of the indexed NLJ. The main weakness of this algorithm is that it does not support pipelined execution, whereas the algorithm without buffering does. For simplicity and scalability, we choose the indexed NLJ without buffering.

4.3.2 *Sort-Merge Joins.* The sort-merge join works in two steps. In the first step, it sorts both relations using a cache-oblivious sorting algorithm. It chooses either the radix sort or the merge sort for each relation. The sorting algorithms for the two relations may be different. In the second step, we perform a merge join on the two sorted relations. This merge join algorithm is similar to the traditional merge join [Ramakrishnan and Gehrke 2003]. The major difference is that we use the cache-oblivious nonindexed nested-loop join algorithm to evaluate the join of tuples with duplicate keys.

4.3.3 *Hash Joins.* We use recursive partitioning with buffer reuse to implement the cache-oblivious hash join. We perform partitioning on the build relation R as well as on the probe relation S.

The join algorithm includes the following five steps. Steps 1–3 are the build phase of the hash join, which constructs a number of hash tables. We construct a hash table for each result partition obtained from Step 3. Steps 4–5 are the probe phase of the hash join, which uses the S tuples to probe the corresponding hash table of R for matching.

Step 1. It constructs a partitioner tree, T, of height $L = \frac{1}{2}\log_2 \frac{|R|}{C_S}$.

Step 2. It uses T to decompose R into p partitions ($p = 2^L$), $R_0, \ldots, R_{p-1}$.

Step 3. It uses T to further decompose R_i ($0 \leq i < p$). The tuples having been processed by the leaf partitioner of T are inserted into the hash table corresponding to the result partition.

Step 4. It decomposes S into p partitions using T, $S_0, \ldots, S_{p-1}$.

Step 5. It uses T to further decompose S_i ($0 \leq i < p$). The tuples having been processed by the leaf partitioner of T are used to probe the corresponding hash table of R_i for matching.

4.4 Cache Complexity

Having presented our cache-oblivious data structures and algorithms, we give their cache complexity.

Propositions 2 and 3 give the cache complexity of recursive partitioning with the partitioner tree and recursive merging with the merger tree, respectively.

PROPOSITION 2. *The cache complexity of partitioning R into N partitions with a partitioner tree is $O(\frac{|R|}{B}\log_C N)$.*

PROOF. We consider the soonest level of the VEB recursion at which all buffers in each subtree (either the top tree or a bottom tree) and the input buffer of its root node can fit into the cache. These subtrees are denoted as base trees [Brodal and Fagerberg 2002]. Suppose each base tree contains k buffers. The total size of the buffers in this tree and the input buffer of its root node is $(2rbk\log_2 k)$ bytes. Since a subtree at the previous recursion level contains approximately k^2 buffers, and can not fit into the cache. The total size of buffers in this tree and the input buffer of its root node is $(2rbk^2\log_2 k^2)$ bytes. Therefore, $2rbk\log_2 k < C$ and $2rbk^2\log_2 k^2 \geq C$. Taking "$\log_2$" on both sides of these two formulas, we have $\frac{1}{4}\log_2\frac{C}{2br} \leq \log_2 k < \log_2\frac{C}{2br}$.

Since the cache lines of a base tree and the input buffer of its root node is reused over $(bk\log_2 k)$ tuples, the average cost of each tuple for partitioning on a base tree is $O(\frac{2rbk\log_2 k}{B}/(bk\log_2 k)) = O(1/B)$. During partitioning, the number of base trees that each tuple must go through is $\log_2 N/\log_2 k$. Given $\frac{1}{4}\log_2\frac{C}{2br} \leq \log_2 k < \log_2\frac{C}{2br}$, we have $\log_2 N/\log_2 k = O(\log_2 N/\log_2 C) = O(\log_C N)$. Thus, the average cost of each partitioning tuple through the entire tree is $O(\frac{1}{B}\log_C N)$. Therefore, the cache complexity of partitioning R into N partitions with a partitioner tree is $O(\frac{|R|}{B}\log_C N)$. □

PROPOSITION 3. *The cache complexity of merging N partitions of size $O(1)$ with a merger tree is $O(\frac{N}{B}\log_C N)$.*

PROOF. Since merging is a dual problem of partitioning, we can prove this proposition in a similar way to Proposition 2. □ □

Propositions 4 and 5 give the cache complexity of the cache-oblivious radix sort and the cache-oblivious merge sort, respectively.

PROPOSITION 4. *The cache complexity of sorting R using the cache-oblivious radix sort algorithm is $O(\frac{|R|}{B}\log_C 2^{Bits})$.*

PROOF. The number of result partitions in the radix sort is $O(2^{Bits})$. According to Proposition 2, the cache complexity of sorting relation R with the radix sort is $O(\frac{|R|}{B}\log_C 2^{Bits})$. □

PROPOSITION 5. *The cache complexity of sorting R using the cache-oblivious merge sort algorithm is $O(\frac{|R|}{B}\log_C |R|)$.*

PROOF. The merge sort starts with $\frac{|R|}{C_S}$ partitions. Each partition consists of C_S tuples. According to Proposition 3, the cache complexity of sorting R with the merge sort is $O(\frac{|R|}{B}\log_C |R|)$. □

Propositions 6–9 give the cache complexity of the cache-oblivious nonindexed NLJ, the cache-oblivious indexed NLJ, the cache-oblivious sort-merge join and the cache-oblivious hash join, respectively.

PROPOSITION 6 (HE AND LUO 2006). *The cache complexity of the cache-oblivious nonindexed NLJ is* $O(\frac{|R|\cdot|S|}{CB})$.

PROPOSITION 7 (HE AND LUO 2006). *The cache complexity of the cache-oblivious indexed NLJ with the COB+-tree is* $O(|R| \log_B |S|)$.

PROPOSITION 8. *The cache complexity of the cache-oblivious sort-merge join is* $O(COM_R + COM_S)$, *where* $COM_R = min(\frac{|R|}{B} \log_C |R|, \frac{|R|}{B} \log_C 2^{Bits})$ *and* $COM_S = min(\frac{|S|}{B} \log_C |S|, \frac{|S|}{B} \log_C 2^{Bits})$.

PROOF. We choose using either the cache-oblivious merge sort or the cache-oblivious radix sort to sort R and S. According to our choice, COM_R and COM_S give the cache complexity of sorting R and S, respectively. The matching cost is $O(|R| + |S|)$, since the merge phase involves one scan on each relation. Thus, the cache complexity of the sort-merge join is $O(COM_R + COM_S)$. □

PROPOSITION 9. *The cache complexity of the cache-oblivious hash join is* $O(\frac{|R|+|S|}{B} \log_C |R|)$.

PROOF. According to Proposition 2, the cache complexity of partitioning R is $O(\frac{|R|}{B} \log_C |R|)$. Additionally, since the cache complexity of partitioning one tuple with the partitioner tree on R is $O(\frac{1}{B} \log_C |R|)$, decomposing S causes $O(\frac{|S|}{B} \log_C |R|)$ cache misses.

For matching, the average number of times of loading each partition of R is roughly $\frac{|S|}{|R|}$. The total cache cost of matching is $O(\frac{|S|}{|R|} \cdot \frac{|R|}{B} + \frac{|S|}{B}) = O(\frac{|S|}{B})$. Summing up the partitioning cost and the matching cost, the total cache cost of the cache-oblivious hash join is $O(\frac{|R|+|S|}{B} \log_C |R|)$. □

Through comparing these complexity results with those in Table III, we conclude that all of our cache-oblivious data structures and algorithms match the cache complexity of their cache-conscious counterparts.

4.5 Reducing the Recursion Overhead

The base case size is important for the efficiency of divide-and-conquer algorithms, even though it does not affect the cache complexity. A small base case size results in a large number of recursive calls, which can yield a significant cache overhead as well as computation overhead. A large base case size may cause cache thrashing. Since the cost estimator gives the cache cost of an algorithm, we use it to compare the costs with and without the divide-and-conquer operation for a given problem size. We then obtain the suitable base case size to be the maximum size of which the problem is small enough to stop the divide-and-conquer process. Even though the cost estimation is about the amount of data transfer, since it is used to determine the base case size, it essentially determines the computation overhead in the recursion as well—the larger the base case, the smaller the computation overhead.

We take CO_NLJ and the hash join as examples. In CO_NLJ, we use the cost estimator to compute the minimum sizes of the relations that are worthwhile to be partitioned in the NLJ. Specifically, we compare the cache costs for the NLJ without and with partitioning: (1) the join is evaluated as a base case, and (2) we divide the join into four smaller joins and evaluate each of these smaller joins as a base case. Since CO_NLJ uses the tuple-based simple NLJ to evaluate the base case, the cost functions of the NLJ without and with partitioning are given as F and F', respectively, in the following two formulas. Note, we define fc to be the size of the data (in bytes) brought into the cache for a recursive call, which includes parameters, function pointers and the return result in a recursive call.

$$F(C_x, B_x) = \begin{cases} \frac{1}{B_x}(||R|| + |R| \cdot ||S||), & ||S|| \geq C_x \\ \frac{1}{B_x}(||R|| + ||S||), & otherwise \end{cases}$$

$$F'(C_x, B_x) = \begin{cases} \frac{1}{B_x}(2||R|| + |R| \cdot ||S|| + 4fc), & ||S|| \geq 2C_x \\ \frac{1}{B_x}(||R|| + ||S|| + 4fc), & ||S|| \leq \frac{C_x}{2} \\ \frac{1}{B_x}(2||R|| + ||S|| + 4fc), & otherwise \end{cases}$$

With the cost estimator, we obtain $\phi = \mathcal{Q}(F)$ and $\phi' = \mathcal{Q}(F')$. Given the condition of recursive partitioning in CO_NLJ, $||R|| = ||S||$, we obtain the minimum base case size when $\phi > \phi'$. In our experiment, the fc value is 64. Suppose $r = s = 128$ bytes. When $|S|$ is larger than 16, we have $\phi > \phi'$. The C_S value is set to be 16 in this example. With this C_S value, we eliminate more than 97% of the recursive calls in CO_NLJ with $C_S = 1$.

The base case of the cache-oblivious hash join can be estimated in a similar way. Specifically, we estimate the cache costs of the build phase without and with partitioning: (1) the tuples of R are directly inserted into the hash table, and (2) we decompose R into p partitions with a partitioner tree, where p is around $|R|^{1/2}$. Next, the tuples in these p partitions are directly inserted into the hash table. The cost functions of the build phase without and with partitioning are given as F and F', respectively. Note, we define z to be the size of a hash bucket header (in bytes).

$$F(C_x, B_x) = \begin{cases} \frac{1}{B_x}(|R|z + 2||R||), & |R| < \frac{C_x}{B_x} \\ \frac{1}{B_x}(|R|\lceil\frac{z}{B_x}\rceil B_x + |R|(r + \lceil\frac{r}{B_x}\rceil B_x)), & otherwise \end{cases}$$

$$F'(C_x, B_x) = \frac{1}{B_x}(||R|| \log_{C_x} |R| + |R|^{1/2} F(C_x, B_x, |R|^{1/2}))$$

F includes two components, the cost of accessing the hash bucket headers, and the cost of reading the tuples and inserting them into the hash table. F' also includes two components, the partitioning cost and the total cost of inserting the tuples from each partition into the hash table.

Given $r = s = 8$ bytes and $z = 8$ bytes, when $|R|$ is larger than 128, we have $\mathcal{Q}(F) > \mathcal{Q}(F')$. Thus, the suitable base case size is 128. With this base case size, we eliminate seven passes of binary partitioning ($\log_2 128 = 7$) in the build or probe phase of the hash join.

Table IV. Machine Characteristics

Name	Intel P4	AMD	Ultra-Sparc	Intel QuadCore
OS	Linux 2.4.18	Linux 2.6.15	Solaris 8	Windows XP
Processor	Intel P4 2.8GHz	AMD Opteron 1.8GHz	Ultra-Sparc III 900Mhz	2.4 GHz × 4
L1 DCache	<8K, 64, 4>	<128K, 64, 4>	<64K, 32, 4>	<32K, 64, 2> × 4
L2 cache	<512K, 128, 8>	<1M, 128, 8>	<8M, 64, 8>	<8M, 128, 4>
DTLB	64	1024	64	64
Memory (GB)	2.0	15.0	8.0	2.0

5. EXPERIMENTAL RESULTS

We present our experimental results both on the overall performance of EaseDB using microbenchmarks and on the detailed performance of its core components, specifically the access methods, the sorting algorithms and the join algorithms.

5.1 Experimental Setup

Our experiments were conducted on four machines of different architectures, namely P4, AMD, Ultra-Sparc, and QuadCore. The main features of these machines are listed in Table IV. The L2 caches on all four platforms are unified. The Ultra-Sparc does not support hardware prefetching data from the main memory [Sun Corp. 1997], whereas the other three do. AMD performs prefetching for ascending sequential accesses only [AMD Corp. 2005] whereas P4 and QuadCore support prefetching for both ascending and descending accesses [Intel Corp. 2004].

In modern CPUs, a TLB is used as a cache for physical page addresses, holding the translation for the most recently used pages. We treat a TLB as a special CPU cache, using the memory page size as its cache line size, and calculating its capacity as the number of entries multiplied by the page size.

Since the L2 cache in QuadCore is shared by concurrent threads, we can evaluate the performance impact of cache coherence on our algorithms. *All experiments were conducted in a single-threaded environment, whereas the ones that are described in Section 5.8 ran multiple threads to evaluate the performance on CMP/SMT processors.*

All algorithms were implemented in C++. The compiler was g++ 3.2.2-5 with optimization flags (*O3* and *finline-functions*) on Linux/Solaris, and was MSVC8 on WinXP. In our experiments, the memory pool was set to be $1.5G$ bytes on all platforms. The data in all experiments were always memory-resident and the memory usage never exceeded 90%.

We first used microbenchmarks to evaluate the overall performance of EaseDB. We chose the microbenchmarks, DBmbench [Shao et al. 2005], proposed by Shao et al. over the TPC benchmarks [TPC 2004], since DBmbench mimics the cache behavior of TPC benchmarks (TPC-H and TPC-C) well on modern architectures and it allows us to study the performance in more detail.

DBmbench consists of three queries, including a scan-dominated query (μSS), a join-dominated query (μNJ) and an index-dominated query (μIDX). Tables V and VI show the table schema and the queries in DBmbench, respectively. Following previous studies [Ailamaki et al. 1999; Shao et al. 2005;

Table V. Table Schema of DBmbench

Table T1	Table T2
CREATE TABLE T1 (a1 INTEGER NOT NULL, a2 INTEGER NOT NULL, a3 INTEGER NOT NULL, <padding>, FOREIGN KEY (a1) references T2);	CREATE TABLE T2 (a1 INTEGER NOT NULL PRIMARY KEY, a2 INTEGER NOT NULL, a3 INTEGER NOT NULL, <padding>,);

Table VI. Queries of DBmbench

μSS	μNJ	μIDX
SELECT distinct (a3) FROM T1 WHERE Lo < a2 < Hi ORDER BY a3	SELECT avg (T1.a3) FROM T1, T2 WHERE T1.a1=T2.a1 AND Lo < T1.a2 < Hi	SELECT avg (a3) FROM T1 WHERE Lo < a2 < Hi

Harizopoulos and Ailamaki 2006], we set the length of "padding" to make a record 100 bytes long. The values of $a1$, $a2$, and $a3$ are uniformly distributed between 1 and 150,000, between 1 to 20,000 and between 1 to 50, respectively. The parameters in the predicate, Lo and Hi, are used to obtain different selectivities. In our study, we set the selectivity of μSS to be 10%, the join selectivity of μNJ to be 20% and the selectivity of μIDX to be 0.1%. These settings for DBmbench represent the characteristics of TPC benchmarks best [Shao et al. 2005].

In addition to this benchmark, we used other homegrown data sets and workloads for more detailed studies. Our homegrown workload contains two selection queries and two join queries on relations R and S. Both tables contain n fields, $a_1, \ldots, a_n$, where a_i is a randomly generated 4-byte integer. We varied n to scale up or down the tuple size. The tree index was built on the field a_1 of R and S. The hash index was built on the field a_1 of R.

The selection queries in our own workloads are in the following form:

Select $R.a_1$
From R
Where <predicate>;

One of the two selection queries is with a non-equality predicate ($x - \delta < R.a_1 < x + \delta$) and the other an equality predicate ($R.a_1 = x$). Note that x is a randomly generated 4-byte integer. We used the B+-tree index to evaluate the nonequality predicate and the hash index to evaluate the equality predicate. Since we focused on the search performance of the tree index, we set δ to a small constant, such as ten, in the nonequality predicate.

The join queries in our own workloads are:

Select $R.a_1$
From R, S
Where <predicate>;

One of the two join queries is an equi-join and the other a nonequijoin. The equijoin takes $R.a_1 = S.a_1$ as the predicate and the non-equijoin $R.a_1 < S.a_1$

Table VII. Performance Metrics

Metrics	Description
TOT_CYC	Total execution time in seconds (*sec*)
L1_DCM	Number of L1 data cache misses in billions (10^9)
L2_DCM	Number of L2 data cache misses in millions (10^6)
TLB_DM	Number of TLB misses in millions (10^6)

$and \ldots and \ R.a_n < S.a_n$. 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. We used the non-indexed NLJ to evaluate the non-equijoin and used the indexed NLJ, the sort-merge join, or the hash join to evaluate the equi-join. The measured results for non-indexed NLJs were obtained when $|R| = |S| = 256K$ and $r = s = 128$ bytes. The measured results for indexed NLJs, sort-merge joins and hash joins were obtained when $|R| = |S| = 32M$ and $r = s = 8$ bytes. These settings were chosen to be comparable to the previous studies on cache-conscious join algorithms [Boncz et al. 1999; Shatdal et al. 1994].

Table VII lists the main performance metrics used in our experiments. We used the C/C++ function *clock*() to obtain the total execution time. In addition, we used hardware profiling tools, PCL [Berrendorf et al. 2002] and Intel VTune [Intel Corp. 2007], to count cache misses on P4 and on QuadCore, respectively.

For the cache-conscious algorithms in our study, we varied their parameter values to examine the performance variance. Given a cache parameter, y, of a target level in the memory hierarchy (either C or B) of an experiment platform, we varied the parameter value x in a cache-conscious algorithm within a range so that the three cases $x < y$, $x = y$ and $x > y$ were all observed. Given a multi-level memory hierarchy, we considered all cache levels and varied the cache parameter value in a cache-conscious algorithm for every level of the cache.

5.2 Microbenchmarks

We first compared EaseDB with a prototype cache-conscious engine that we developed using DBmbench [Shao et al. 2005]. We stored the relations in the benchmark in the row-based or the column-based manner. The DBmbench data set was 512M bytes. We built a B+-tree on the attribute $a2$ of table T1. We manually tuned the parameter values for the cache-conscious algorithms for the best performance. Both cache-oblivious and cache-conscious cost estimators give the same query plan for each of the three queries. They choose the table scan and next the radix sort to evaluate μSS, the hash join to evaluate μNJ, and the index scan to evaluate μIDX.

We issued each kind of query as a group from DBmbench continuously to both EaseDB and the cache-conscious engine and measured the average execution time per query for μSS and μNJ, and per thousand queries for μIDX. Figure 6 shows the measured average time on P4 when the relations are stored in the row-based or the column-based manner. The number of queries of each kind executed was $20K$. We do not show the results on other three platforms, because

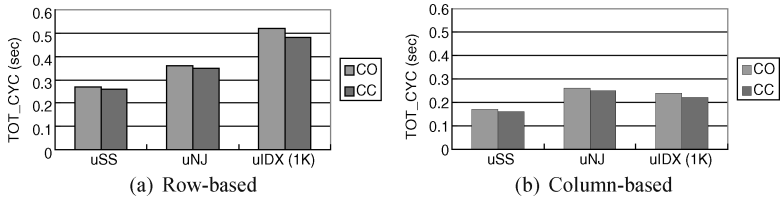


Fig. 6. The average execution time of each query group in DBmbench on P4. We compute the execution time per query for μSS and μNJ, and per thousand queries for μIDX.

the performance comparison between EaseDB and the cache-conscious engine is similar to that on P4. Regardless of the storage models, EaseDB has a good performance that is similar to the fine-tuned cache-conscious engine. It is less than 5% slower than the cache-conscious engine for μSS and μNJ, and around 10% slower for μIDX. Thus, we present the results for the row-based storage model only in the remainder of the study.

5.3 Model Evaluation

We evaluated our cache-oblivious cost model with two basic access patterns including the linear scan and the random traversal, and our join algorithms on our homegrown workloads. A linear scan sequentially accesses all the tuples in a relation. A random traversal accesses the tuples at random locations in a relation. We first compared our estimation with the estimation of a cache-conscious cost model [Manegold et al. 2002]. The cache-conscious cost model used the cache parameter values of the L2 cache on the three architectures. We only report the results on P4, because the results on the other three architectures are similar. Next, we verified the effectiveness of our estimation on the suitable base case size.

Figure 7 shows the volume of data transfer based on our measurement and the estimation using cache-conscious and cache-oblivious cost models on P4. We limit the range of the cache capacity and the cache block size with the knowledge of the configurations on the modern machine; that is, the cache block size is in the range between $32B$ and $256B$, and the cache capacity is between $8KB$ and $16MB$. We denote the extension to our cost model with the range knowledge as "CC(Range)". We compute the volume of data transferred in our measurement to be the measured number of L2 cache misses multiplied by the L2 cache block size. We fixed the tuple size to be 8 bytes and varied the number of tuples in both relations. Regardless of the data size and the algorithm, both CO and CC(Range) correctly predict the order of relative performance of the four join algorithms. For example, both models predict that the hash join is the most cache efficient and the nonindexed nested-loop join is the least cache efficient among the four join algorithms. Moreover, CC(Range) correlates well with the estimation by the cache-conscious cost model.

In our estimation, cache misses in the linear scan are all compulsory misses, whereas those in other algorithms include compulsory and capacity misses. The three models achieve the same accuracy for the linear scan. In comparison, for the other algorithms, our estimation is off the measurement by various degrees.

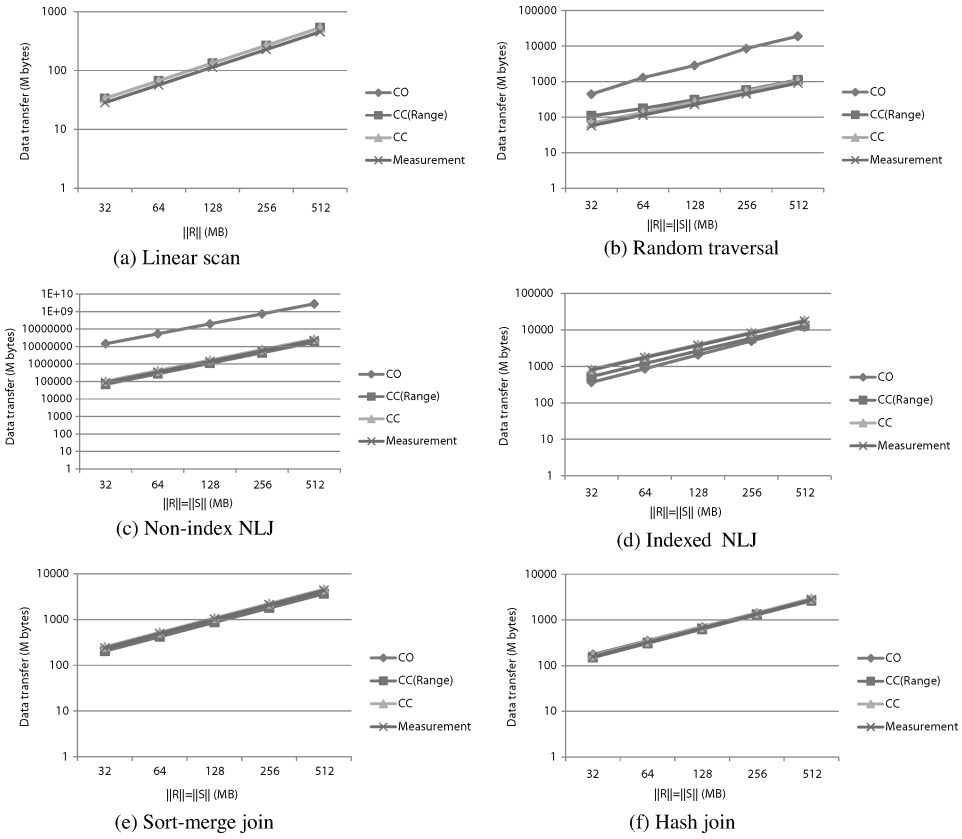


Fig. 7. Evaluation of cache-conscious and cache-oblivious cost models on P4.

In particular, for those algorithms where capacity misses dominate compulsory misses, such as random traversal and non-indexed nested-loop joins, our estimation results in a low accuracy. On the other hand, for those algorithms where compulsory misses dominate capacity misses, such as hash joins and sort merge joins, our estimation is close to the measurement. Additionally, CC(Range) is more accurate than our CO estimation, because the range knowledge excludes from consideration the cache capacities that are unrealistically small.

Figure 8 shows the time breakdown of two representative cache-oblivious join algorithms, non-indexed NLJs and the build phase of hash joins, with the base case size varied on P4. The results for the sort-merge join algorithm are not shown in this figure, since they were similar to those of the hash join. The busy time is obtained by subtracting the three types of cache stalls from the total elapsed time. The base case for the nonindexed NLJ is a join on two equal-sized partitions. The size of each partition is compared with the capacities of the L1 and L2 caches, and the TLB. The base case for the build phase of the hash join is to construct a small hash table. Since each hash bucket is likely to be one cache line, the size of the hash table is compared with the number of cache lines in the L1 and L2 data caches, and the number of entries in the TLB.

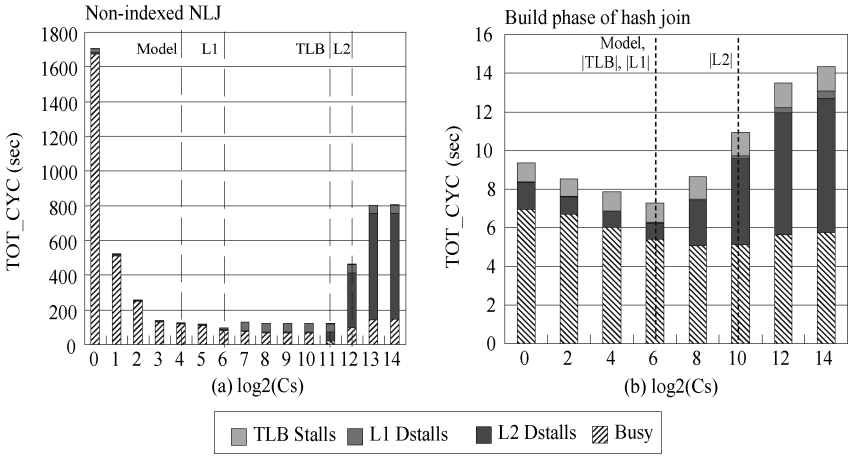


Fig. 8. Performance of join algorithms with the base case size varied on P4: (left) the non-indexed nested-loop join, (right) the build phase of the hash join.

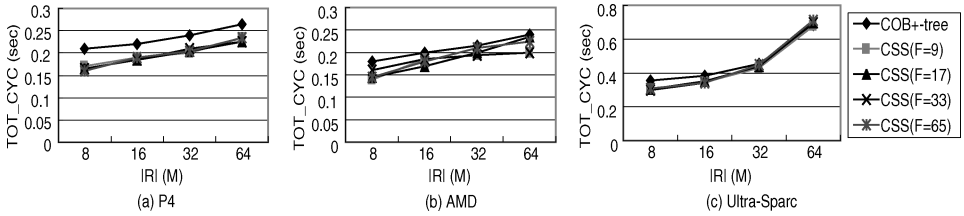


Fig. 9. Performance comparison between COB+-trees and CSS-trees.

Our estimated base case size greatly reduces the recursion overhead. With the estimated base case size, the cache stalls of both algorithms are greatly reduced, by over 80% and 60% compared with those when the base case is one. The busy time is also greatly reduced. These results verify the effectiveness of our cost estimator. One may conjecture from Figure 8(a) that the L1 cache capacity is a better guess than our cost estimator for the base case size on P4; unfortunately, this particular phenomenon does not hold for different platforms, different algorithms, or different data sizes [Bender et al. 2006; Brodal et al. 2004; He and Luo 2006].

5.4 Access Methods

We evaluated our access methods including the B+-Tree and the hash index using a number of selection queries.

5.4.1 *B+-Trees.* We compared the search performance of COB+-trees and CSS-trees [Rao and Ross 1999]. On each platform, we varied the fanout of the CSS-tree from 9 to 65. The node size was varied from 32 to 256 bytes. Figure 9 shows the performance comparison with the number of tuples in the relation varied. The number of selection queries executed was $200K$.

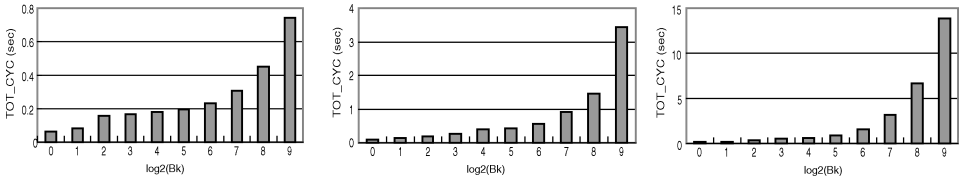


Fig. 10. Performance of hash indexes. The hash index with $Bk = 1$ is our cache-oblivious hash index in EaseDB.

On all platforms, COB+-trees have a similar performance to CSS-trees. The performance difference between them becomes smaller as the relation size increases. When $|R| = 64M$, the COB+-tree is around 20% slower than the best CSS-tree on P4 and AMD, and it is less than 1% slower than the best CSS-tree on Ultra-Sparc. The results on Ultra-Sparc are similar to those in the previous study on a P3 machine [Brodal et al. 2002].

5.4.2 *Hash Indexes.* We evaluated the hash index with the average number of tuples in each bucket, Bk, varied. The number of selection queries executed was $200K$. Figure 10 shows the performance of hash indexes when Bk is varied from one to 512. Since each tuple takes 8 bytes (four bytes for the value and four bytes for the record identifier), the bucket size is around $(8 \times Bk)$ bytes. That is, the bucket size is varied from 8 to $4K$ bytes. The cache-oblivious hash index is the one with $Bk = 1$, whereas the cache-conscious one chooses the Bk value according to the cache block size.

On all platforms, the performance degrades dramatically as the Bk value increases. Due to its hardware prefetching capability, P4 has a smaller performance degradation than Ultra-Sparc. The cache-oblivious hash index is faster than the cache-conscious one, because its bucket size is smaller and the computation time on each bucket is likely to be smaller.

5.5 Sorting Algorithms

We compared the overall performance of the radix sort and the merge sort. We fixed the number of tuples to be $32M$ and the tuple size to be 8 bytes. The minimum value of the sorting key was one and the maximum value was max. The number of bits used in the radix sort is $Bits = \log_2 max$. According to our estimation, if $max \geq 2^{18}$, the merge sort outperforms the radix sort.

Figure 11 shows the performance comparison between the cache-oblivious radix sort and the merge sort with max varied. We observed the same performance trend on all three platforms. When max is small, the radix sort outperforms the merge sort. As max becomes larger, the performance of the radix sort becomes close to that of the merge sort. The merge sort outperforms the radix sort when max is larger than 2^{18} on Ultra-Sparc and 2^{19} on P4 and AMD. Therefore, our estimation correctly predicts the performance comparison between the radix sort and the merge sort.

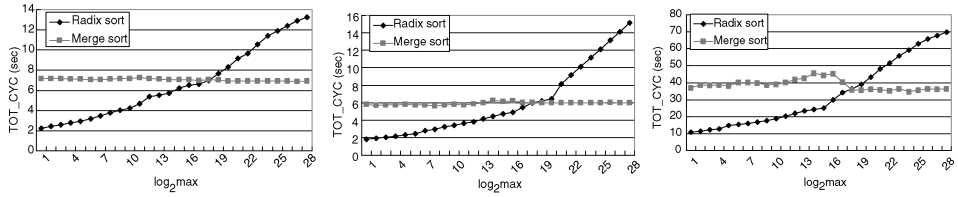


Fig. 11. The cache-oblivious radix sort and cache-oblivious merge sort. We varied the maximum value of the sorting key, *max*.

5.6 Join Algorithms

We evaluated the performance of our cache-oblivious join algorithms in comparison with the best performance of their cache-conscious counterparts. Figure 12 shows the performance measurements of join algorithms on P4, AMD and Ultra-Sparc.

Figures 12(a–c) show the performance of nonindexed NLJs. The cache-conscious algorithm is the blocked NLJ [Shatdal et al. 1994], whose parameter is the block size of the inner relation. We varied the block size of the blocked NLJ from $4K$ to $16M$ bytes.

Figures 12(d–f) show the performance of indexed NLJs with COB+-trees and CSS-trees. We varied the node size of the CSS-tree from 32 to 256 bytes.

Figures 12(g–i) show the performance of sort-merge joins. The number of bits in the radix sort, *Bits*, is 32, which is larger than our estimated threshold value, 17. Therefore, we chose the merge sort to sort both relations. The cache-conscious merge sort [Ramakrishnan and Gehrke 2003; LaMarca and Ladner 1997] works in two phases. First, given the partition granularity, *gr* bytes, it divides the relation into multiple partitions. Each partition is around *gr* bytes. It sorts each partition using the quick sort. Second, given the merge fanout F, it recursively merges F partitions. These figures show the best performance obtained in our tuning on the merge fanout for a given partition granularity.

Figures 12(j–l) show the performance of hash joins. The cache-conscious hash join is the radix join [Boncz et al. 1999]. In the radix join, we need to tune the partition fanout and the partition granularity. Given a certain partition granularity, *gr* bytes, we varied the partition fanout and measured the execution time. Again, these figures show the best performance obtained in our tuning for a given partition granularity.

We analyze the results of Figure 12 on three aspects. First, we study the performance variance of each cache-conscious algorithm. On a given platform, the performance variance of cache-conscious join algorithms (except the indexed NLJ) with different parameter values is large. Furthermore, the performance variance of a cache-conscious algorithm differs across platforms. For example, the performance variance of cache-conscious non-indexed NLJs, sort-merge joins and hash joins on Ultra-Sparc is smaller than on the other two platforms.

Second, we study the best parameter values for the cache-conscious algorithms. Table VIII shows the best parameter values for the four cache-conscious join algorithms on our platforms. On a given platform, the best parameter value

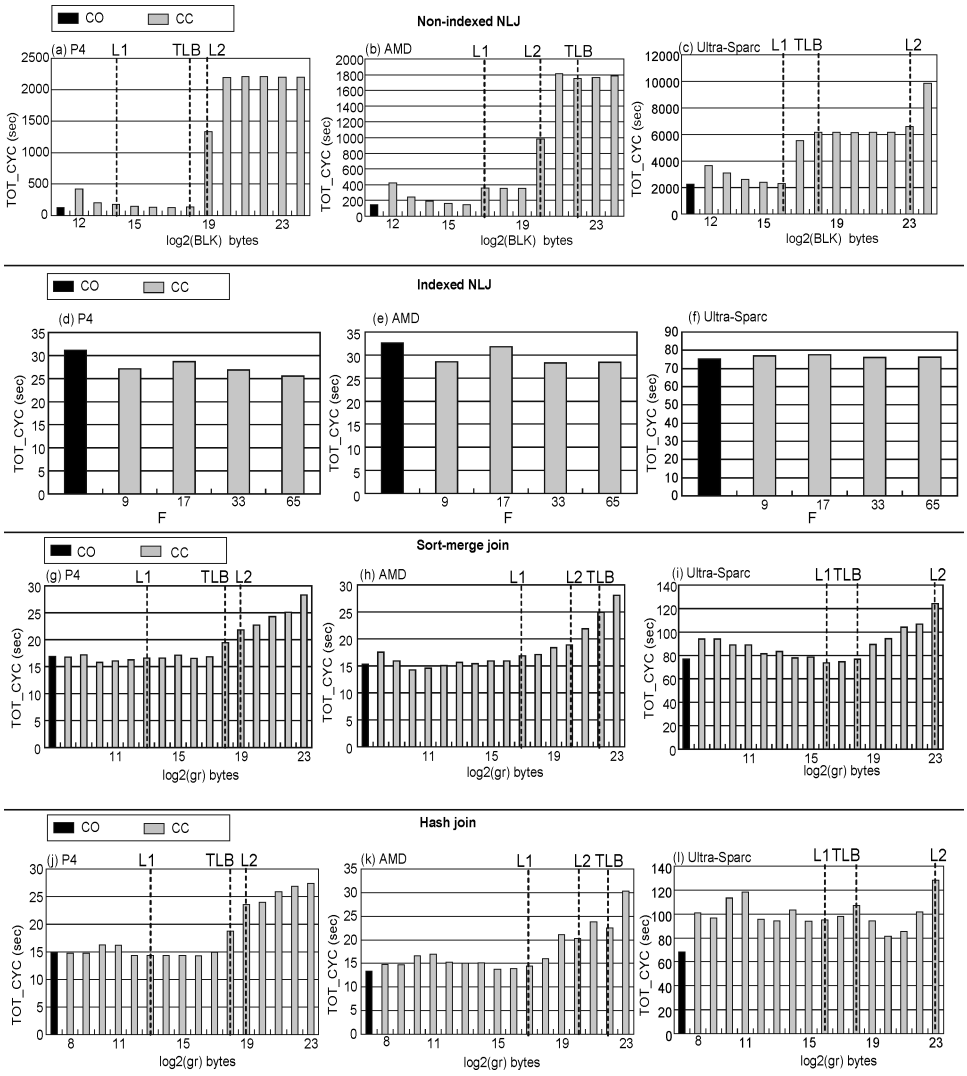


Fig. 12. Performance study for join algorithms. From top down: the non-indexed NLJ, the indexed NLJ, the sort-merge join and the hash join.

for a cache-conscious join algorithm may be none of the cache parameters, for example, the sizes of the L1 and L2 data caches, or the number of entries in the TLB. Moreover, for a given cache-conscious algorithm, the best parameter values differ across platforms. These results show it is difficult to determine the best parameter values on different platforms even with the knowledge of the cache parameters.

Third, we compare the overall performance of cache-conscious and cache-oblivious algorithms on three platforms. Regardless of the architectural differences among the three platforms, our join algorithms provide a robust and good performance. Specifically, the performance of our join algorithms is close

Table VIII. The Best Parameter Values for the Four Cache-Conscious Join Algorithms

Platform	P4	AMD	Ultra-Sparc
Non-indexed NLJ	$L1\_Cap < x < L2\_Cap$	$x < L1\_Cap$	$x = L1\_Cap$
Indexed NLJ	$x > L2\_B$	$x = L2\_B$	$x > L2\_B$
Sort-merge join	$x < L1\_Cap$	$x < L1\_Cap$	$x = L1\_Cap$
Hash join	$L1\_Cap < x < L2\_Cap$	$x < L1\_Cap$	$L1\_Cap < x < L2\_Cap$

We denote the parameter of the cache-conscious algorithm as x. On each platform, $L1\_B$ and $L2\_B$ denote the block sizes of the L1 and L2 data caches, respectively; $L1\_Cap$ and $L2\_Cap$ denote the cache capacities of the L1 and L2 data caches, respectively.

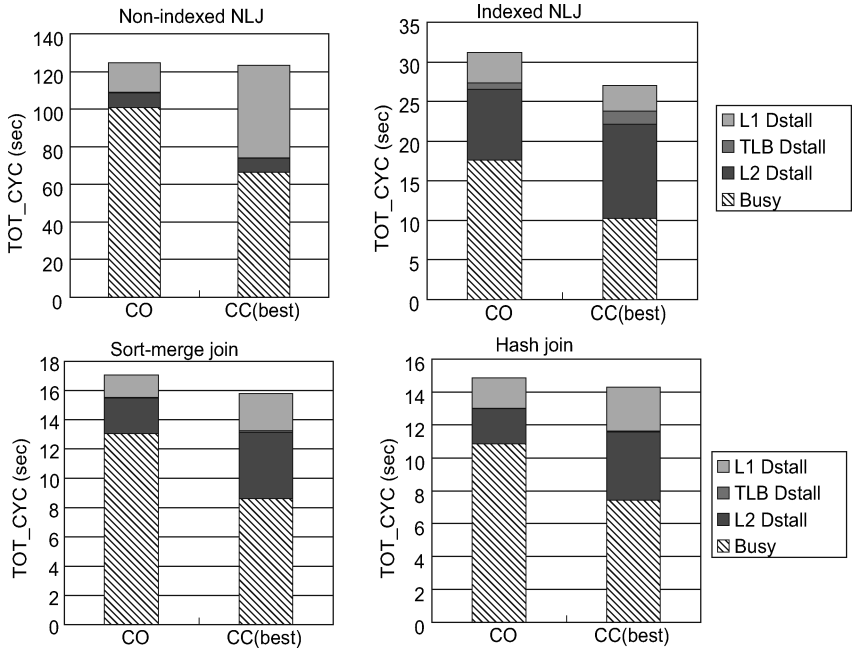
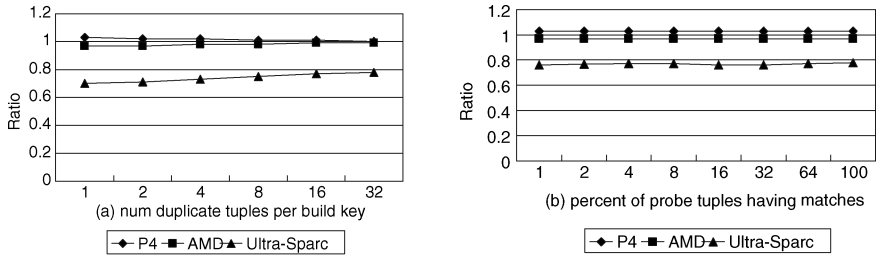


Fig. 13. Time breakdown of cache-oblivious algorithms and the best cache-conscious algorithms on P4.

to, if not better than, the best performance of cache-conscious join algorithms on P4, and is mostly better than the best performance of cache-conscious join algorithms on both AMD and Ultra-Sparc.

We further examine the time breakdown of our cache-oblivious algorithms and the best cache-conscious algorithms in Figure 13. The total cache stalls of the cache-conscious join algorithms are significant, because these algorithms typically optimize only for the cache of a chosen level and cache thrashing may occur at the levels other than the chosen one. In contrast, the cache stalls of our cache-oblivious algorithms are less significant due to their automatic optimization for the entire memory hierarchy. The busy time is significant among all cache-oblivious join algorithms.

Finally, we study the effect of data skews and join selectivities on the TOT_CYC ratio of cache-oblivious joins over fine-tuned cache-conscious joins with the suitable parameter values on the three platforms. We omit the



(a) Varying the number of duplicates per key (b) Varying join selectivity

Fig. 14. The TOT_CYC ratio of cache-oblivious hash joins over cache-conscious hash joins.

results on sort-merge joins and NLJs from the figures, because the trend of the TOT_CYC ratio of the sort-merge join is similar to that of the hash join, and the nested-loop join with and without indexes have a stable TOT_CYC ratio regardless of skewness and join selectivities. Figure 14(a) shows the ratio for the hash join when the number of duplicate tuples per key is varied. The larger this number is, the more skewed the relation becomes. As both relations become skewed, the matching cost becomes more significant, which is the same between cache-oblivious and cache-conscious hash joins. Hence, the ratio becomes closer to one. The performance ratio is consistently close to one on P4, and smaller than one on AMD and Ultra-Sparc. Figure 14(b) shows the ratio for the hash join when the percentage of probe tuples having matches is varied. The smaller this percentage is, the higher the join selectivity is. Regardless of the join selectivities, the performance ratio is stable on all platforms.

5.7 Comparison with MonetDB

To check whether our cache-centric implementation has a comparable performance with state-of-the-art main memory databases, we performed a comparison between our engines and MonetDB 5.0 [MonetDB 2006]. The comparison was done on the core query processing algorithms, excluding the other components such as query parsing, plan generation, and optimization. We used our homegrown datasets for better control on the data size, the data distribution and the query workload. To measure the query processing performance on MonetDB, we first stored the data sets into text files, and imported these files into MonetDB using the *copy* command. Next, we issued SQL queries to MonetDB through JDBC.

Figure 15 compares the performance of the sort and the hash join in our cache-oblivious and cache-conscious engines with those in MonetDB on the QuadCore machine. The results were obtained from running the system on a single core of the QuadCore processor. We also performed the experiments on P4 and obtained similar results. We varied $|R|$ from $4M$ to $16M$ for the sort, and we kept $|R| = |S|$ and varied both $|R|$ and $|S|$ from $4M$ to $16M$ for the hash join. As the data size increases, our sorting algorithm has a similar performance as that in MonetDB, and our join algorithm outperforms that in MonetDB. This figure indicates that the efficiency of our implementation for the cache-efficient algorithms is comparable to MonetDB.

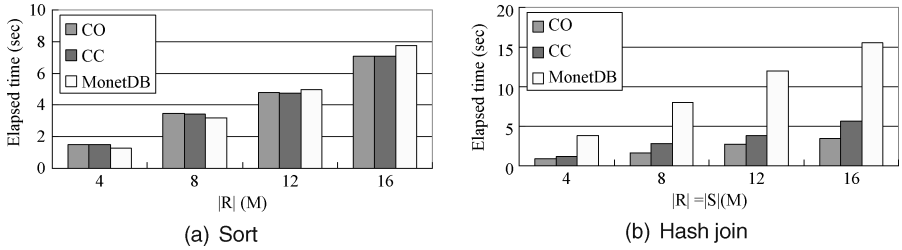


Fig. 15. Performance comparison between our engines and MonetDB on QuadCore.

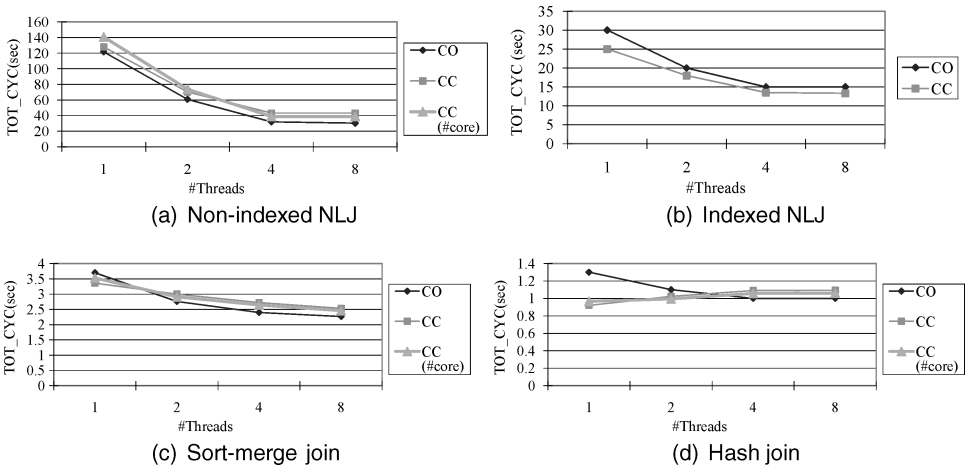


Fig. 16. Average execution time of the four join implementations on QuadCore.

5.8 Evaluation on CMP/SMT Processors

We studied the performance impact of cache interference when multiple concurrent threads run on CMP/SMT processors. The cache behavior of an algorithm on a CMP/SMT processor may be different from that in a single-threaded execution environment. A thread may reuse the data that have been brought into the cache by other threads, or its own working set may be evicted from the cache due to other threads loading other cache lines.

Figures 16 and 17 show the average execution time and the cache miss rate of join algorithms when we varied the number of concurrent threads. The average execution time is the total execution time divided by the number of threads. The cache miss rate is defined to be the ratio of the number of L2 data cache misses and the number of retired instructions [Intel Corp. 2007]. For the non-indexed NLJ, $|R| = |S| = 32K$ and $r = s = 128B$; for other three joins, $|R| = |S| = 4M$ and $r = s = 8B$. Each thread ran either a cache-oblivious algorithm without any tuning, or a cache-conscious algorithm with tuning. The measurements of the cache-conscious algorithm include those with the suitable cache parameter values in the single-threaded execution, denoted as "CC". Additionally, for the join algorithms that are aware of the cache capacity including the non-indexed

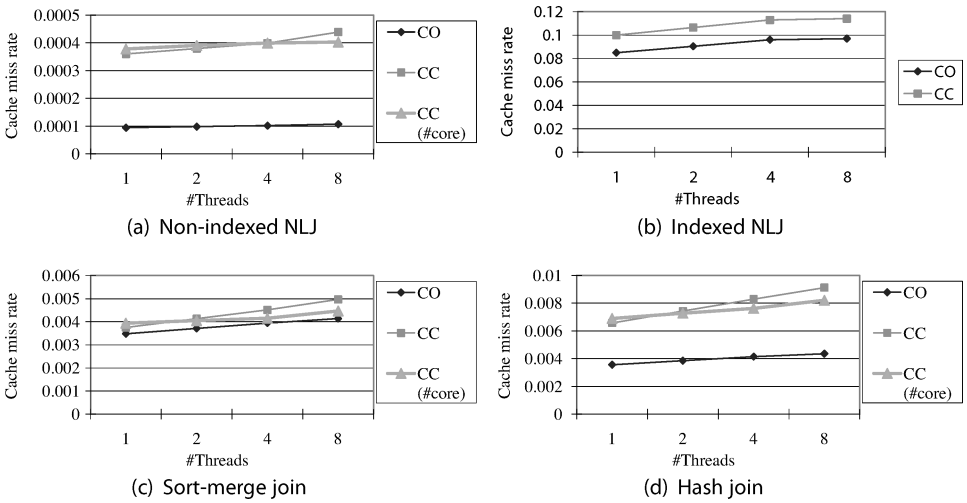


Fig. 17. L2 data cache miss rate of the four join implementations on QuadCore (obtained using Intel VTune).

NLJs, the sort-merge join and the hash join, we consider the tuning with respect to the number of cores in the CMP processor. Suppose the suitable cache parameter values in the single-threaded execution is x_0, and the number of cores in the CMP processor is #*core*, we obtained the measurements with the parameter value of $\frac{x_0}{\#core}$. These measurements are denoted as "CC(#core)." This setting reduces the cache interference among threads. However, it does not fully utilize the cache when the number of threads is smaller than the number of cores. As a result, the CC(#core) algorithm does not necessarily improve the overall performance.

When the number of threads is smaller than the number of cores, the average execution time of both algorithms decreases due to the parallelism in the execution. When the number of threads is larger than the number of cores, the average execution time becomes stable. This indicates the processor is fully utilized. Additionally, when the number of threads is one, the CC(#core) algorithm is slower than the CC algorithm. When the number of threads is equal to the number of cores on the CMP, the CC(#core) algorithm is faster than the CC algorithm. Since the runtime dynamics on the CMP is difficult to predict, the tuning of the CC(#core) algorithm does not necessarily improve the overall performance.

Comparing cache-conscious and cache-oblivious algorithms on the four joins, we find that they achieve a similar average execution time on the indexed NLJ regardless of the number of threads, whereas the cache-oblivious algorithm is better than the cache-conscious algorithm on other three joins as the number of threads increases. When the number of threads is four, cache-oblivious algorithms are 28%, 10%, and 8% faster than their fine-tuned cache-conscious counterparts for the nonindexed NLJ, the sort-merge join and the hash join, respectively.

We analyze the cache performance of the cache-conscious and the cache-oblivious algorithms, as shown in Figure 17. The cache miss rate of both cache-conscious and cache-oblivious algorithms increases, as the number of threads increases. This increase suggests the presence of cache interference. For all joins except the indexed NLJ, the increase in the cache miss rate of the cache-oblivious algorithm is smaller than that of the CC algorithm, and is similar to that of CC(Range) algorithm. When the number of threads increases from one to four, the increases in the cache miss rate of the cache-oblivious non-indexed NLJ, sort-merge join and hash join are 8%, 13% and 15%, respectively; the increases for the CC non-indexed NLJ, sort-merge join and hash join are 11%, 21% and 26%, respectively; the increases for the CC(#core) non-indexed NLJ, sort-merge join and hash join are 6%, 7% and 10%, respectively. This shows that the cache-oblivious algorithm has a more robust cache performance than the cache-conscious algorithm, or has a similar robustness on the cache performance to the cache-conscious algorithm with further tuning with respect to the number of cores in the CMP processor. Since the cache performance of our cache-oblivious algorithm matches that of its cache-conscious counterpart in the single-threaded execution environment, our cache-oblivious algorithm may outperform its cache-conscious counterpart when cache interference occurs, as shown in Figure 16.

5.9 Summary

In summary, EaseDB, without any platform-specific tuning, is less than 30% slower than its fine-tuned and platform-specific cache-conscious counterpart on all of our platforms. This comparable performance applies to both the row-based and the column-based storage models. The performance of each of the four cache-oblivious join algorithms in EaseDB is close to the best performance of its cache-conscious counterpart on P4 and AMD, and is better than the best performance of its cache-conscious counterpart on Ultra-Sparc, regardless of data skews and join selectivities. Additionally, our join algorithms have a more robust cache performance than the cache-conscious join algorithms on CMP/SMT processors. As a result, they are up to 28% faster than their fine-tuned cache-conscious counterparts on the quad-core machine.

6. DISCUSSION

Through the design, implementation and evaluation of EaseDB, we have identified a number of limitations and opportunities for cache-oblivious query processing in comparison with cache-conscious query processing.

6.1 Limitations

We identify four limitations of cache-oblivious algorithms. All of these limitations are rooted at the philosophy of the cache-oblivious approach—the optimal performance must be achieved without any platform-specific tuning. In the following, we discuss these limitations and our experience in addressing them.

First, without the knowledge of any cache parameters, cache-oblivious techniques usually employ sophisticated data structures and mechanisms, for example, the VEB layout and our buffer hierarchy, in order to achieve the same cache complexity as their cache-conscious counterparts. From a systems' point of view, we make great effort to reuse these efficient data structures and mechanisms as building blocks to develop a cache-oblivious query processing system. For example, we use the partitioner tree as a common data structure for the radix sort and the hash join.

Second, as the base case size in the recursion is usually small, the recursion process may go unnecessarily deep. Consequently, the recursion overhead, especially the computational overhead, becomes significant. For instance, the cache-oblivious non-indexed nested loop join had three times less CPU busy time when the base case increased from one to two tuples in our experiments. Moreover, the dominance of computational cost in the overall time remained for all base cases smaller than 64 tuples. On the other hand, a large base case may result in cache thrashing. Thus, it is necessary to determine a suitable base case size to eliminate the unnecessary recursions and to avoid cache thrashing. Furthermore, this reduction of the recursion overhead must be achieved in a cache-oblivious way, as opposed to any platform-dependent approach, self-learning or not. In our work, we have developed a simple cost model to estimate the expected cache cost of an algorithm on an arbitrary memory hierarchy and subsequently to determine a suitable base case size for the algorithm.

Third, the cache-oblivious cost estimation may not be as accurate as the cache-conscious cost estimation, as we have shown in the experiments. Again, the reason is that, instead of estimating the cache cost for a specific memory hierarchy, our model gives an expected value of cache costs on an arbitrary memory hierarchy. Furthermore, this cache-oblivious estimation is difficult to devise an error bound, if feasible at all. Fortunately, our experiments demonstrated that the expected value estimated by our model followed the trend of the cache-conscious estimation well. Additionally, the base case size obtained from our estimation greatly reduces the recursion overhead in our cache-oblivious algorithm. Nevertheless, the cache-conscious approach is still a valid choice for the cost estimation. Extending our cache-oblivious cost estimation with the range of the cache parameter values is a good tradeoff between the accuracy and the automaticity of the cost model.

Fourth, cache-oblivious algorithms may underutilize the hardware capability of the caches. Without the knowledge of the cache block size and the cache capacity, it is difficult to derive the accurate state of the cache. Without the knowledge of the cache state, cache-oblivious algorithms can not explicitly take advantage of other cache characteristics, such as the prefetching capability and the number of concurrent threads sharing the cache. Fortunately, most of the cache-oblivious algorithms pack the data into a consecutive memory area, such as the VEB layout in the COB+-tree, so that they can exploit the hardware prefetching capability of the cache (if available) automatically. Furthermore, since cache-oblivious algorithms have a robust cache performance, they are likely to achieve a high and robust throughput on CMP/SMT processors.

6.2 Opportunities

The major opportunity of a cache-oblivious database also comes from its independence from any platform-specific tuning. As a result, it can run efficiently on a diversity of machines without parameter setting or tuning. Furthermore, it automatically achieves a good cache performance on all levels of the memory hierarchy. This automaticity is especially desirable when the significant level of caches in the memory hierarchy changes, for example, from the disk to the L2 data cache [Ailamaki et al. 1999]. A recent study [Hardavellas et al. 2007] shows that the L1 data cache will become an important factor in the total execution time of query processing on CMP/SMT machines. This finding implies that cache-conscious algorithms require new tuning or complete redesign to adapt to such changes. In contrast, cache-oblivious algorithms will thrive through hardware changes. This feature reduces the ownership cost of a cache-oblivious database system.

With high automaticity, cache-oblivious algorithms have great potential in parallel computing [Chowdhury and Ramachandran 2007]. They naturally fit into the parallel computing paradigm due to their divide-and-conquer design. Subproblems in the divide-and-conquer methodology can be run independently on different cores on a CMP/SMT machine or on different nodes in a share-nothing system. Moreover, the automaticity of cache-oblivious algorithms greatly reduces the amount of tuning in parallel computing, which is more complicated than in single-node systems.

In addition to automaticity, the performance of a cache-oblivious algorithm is more robust than its cache-conscious counterpart when the cache state is highly dynamic, such as in CMP/SMT machines. The robust cache performance of the cache-oblivious algorithm is due to its small *reuse distance*. The reuse distance is defined to be the number of distinct cache accesses between two consecutive accesses to a certain cache line [Ding and Zhong 2003]. A cache-conscious algorithm typically sets the parameter value close to the cache parameter value of the target level of caches in a specific memory hierarchy. For example, the block size of the inner relation in the blocked NLJ [Shatdal et al. 1994] is set to around the cache capacity of the significant level of cache. The reuse distance of the cache hits in the blocked NLJ is close to the number of cache lines in the cache. In contrast, the cache-oblivious algorithm follows the divide-and-conquer process. As the recursion gets deeper, the cache accesses become localized. The base case is sufficiently small so that the reuse distance is small. Thus, our cache-oblivious algorithm is likely to have a more robust cache performance than its cache-conscious counterpart on CMP/SMT processors.

In summary, both cache-oblivious and cache-conscious algorithms share the common wisdom of optimizing the cache performance in a multi-level memory hierarchy, such as exploiting the sequential access nature and improving the locality of cache accesses. They differ in optimization strategies. Cache-conscious algorithms utilize the knowledge of the cache parameter and explicitly pack data into the cache or into a cache block of a specific level. In contrast, cache-oblivious algorithms utilize more sophisticated techniques to improve the locality. The difference in the optimization strategy results in different strengths

and weaknesses for both kinds of algorithms. We consider the cache-oblivious approach as an alternative way of improving the cache performance for in-memory query processing. As memory systems and processors become more complicated, there will be times that cache-oblivious query processing is more desirable than its cache-conscious counterpart.

7. CONCLUSION

As the memory hierarchy becomes an important factor for the performance of database applications, we propose to apply cache-oblivious techniques to automatically improve the memory performance of relational query processing. While database researchers have demonstrated the effectiveness of the cache-conscious approach on CPU caches, we provide a practical alternative that achieves a high automaticity as well as a comparable performance to the cache-conscious approach.

In this article, we present our design, implementation, and evaluation of our cache-oblivious in-memory query processor, EaseDB. To the best of our knowledge, EaseDB is the first relational query processor to employ cache-oblivious techniques for improving data cache performance. Our results show that our cache-oblivious algorithms provide a good performance on various platforms, which is similar to or even better than their fine-tuned cache-conscious counterparts. Moreover, this good performance is achieved without any tuning on the platforms in our experiments.

We expect the ideas behind EaseDB to become even more relevant and applicable to the upcoming generation of machines whose memory hierarchies become deeper and more diversified [Hennessy and Patterson 2002]. We also conjecture that the advantage of the automaticity of the cache-oblivious algorithms will be greater in the future when multicore and multithreading chips support more concurrent threads [Hardavellas et al. 2007].

The EaseDB software packages are available on our project site, http://www.cse.ust.hk/cactus/.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments on the earlier versions of the draft.

REFERENCES

AILAMAKI, A. 2005. Database architectures for new hardware. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, 1148.

AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS, M. 2001. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 169–180.

AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. 1999. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)* Morgan Kaufmann Publishers Inc., San Francisco, CA, 266–277.

AMD CORP. 2005. *Software Optimization Guide for AMD64 Processors*.

BAULIER, J., BOHANNON, P., GOGATE, S., GUPTA, C., AND HALDAR, S. 1999. Datablitz storage manager: Main-memory database performance for critical applications. In *Proceedings of the ACM*

SIGMOD International Conference on Management of Data (SIGMOD '99). ACM Press, New York, NY, 519–520.

BENDER, M. A., DEMAINE, E. D., AND FARACH-COLTON, M. 2000. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS '00)*. IEEE Computer Society, 399.

BENDER, M. A., DUAN, Z., IACONO, J., AND WU, J. 2004. A locality-preserving cache-oblivious dynamic dictionary. *J. Algorithms 53,* 2, 115–136.

BENDER, M. A., FARACH-COLTON, M., AND KUSZMAUL, B. C. 2006. Cache-oblivious string B-trees. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '06)*. ACM Press, New York, NY, 233–242.

BERRENDORF, R., ZIEGLER, H., AND MOHR, B. 2002. PCL: Performance Counter Library. http://www.fz-juelich.de/zam/PCL/.

BOHANNON, P., MCLLROY, P., AND RASTOGI, R. 2001. Main-memory index structures with fixed-size partial keys. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data (SIGMOD '01)*. ACM Press, New York, NY, 163–174.

BONCZ, P. A., MANEGOLD, S., AND KERSTEN, M. L. 1999. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 54–65.

BRODAL, G. S. AND FAGERBERG, R. 2002. Cache oblivious distribution sweeping. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP '02)*. Springer-Verlag, Berlin, Germany, 426–438.

BRODAL, G. S., FAGERBERG, R., AND JACOB, R. 2002. Cache oblivious search trees via binary trees of small height. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 39–48.

BRODAL, G. S., FAGERBERG, R., AND VINTHER, K. 2004. Engineering a cache-oblivious sorting algorithm. In *ALENEX/ANALC*. 4–17.

CHEN, S., AILAMAKI, A., GIBBONS, P. B., AND MOWRY, T. C. 2004. Improving hash join performance through prefetching. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*. IEEE Computer Society, 116.

CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. 1999. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming language Design and Implementation (PLDI '99)*. ACM Press, New York, NY, 1–12.

CHOWDHURY, R. A. AND RAMACHANDRAN, V. 2007. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07)*. ACM Press, New York, NY, 71–80.

COMER, D. 1979. Ubiquitous B-tree. *ACM Comput. Surv. 11,* 2, 121–137.

COPELAND, G. P. AND KHOSHAFIAN, S. N. 1985. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data (SIGMOD '85)*. ACM Press, New York, NY, 268–279.

DEMAINE, E. D. 2002. Cache-oblivious algorithms and data structures. Lecture Notes from the EEF Summer School on Massive Data Sets, BRICS.

DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. 1984. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'84)*. ACM Press, New York, NY, 1–8.

DING, C. AND ZHONG, Y. 2003. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*. ACM Press, New York, NY, 245–257.

FASTDB. 2002. http://www.ispras.ru/ knizhnik/fastdb.html.

FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*. IEEE Computer Society, 285.

GARCIA, P. AND KORTH, H. F. 2006. Database hash-join algorithms on multithreaded computer architectures. In *Proceedings of the 3rd Conference on Computing Frontiers (CF06)*. ACM, New York, NY, 241–252.

GARCIA-MOLINA, H. AND SALEM, K. 1992. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Engin. 4,* 6, 509–516.

HANKINS, R. A. AND PATEL, J. M. 2003. Data morphing: An adaptive, cache-conscious storage technique. In *VLDB*. 417–428.

HARDAVELLAS, N., PANDIS, I., JOHNSON, R., MANCHERIL, N., HARIZOPOULOS, S., AILAMAKI, A., AND FALSAFI, B. 2007. Database servers on chip multiprocessors: Limitations and opportunities. In *Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR '07)*. Asilomar, CA.

HARIZOPOULOS, S. AND AILAMAKI, A. 2006. Improving instruction cache performance in OLTP. *ACM Trans. Datab. Syst. 31,* 3, 887–920.

HE, B., LI, Y., LUO, Q., AND YANG, D. 2007. EaseDB: A cache-oblivious in-memory query processor. In *Proceedings of the International SIGMOD Conference*. 1064–1066.

HE, B. AND LUO, Q. 2006. Cache-oblivious nested-loop joins. In *Proceedings of the ACM 15th Conference on Information and Knowledge Management (CIKM '06)*.

HE, B. AND LUO, Q. 2007. Cache-oblivious query processing. In *Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR '07)*. Asilomar, CA.

HE, B., LUO, Q., AND CHOI, B. 2006. Cache-conscious automata for XML filtering. *IEEE Trans. Knowl. Data Engin. 18,* 12, 1629–1644.

HE, B., LUO, Q., AND CHOI, B. 2007. Adaptive index utilization in memory-resident structural joins. *IEEE Trans. Knowl. Data Engin. 19,* 6, 772–788.

HENNESSY, J. L. AND PATTERSON, D. A. 2002. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers.

INTEL CORP. 2004. *Intel(R) Itanium(R) 2 Processor Reference Manual for Software Development and Optimization*.

INTEL CORP. 2007. Intel vtune performance analyzer. http://www3.intel.com/cd/software/products/asmo-na/eng/239144.htm.

KIM, S., CHANDRA, D., AND SOLIHIN, Y. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, 111–122.

KNUTH, D. E. 1998. *The Art of Computer Programming*, 2nd ed. Addison-Wesley.

LAMARCA, A. AND LADNER, R. E. 1997. The influence of caches on the performance of sorting. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 370–379.

LEHMAN, T. J. AND CAREY, M. J. 1986a. Query processing in main memory database management systems. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86)*. ACM Press, New York. 239–250.

LEHMAN, T. J. AND CAREY, M. J. 1986b. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 294–303.

MANEGOLD, S. 2004. The Calibrator (v0.9e), a cache-memory and TLB calibration tool. http://www.cwi.nl/~manegold/Calibrator/.

MANEGOLD, S., BONCZ, P., AND KERSTEN, M. 2002. Generic database cost models for hierarchical memory systems. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB '02)*.

MARR, D. T., BINNS, F., HILL, D. L., HINTON, G., KOUFATY, D. A., MILLER, J. A., AND UPTON, M. 2002. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technol. J. 6,* 1.

MONETDB. 2006. http://monetdb.cwi.nl/.

RAMAKRISHNAN, R. AND GEHRKE, J. 2003. *Database Management Systems*, 3rd ed. McGraw-Hill.

RAO, J. AND ROSS, K. A. 1999. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 78–89.

RAO, J. AND ROSS, K. A. 2000. Making B+-trees cache conscious in main memory. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM Press, New York, 475–486.

SAMUEL, M., PEDERSEN, A. U., AND BONNET, P. 2005. Making CSB+-trees processor conscious. In *Proceedings of the 1st International Workshop on Data Management on New Hardware (DAMON '05)*. ACM Press, New York, NY, 1.

SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*. ACM Press, New York, NY, 23–34.

SHAO, M., AILAMAKI, A., AND FALSAFI, B. 2005. DBmbench: Fast and accurate database workload representation on modern microarchitecture. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '05)*. IBM Press, 254–267.

SHATDAL, A., KANT, C., AND NAUGHTON, J. F. 1994. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 510–521.

STONEBRAKER, M., ABADI, D. J., BATKIN, A., CHEN, X., CHERNIACK, M., FERREIRA, M., LAU, E., LIN, A., MADDEN, S., O'NEIL, E., O'NEIL, P., RASIN, A., TRAN, N., AND ZDONIK, S. 2005. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very large Data Bases (VLDB '05)*. VLDB Endowment, 553–564.

SUN CORP. 1997. *UltraSPARC (R) III Cu Users Manual*.

TIMESTEN. 2006. http://www.oracle.com/timesten/index.html.

TPC. 2004. http://www.tpc.org/.

VAN EMDE BOAS, P., KAAS, R., AND ZIJLSTRA, E. 1977. Design and implementation of an efficient priority queue. *Math. Syst. Theory 10*, 99–127.

YOON, S.-E. AND LINDSTROM, P. 2006. Mesh layouts for block-based caches. *IEEE Trans. Visual. Comput. Graph. 12,* 5, 1213–1220.

ZHOU, J., CIESLEWICZ, J., ROSS, K. A., AND SHAH, M. 2005. Improving database performance on simultaneous multithreading processors. In *Proceedings of the 31st International Conference on Very large Data Bases (VLDB '05)*. VLDB Endowment, 49–60.

ZHOU, J. AND ROSS, K. A. 2003. Buffering accesses to memory-resident index structures. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB '03)*. 405–416.

Received February 2007; revised September 2007; accepted January 2008