Scheduling for Lazy Evaluation of Transactions in Database Systems

Daniel Tahara
daniel.tahara@yale.edu
Yale University

1 Background

A typical database workload comprises a series of reads and writes on often overlapping but possibly disjoint portions of data. In order to allow developers to make certain assumptions about their data, notably the ACID guarantees (atomicity, consistency, isolation, durability), most modern RDBMSs utilize techniques such as locking, two-phase commit protocols, and multi version concurrency control. Together, these techniques ensure that, despite concurrent access to the data, one will never reach or remain an intermediate state that can be considered somehow ‘invalid.’

Regardless of the specific implementation, all RDBMSs rely on a transaction manager in order to effect the mechanisms necessary to ensure that transactions are executed and decide whether each given transaction has committed or needs to be aborted and rolled back. In particular, through locking and the other techniques described above, the transaction manager creates and then executes a determines a possibly interleaved execution plan that reflects some serial order of those transactions. It then, by immediately attempting to make the changes to the underlying physical store, can determine if the transactions violate any integrity or other constraints. If not, the transaction succeeds and is ‘commited’; if not, the transaction is aborted and rolled back.

Until recently, the status quo has been that the transaction managers should execute transactions immediately. Such a perspective adheres to our intuition; in order to provide the correct commit/abort decision, the transaction must actually be executed to completion and the end state of the RDBMS verified. In typical, nondeterministic database system, this is the only method of execution because many situations such as deadlock or database crashes may cause a transaction to abort independently of the transaction logic. However, when we switch our attention to deterministic DBMSs such as [1] and [3], these nondeterministic failures are disallowed, so every transaction will be committed as long as it does not violate any integrity constraints.

Consider then, the following: What if, rather than immediately executing transactions as they arrive in the database, the database delayed the execution to some later point in time as determined by some scheduling algorithm. Minus some formalisms, we have the beginnings of what we might call a lazy transaction manager, an idea which we explore in more detail below.
2 Discussion

Define an *eager transaction manager* as one that, upon receiving a transaction (or set of potentially competing transactions), immediately creates an execution plan and applies the operations to the database. Notably, the issuers of the query are notified immediately of the result of their transactions, and the database state changes immediately.

In contrast, define a *lazy transaction manager* as one that only applies transactions when the specific data to which they refer are referenced by some other transaction; that is, when a read request is issued for that data. The transaction manager still provides an immediate commit/abort decision, but the guarantees associated with the decision may not be reflected in the physical state of the database state until some time in the future.

A lazy transaction system has a number of major benefits. Since transactions are executed only when one or more of the data that they modify are actually requested by another operation, we can reap tremendous gains from cache locality. To understand why, consider a chain of ten successive writes on datum A. The system receives a read request for A, and then brings the appropriate database page into memory. The writes are then immediately applied to the in memory object, with no further calls to disk. We ammortized the cost of a single disk read over eleven operations. This ammortization occurs in contrast to the situation in a traditional, eager system, where the writes might be interleaved among other operations that cause the relevant page to be evicted from the page cache. In that case, each write must first fetch the page from disk before executing.

This is just one of the benefits of a lazy transaction system, others of which include decreased communication costs among elements of the RDBMS, and the lack of a centralized lock manager (for a number of reasons, a locking system must be replaced by mutexes on the individual transactions), and we will attempt to quantify the results in our work, as described below.

3 Implementation

Actually implementing the lazy transaction manager defined above is incredibly non-trivial, since a lot of our intuitive reasoning about serializability and consistency break down when we remove the immediate temporal relations among a set of transactions. This is done in two phases, *stickification* and *substantiation*. Stickification is the part of transaction execution that occurs immediately, and in particular, evaluates the transaction to determine its record dependencies. It then generates system records to indicate which other transactions it depends on (that is, it reads a value written to by a previous, potentially unexecuted transaction). In addition, the stickification phase executes any parts of the transaction that might cause the transaction to abort—specifically, integrity and other user-defined constraints or conditions.

Stickified transactions are then stored for later execution in the substantiation phase. The transaction records implicitly create a partial ordering based on the dependencies among the stickified transactions. Using this partial ordering, we can create a directed acyclic graph with edges representing dependencies. When the DBMS receives a request for record R, these records are examined for all transactions that modify R, and then executes those transactions to generate the present state of the tuple.

The interesting question for both generating this partial ordering and actual executing it is in which direction we should mark the dependencies. Intuition says that transactions
should know which other transactions they are dependent upon in order to ensure that those can be executed, but it also significantly complicates decisions like when to free the memory representing a transaction. Thus, there are a number of tradeoffs that one must consider in building such a system.

4 Deliverables

PVLDB accepts submissions on a monthly basis, with March 1st being the deadline for consideration for VLDB 2014. I will be contributing to an ongoing effort [2] to complete an implementation of a lazy transaction manager for submission by that time. However, if we do not meet this deadline, we will aim for a later VLDB deadline (April or May) as appropriate. In addition to the implementation, we hope to produce experimental results demonstrating which of the two alternative scheduling strategies is more optimal and under what workloads it is preferable to a traditional transaction manager.

Specifically, I will be working on the substantiation component of the transaction manager. Apart from the challenge of representation, as mentioned above, there are a number of challenges with regards to actually scheduling the execution of the transactions, especially when execution is split across multiple threads. For example, what if the DAG contains two mutually independent transaction nodes (A and B) that depend on the same single transaction (C). If execution of the leaves of these two branches begins in separate threads, there is a point at which they will both attempt to execute the same set of logic/transactions, which could not only lead to wasted resources (in the best case when operations or idempotent), or worse, deadlock conflicts. One possible solution might be to use mutex locks local to each transaction, and, after the thread that is executing the transaction dependencies for A executes transaction C, deleting B’s dependence on C. Of course, there are numerous tradeoffs for any approach, but I will hope to identify an optimal (or at least correct) approach that results in minimal wasted resources and minimal contention.

References

