Technical Perspective of Concurrent Prefix Recovery: Performing CPR on a Database

Philip A. Bernstein
Microsoft Research
philbe@microsoft.com

Where do novel database system research results come from? In the 1970's, most systems research papers proposed mechanisms to support abstractions that were being explored for the first time, such as data translation, indexing, query optimization, high performance transactions, distributed databases, heterogeneous databases, and replicated databases. Novelty was easy to come by. These abstractions now form the core of the database systems field.

Since then, the main abstractions of database systems have not changed much. So where do novel solutions come from now? I claim they are driven by six trends, listed below with some recent examples:

1. New hardware mechanisms – multicore, solid-state disks, vector processing, non-volatile RAM, RDMA, GPUs, FPGAs, enclaves.
2. New software mechanisms – log-structured storage, column storage, transactional memory, blockchain, consensus algorithms, distributed hash tables, machine learning.
3. New data models – key-value stores, XML, JSON, graphs.
4. New system platforms – cloud computing, cloud storage, large main memories, cloud-fog-edge, serverless computing.
5. New workloads – stream processing, OLAP, map-reduce, training and serving ML models, graph algorithms over big data, data science, stateful web services.
6. Different system-level goals – scalability, throughput, consistency, latency, fault tolerance, availability, elasticity, cost, extensibility, security, privacy, manageability, robustness.

There is a well-known repertoire of techniques to address these challenges. They include access control, asynchronous operations, batching, caching, checkpointing, compare-and-swap, compression, cost-based optimization, encryption, function shipping, indirection, lazy updates, locking, materialization, multi-versioning (copy-on-write), parallelism, partitioning, pipelining, pre-fetching, replication, speculation, state machines, timeouts, timestamping, transactional queues, triggers, watchdogs, workflow, and those in (2) above. There are many more of course, but probably not hundreds.

Let us use the paper I am introducing as an example. It addresses the problem of checkpoint and recovery for a transactional key-value store—a well-known workload. Its novelty arises from its ability to scale throughput linearly on a large multicore server with negligible increase of latency and from the way it attains this goal.

To appreciate the significance of the paper's novel contributions, let us consider classical solutions to the problem it solves. A simple approach is partitioning, that is, partition the workload so that each core is responsible for reads and writes on a distinct partition of the database. This would ensure there is no interference between the cores. It would enable each core to log its updates independently of the others, thereby ensuring recoverability. However, it would not be robust with respect to changes in the fraction of operations that are directed to each partition. One core could be overloaded while another has spare capacity.

If we relax the partitioning assumption, the problem becomes much harder. For example, updates by different cores may conflict, which creates dependencies between them. Suppose transaction T updates x in one core and a transaction T' in another core reads x's updated value and updates y. Then if a checkpoint includes that updated value of y, it must also include the updated value of x. That is, the checkpointed state needs to be consistent with respect to updates that were acknowledged to users and with respect to each other. In a classical database system, these problems are addressed by locking and logging. However, locking introduces contention among parallel operations, which limits scalability. Logging also introduces contention, such as the need for parallel threads to coordinate their append operations to the shared log, plus it has many other inefficiencies. Although effective techniques have been developed for many of these problems, logging still poses limits to multicore scalability.

This paper's solution circumvents the scalability bottleneck by combining several techniques: a new recovery model called *concurrent prefix recovery*, the use of a 2-version data model, and a state machine.

With concurrent prefix recovery, the *system* defines a commit point, in contrast to standard techniques where clients issue commit. Clients independently synchronize with this commit request, never blocking each other. In the Rest state (i.e., normal operation), each client runs transactions serially on the latest active version, v. When the system issues a commit, it moves to the Prepare state. Each client periodically reads the system's state. After a client moves into the Prepare state, it continues executing normally. However, if its transaction accesses an item already at version $v+1$, the transaction aborts, the client moves to the next state, called In-Progress, and the transaction re-executes. After all clients have entered Prepare, the system moves its state to In-Progress. At this point, version v of all items are immutable and can be checkpointed without interference from clients. Now, if a transaction wants to update an item x at version v, it creates a new version $v+1$ of x instead of updating version v. (Another client that is still in Prepare state might see this version $v+1$, leading to an abort as explained above.) If x is already at version $v+1$, the transaction updates x in place. After all version v items have been checkpointed, the system returns to the Rest state with $v+1$ as the latest active version.

Voilà. Simple, scalable, and novel. Read on, for details.