CAS CS 561
Research Project Presentation
Range Deletes in LSM-Trees

Presented by: Guanzhang Li, Kaize Shi, Shirene Cao
Mentor: Subhadeep Sarkar

Problem Statement & Objectives

Logical deletes (invalidations) harm the read performance of LSM-tree

- The actual elimination of deleted data is deferred
- CPU overhead for managing the range-delete map
- Read amplification (number of disk-reads per query)
- Cost of ensuring consistency

Our goals

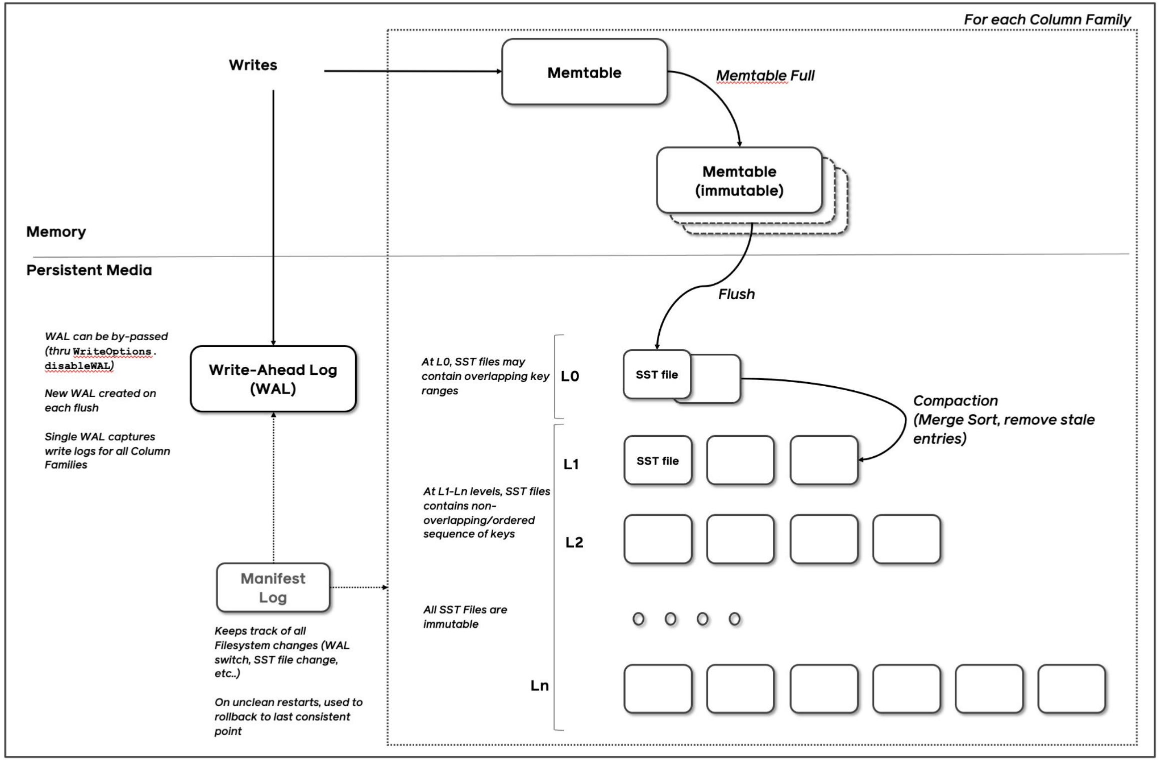
- Understand how LSM-tree works in RocksDB
- Measure the impacts of range deletes in RocksDB on read performance
    - Read throughput
    - I/Os
    - Memory footprint
    - CPU cycles

Introducing RocksDB

Persistent Key-Value store developed at Facebook based on Google’s LevelDB

Core Components

- memtable
- logfile (Write Ahead Log)
- sstfile (Sorted Strings Table)



For each Column Family

Writes

Memtable

Memtable Full

Memtable (immutable)

Memory

Persistent Media

Flush

WAL can be by-passed (thru `WriteOptions.disableWAL`)

New WAL created on each flush

Single WAL captures write logs for all Column Families

Write-Ahead Log (WAL)

At L0, SST files may contain overlapping key ranges

L0

SST file

Compaction (Merge Sort, remove stale entries)

L1

SST file

At L1-Ln levels, SST files contains non-overlapping/ordered sequence of keys

L2

Manifest Log

Keeps track of all Filesystem changes (WAL switch, SST file change, etc..)

On unclean restarts, used to rollback to last consistent point

All SST Files are immutable

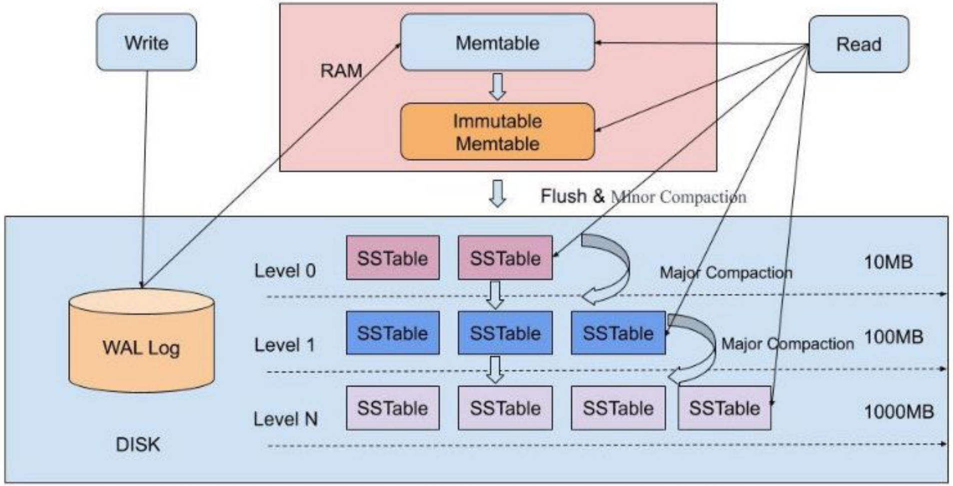
o o o o

Ln

LSM-Tree in RocksDB

- RocksDB uses leveled compaction by default, but can use a hybrid structure
  - Tiering (level 0): each level has multiple runs, sort-merge compaction triggered by threshold
  - Leveling (level 1 - *N*): each level has at most only 1 run
- When a level is full, compaction will be triggered
- Mutable buffer → immutable buffer → immutable file
- Each chunk of data is an SST file

Log Structured Merge Trees



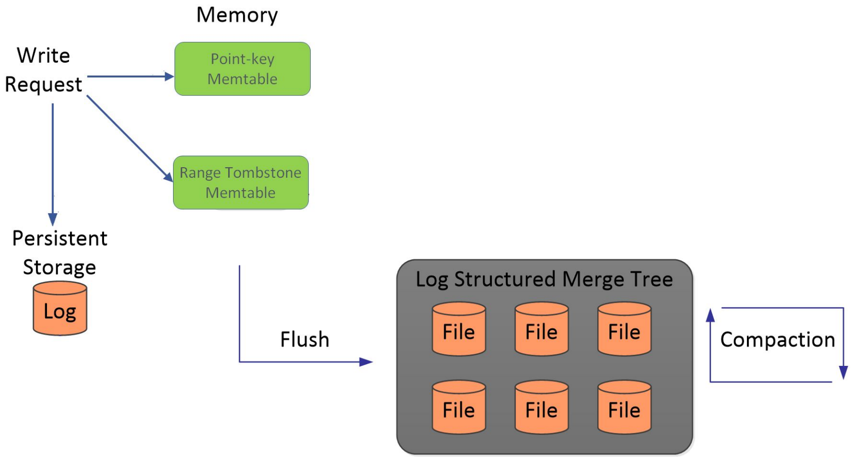
LSM-Tree Range Delete in RocksDB

Tombstones first enter the mutable buffer with timestamps

- During new operations, mutable buffer is queried first and the range tombstones are checked
- `timestamp -> (start_range, end_range)`

<span style="color:red">Skyline</span> facilitates lookups

- Merging all the range tombstones
- 2 dimensions: key range & timestamp range



Memory

Write Request

Point-key Memtable

Range Tombstone Memtable

Persistent Storage

Log

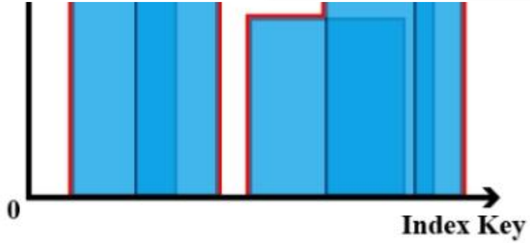
Flush

Log Structured Merge Tree

File File File

File File File

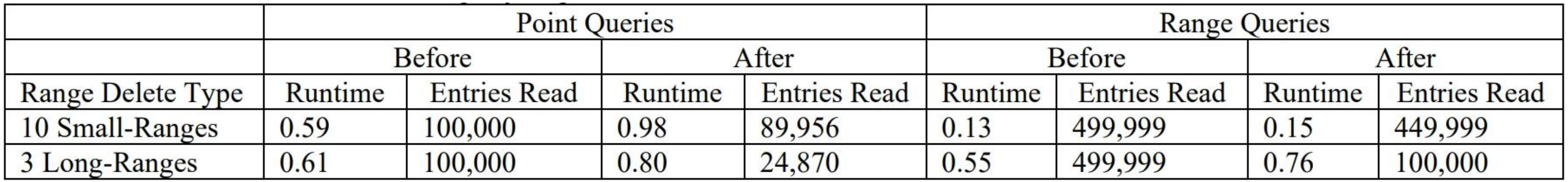
Compaction



Experiment Setup - RocksDB API

- Platform: Azure VM, Linux CentOS 7.9 Standard B2s (2 vCPUs, 4 GiB RAM)
- Range delete types: “many small-range” vs. “a few long-range”
    - 10 small-range deletes, each one invalidates 9,999 entries
    - 3 long range deletes, each one invalidates 249,999 entries
- Data:
    - 1,000,000 key-value pairs
    - Key range: from "0000000" to "0999999"
    - Values: random 500-character strings
- Point queries: 100,000 random and non-repetitive lookups
- Range queries: 499,999 keys, from "0250000" to "0749999"

Preliminary Results - RocksDB API



	Point Queries				Range Queries			
	Before		After		Before		After	
Range Delete Type	Runtime	Entries Read	Runtime	Entries Read	Runtime	Entries Read	Runtime	Entries Read
10 Small-Ranges	0.59	100,000	0.98	89,956	0.13	499,999	0.15	449,999
3 Long-Ranges	0.61	100,000	0.80	24,870	0.55	499,999	0.76	100,000

- Read throughput: number of entries read per second
- “Many small-range”
  - Point query read throughput drops 45.8%
  - Range query read throughput drops 22.0%
- “A few long-range”
  - Point query read throughput drops 81.0%
  - Range query read throughput drops 85.5%
- The performance drop is too high

Preliminary Results - RocksDB db_bench Tools

Db_bench is the main tool used for benchmarking RocksDB performance

Set up:

RocksDB:	Version 7.1
CPU:	2 * Intel(R) Xeon(R) Platinum 8171M CPU @ 2.60GHz
Keys:	64 bytes each
Values:	512 bytes each
Entries:	2500000
Block cache:	8MB

Number of range tombstone: 2

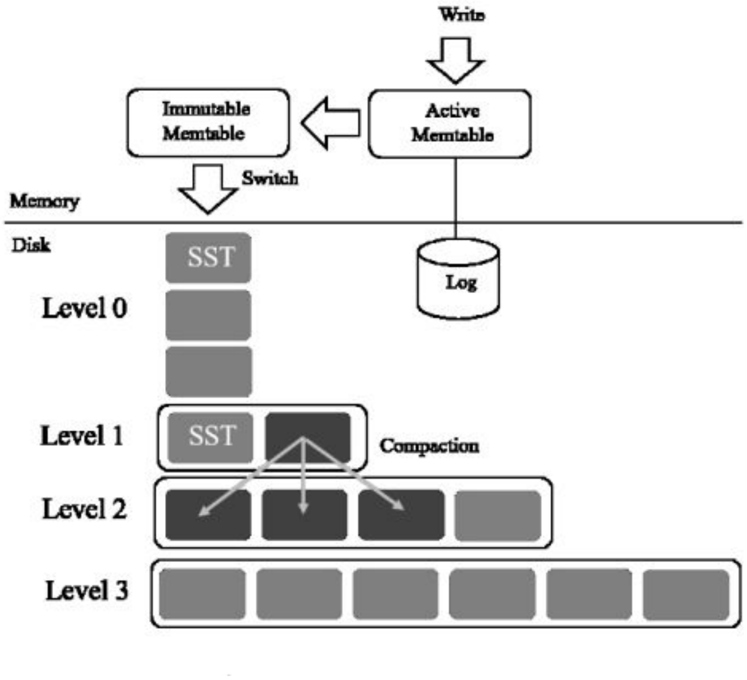
Range tombstone width: 10000

Preliminary Results - RocksDB db_bench Tools

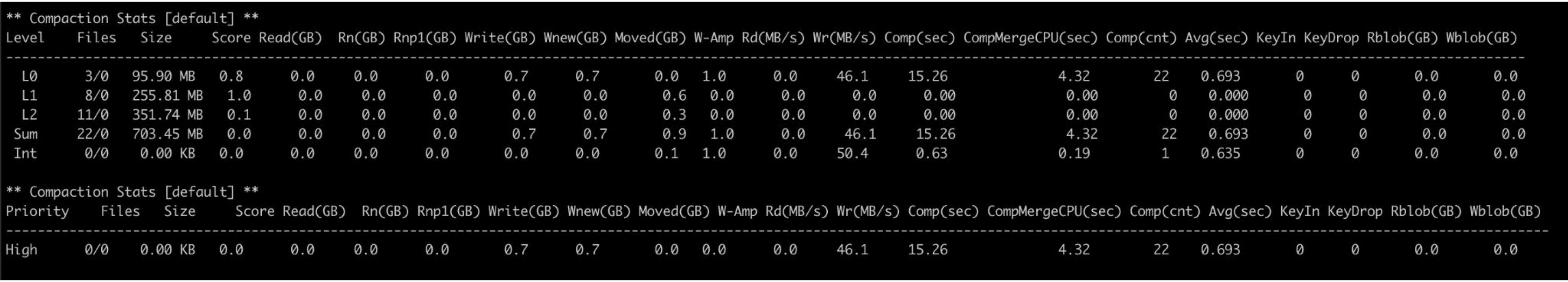
Details of the range del million data keys, and
we place the first range cond range
tombstone at a higher l

The total delete keys a

Compare the reading th delete.



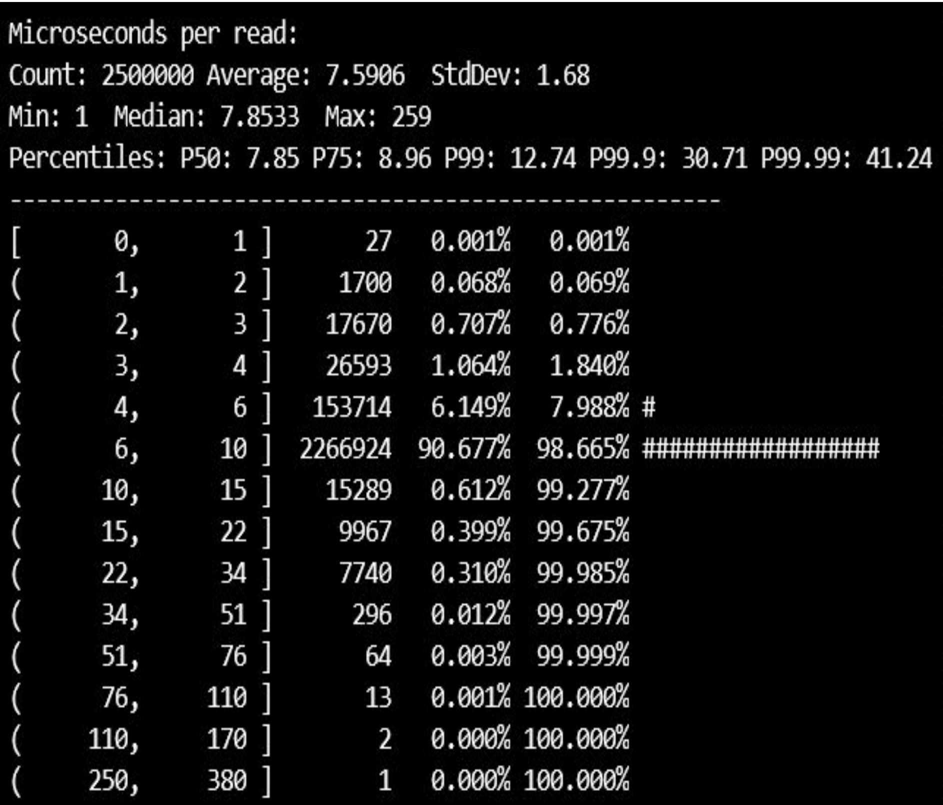
Db_bench – compaction stats



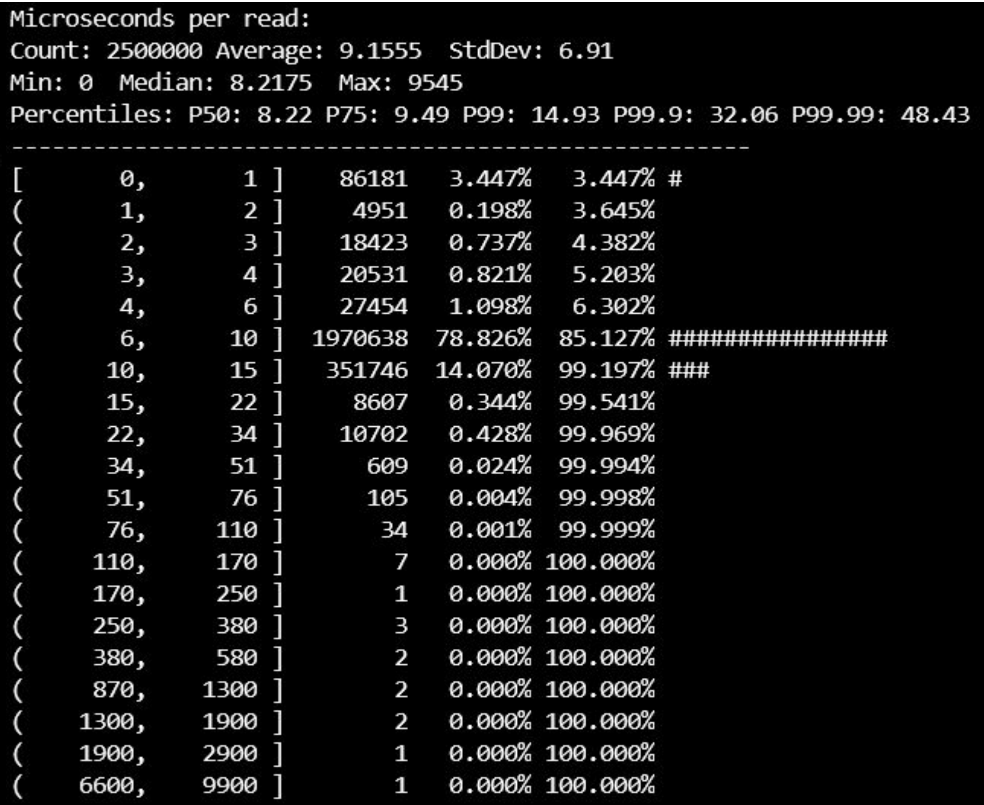
```
** Compaction Stats [default] **
Level    Files   Size     Score Read(GB)  Rn(GB) Rnp1(GB) Write(GB) Wnew(GB) Moved(GB) W-Amp Rd(MB/s) Wr(MB/s) Comp(sec) CompMergeCPU(sec) Comp(cnt) Avg(sec) KeyIn KeyDrop Rblob(GB) Wblob(GB)
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  L0      3/0   95.90 MB   0.8      0.0     0.0      0.0       0.7      0.7       0.0   1.0      0.0     46.1     15.26              4.32        22    0.693       0      0       0.0       0.0
  L1      8/0   255.81 MB   1.0      0.0     0.0      0.0       0.0      0.0       0.6   0.0      0.0      0.0      0.00              0.00         0    0.000       0      0       0.0       0.0
  L2     11/0   351.74 MB   0.1      0.0     0.0      0.0       0.0      0.0       0.3   0.0      0.0      0.0      0.00              0.00         0    0.000       0      0       0.0       0.0
 Sum     22/0   703.45 MB   0.0      0.0     0.0      0.0       0.7      0.7       0.9   1.0      0.0     46.1     15.26              4.32        22    0.693       0      0       0.0       0.0
 Int      0/0    0.00 KB   0.0      0.0     0.0      0.0       0.0      0.0       0.1   1.0      0.0     50.4      0.63              0.19         1    0.635       0      0       0.0       0.0

** Compaction Stats [default] **
Priority    Files   Size     Score Read(GB)  Rn(GB) Rnp1(GB) Write(GB) Wnew(GB) Moved(GB) W-Amp Rd(MB/s) Wr(MB/s) Comp(sec) CompMergeCPU(sec) Comp(cnt) Avg(sec) KeyIn KeyDrop Rblob(GB) Wblob(GB)
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
High      0/0    0.00 KB   0.0      0.0     0.0      0.0       0.7      0.7       0.0   0.0      0.0     46.1     15.26              4.32        22    0.693       0      0       0.0       0.0
```

Score: for levels other than L0 the score is (current level size) / (max level size).

Preliminary Results - db_bench Test



```
Microseconds per read:
Count: 2500000 Average: 7.5906  StdDev: 1.68
Min: 1  Median: 7.8533  Max: 259
Percentiles: P50: 7.85 P75: 8.96 P99: 12.74 P99.9: 30.71 P99.99: 41.24
------------------------------------------------------
[       0,       1 ]       27   0.001%   0.001%
(       1,       2 ]     1700   0.068%   0.069%
(       2,       3 ]    17670   0.707%   0.776%
(       3,       4 ]    26593   1.064%   1.840%
(       4,       6 ]   153714   6.149%   7.988% #
(       6,      10 ]  2266924  90.677%  98.665% ##################
(      10,      15 ]    15289   0.612%  99.277%
(      15,      22 ]     9967   0.399%  99.675%
(      22,      34 ]     7740   0.310%  99.985%
(      34,      51 ]      296   0.012%  99.997%
(      51,      76 ]       64   0.003%  99.999%
(      76,     110 ]       13   0.001% 100.000%
(     110,     170 ]        2   0.000% 100.000%
(     250,     380 ]        1   0.000% 100.000%
```



```
Microseconds per read:
Count: 2500000 Average: 9.1555  StdDev: 6.91
Min: 0  Median: 8.2175  Max: 9545
Percentiles: P50: 8.22 P75: 9.49 P99: 14.93 P99.9: 32.06 P99.99: 48.43
------------------------------------------------------
[       0,       1 ]    86181   3.447%   3.447% #
(       1,       2 ]     4951   0.198%   3.645%
(       2,       3 ]    18423   0.737%   4.382%
(       3,       4 ]    20531   0.821%   5.203%
(       4,       6 ]    27454   1.098%   6.302%
(       6,      10 ]  1970638  78.826%  85.127% ################
(      10,      15 ]   351746  14.070%  99.197% ###
(      15,      22 ]     8607   0.344%  99.541%
(      22,      34 ]    10702   0.428%  99.969%
(      34,      51 ]      609   0.024%  99.994%
(      51,      76 ]      105   0.004%  99.998%
(      76,     110 ]       34   0.001%  99.999%
(     110,     170 ]        7   0.000% 100.000%
(     170,     250 ]        1   0.000% 100.000%
(     250,     380 ]        3   0.000% 100.000%
(     380,     580 ]        2   0.000% 100.000%
(     870,    1300 ]        2   0.000% 100.000%
(    1300,    1900 ]        2   0.000% 100.000%
(    1900,    2900 ]        1   0.000% 100.000%
(    6600,    9900 ]        1   0.000% 100.000%
```

Preliminary Results - db_bench Point Query

Time taken for one operation(random reading / point query) 10 times average:

Before range delete: 7.6298 micro sec / operation (2500000 of 2500000 found)

After range delete: 8.9799 micro sec / operation (2299811 of 2500000 found)

Performance dropped by 17%.

Conclusion & Future

- Preliminary observations
  - Ranges deletes indeed have significant damage to read performance
  - “A few long-range” is worse than “many small-range”
- On-going
  - More rigorous controlled conditions & more experiments
  - Better workload generator
  - Debugging the RocksDB API experiment code
  - Finding metrics for I/O, memory footprint (sizes of tombstones)
  - Comparing the utilities of RocksDB API & db_bench

Lessons Learned & Challenges

- Lessons
  - A better understanding on LSM-tree
  - Always have a plan B in case of emergency
  - Start EARLY
- Challenges
  - Compiling and getting started
  - Finding the correct metrics & functions
  - Programming in C++
  - Using db_bench