Plan-ahead concurrency control for highly scalable database systems

Abstract

In this work, I investigate the algorithmic problem of plan-ahead concurrency control, where a database system aims to reorder transactions as they enter the system to build more concurrent schedules. First, I propose that the theoretical aim of a plan-ahead concurrency control scheduler ought to be reducing “schedule depth” and so minimizing the number of independent sets of transactions that must be executed in order to carry out the full set of transactions. This objective allows the planning problem to be formatted as a minimization problem, which can be reduced to graph coloring. The graph coloring reduction allows us to draw on a number of properties from the graph coloring literature, most importantly that the problem is NP-hard. Finally, I analyze a simple greedy approach to transaction reordering. While not having particularly strong theoretical properties, the greedy approach performs well in simulation. I also look at how the greedy approach performs in the more realistic setting of a multi-threaded test database system.

Introduction

Concurrency control is the mechanism by which a database system allows multiple transactions to run simultaneously while ensuring the results are “serializable”, or equivalent to a serial ordering. Traditional methods of concurrency control typically fall into two categories: those which, like two-phase locking, prevent conflicts by dynamically ordering transactions at runtime, and those which, like optimistic concurrency control (OCC), check for conflicts after the transaction has run and abort if any are found.

While historically improvements in hardware often came in the form of improved single-threaded performance, more recently a number of limitations have driven hardware manufacturers to turn to parallelism achieve greater performance. As both the amount of available parallel execution power and the demand for high-throughput database systems grows, the concurrency control system becomes an increasingly important component of the database system’s performance. However, most available methods, including pessimistic concurrency control (like two-phase locking) and optimistic concurrency control, struggle to scale, especially on high-contention workloads. These workloads, in which the same record is read or written by a large number of transactions at once, cause serious problems for optimistic concurrency control mechanisms because they lead to a large number of transaction retries—OCC systems are built on the assumption that transaction conflicts are rare and retrying in case of the occasional conflict is acceptable, but as the number of retries increases the overall throughput of the system drops off quickly. Pessimistic concurrency control methods deal better with high-contention workloads, as these systems ensure that no two conflicting transactions will run at the same time.
However, even these systems struggle to scale to multithreaded settings, as the cost of communicating between CPUs increases. Further, lock-based approaches often run into issues when multiple threads request a lock at the same time, as the cost of synchronizing multiple cores grows quickly. As shown below, two-phase locking struggles to scale even on read-only workloads due to the high cost of synchronizing across threads.

![Two Phase Locking Graph](image)

Two phase locking on a high-contention read-only workload. Notice that throughput actually decreases when the number of CPU cores increases from 60 to 80 due to communication overhead. Figure from [1].

We aim to investigate a newer method, “plan-ahead concurrency control”, which determines a serializable schedule for the transactions before runtime, allowing more concurrent schedules, less inter-thread communication, and overall higher throughput.

**Problem Statement**

To formulate transaction scheduling as an algorithmic problem, we will characterize a *transaction* by two sets of records, a “read set” of records that the transaction requires to be read but not modified, and a “write set” of records that the transaction modifies (or may modify), potentially in addition to reading. While standard in the database literature, note that these terms may be slightly confusing, as the transaction must be able to read all records in the write set in addition to those in the read set. We define conflicting transactions as follows:

Transactions T1 and T2 *conflict* if any of the following hold:

- T1’s read set intersects with T2’s write set, or $r_1 \cap w_2 \neq \emptyset$
- T2’s read set intersects with T1’s write set, or $r_2 \cap w_1 \neq \emptyset$
- T1’s write set intersects with T2’s write set, or $w_1 \cap w_2 \neq \emptyset$

Non-conflicting transactions can be run at the same time and will always produce a serializable outcome. If we define a *batch* as a set of non-conflicting transactions, then a schedule can be written as an (ordered) set of batches, where execution might entail simply running the batches in order. We will also refer to schedules as *packings*, as creating a schedule entails packing transactions into batches. A reasonable objective for plan-ahead concurrency control scheduling
is to minimize \textit{schedule depth}, the number of batches in a packing. As illustrated below for an example with five transactions, while it is possible to place each transaction in its own batch and execute them serially, by reordering them, the whole thing can be done two steps by grouping odd and even transactions. Minimizing the number of batches in a packing is desirable for a number of reasons. Not only does this method result in a schedule that will finish fastest under the assumption of infinite parallelism and fixed transaction execution time, but it also reduces the amount of communication necessary between threads. Since every transaction in a batch can run independently of others in the batch, cross thread communication is only truly required between batches. While in practice it may be desirable to offer more fine-grained synchronization, minimizing the number of batches in a schedule should generally lead to lower synchronization requirements.

<table>
<thead>
<tr>
<th>Record</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction T1</td>
<td>Write</td>
<td>Write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>Write</td>
<td>Write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>Write</td>
<td>Write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4</td>
<td>Write</td>
<td>Write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T5</td>
<td>Write</td>
<td>Write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

An example schedule where the shortest schedule consists of two batches.

**Graph Coloring**

If we define the \textit{batch packing problem} as the minimization of schedule depth for a given set of transactions and the \textit{k-batch packing} problem as the decision problem of whether a given set of transactions can be packed into k or fewer batches, it is fairly simple to reduce these problems to graph coloring and vis-versa. First, to reduce batch packing to graph coloring, construct a so called conflict graph by creating a node for each transaction and drawing an edge between any two nodes which conflict, as defined in the last section. Then if you are able to color this conflict graph with k-colors, the set of all nodes with a given color corresponds to a set of non-conflicting transactions, and since every node must be colored, every transaction must be included in one of the k batches implied by the coloring, resulting in a packing of size k.
Example conflict graph for the above set of transactions.

In the other direction, given a general graph, we can create a new transaction for every node in the graph. For every edge, create a new record that no other transaction touches, and insert it into the write sets of both transactions corresponding to the endpoints of the edge. Each batch in a successful k-packing of these transactions corresponds to a set of non-conflicting transactions, and since each edge is represented by a conflicting write, no two transactions in a batch correspond to nodes with an edge between them, and thus the original graph is k-colorable.

These reductions lead immediately to some discouraging theoretical results. First, since k-colorability NP-complete, so is k-batch packing, and minimizing the number of batches in a packing is NP-hard in general. Further, it has been shown that approximating the minimum number of colors for a graph within a nearly-linear factor $n^{1-\varepsilon}$ is NP-hard for any $\varepsilon > 0$, meaning that even guaranteeing a packing is close to the optimal packing becomes computationally infeasible as the number of transactions in a packing grows [2]. Despite these difficult bounds, it is worth examining how a packing algorithm behaves in practice.

A Greedy Approach

The most basic packing algorithm can be derived from the first fit algorithm for graph coloring, and involves simple trying to insert each transaction into each existing batch until finding a batch it doesn’t conflict with. The pseudo code is included below.

Since first-fit for batch packing is nearly identical to first-fit for graph coloring, many of the results carry over. For every set of transactions, there is always some initial ordering for which the greedy algorithm is optimal, simply by taking the optimal coloring and listing the transactions from each batch one by one in batch order. Forming transactions from the crown graph (shown below), the greedy algorithm may, in the worst case, end up creating a schedule (coloring) with $n/2$ batches (colors) rather than the 2 required by the graph. This means that, in the worst case, the greedy algorithm does arbitrarily badly. While the crown graph relies on the worst possible input ordering (requiring you to get extremely unlucky), randomizing the input doesn’t necessarily
help: it has been shown that some graphs exists for which you get a “bad” ordering with high probability (here “bad” is roughly a $n^{1-\varepsilon}/\log_2(n)$ factor, see Kučera 1991 for details)[3].

**First-fit:**
transactions = [t1, ..., tn]
batches = []
For txn in transactions:
  Added = false
  For batch in batches:
    If not txn.conflicts_with(batch):
      batch.add(txn)
      Added = true
    If not added:
      batches.append(newbatch(txn))
Return batches

Pseudocode for greedy packing algorithm.

The crown graph, with an example of how certain node orderings lead to arbitrarily bad colorings. From https://en.wikipedia.org/wiki/Greedy_coloring.

However, these results are fairly pessimistic, generally assuming worst case scenarios and often relying on schedules with unrealistically large numbers of records per transaction or extremely contrived patterns of conflict. To see how the greedy algorithm performs in a more realistic scenario, I simulated its performance on a high contention write-only scenario assuming each transaction would choose its records uniformly at random from a fixed size database. Below, the results for transactions of 4 writes out of a database of 200 records (extremely high 2% conflict) are shown, with the total number of transactions in the packing (n) on the x-axis and the average number of transactions per batch on the y-axis. Since the absolute maximum number of transaction per batch is given by 200 / 4 = 50, we can see that the greedy algorithm very quickly reaches more than half the optimal number of batches. This is important for two reasons: first, the more transactions that must be included in a packing before sending it to the database for execution is directly related to the latency of processing each transaction, so keeping batches small is important for getting results back faster. This is a natural result of the pipelined nature of this kind of pre-planning approach. Second, the algorithm itself is non-linear, so packing larger batches takes more time per transaction.
Simulated results. Optimal packing is less than 50 transactions per batch.

Overall, the time complexity is given by number of transactions (n) * number of batches (m) * cost of checking for conflicts in the worst case. Number of batches is typically and worst case $O(n)$, although for low conflict workloads (eg read only workloads) it may be lower. Cost of check for conflicts is $O(n)$ in the worst case, given $O(n^3)$ overall, but this assumes each transaction will touch a growing number of records as the database grows, which is unrealistic. Assuming each transaction touches only a constant number of records, overall complexity is $O(n^2)$, which is much better than exponential, but still worth being careful about.

**Conclusion**

Overall, the comparison between graph coloring and schedule packing is a valuable one for putting some bounds on what is possible. However, despite the lack of strong theoretical guarantees, the greedy algorithm shows promise on simulated workloads. While the practical efficiency of the algorithm and plan-ahead concurrency control in general remains to be proven, initial results (both here and in Stan’s report) demonstrate that the problem, while tricky, should be tractable for at least some workloads.
References

