Association for Computing Machinery
Advancing Computing as a Science & Profession

Ubiquity, an ACM publication
August 2014

Ubiquity Symposium

The Multicore Transformation

Multicore Processors and Database Systems

by Kenneth A. Ross[1]

Editor's Introduction

Database management systems are necessary for transaction processing and query processing. Today, parallel database systems can be run on multicore platforms. Presented within is an overview of how multicore machines have impacted the design and implementation of database management systems.

[1] Work supported by NSF grant IIS-09-15956.

Ubiquity Symposium

The Multicore Transformation

Multicore Processors and Database Systems

by Kenneth A. Ross

Database systems have been a success story for parallelism. Historically, concurrent/parallel processing of transactions and queries allowed database systems to efficiently use system resources, primarily disk drives that were the main performance bottleneck. In modern systems, the performance bottlenecks have shifted. Central performance issues now include CPU cache performance, memory-resident concurrency management, and the efficient implementation of critical sections. In this short article, I will give an overview of how multicore machines have impacted the design and implementation of database management systems.

A central function in database management systems is transaction processing. Transactions allow users to update data atomically with guarantees of consistency and durability. The fundamental correctness criterion of transaction processing is serializability, i.e., equivalence to a serial execution. Nevertheless, database systems do not execute transactions serially: There are often plenty of opportunities to do independent work in parallel.

A second important function in database management systems is query processing. Users query the database state using a declarative query language such as SQL. For query processing in a read-mostly system with batch updates, parallelism helps to scale the performance of the system: The system can partition the data (either in advance or at query time) and work on the pieces independently.

The core database algorithms, joins, aggregations, partitioning, and sorting, are all relatively easy to parallelize. Unlike in general-purpose programming, the impediments to effective parallelism are usually not serial bottlenecks in the algorithms themselves. Instead, difficulties primarily occur as a result of the data distribution. Skew can cause imbalances in workloads. Data hotspots can block transactions and serialize execution.

The recent trend toward large main memories has significantly reduced the impact of magnetic disks (particularly the high random access latency) on overall system performance. The database system can cache the working set in RAM. Durability (persistence) can be achieved via logging, which has a more I/O friendly sequential write pattern, or via on-line replication to other database instances. Additionally, magnetic disks are being supplemented by solid-state disks that provide persistence with much better random I/O times.

The net result of these trends is that in-memory database system performance has become even more critical. The problems of concurrency remain, but have shifted a level up in the memory hierarchy. Just as implementers used to architect data structures such as B-trees to give good locality and concurrent access on disk, today's implementers need to architect for locality and concurrency in RAM and cache. One can no longer ignore the cost of cache misses or synchronization primitives such as mutexes on the grounds that they are cheap relative to a disk I/O.

Parallel database systems can today be run on multicore platforms, with each core being treated as if it were a separate machine. On machines with simultaneous multithreading, one could go even further and treat every hardware thread as a separate processor. However, such a mapping fails to capture the intrinsic sharing of RAM and cache resources that characterizes a multicore chip. For example, consider the simple but common task of aggregating data to answer a query. Imagine a large table containing sales data, and a query that asks for the total sales (in dollars) for each product. A simple technique for answering this kind of query is to build a hash table on the product key. Each hash table entry stores the running sales subtotal. After a complete pass through the data, the hash table contains the query result.

To parallelize this computation, one could divide the sales table evenly among the cores/threads, each with its own hash table. However, this approach multiplies the memory requirements for the hash tables by the number of threads. If there are many products, the total memory required will be much higher. Also, threads that share cache resources will now be competing for the same fixed cache space using a much larger memory footprint. As a result, they may suffer from reduced performance due to thrashing in the cache.

An alternative solution would be to use a shared table that all threads access. The memory is better utilized, but a different problem surfaces. Because the threads are updating a shared structure, the hash cells need to be protected by a mutex, or need to be updated by operations that the hardware guarantees to be atomic. The overhead of extra instructions for atomic operations or mutexes is noticeable, but is not the most significant overhead. Suppose that the sales table contains some very popular items, i.e., products with a particularly large number of

sales. If these popular items are sufficiently common, threads may stall waiting for access to the item they need to update, leading to serial rather than parallel execution.

The problem of aggregation on multicore machines has been studied by my group at Columbia on a variety of recent multicore platforms [1, 2, 3]. A high-level solution to the problem is to selectively replicate the popular items on the various cores, while keeping just one copy of the less frequent items in a shared table. The replication can be done either explicitly by a database operation using sampling, or by a library designed to detect contention and respond by cloning hotspot data items. However, on some platforms, the penalty for updates to any kind of shared structure is very high. On such platforms an explicit partitioning of the data by product key, so that each thread's hash table has a disjoint set of product keys, seems to be the best approach even though it requires an extra partitioning pass through the data. The general principle of partitioning disk-resident data into smaller local disjoint working sets is a standard technique in parallel database systems [4].

For joins on multicore machines, highly tuned parallel implementations can effectively use the parallel cores [5, 6, 7]. Using the SIMD resources available on modern machines is an important source of additional parallelism [5]. Partitioning work among the cores needs to be done carefully for joins, because an even partition of join keys does not guarantee an even partition of join work [7, 8]. Parallelization of serial query plans for multicore execution has been studied by Pankratius and Heneka [9].

The fact that different algorithms are preferred on different multicore platforms means that implementers may need to make architecture-dependent choices. This quandary would not come as a surprise to users of Graphics Processing Units (GPUs), which are high performing multicore devices with a very particular paradigm for efficient programming. (In fact, several recent research projects have considered using GPUs as accelerators for database workloads.) To maintain good performance on multiple platforms, implementers must either maintain multiple versions of the performance sensitive code, or specialize the algorithms based on a performance calibration [10]. Adapting algorithms dynamically based on current estimates of various compute and memory resources remains an important open problem for databases.

For high-performance transaction processing in RAM, the cost of managing mutexes can become the primary bottleneck. One promising approach to this problem is to partition the database in such a way that each core is exclusively responsible for processing a subset of the data. Besides the previously mentioned enhancement of parallelism, this arrangement means that no locking is needed to ensure correct data access, since each partition is accessible by just one core. Suppose the partitioning is performed in a way that allows transactions to need data

from just one partition. Then a (lock-free) serial execution at each node can guarantee serializability. For applications with this kind of local data access, the performance improvements can be significant [11]. Efficiently handling a mix of local and nonlocal transactions remains an open research question [12].

Even when processing data that is primarily disk-resident, multicore architectures demand modifications to traditional database system design. For example, the Shore-MT system reorganized and reimplemented code for critical sections, focusing on optimizations for scalability rather than for single-threaded performance [13]. In some cases, abstractions and encapsulation had to be broken in order to optimally synchronize the central data structures.

Within the class of multicore CPUs, the architectural details can be important. The Sun Niagara T3 architecture, for example, is designed for throughput using a large number (8 per core) of lightweight threads that are interleaved by the processor; there are 16 cores on a single chip. Each thread gets the CPU typically every fourth cycle, meaning that single-threaded performance is not great.

Nevertheless, the interleaving means that the latency of certain operations such as branch resolution can be hidden between stages where the thread is active. As a result, the CPU does not need extensive hardware for branch misprediction, and more cores/threads can be supported in a fixed transistor budget.[2] Throughput is a goal that is well aligned with parallel database workloads. In fact, the Niagara architecture has been able to generate impressive TPC-C and TPC-H benchmark numbers, including the top-performing TPC-C result.[3]

Another important architectural detail is the memory subsystem. On recent machines with multiple multicore chips, memory is partitioned among the chips, leading to a Non-Uniform Memory Access (NUMA) architecture. On one hand, operations should try to use local memory if possible, because access is faster. For example, the join processing algorithms of Albutiu et al. [7] are NUMA-aware. On the other hand, it is important to balance memory allocation among the cores, and naive acceptance of operating system default allocation policies can lead to unbalanced (and sometimes degenerate) behavior.

Looking into the near future, it appears that chips will contain even more cores. There is a strong possibility that memory bandwidth to the chip will become the new bottleneck for data-

[2] The Niagara T4 improves single-threaded performance with a higher clock frequency and more branch prediction logic, at the cost of reducing the number of cores relative to the T3.

[3] www.tpc.org, April 2013. Oracle Database 11g R2 Enterprise Edition w/RAC w/Partitioning running on SPARC SuperCluster with T3-4 Servers and achieving US$1.01 per tpmC.

intensive applications like databases. In such a scenario, it may pay to compress the data in RAM, and to decompress it in cache [14].

Another possible trend is that chip manufacturers will be constrained by energy considerations from having a large number of general-purpose cores operating concurrently. If so, it may pay to implement special purpose accelerators for commonly used primitives, and to budget some chip area for these accelerators [15]. Database systems represent a significant workload for modern server machines, and specialized accelerators for database primitives could be orders of magnitude more efficient than general-purpose cores, both in terms of work done per transistor, and in terms of energy cost.

References

[1] John Cieslewicz and Kenneth A. Ross. Adaptive aggregation on chip multiprocessors. In *International Conference on Very Large Databases, VLDB*, pages 339–350, 2007.

[2] John Cieslewicz, Kenneth A. Ross, Kyoho Satsumi, and Yang Ye. Automatic contention detection and amelioration for data-intensive operations. In *SIGMOD Conference*, pages 483–494, 2010.

[3] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. Scalable aggregation on multicore processors. In *International Workshop on Data Management on New Hardware, DaMoN*, pages 1–9, 2011.

[4] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems* (Third ed.). McGraw-Hill, 2003.

[5] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multicore CPUs. *PVLDB* 2, 2 (2009), 1378–1389.

[6] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD Conference*, pages 37–48, 2011.

[7] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main-memory multi-core database systems. *PVLDB* 5, 10 (2012), 1064-1075.

[8] Kenneth A. Ross and John Cieslewicz. Optimal splitters for database partitioning with size bounds. In *International Conference on Database Theory, ICDT*, pages 98–110, 2009.

[9] Victor Pankratius and Martin Heneka. Moving database systems to multicore: An auto-tuning approach. In *International Conference on Parallel Processing, ICPP*, pages 582–591, 2011.

[10] Stefan Manegold. The Calibrator, a Cache Memory and TLB Calibration Tool. 2004.

[11] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB* 1, 2 (2008), 1496–1499.

[12] Andrew Pavlo, Evan P. C. Jones, and Stanley B. Zdonik. On predictive modeling for optimizing transaction execution in parallel OLTP systems. *PVLDB* 5, 2 (2001), 85–96.

[13] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '09*, pages 24–35, ACM, New York, 2009.

[14] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter A. Boncz. Superscalar RAM-CPU cache compression. In *International Conference on Data Engineering, ICDE*, page 59, 2006.

[15] Lisa K. Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. In *International Symposium on Computer Architecture, ISCA*, 2013.

About the Author

Kenneth Ross is a Professor in the Computer Science Department at Columbia University in New York City. His research interests touch on various aspects of database systems, including query processing, query language design, data warehousing, and architecture-sensitive database system design. He also has an interest in computational biology, including the analysis of large genomic data sets. Professor Ross received his PhD from Stanford University. He has received several awards, including a Packard Foundation Fellowship, a Sloan Foundation Fellowship, and an NSF Young Investigator award.

DOI: 10.1145/2618403

