From DAGs to Riches: Improving transaction throughput in a deterministic system

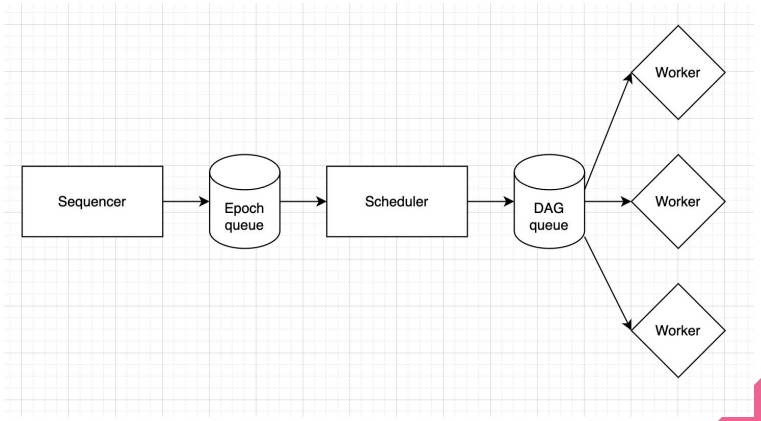
Rakrish Dhakal, Alexander Movsesyan, Pranav Sivaraman, Daniel Wei, Sathwik Yanamaddi

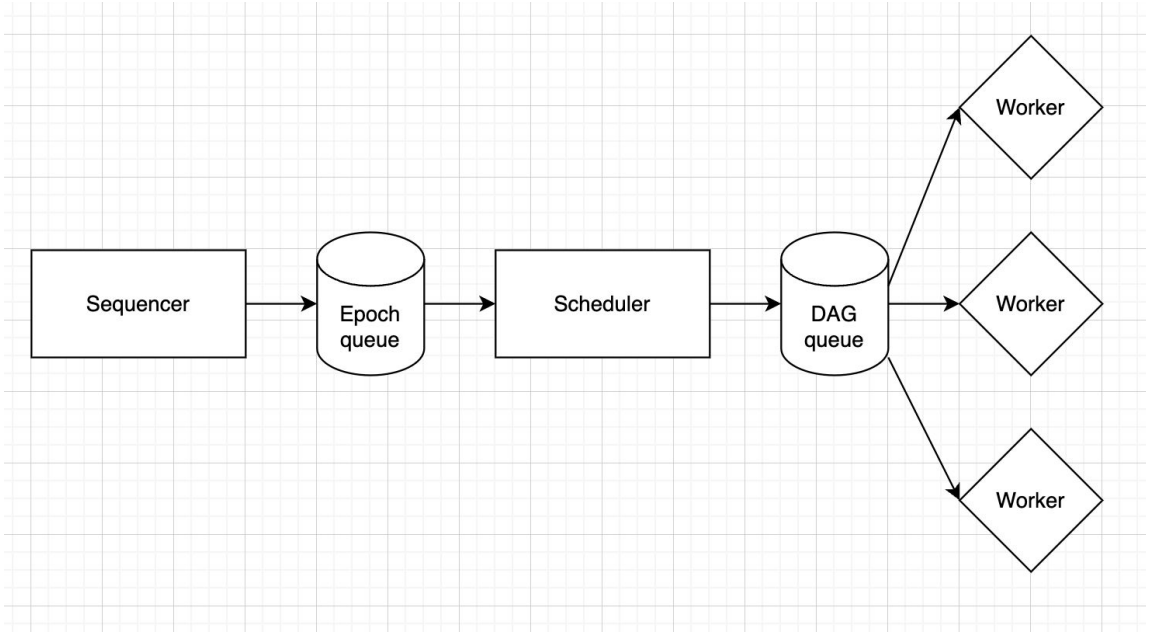
Project 2 Scheduler - Bottlenecks

- Scheduler maintains ready_txns queue
- Scheduler commits/aborts completed_txns
- Transaction locking, commit, validation, and assignment to workers, all occur in the same thread
- Goal: parallelize these different phases
    - i.e. offload them to the workers

Solution

- Parallelize transaction assignment to threads
- Build a dependency graph of transactions using the following rules:
  - txn *v* depends on txn *u* iff *u* < *v*, and either:
    i. *u*'s writeset intersects *v*'s readset
    ii. *u*'s readset intersects *v*'s writeset
    iii. *u*'s writeset intersects *v*'s writeset
- Use worker threads to continuously consume nodes in dependency graph with in-degree of zero

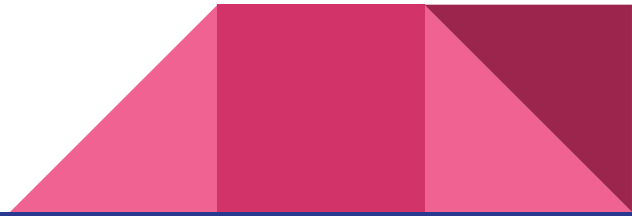




Two-tier producer-consumer system

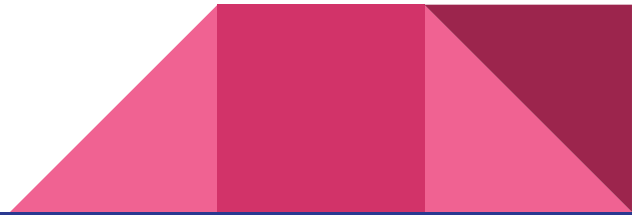
Technical Approach - Sequencer

- Single-threaded
- Batch transactions into epochs of 10ms (like Calvin)
- Places epochs in the epoch queue which can be picked up by scheduler



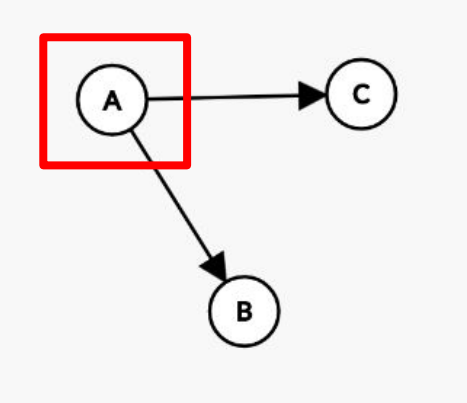
Technical Approach - Scheduler

- Single-threaded
- Picks up epochs from the epoch queue
- Builds a DAG of dependencies using our novel algorithm - DAGER,
  - Can handle both exclusive locks and shared locks, similar to Lock Manager B
- Pushes all 0-indegree nodes (txns w/o dependencies) into the worker queue
  - Tracking indegree is done as we build DAG, no traversal of the graph required



Example DAG

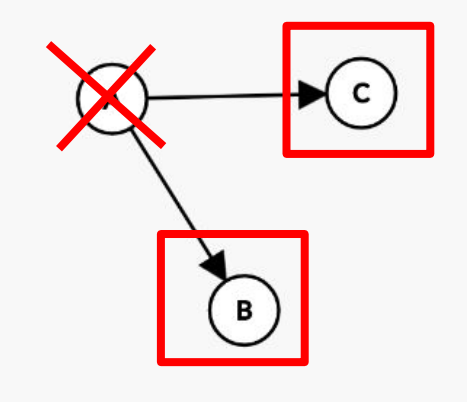
Txn	read_set	write_set
A	{}	{1}
B	{1}	{}
C	{1}	{}




```
{A: {B, C}, B: set(), C: set()}
{A: 0, B: 1, C: 1}
```

Example DAG

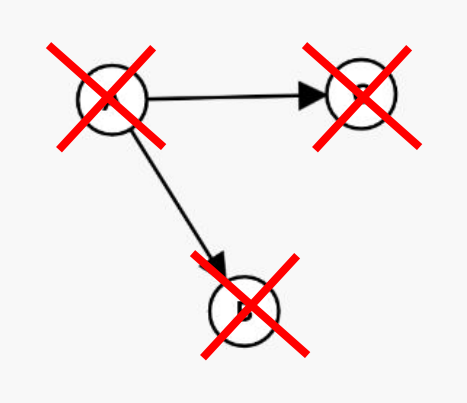
Txn	read_set	write_set
A	{}	{1}
B	{1}	{}
C	{1}	{}




```
{A: {B, C}, B: set(), C: set()}
{A: 0, B: 0, C: 0}
```

Example DAG

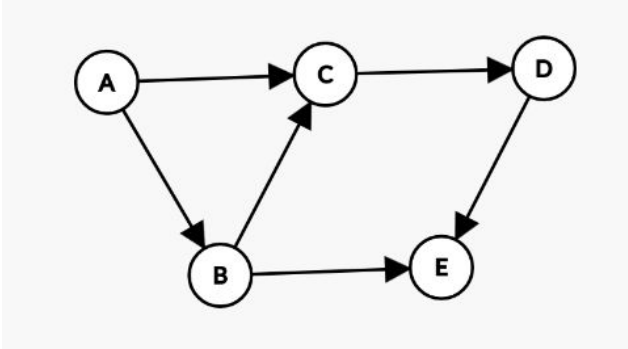
Txn	read_set	write_set
A	{}	{1}
B	{1}	{}
C	{1}	{}



```
{A: {B, C}, B: set(), C: set()}
{A: 0, B: 0, C: 0}
```

Example DAG

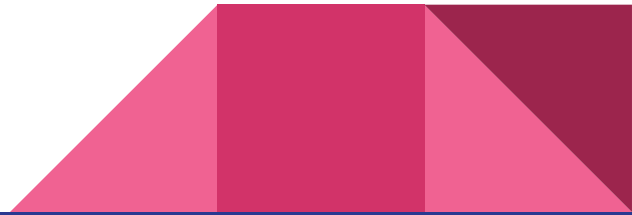
Txn	read_set	write_set
A	{1, 2}	{3, 4}
B	{3}	{5}
C	{4, 5}	{1, 2}
D	{1}	{2, 5}
E	{4, 5}	{1, 3}




```
{A: {B, C}, B: {C, E}, C: {D}, D: {E}, E: set()}
{A: 0, B: 1, C: 2, D: 1, E: 2}
```

Technical Approach - Worker

- Each worker continuously polls the worker queue and picks up work
  - These are the 0 in-degree nodes – txns with no dependencies that can now be completed
- Each worker is now responsible for committing/aborting txn (instead of scheduler)
- After txn is finished, worker updates in-degree for each neighbor and places it in the queue if in-degree is zero
  - i.e. after processing current txn, another txn is free of dependencies



Other Optimizations

- Performance comparison for epoch-based vs continuously-updating DAG
- Using a lockless atomic queue instead of provided implementation
- Using a local queue per worker instead of a global queue
  - Global queue: contention for access to one queue
  - Queue-per-worker: no contention… but how to fix load imbalance?
- Work stealing
  - Worker adds tasks to the back of its queue
  - Worker processes tasks from the front of its queue
  - But any other workers, if they run out of tasks, can steal work from the back of any other workers' queues