# $k$-LSM

## A Relaxed Lock-Free Priority Queue

Jakob Gruber

February 26, 2016

# Introduction

## The $k$-LSM

- Concurrent priority queue by Wimmer
- *Lock-free*: Progress condition. At least one thread makes progress at any time.
- *Linearizable*: Safety condition. Operations appear to take effect at one instant in time.
- *Relaxed*: Deletions may return one of $kP$ smallest elements ($k$ is configurable, $P$ is the thread count).

# Introduction

## Motivation

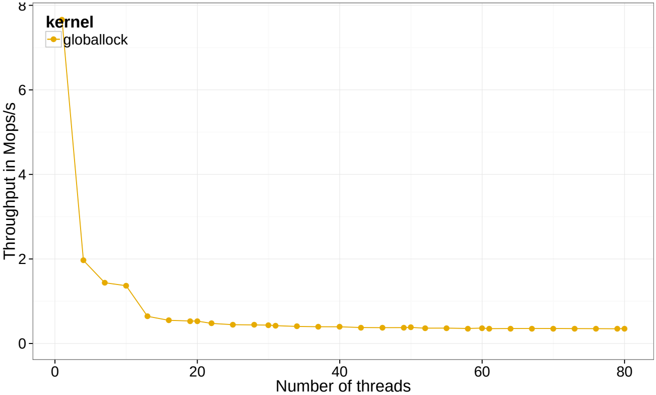



Figure: Throughput on `mars`, uniform workload, uniform key generation.

# Introduction

## Motivation

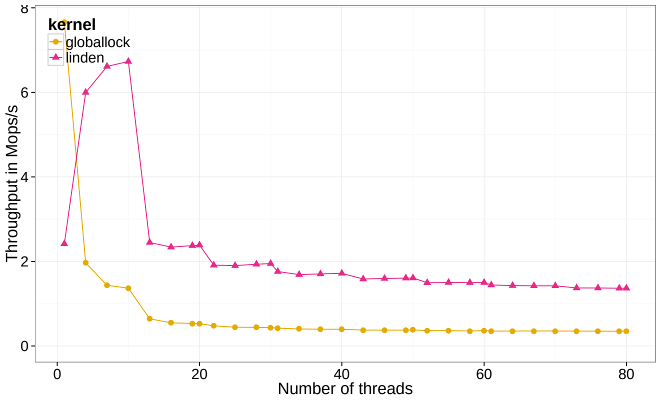



Figure: Throughput on `mars`, uniform workload, uniform key generation.

# Introduction

## Motivation

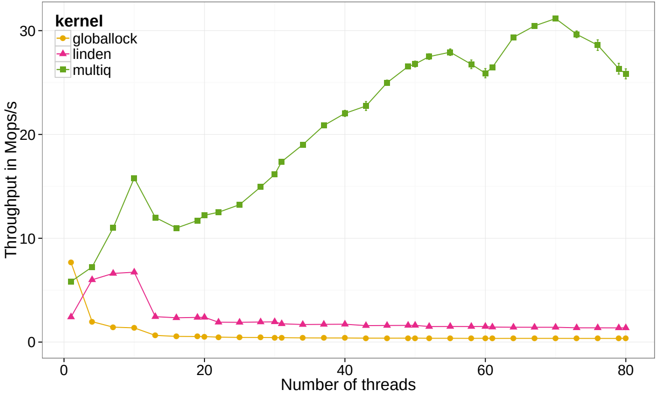



Figure: Throughput on `mars`, uniform workload, uniform key generation.

# Introduction

## Motivation

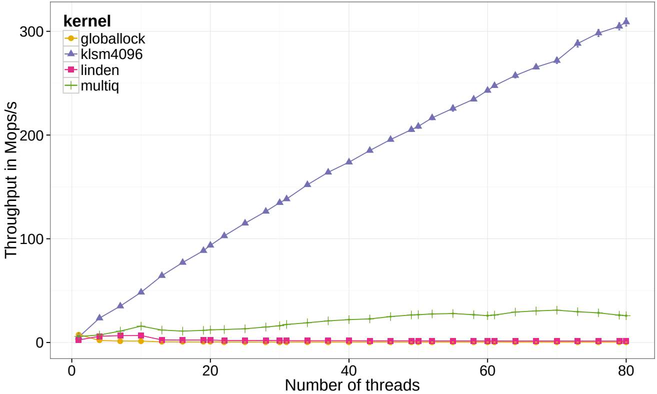



Figure: Throughput on `mars`, uniform workload, uniform key generation.

# Introduction

## In this thesis

Previously, $k$-LSM only part of task-scheduling framework *Pheet*.

- Standalone reimplementation (C++11, 4800 SLOC).
- Extensive evaluation against related structures on multiple machines.
- Different benchmarks demonstrate behavioral changes.
- Semantic quality benchmarks show results are within theoretical bounds.

# Introduction

## Contents

- Background.
- Related work.
- The *k*-LSM design.
- Results.

Priority Queues (PQs):

- Standard abstract data structure.
- Interface consists of two $O(\log n)$ operations:

```
void insert(const K &key, const V &val);
bool delete_min(V &val);
```

- Typical backing data structures: heaps & search trees.

# Related Work

- Strict concurrent PQs
  - Hunt et al.: Fine-grained locking heap.
  - Shavit and Lotan: First SkipList-based PQ.
  - Sundell and Tsigas: First lock-free PQ.
  - Lindén and Jonsson: Minimizes contention.
  - Braginsky: List of arrays, efficient.
- Relaxed concurrent PQs
  - Alistarh et al.: SprayList.
  - Rihani, Sanders, and Dementiev: Multiqueues.
  - Wimmer et al.: k-Log-structured Merge Tree (LSM).

# Relaxed Priority Queues

- Strict PQs have inherent bottleneck at minimal element.
- Another approach is to relax semantics, i.e. instead of returning *the* minimal element, return one of the $k$ minimal elements.

# Relaxed Priority Queues

## Alistarh et al.: SprayList

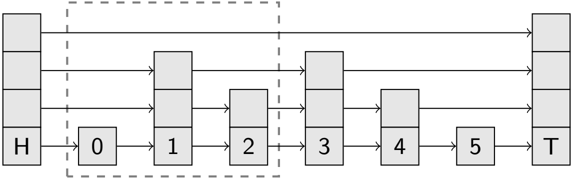



Figure: The SprayList is based on a lock-free SkipList. Deletions randomly pick one of the $O(P \log^3 P)$ smallest items.

# Relaxed Priority Queues

## Rihani, Sanders, and Dementiev: Multiqueues

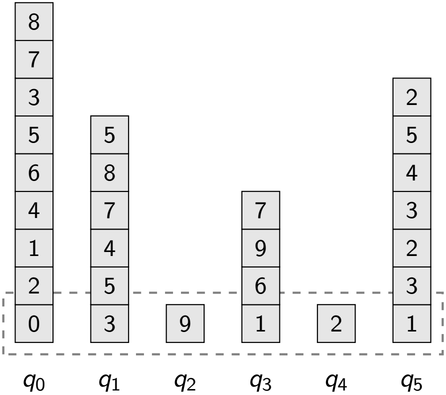
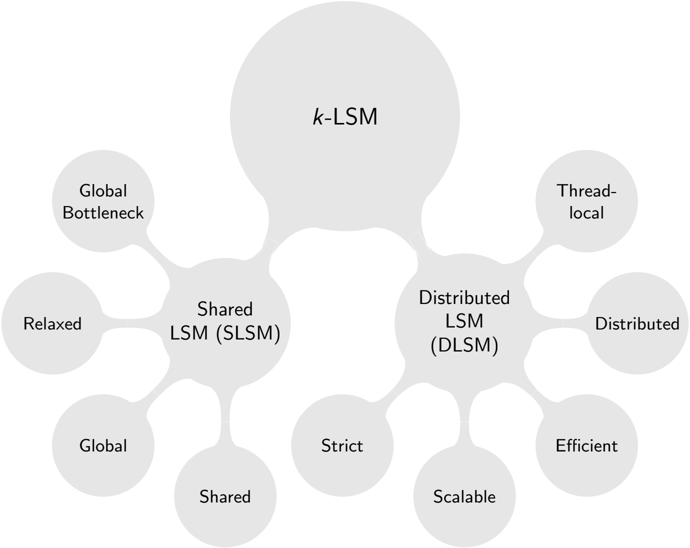



Figure: Multiqueues are a collection of priority queues. Deletions remove the minimal item from a randomly selected queue.

# *k*-LSM

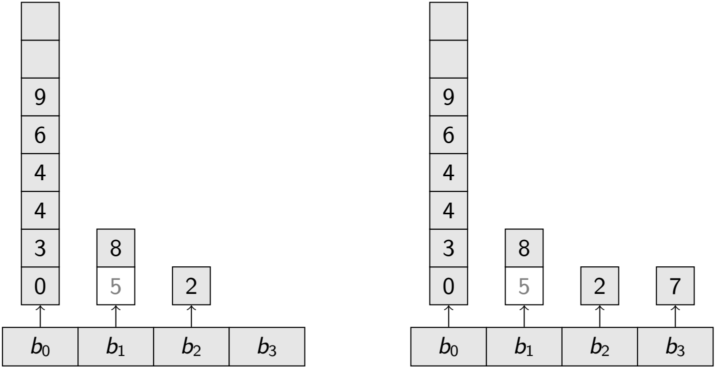


*k*-LSM
Global Bottleneck
Relaxed
Global
Shared
Shared LSM (SLSM)
Distributed LSM (DLSM)
Thread-local
Distributed
Efficient
Scalable
Strict

- Based on the concept of LSMs.
- Maintain items in a logarithmic number of sorted arrays.
- Array merges are the central operation.
- Both insertions and deletions are in $O(\log n)$.

# $k$-LSM

## LSM Insertions



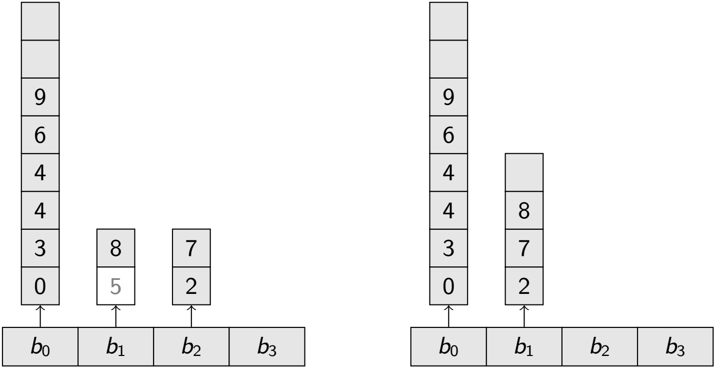

(a) The initial state.

(b) Item 7 inserted as a new block.

Figure: Insertion of a new element into the LSM.

# $k$-LSM

## LSM Insertions



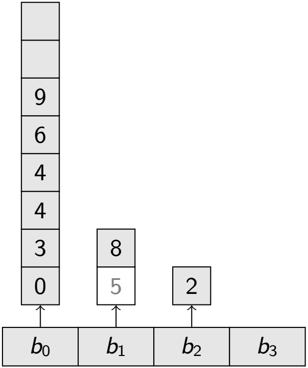

(a) After the first merge.

(b) After the second merge.

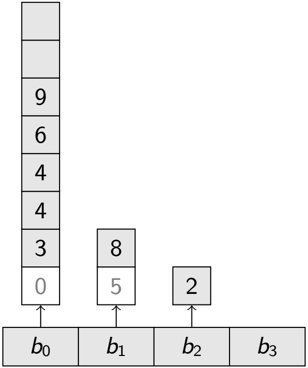
Figure: Insertion of a new element into the LSM.

# $k$-LSM

## LSM Deletions




(a) The initial state.




(b) Item 0 marked as deleted.

Figure: Deletion from the LSM.

# *k*-LSM

## DLSM

- One LSM per thread (thread-local).
- Inter-thread communication only when local LSM is empty (`spy`).

- One global LSM shared by all threads.
- Relaxation through so-called *pivot* ranges.

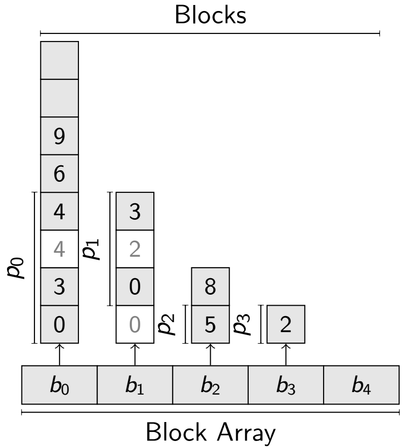



Figure: Pivot ranges contain subset of $k+1$ smallest elements.

- $k$-LSM deletions peek at DLSM and SLSM and remove the smaller item.
- Each thread-local DLSM has a capacity of $k \rightarrow$ deletions skip at most $k(P-1)$ items.
- The SLSM pivot range contains $k+1$ smallest items $\rightarrow$ deletions skip at most $k$ items.
- Combining both: at most $kP$ items skipped.

# Results

## Throughput: Uniform workload, uniform key generation

- Most commonly used benchmark.
- Throughput: Number of operations per second.
- Each thread: 50% insertions, 50% deletions.
- Keys taken at random from 32-bit integer range.

# Results

## Throughput: Uniform workload, uniform key generation

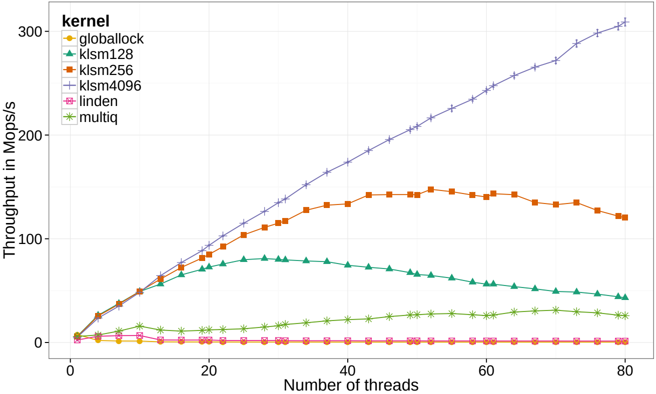



Figure: `mars` (80 core Intel Xeon at 2 GHz)

# Results

## Throughput: Uniform workload, ascending keygeneration

- Keys taken from $[0, 512[+t$.
- Induces FIFO-like behavior.

# Results

## Throughput: Uniform workload, ascending keygeneration

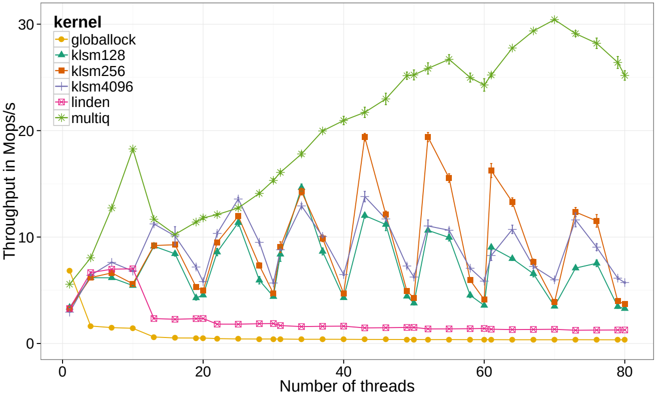



Figure: `mars`.

# Results

## Throughput: Split workload, uniform key generation

- 50% of threads insert, others delete.

# Results

## Throughput: Split workload, uniform key generation

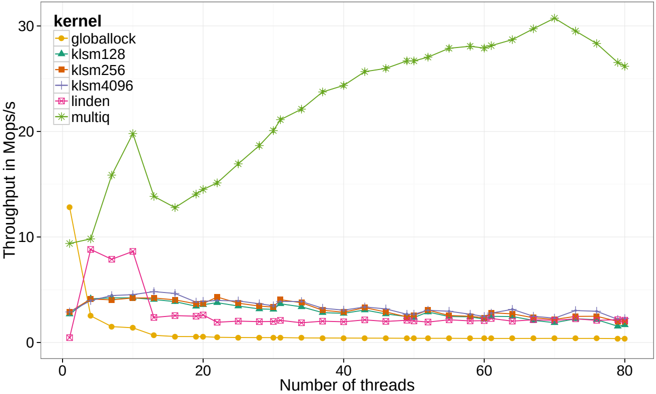



Figure: `mars`.

# Results

## Throughput: Split workload, ascending keygeneration

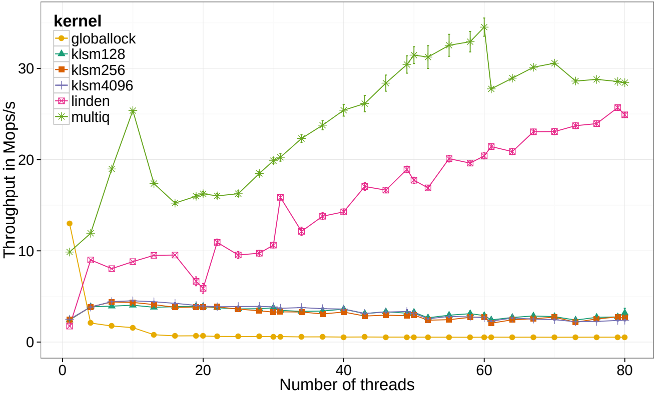



Figure: `mars`.

# Results

- Performance directly dependent on utilization of the DLSM.
- Uniform/uniform benchmark: 95% of all deletions from the DLSM.

# Results

## Quality

- And how good is the quality of results?
- Determined by rank: for each `delete_min`, if the removed item is the $k$-smallest item at that time, it has rank $k$.
- $k$-LSM results are usually within $5^{\text{th}}$ percentile of allowed relaxation.
- `multiq` approximately comparable to `klsm4096`.

# Conclusion

- Standalone reimplementation completed successfully.
- The $k$-LSM can have exceptional performance.
- ... under the right conditions.
- The standard uniform workload, uniform key generation benchmark is not enough to evaluate performance.
- Very good quality results in practice.
- Multiqueues have lower peak performance, but have stable behavior in all experiments.

# Conclusion

Questions?

# Results

## Throughput: Uniform workload, uniform key generation

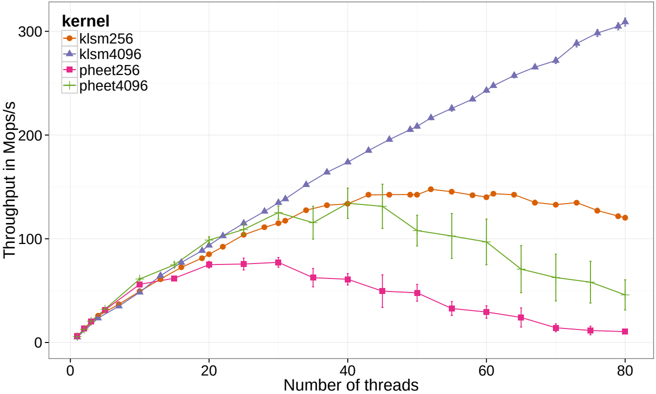



Figure: `mars`, comparison against the Pheet *k*-LSM

# Results

## Throughput: Uniform workload, uniform key generation

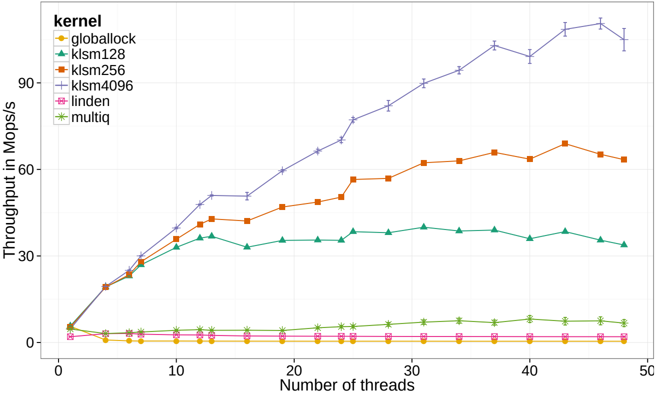



Figure: `saturn`

# Results

## Throughput: Uniform workload, uniform key generation

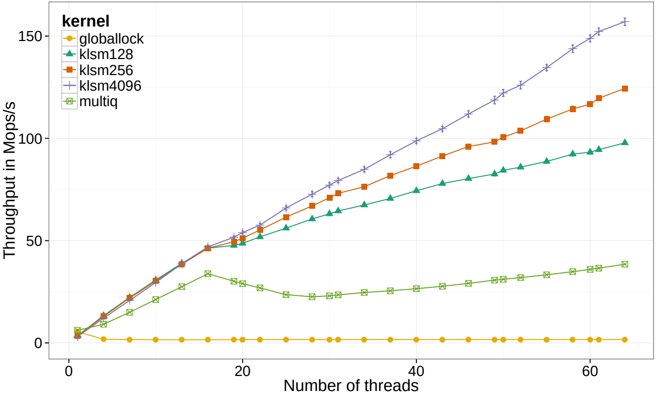



Figure: `ceres`

# Results

## Quality: Uniform workload, uniform key generation

| | 20 threads | | 40 threads | | 80 threads | |
|---|---|---|---|---|---|---|
| | Mean | SD | Mean | SD | Mean | SD |
| klsm256 | 42 | 42 | 68 | 57 | 635 | 464 |
| klsm4096 | 422 | 729 | 1124 | 1287 | 13469 | 13980 |
| multiq | 1163 | 3607 | 2296 | 7881 | 3753 | 12856 |

Table: `mars`.