Passive-Aggressive Locking: A Novel Database Concurrency Control Protocol

Arjun Kavi
Adviser: Daniel Abadi
2012 May 6th

Abstract
Maintaining ACID compliance in database represents a significant impediment to concurrent database throughput. Addressing this problem, a novel locking-based concurrency protocol called Passive-Aggressive Locking is presented in which a transaction either receives the entirety of locks or none at all. This protocol shows promise of significantly increasing concurrent transaction throughput under certain workloads. This work presents a characterization of the performance of the protocol for certain load distributions in simulation and implemented in PostgreSQL, where it performs markedly better than the native MVCC-style concurrency control.

Protocol and Characterization
The protocol for Passive-Aggressive Locking (PAL) is conceptually straightforward. Each transaction prior to execution requests all the locks it requires. If at that point in time all the locks can be granted, the transaction acquires them and executes. However, if at least one of the locks cannot be immediately granted, the lock requests are placed in queues for their respective locks and a counter tracking how many of the requested locks are available is incremented. On finishing execution, each transaction releases the locks that it acquired, and decrements the lock counter for each of the transactions waiting on the lock that could now acquire it. The lock manager then selects the earliest transaction in the queue with a lock counter of zero (implying that all of the locks it is requesting are now available to be granted), grants the lock requests, increments the counters of any waiting transactions that are newly blocked, and starts execution of the transaction.

Under this protocol, it is possible that a transaction may never acquire the locks it needs in order to run. A transaction could potentially be perpetually stymied by a constant inflow of other transactions requesting a subset of its locks which become available before the set in its entirety. However, the willingness to trade starvation for throughput is not without precedent in database
concurrency control protocols; optimistic concurrency control also sacrifices fairness to any one transaction for overall throughput. However, in OCC, repeated validation failures come not only at the cost of starvation, but also at the cost of wasted computation, a cost not present in PAL. Furthermore, unlike 2PL where the need to clear deadlocks also could result in wasted computation, it is impossible for PAL to deadlock due to the guarantee that a transaction receives all of its locks at once. Finally, it should be noted that unlike MVCC, PAL provides a full serializability guarantee.

Use of PAL, however, has a prerequisite, that a transaction is able to declare all of the locks that it will need prior to execution. This is not always possible, for example in a case where an update learns what tuple to update by reading the values of another. This is identical to the problem faced by Calvin with such "dependent transactions", wherein to definitively acquire all the locks needed by the transaction, it would be necessary to lock over the entire table, with the consequent loss of concurrency. The solution presented in Calvin, that of decomposing the transaction into a set which progressively "discover" the lockset needed by the transaction, would likely be viable for PAL as well. Another potentially possible solution to this problem is predictive modeling to attempt to guess which rows might need to be locked, permitting a subset, rather than the table in its entirety to be locked.

**Non-Optimality**

It's trivial to present scenarios in which PAL outperforms 2PL, which might make it tempting to suggest that PA locking could produces an optimal ordering for any given load of transactions. Unfortunately, pathological scenarios can be created which demonstrate this is not the case. Note that throughput (ie, number of transactions per second) is not a reasonable criterion for optimality, as it implies that a handful of very small transactions is "more optimal" than running a single longer transaction. Instead, take optimality to mean utilization of resources (ie, the percentage of time a set of resources is being used by a transaction to perform work that will eventually be committed).

Consider the following scenario: There are two resources which the lock manager is overseeing. The load consists of transactions which either lock on one or both of the resources. The run starts with a single transaction locking on a single resource, which blocks any number of transactions which require both the resources. The moment before this transaction finishes, another transaction requiring a lock on the other resource is requested, and receives the lock it requests. The running continues in this manner, with a series of single transactions requiring one of the two resources blocking all transactions requiring both continuously, alternating from one resource to the other. Under these circumstances, resource utilization is roughly 50%, while it's clear under 2PL it would be much higher, approaching 100%
Zipfian Skew

Whether PAL is likely to perform better than another concurrency protocol is thus likely a function of the distribution of lock-sets of a transaction load. One plausible distribution is a Zipfian distribution, wherein the frequency with which a given item appears is inversely proportional to its rank in the frequency ordering, raised to a power known as the "skew" factor (Note that a skew of 0 produces a uniform distribution, and 1 a canonical Zipfian distribution). Equivalently:

\[ f_n = \frac{1}{n^s} \]

Where \( n \) = frequency rank of the item, and \( s \) is the skew factor.

Zipfian distributions are found frequently in real world database applications (e.g., the frequency with which a book in Amazon's database is ordered by a customer), and thus a locking protocol that performs well under it has potential in a wide array of niches. Such distributions are problematic, as they represent a relatively high degree of contention (A given transaction is highly likely to require access to one of the top handful of resources). However, it seems likely that PAL would perform well under this sort of load compared to traditional 2PL. While, by necessity, any transaction accessing several of the most common resources will be forced into near-serial execution, a transaction touching on only a handful, which would be stalled by the serial ordering of the other transaction in 2PL, would under PAL be more likely to "sneak" on through.

Simulation

In order to provide a picture of the performance characteristics of PAL without confounding factors that would be present in a real database, a simulation was created. The simulation maintains 100 transactions, either executing or waiting to execute, continuously for a period of 1 second, reporting the number of transactions carried out. Furthermore, to shed some light on the degree to which starvation represents a problem for PAL, the amount of time each transaction waited to acquire its locks was recorded. Using this framework, an implementation of PAL was compared against an implementation of strict 2PL, with the acquisition phase modeled by having the transaction request its lock set evenly spaced over the run of the transaction.

The upper graph depicts transaction throughput and the lower the ratio of throughput; both demonstrates that no matter what the zipfian skew of resource access, PAL outperforms 2PL by a factor of between two to four. However, as predicted, PAL degrades much more gracefully in the face of increased contention cause by greater skew, with a fairly dramatic increase in relative throughput between skews of 0.4 and 0.6. The last graph demonstrates that worries about starvation are probably unfounded, as even the 95th percentile of PAL transactions wait a shorter amount of time to execute than those of 2PL.
The 95th percentile is depicted rather than the median as it better captures the phenomenon of starvation; were starvation to be present, it might not affect the median queuing time at all, while causing the handful of transactions at the far end of the wait time distribution to have to wait near indefinitely.
The applicability of simulations to real world behavior is always tenuous at best, and this simulation makes several simplifying assumptions, chief among them is the assumption that transaction lengths are relatively uniform and even in distribution, which could potentially have a large effect on the results, as demonstrated in the non-optimality proof. The validity of the uniformly spaced acquisition of locks in the 2PL simulation is also potentially problematic.

**Implementation in Postgres**

In order to test the efficacy of PAL in "real world" situations, the protocol was implemented in PostgreSQL. PostgreSQL natively uses MVCC for concurrency control, providing, by default, snapshot isolation level transaction isolation. This is effected in practice by a transaction acquiring access locks on the tables involved (to prevent, for example, dropping of the table while the transaction is in progress), selecting the appropriate tuples to read by timestamp, and grabbing exclusive locks over any tuples it updates or deletes, which it releases once the transaction has committed. Thus, PostgreSQL’s lock manager can be viewed as a read snapshot accompanied by strict 2PL over the write-set.

The default usage model of PostgreSQL, a server to which clients connect, did not readily lend itself to the purpose of benchmarking these two protocols against each other. Since the server spawns a single process to execute queries for each incoming client, the test framework would require the ability to create multiple clients to interact with the server, as well as the ability to closely synchronize them. Instead, the backend was altered to generate a hardcoded
number of query execution processes, which, once instructed to by the post-
master process, attempt to successively execute as many queries successively as
possible. After a time limit, the postmaster process halts the query execution
processes and calculates throughput. Furthermore, to reduce the number of
causes of variability, background processes, such as the Autovacuum process,
are disabled.

The benchmark itself consists of the query “UPDATE test_table SET quan-
tity = quantity + 1 WHERE #{IDs}“ executed on a table with cardinality
1,000, five concurrent query execution processes and a runtime of 10 seconds.
The set of ids #{IDs} is randomly generated at runtime, with an average set
size of 10 and distribution with varying Zipfian skews. The benchmark was run
on a Intel Xeon x5550 at 2.67 Ghz with 12 GB of RAM. In order for the PAL
version of the database to have knowledge of the write-set prior to execution,
the query constructor informs the process of the lockset it will need to acquire,
information not available to the native lock manager. However, since the execution
planner, being identical in both, does not have access to this information
in either version and thus cannot take advantage of it, this has no effect on
run time of the transaction. Note that since the query only requires writes,
the benchmark is effectively a comparison between strict 2PL and PAL. While
measuring throughput is an imperfect benchmark, as mentioned above, it is
adequately informative given that the transactions are of roughly equal size, as
they are here.

Figure 4: Throughput in PostgreSQL

![Throughput in PostgreSQL](image)

There are several caveats to be noted about these results. The first is that in
reviewing the logs of the runs, standard Postgres suffers from a not insignificantly
number of deadlocks (which Postgres deals with by aborting then restarting
one of the deadlocked transactions). This significantly reduced throughput, and while this would represent a legitimate victory for PAL, which cannot deadlock, if it is characteristic of real-world behavior. However, this is difficult to ascertain. The second is that ittermitently, runs using both protocols threw execution terminating errors reported by Postgres as "unexpected data beyond EOF" and attributed by Postgres to a faulty kernel. Runs displaying this error were discarded and reran. Furthermore, there were drastic swings of nearly 100% in throughput between runs with the same settings. To ameliorate the effects of this last problem, five runs were conducted for each data point, with the median selected. As a result of these issues, these results must be taken as being extremely preliminary and tentative. The data, however, is extremely clear in the fact that PAL produces significantly higher throughput than the standard MVCC of PostgreSQL for this kind of transaction. Furthermore, although not sufficient to conclusively demonstrate it, the data suggests PAL implementation decreases more gracefully under increasing Zipfian skew than the native concurrency control, as predicted by the simulation.

**Conclusion**

Although tentative, these results strongly indicate that Passive-Aggressive Locking represents an improvement over two-phase locking for at least a subset of transaction loads. It not only promises to improve transaction throughput, but also provides a dead-lock free guarantee, and the tradeoff made on the risk of starvation is relatively mild. This work, while not conclusively answering questions regarding the behavior and efficacy of Passive-Aggressive Locking, demonstrates the validity of the protocol as a topic of further study and interest.