A Multicore Database Is Not a Distributed System

Neha Narula
MIT CSAIL

Database concurrency control is evolving to parallelize well on machines with many cores. Some researchers advocate viewing a multicore machine as a shared-nothing distributed system, where data is exclusively partitioned between cores [8]. This approach breaks down for changing workloads or workloads that do not partition well. In the meantime, other systems have shown that concurrency control need not be too expensive. We advocate scheduling transactions according to workload characteristics in order to address the problems that arise with true data contention.

In a shared-nothing main-memory database, transactions use a single-threaded execution model [4]. This technique eliminates the need for synchronization protocols like locking or validating for all transactions which only touch data on a single partition. Many systems use this approach on multicore machines, with a partition per core [6–8]. This technique is effective when a workload partitions perfectly, potentially providing 30% gains over a database which uses concurrency control [10]. Unfortunately, this strategy suffers from many serious drawbacks with more general workloads:

Load imbalances. If a single core becomes overloaded, overall throughput suffers. Solutions have been suggested that re-partition data on the fly, but this can be expensive if it must happen often. Data like popular stocks, news articles, and frequently used search terms are constantly changing with time. The monitoring and data movement associated with workloads that change frequently can be very costly.

Badly partitioning workloads. Many important workloads on the Web do not partition well. Facebook and Twitter store data based on a two-way follows relationship, and Google's ad serving framework requires keeping track of ad campaigns by customer and by keyword, which is also a many-to-many relationship. These types of applications will have a significant number of multi-partition transactions given *any* partitioning, requiring a protocol like two-phase commit or timestamp ordering [1, 9]. Two-phase commit is expensive due to the required rounds of communication and stalls on cores. Timestamp ordering can cause stalls when a transaction has to wait for a slower one assigned earlier in the timestamp ordering. Even in workloads that mostly partition, the cost of executing multi-partition transactions can be so high that it overwhelms the cost of executing single partition transactions [2].

A *soft partitioning* approach using shared memory provides more flexibility. Instead of exclusive access to partitions, each core has "preferred" data partitions, and uses concurrency control. Transactions on a single soft partition are mostly sent to the same core, reducing the potential for conflicts. However, any core is able to correctly handle any transaction since all use concurrency control. Systems like Silo and Hekaton have shown that a soft partitioning approach can get very good performance, achieving millions of transactions per second on up to 32 and 24 cores, respectively [3, 10]. A big new challenge lies in using a good strategy depending on a workload's characteristics, and extracting whatever concurrency can be had when there is contention.

We have an approach which, for some operations, parallelizes contentious writes on many cores [5]. This technique can schedule transactions into phases according to their properties; in workloads with many contending writes, reads are delayed in order to perform writes in parallel. We are investigating potential extensions to phase scheduling, for example, in a workload which partitions well, data can be marked as strictly partitioned (owned by a single core), and cross-partition transactions can be delayed until concurrency control is used again.

Modern multicore hardware provides cache coherence whether we want it or not, so instead of building systems which reimplement cache coherence via communication, we should leverage it. Phase scheduling transactions lets a database benefit from different concurrency control techniques depending on the workload.

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4 - 7 2015, Asilomar, California, USA

References

[1] Cowling, J., and Liskov, B. Granola: low-overhead distributed transaction coordination. In *USENIX ATC* (2012), USENIX Association, pp. 21–21.

[2] Curino, C., Jones, E., Zhang, Y., and Madden, S. Schism: a workload-driven approach to database replication and partitioning. *VLDB 3*, 1-2 (2010), 48–57.

[3] Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N., and Zwilling, M. Hekaton: Sql server's memory-optimized oltp engine. In *ICDE* (2013), ACM, pp. 1243–1254.

[4] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-Store: A high-performance, distributed main memory transaction processing system. *VLDB 1* (August 2008), 1496–1499.

[5] Narula, N., Cutler, C., Kohler, E., and Morris, R. Phase reconciliation for contended in-memory transactions. In *OSDI* (2014), USENIX Association, pp. 511–524.

[6] Pandis, I., Johnson, R., Hardavellas, N., and Ailamaki, A. Data-oriented transaction execution. *VLDB 3*, 1-2 (2010), 928–939.

[7] Pandis, I., Tözün, P., Johnson, R., and Ailamaki, A. Plp: page latch-free shared-everything oltp. *VLDB 4*, 10 (2011), 610–621.

[8] Salomie, T.-I., Subasu, I. E., Giceva, J., and Alonso, G. Database engines on multicores, why parallelize when you can distribute? In *Eurosys* (2011), ACM, pp. 17–30.

[9] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD* (2012), ACM, pp. 1–12.

[10] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy transactions in multicore in-memory databases. In *SOSP* (2013), ACM, pp. 18–32.