Consistency in Motion

Chris Douglas
UC Berkeley

Abstract

All techniques for concurrency control establish an effective execution order for transactions. Invariants that guarantee consistency rely on this execution order, either implicitly by construction (the *locked point* in 2PL, deterministic locking in Calvin [7]), or explicitly by assigning transaction timestamps [2]. Scheduling workloads that mix short and long-running transactions has been a consistent challenge for data management, as each *reordering* of running transactions discards completed work. Variance in query runtimes caused by mixing short and long-running transactions motivates either partitioned or hybrid systems that explicitly separate these workloads, admitting anomalies by running transactions at lower isolation levels.

Executing long-running queries as materialized views may improve throughput by allowing concurrent transactions to be reordered. We believe this can be implemented without radical changes to query execution, by embedding streaming incremental computation into existing concurrency control techniques.

Views are defined by queries, raising the question: how does the execution of a query, particularly a long-running query, differ from the maintenance of a materialized view? The overhead of creating and maintaining a view can be significantly higher than executing a query in isolation, so any benefit to workload throughput or makespan must come from improved concurrency. For incremental view maintenance (IVM), this is a constraint: while view maintenance can be deferred [5], any enumeration of a view must remain consistent with the transaction ordering generated by updates. In contrast, a *running* query need only be ordered and consistently enumerable when it commits.

Further, *any* query may reference a view, so IVM must be fully general and available in ways that running queries are not. For example, the query's intermediate state is not visible to other queries. Deferred maintenance does not block other queries' progress. Unlike materialized views, queries can restart without affecting concurrent transactions. Most importantly, permutations of per-object histories among running transactions can *effect* the transaction order, rather than binding a view to an established transaction order.

To consume the stream of conflicting changes to a long-running query, we adopt DBSP [3]. DBSP compiles relational queries— including aggregation and recursion— to a dataflow that consumes changes to the base tables and eagerly emits changes to the view. For brevity, we note only that DBSP supports commutative groups. For relational algebra, DBSP uses $\mathbb{Z}$-sets to model the effects of adding, removing, and updating tuples from relations, similar to the derivation-counting approaches for view maintenance and view adaptation [4]. The circuit itself maintains only the state necessary to emit changes to the view, but the "integral" of its output materializes the view or in this case, the query result.

1 Example: 2PL and IVM

In the following, we sketch how one could graft incremental computation to traditional 2PL. Consider a long-running, serializable query T_γ holding long read and write locks. Transactions writing into conflicting ranges must wait until the query releases its lock, even when the update does not affect the output of T_γ. This *irrelevant update* problem is well known in IVM literature, and a sub-case of views that are *self-maintainable* with respect to an update [1]. When $T\gamma$ reaches its *locked point* owning all the locks it will ever own, its serialization order is determined [2], but if these locks could be made porous then conflicting transactions could precede T_γ by merging writes into the running query— maintaining the view— consistent with a serializable execution.

In the lock table, instead of acquiring a shared lock, T_γ acquires an *incremental shared lock* on tuples in its read set, connected to the incremental query Q_γ^Δ obtained by incrementalizing Q_γ with DBSP. A short-running transaction T_1 writing into an locked range would normally be ordered after T_γ, but if its write is absorbed by Q_γ^Δ, then T_1 can be ordered before T_γ without aborting and restarting either transaction. Incremental shared locks can only be shared among transactions that support incremental updates, and a write succeeds only if all incremental queries absorb the write.

Admitting a write also affects write locks held by T_γ as T_1 should read the value before T_γ obtained the lock; this avoids circular information flow (G1c [1]). Correct ordering of read-write dependencies entails a corresponding *incremental write lock* that maintains an anchor value, with deltas ordering one or more active, incremental transactions.

Note that T_γ taking a dependency on T_1 does not entail a cascading abort of T_γ if T_1 aborts. To undo the effect on the incremental query, adding the inverse element— which must exist, as $\mathbb{Z}$-sets form a commutative group— to the incremental query should remove the effect of T_1 from the integral of Q_γ^Δ. This capability can be used to reorder and speculate on the effects of transactions without restart.

As we locate the handle for the incremental query in the lock table, writes are *proposed* to incremental queries and may be refused, gracefully degrading to a standard lock. An incremental query could accept only irrelevant updates, reject updates as it converges, or accept updates until a threshold/deadline. Refusing a write may cascade into removing the effects of the transaction from other incremental queries to ensure a consistent ordering. A write not unanimously accepted by incremental shared locks could be rejected and its effects undone, or ordered among incremental queries.

[1] i.e., can update the view using only the change and the view contents without referring to the base data