CS5280: Optimizing Scheduling Algorithms

Raj Patil[1] **and Suparna Kar**[2]

[1]**CS18BTECH11039**
[2]**CS21RESCH11011**

ABSTRACT

We first briefly describe the specific problems that exist in the schedulers under observation, followed by the proposed tweaks in the schedulers that tackle the issues accompanied with corresponding arguments for correctness. Lastly, we see how the tweaked scheduler deals with the history that was misdealt with by the vanilla schedulers.

Keywords: BOCC, BTO, MVTO

CONTENTS

1 OPTIMIZING BOCC : DATUM-LEVEL TEMPORAL VALIDATION

1.1 Issue

Consider the history:

```
T1 : [----w1(x)----c1----]

T2 :              [----r2(x)----w2(x)----a2----]
```

T2 gets aborted due to T1 even though there are no incoming edges into T1 due to T2 in the conflict graph G. T2 gets aborted because the read set of T2 intersects with the write set of T1. If T2 committed, this history would still be in CSR, and we aim to fix unnecessary aborts like these.

1.2 Proposition

The cause for aborting T2 in the vanilla BOCC scheduler is that its read set intersects with the write set of T1: the scheduler would still output a correct history if it allowed this as the corresponding read of T2 arrives after the transaction T1 wrote on the corresponding data item. This could be dealt with if we kept note of the time stamps of each read that arrived for each transaction. We then consider that commit time of a transaction as the time stamp for all its writes.

Such requirements are trivial and what follows is the description of what should be done when a transaction requests a particular operation to be executed:

1.2.1 Transaction tj reads x

Insert a time-stamped tuple (now, x) into the read-log of tj

1.2.2 Transaction tj writes x

Insert the datum x into the write-set of tj Note that the final-commit time will be considered as the time-stamp for these writes.

1.2.3 Transaction tj requests validation

Now, as this is a backward oriented check, all committed transactions (ti's) can be bucketed into two categories: ones that committed before tj started and ones that committed after. The ones that were committed before wouldn't be an issue and don't need to be checked.

For the ones that committed after tj started, observing each datum in the intersection (based on datum) of the read-log of tj and write-log of ti. For each such datum, check if the read-time of tj was greater than the corresponding write time-stamp (i.e. commit time) for ti. If this does not hold for any datum, abort tj, otherwise its writes are safe to be committed.

1.3 Correctness

Building upon the correctness of the vanilla BOCC scheduler: notice that it validates the tj if the intersection of its reads with previously committed transactions (ti's) is null. This ensures that no edges go from tj to ti in the conflict graph in a crude manner, rejecting correct schedules as well. Here, we refine that check to ensure conflict graph acyclicity with some leeway: the read should have arrived after the write is written in the previously committed transactions. This still ensures that no edges are going back from tj to ti, hence guaranteeing correctness.

1.4 Consequences

In the history presented in section 1.1, instead of aborting due to a non-empty intersection of the read set of the current transaction (t2) and one that was committed after t2 started, we check that the corresponding read of t2 is scheduled after the commit of t1 and t2 continues execution.

2 OPTIMIZING BTO : CASCADING ABORTS

2.1 Issue

Consider the history:

```
T1: [--r1(x)------------------------------w1(x)-----a1---]

T2:      [------r2(x)--------------w2(y)---a2----]

T3:                [---------r3(y)----c3--]
```

T2 is aborting while performing W2(y) because the max-read timestamp of y is three which is greater than T2's timestamp. Similarly, T1 gets aborted while performing W1(x) because the max-read timestamp of x is two which is greater than T1's timestamp.

Observe that T1 need not be aborted as T2 is no longer part of the conflict graph: we try to address this problem of cascading aborts.

2.2 Proposition

The crux of the issue that leads to these cascading aborts is the laziness involved when dealing with aborts. When aborting a transaction, one should also make sure to clean up the effects it had on the scheduler and the database. An abort should render the scheduler and the database in a state that is indistinguishable from hypothetical case where the transaction being aborted did not arrive in the first place.

The vanilla BTO scheduler fails to do so while maintaining the max read and time stamps for each datum - if the timestamp of the transaction being aborted is stored in one of these meta-variables, its effect is retained after the abortion.

We can directly target this issue by recording the past values of these meta-variables for easier reversion. There are multiple ways to do so but we model these as a max-heap. We then summarize how the scheduler behaves with the tweaks in critical situations.

2.2.1 Meta-variables as max-heaps

Instead of storing a single time stamp for each datum for latest reader and writer, we model them as max-heaps and the root of the heap is the max time-stamp that the scheduler uses to decide the fate of incoming operation requests.

Instead of over-writing this meta-variable as in the vanilla BTO scheduler, we insert the new time stamp into the corresponding max heap and it bubbles up to the top, ready to be used by the scheduler.

This also allows us to deal with aborts elegantly: simply delete that time-stamp from the heap and the second largest time-stamp bubbles back up to the top of the heap - ready to be used by the scheduler as.

Note that we explicitly disallow duplicate insertions in the max-heaps (trivial modifications to the implementation).

2.2.2 transaction tj reads x

Reads can conflict with writes, hence we only check the max-write-heap of x for this arrival. If the time stamp of tj is greater than the max-writer's time-stamp, than the read request is safe and can be processed. The administrative work then involves inserting the time-stamp of tj into the max-read-heap of x for future use.

If this was not the case, we will have to abort tj. This involves deleting its timestamps from all the max read and write heaps for each datum to completely revert its effects. Note that due to the nature of correctness constraints on the heap data structure, the second largest time-stamp will bubble up to the top and the scheduler is ready to go as if tj didn't even arrive.

2.2.3 transaction tj writes x

Writes can conflict with writes and reads, hence we check both the max-read-heap and max-write-heap of x for this arrival. Now, again, if the time-stamp of tj is greater than both these maxes, then this write is safe and can be processed. The administrative steps now involve inserting the time-stamp of tj into the max-write-heap for x.

Aborts are dealt with just as in the case when a read operation was rejected. We delete time-stamps of tj from all the max read and write heaps for each datum.

2.3 Correctness

Correctness trivially follows from the same argument that was applicable to a vanilla BTO scheduler. The tweaked scheduler always ensures that conflicts always go from a lower time stamp to a larger time stamp which in turn ensures acyclicity of the conflict graph.

2.4 Consequences

In the history in section 2.1, the first abort is justified but the second one is unreasonable. With the modified BTO, when aborting t2, its time-stamp will be deleted from the max-read-heap of x and w1(x) will no longer be an invalid request as it is greater than (or equal) to the largest max-read timestamp and the largest write-time-stamp of x (t1 and t0 respectively). Hence, we have avoided the cascading abort.

3 OPTIMIZING MVTO : DATUM-LEVEL SERIALIZABILITY CHECK

3.1 Issue

Consider the History:

```
T1: [----w1(x1)----c1---]

T2:             [-----r2(x1)----------------w2(x2)---a2--]

T3:                              [---r3(x1)--c3--]
```

T2 gets aborted by T3 because there exists a read operation on x R3(x1). Notice that, T1T3T2 is a mono-version serial schedule for this multi-version history which is not accepted by MVTO. Unnecessary aborts like these can be avoided.

3.2 Proposition

Recall that the correctness of MVTO is contingent upon the output history being in MVSR i.e. one should be able to identify a serial monoversion schedule that has the same reads-from relation with the history being output. We build this proposition with this notion of correctness in mind.

3.2.1 transaction tj reads x

Here just as in the vanilla MVTO scheduler, we follow suit and simply map the version function to the writer of x with the largest time stamp less than that of tj.

3.2.2 transaction tj writes x

The vanilla MVTO scheduler is more crude in terms of its checks in this case and hence rejects some histories that could be in MVSR. We intend to refine these checks.

Observing the conditions for an abort: tj is aborted if a read (from tk), with a time-stamp greater than tj, was already scheduled to read from a write(from ti) with a time-stamp older(before in time) than tj (for time-stamps: $t_i < t_j < t_k$). The vanilla MVTO aborts this, but there is a chance for the complete history to still be in MVSR.

Trying to formalize that situation - our intention is to allow the output history to have the same reads-from-relation as that of a serial monoversion schedule. Assuming this schedule exists for now. In such a serial schedule tj will be placed after tk as otherwise, tk(the early reader) will read from tj(the writer) in the serial version instead of ti as it is doing in the original history. This relation could be managed if for each datum in the intersection of the read set of tk and the write set of tj, it was read by tk before it was written by tj.

If this does not hold for each datum in that intersection, then one cannot place tk before tj when finding a serial version and we abort tj. Otherwise, tj can continue.

This modification in the checks requires some additional administrative work, namely:

- maintaining read and write sets for each transactions
- logging read and write time-stamps for each operation of each transaction.

3.3 Correctness

Note that we constructed the tweaked algorithm with correctness in mind at all times and hence it follows trivially.

3.4 Consequences

For the history in section 3.1 when w2(x2) is requested this time, we can identify the previously problematic timestamp order of $t_1 < t_2 < t_3$ with t_3 reading from t_1. Instead of aborting right away, we notice that all writes of t2 come after their corresponding reads (if any) in t3. According to the modified scheduler this will allow t2 to continue execution without hampering correctness.

4 REFERENCES

Transactional Information Systems
Gerhard Weikum and Gottfried Vossen