Project description - first semester
Deterministic highly-scalable concurrency control within database systems

1 ABSTRACT

Prior research on deterministic database systems has proposed pre-determining a serializable schedule of transactions prior to their execution. The pre-determined schedule is represented via an explicit dependency graph, which determines the order in which transactions must execute and effectively obviates the necessity for explicit locking.

Within this work we show that building a dependency graph by considering batches of transactions rather than single transactions independently from one another may lead to significant performance improvements. We devise an algorithm for creating execution schedules from a set of transactions divided into batches based on an approximation of the set packing problem. We subsequently investigate its performance through a series of simulations and show that our algorithm produces schedules that are much more performant than the baseline schedules created without batching. The performance increase is present throughout a series of scenarios which approximate real workloads. We further investigate the source of the increase in performance and show that the process of transaction batching leads to an increased utilization of shared locks within the system which results from transaction reordering within batches. The resulting schedule is highly parallelizable and may offer a new highly performant concurrency control mechanism.

The second semester of work on this project will be focused on replacing the simulation with a functional system to show that the overhead of the schedule creation mechanism does not render the approach impractical. Moreover, the created system will be compared with two phase locking to determine the scenarios in which currently investigated system is superior.

2 INTRODUCTION

As explained within the abstract, the goal of this project is to create a concurrency control mechanism that does not use explicit locking and which builds an execution dependency graph using batches of transactions rather than single transactions independently. The general functioning of the system we will be considering may be described as:

*dna@yale.edu
†NetID: ss2755, stanislaw.swidwinski@yale.edu
1. Transactions arrive between time \( t \) and \( t + \Delta t \) forming a batch.

2. A schedule is created from these transactions. This schedule will be referred to as a "batch schedule"

3. The "batch schedule" is merged with the schedule that existed before the current batch arrived.

Hence, there is a notion of a "schedule" and a "batch schedule." Logically, any "schedule" can be thought of as a classical lock table with a ready queue that holds all transactions that can begin executing at a given time. The major logical difference is that the executing threads in the system never communicate with the lock table directly. Rather, they use the schedule API to obtain a transaction ready to execute and report the finished execution back to it.

![Figure 2.1: General structure and API of a schedule.](image1)

![Figure 2.2: General structure and API of a batch schedule.](image2)

### 3 Creating a Batch Schedule

A high-level batch schedule creation algorithm looks as follows:

```c
create_batch_schedule(unordered_set<Transaction> txns_within_batch) {
    batch_schedule bs = {}
    unordered_set<Transaction> not-scheduled = txns_within_batch
    while (not-scheduled.size() != 0) {
        packing P = get_transaction_packing(not-scheduled)
        bs.add_to_batch_schedule(P)
        not-scheduled = not-scheduled - P
    }
}
```
return batch_schedule
}

Each of the steps involved in creating the schedule will be expanded upon. The overarching
goal of the algorithm is to create a batch schedule, which attempts to maximizes the number
of "in-flight" transactions.

3.1 TRANSACTION PACKING. WHAT IS IT AND WHY DO WE USE IT?

The presented algorithm divides the batch into "packings," each of which contains non-
conflicting transactions. Since we would like to execute "largest" packings first, the problem
of finding transactional packings is equivalent to finding a maximum set packing.
To see the parallel, let us consider the set of transactions as a set of sets of locks $A = T_1, \ldots, T_k \subset \Omega^2$. Every "packing" that we require is equivalent to a maximal set packing $B \subset A$ by definition of the problem. At every iteration of the while loop above we are searching for a maximum set packing from within the transactions that have not yet been scheduled.

3.2 OBTAINING A TRANSACTION PACKING

Let $T$ be the input to the get_transaction_packing routine. It is the set of transactions that
have not yet been added to a schedule. Let $f(x, y)$ be the weight-function for transactions,
where $x$ is the number of exclusive locks requested and $y$ be the number of shared locks
requested. $f(x, y)$ can also be abbreviated as $f(t)$ when written in terms of a transaction $t \in T$.

Call the output of the following routine a "packing":

```c
get_transaction_packing (unordered_set<Transaction> T) {
    packing P = {};
    while (there exist transactions not colliding with members of packing) {
        Find transaction t within T that
        a) minimizes f(t)
        b) does not collide with txns in the packing
        c) Is not already present in the packing
        Add t to packing
    }
    return packing
}
```

1. The notion of size may be defined arbitrarily and will be made more concrete within the following sections
2. Of course, the locks may be exclusive or shared, but for simplicity of explanation let us assume that the locks are
   exclusive. This assumption is relaxed when discussing the way packings are obtained and the logic presented
   here is easily extended to include shared locks.
3. The name comes from the parallel to the set packing problem which is discussed above.
The output is an approximation of maximal set of transactions that can run at once. Notice that depending on the definition of $f(t)$ one can prioritize different types of transactions: those with the fewest exclusive locks, the fewest shared locks, or some efficiently-computable function of the two. This might allow the system to adapt to different types of workload.

Notice too, that we check that a transaction "does not collide with txns in packing". This is a check very reminiscent with that within OCC systems. The major difference is that we perform the check before a transaction is run rather than after, which may save a lot of time for long transactions.

### 3.3 Adding a Packing to a Batch Schedule

```c
batch_schedule::add_to_batch_schedule(packing P) {
    for (Txn t in P) {
        for (Lock l in t.write_set) {
            Add_to_lock_queue(l, exclusive_lock)
        }
        for (Lock l in t.read_set) {
            Add_to_lock_queue(l, shared_lock)
        }
    }
}
```

Adding a lock to the queue is straightforward. The only tricky thing is coalescing shared locks across packings and within packings. The following example shows how the process works:

![Example Diagram](image)

**Figure 3.1:** An example of the process of creation of a batch schedule. Green lock requests represent shared lock requests and orange represent an exclusive lock request. $f(t)$ used is the number of exclusive locks requested.
4 JOINING BATCH SCHEDULES WITH THE GLOBAL SCHEDULE.

Logically, adding a batch schedule to a global schedule requires merging the two logical graphs. Notice that this process does not incur big overhead in terms of lock contention, since it may merge the graphs one dependency at a time.

5 POSSIBLE OPTIMIZATIONS

Notice that up until now we have considered creating the batch schedules with no consideration of the pre-existing schedule. To increase the number of transaction in flight, it makes sense to schedule transactions that do not collide with those from the formerly existing schedule first. This may be incorporated within the get_transaction_packing routine as an optimization. This possibility has not yet been investigated and remains within the realm of future work.

6 SIMULATION SETUP

To evaluate the schedules created by our algorithm we built a simulation, which assumes lack of overhead associated with schedule creation. First, we describe the structure of the simulation framework in detail, then we present the results.

Most any simulation functions according to the following outline.

1. Initialization
2. A series of measurements
3. Data dump
4. Data aggregation, parsing and plotting

Each of the steps will be expanded upon. Initialization is done mainly within the Python script responsible for coordination of experiments. The middle two steps are done by the C++ code and the last step is, again, done from within Python. For code and implementation details please consult the source code available at https://github.com/stswidwinski/final_project.

6.1 INITIALIZATION - CONTROL OVER EXPERIMENTS

For our purposes an experiment is a series of measurements that keeps all parameters but for a single parameter constant. The parameters that control the generation of transactional load and the experiments are:

Time Period. This is the time during which transactions may enter the system. Time is considered to be quantized into integral intervals (of length 1). Hence, one can think about time as a counter from 0 to time period - 1.
**Transaction Number.** This is the total number of transactions within a workload.

**Linear Time Factor.** As described later on, the length of a transaction is linearly dependent on the number of locks requested by a transaction. In fact, this is just the number of locks × Linear Time Factor.

**Uncontested Lock Space.** The number of locks within the "uncontested lock space." The purpose of dividing the lock space into contested and uncontested spaces is to enable a forced lock contention within a subspace of the locks (simulating "hot records").

**Uncontested Average Held Locks.** The number of locks from the uncontested lock space that are held by a transaction.

**Uncontested Held Locks Standard Deviation.** The standard deviation of the number of locks from the uncontested lock space held by transactions.

**Contested Lock Information.** The simulation offers three parameters mirroring those above. They provide information about the "hot records" within the system. Again, one may control the size of this space, the average number of held locks and the standard deviation of it.

**Percentage of write txns.** Transactions within the simulation are either write-only or read-only. This variable specifies the percentage of write transactions.

**Bursty Seed Chance.** Parameter used to determine the generation of bursty work loads. More on this and the following item in further sections.

**Bursty Linear Factor.** As above.

**Batch Length.** The length of a batch for the batching model.

**Models.** Models that are to be run during the experiments. Right now we can choose from serial, batched, real-time (lack of batching). Currently the latter two are simulated with the assumption of infinite number of threads available, which further allows us to ignore physical capabilities of machines.

**Data.** Data to be gathered. More on that later.

**Repetitions.** Every measurement may be repeated on a newly generated workload according to the distributions that follow from set parameters. This factor establishes the number of repetitions made.

### 6.2 Workload Generation

Before any experiments may be run, a workload of transactions must be generated. Within any workload in the system the following are invariant:

1. The number of locks held by any transaction is described by a normal distribution with the average and the standard deviation specified by the user.

2. The locks are chosen uniformly at random from the corresponding lock space, whose

---

4that is, a transaction may not request both exclusive and shared locks.
size is specified by the user.

3. The duration of transaction execution is linearly dependent on the number of locks the transaction holds. It is just the number of locks requested × Linear Time Factor.

There are two kinds of workloads that are used within the simulation with distinct distributions of arrival times of the transactions: a bursty and a uniform distribution.

6.3 Generation of uniformly distributed workloads.

To generate a uniform workload, a uniform integral distribution from 0 to Time Period - 1 is utilized to randomize the arrival times for transactions. Subsequently, the number of contested and uncontested locks (and the specific locks) are randomized and added to transactions with the transaction duration being a byproduct of this. As one can expect the following is a sample histogram of transaction arrival times:

![Sample uniformly distributed workload](image)

Figure 6.1: Sample uniformly distributed workload

6.4 Generation of bursty workloads.

Generation of bursty workloads is more complex. Let \( d(i) \) indicate a transaction within the workload \( W \) such that \( \forall t \in W |d(i).arrival_time - i| \leq |t.arrival_time| - i \). Moreover, let \( d(i).arrival_time \) be defined as the maximal integer in case of \( W = \emptyset \). The generation proceeds as follows:

1. While (not all arrival times have been generated):
   a) Uniformly at random obtain \( i \) form within the integral interval 0 to Time Period - 1.
   b) Let \( P = \text{Bursty Seed Chance} + \frac{\text{Bursty Linear Factor}}{d(i).arrival_time} \).
   c) Let \( p \) be chosen uniformly at random from the real interval 0 to 1.
   d) If \( p < P \), we proceed to add a transaction with arrival time \( i \). Otherwise, we go back to first step without adding a transaction.

Clearly, the above algorithm will produce bursty (clustered) workloads with invariant num-
bers of transactions within them. The figure below shows sample distribution generated using this method.

Figure 6.2: Sample bursty workload generated using the described method. Parameters used are: Time period of $10^6$, Transaction number of $10^5$, Bursty Seed Chance of $4 \times 10^{-4}$, Bursty Linear Factor of 50.

6.4.1 Run time of the generation method

Clearly, the expected run time of the generation method per iteration is proportional to $\frac{1}{\text{Bursty Seed Chance}}^5$ and on average quickly approaches a small constant as more arrival times are added.

6.5 Data

For every experiment a set of data can be specified that should be gathered. Those for now include:

Workload Data. Data considering the workloads used for the experiment. Used to automatically create graphs of the arrival time distributions.

Transaction Completion Data. This includes the time of transaction's arrival, the time of its beginning of execution.

In-flight Transaction Data. The number of transactions being "in-flight" at every step of the simulation.

7 Experimental results.

We present two experiments measuring the total time necessary to execute transactional workloads in medium and high-contention scenarios. The created system is evaluated against the traditional non-batched system.

\footnote{in the worst case of very sparse transactions}
7.1 The impact of batch duration on medium contention loads.

The following parameters were used:

**Generation information**
- Time Period: 1000000
- Transaction Number: 100000
- Linear Time Multiplier: 35
- Uncontested lock space size: 1000
- Uncontested lock held avg: 10
- Uncontested lock held std dev: 5
- Contested lock space size: 0
- Contested lock held avg: 0
- Contested lock held std dev: 0
- Write Txn Perc: 0.2
- Bursty Seed Chance: 0.0004
- Burst Linear Factor: 50

**Experiment information**
- Batch Length: 10 20 30 40 50 100 200 250 300 350 400 450 500 600 700 800 900 1000 1250 1500 1750 2000 2500 3000 3500 4000 4500 5000 6000 7000 8000 9000 10000 15000 20000 30000 50000 75000 100000 125000 200000 300000 400000
- Repetitions: 3
- Models used: batched real_time
- Data gathered: avg_proc_time std_dev_proc_time

The following graphs summarize the experiment and are taken as an average of 6 runs, 3 for a bursty distribution and 3 for an uniform distribution.

![Workload completion time as a function of batch length](image.png)

Figure 7.1: Average completion time of workload for a medium contention workload.
The run time of the workload decreases significantly with the increase of the batch length. This seems to indicate that the batching algorithm helps improve the parallelizability of execution of transactions. This claim seems to be supported by the histogram of the execution time by the number of in-flight transactions seen below.

![Figure 7.2: The aggregated execution time by the number of concurrent transactions for batch size of 10](image)

![Figure 7.3: The aggregated execution time by the number of concurrent transactions for batch size of 800](image)

![Figure 7.4: The aggregated execution time by the number of concurrent transactions for batch size of 1900](image)

We see that with increase in the batch length the distribution "moves towards right," which means that effectively the number of parallel transactions increases.

### 7.2 The Impact of Batch Duration on Higher Contention Loads

The following parameters were used:

**Generation information**

<table>
<thead>
<tr>
<th>Time Period</th>
<th>Transaction Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>1000000</td>
</tr>
</tbody>
</table>
Linear Time Multiplier 35
Uncontested lock space size 1000
Uncontested lock held avg 10
Uncontested lock held std dev 5
Contested lock space size 0
Contested lock held avg 0
Contested lock held std dev 0
Write Txn Perc 0.5
Bursty Seed Chance 0.0004
Bursty Linear Factor 50

Experiment information

Batch Length 10 20 30 40 50 100 200 250 500 750 1000 1250 1500 2000 2500 3000 3500 4000 4500 5000 6000 7000 8000 9000 10000 15000 20000 30000 50000 75000 100000 125000 150000 175000 200000 300000 400000
Repetitions 3
Models used batched real_time
Data gathered avg_proc_time std_dev_proc_time

The following graphs summarize the experiment and are taken as an average of 6 runs, 3 for a bursty distribution and 3 for an uniform distribution.

Figure 7.5: Average completion time of workload for a higher contention workload.

We see the same trends as before with the histograms, again, confirming that batching increases the parallelizability of the load.
8 EXPLANATION OF IMPROVED EFFICIENCY

To explain the obtained results, we consider scenarios of increasing complexity. The two main forms of graphs used will be Gantt plots of transactions in time and dependency graphs of transactions. The Gantt plots portray the total time a transaction spends in the system and shows the time during which a transaction is actually being executed. Vertical dashed lines portray the end of a batch. Within the dependency graphs, the transactions (identified by their IDs) are the nodes and the directed edges portray the partial ordering of execution. In both the Gantt plots and dependency graphs read transactions are depicted using the green color and write transactions using the red color.
8.1 One Batch

The following simulations were run using the specified parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Period</td>
<td>60</td>
</tr>
<tr>
<td>Transaction Number</td>
<td>5 10 15 20 25 30</td>
</tr>
<tr>
<td>Linear Time Multiplier</td>
<td>2</td>
</tr>
<tr>
<td>Uncontested lock space size</td>
<td>50</td>
</tr>
<tr>
<td>Uncontested lock held avg</td>
<td>6</td>
</tr>
<tr>
<td>Uncontested lock held std dev</td>
<td>3</td>
</tr>
<tr>
<td>Contested lock space size</td>
<td>0</td>
</tr>
<tr>
<td>Contested lock held avg</td>
<td>0</td>
</tr>
<tr>
<td>Contested lock held std dev</td>
<td>0</td>
</tr>
<tr>
<td>Write Txn Perc</td>
<td>0.5</td>
</tr>
<tr>
<td>Bursty Seed Chance</td>
<td>0.0004</td>
</tr>
<tr>
<td>Bursty Linear Factor</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Experiment information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch Length</td>
<td>30</td>
</tr>
<tr>
<td>Repetitions</td>
<td>3</td>
</tr>
<tr>
<td>Models used</td>
<td>batched real_time</td>
</tr>
<tr>
<td>Data gathered</td>
<td>dep_graph txn_gant avg_proc_time std_dev_proc_time</td>
</tr>
</tbody>
</table>

8.1.1 Few Transactions (5)

With few transactions in play, there is a limited number of permutations and the transaction load is of relatively low contention. Even so, it is evident that the batching process parallelizes the workload, which is evident in the dependency graphs below.

Record that the batching process defined earlier in the work attempted to increase the number of concurrently running transactions. Clearly, transaction 0 collides with all the other transactions and the batching process moves such a transaction either to the "front" of the graph (if such a transaction is "short") or to the "back" of the graph (if such a transaction is long). Moreover, notice that the former happens only if there is no transaction that is shorter than the blocking transaction that may collide with it. Such reordering leads to a higher "utilization" of a shared lock, meaning that when a shared lock is held, the number of transactions holding said lock is higher for the batched system.

Of course, the above reasoning only holds for workloads without particular "hot set" of locks. Even so, it can be partially transferred onto workloads with "hot sets" of locks, since within a batch a system will still move blocking transactions that hold "hot" locks either to the beginning or the end of the dependency graph as seen above.

Even though the dependency graph seems to be "better" in the batched scenario, the time necessary for the batch to "come in" is a factor big enough for the batched system to perform worse than a real time system for a single batch in this scenario.
8.1.2 More Transactions (15)

The reasoning from above transfers completely to a situation with more transactions. The graphs below prove the points made and show that in scenarios where the execution of a batch is even longer than the batch itself, the execution times in real time and batches scenarios are nearly the same with slight advantage of the batched system. The details of the dependency graphs are not the central point of the discussion - the general shape of the graph is. The graph within a batched system is "higher" while the graph within a real time system is "longer." Notice also that some of the nodes within the real time system do not have dependencies at all. That is because they arrive when none of the locks required by them are being held (txn 4 and 14).
While for many transactions the dependency graphs are rather hard to see and reason about on a fine level, they may still be utilized for shape comparison described earlier. The Gantt charts shown below prove that when the batch length is much smaller than the length of execution, the more highly parallelizable schedule leads to shorter execution time of the batch. Hence, the take-away from the experiments with a single batch is that batching allows to move the blocking transactions to either "side" of parallelizable transactions (either before them or after them), so that as many transactions as possible hold a single shared lock when such is held.
The experiments were run for the following configuration:

Generation information
Time Period 60
Transaction Number 5 10 15 20 25 30
Linear Time Multiplier 2
Uncontested lock space size 50
Uncontested lock held avg 6
Uncontested lock held std dev 3
Contested lock space size 0
Contested lock held avg 0
Contested lock held std dev 0
Write Txn Perc 0.5
Bursty Seed Chance 0.0004
Bursty Linear Factor 50

Experiment information

Batch Length 30
Repetitions 3
Models used batched real_time
Data gathered dep_graph txn_gant avg_proc_time std_dev_proc_time

The dependency graphs and Gantt plots for two batches become complex to reason about relatively fast. The points within this experiment are the same as before. Since the batching process leads to higher lock "utilization" as described before, the number of times a lock "switches" from exclusive to shared mode is lower within a batched system. Accordingly, the execution time of multiple batches within a batched system will be lower than in the real time system, since the batched system "sees" fewer transactions (fewer flips from exclusive to shared mode).

The above hypothesis is confirmed by the fact that dependency graphs for batched systems have clearly defined "blocks" of read transactions that may execute concurrently whereas a dependency graph for a real time system does not posses such a feature. The following example shows the point and uses 30 transactions.
Figure 8.13: Dependency graph of transactions within a batched system.

Figure 8.14: Dependency graph of transactions within a real time system.

Figure 8.15: Gantt plot of transaction within a batched system.

Figure 8.16: Gantt plot of transaction within a real time system.
8.3 Many Batches

The logic presented transfers completely to formerly shown results. Please notice that the best completion times reported are very close to the range of allowed arrival times in the case of smaller contention, which implies that the load has been processed in time nearly optimal. The result must be taken with a grain of salt, since we are considering a simulation with an unlimited number of cores available, but it gives faith that a system with such a scheduling mechanism could be practical.

8.4 Future Work

In the remaining semester of this project we will focus on the following issues:

1. Investigation of quantitative measures for determining the extent of lock utilization
2. Investigation of write-only workloads within built algorithm
3. Building of a true system to measure the effects of batch-creation overhead and synchronization overheads. Comparison with a real two phase locking system to determine the benefits of the new method