25 minutes

CS 736: Thursday, February 8, 2018 – SSD-Conscious Storage

Da Zheng and Disa Mhembere and Randal Burns and Joshua Vogelstein and Carey E. Priebe and Alexander S. Szalay, FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs Conference on File and Storage Technologies (FAST 2015)

1. What does it mean for a graph-processing engine to be semi-external?

- Meets (or exceeds) performance of in-memory engines and allows graph problems to scale beyond pure in-memory solutions

Definition:
- Enough memory to fit nodes/vertices in memory (edges in storage)

2. How does FlashGraph describe previous work of GraphChi and X-stream? Do you agree with their descriptions?

"specifically designed for magnetic disks"

Both run experiments on SSDs (and RAM for X-stream) → Don't agree.

But, true that previous work had overly simplistic model of what is needed for good SSD perf.

3. What are the four goals of the FlashGraph design?
   Do they show that they meet these goals?

Reduce I/O.
- compact data structures
- maximize hit rates
- selective data access

Perform sequential I/O when possible.

Overlap I/O + computation.

Minimize wearout (by minimizing writes)

Any experiments for wearout?
Not emphasized.

Experiments showing impact of
- seq ops
- cpu utilization

Could have more w/ I/O amounts

4. Do you find anything interesting about the programming model of FlashGraph (compared to previous systems)? Does it contain any optimizations or features that seem useful?

- Only work for active nodes
- Vertex must explicitly ask for edge list (not always needed)
- Separate in + out edge lists (often need only one)
- multicast-
- Can communicate with ~~et~~ non-neighbor vertices + get their edge lists (useful when topology changes)

Application knowledge is useful!

5. What data structures does FlashGraph keep in memory (instead of SSD)?
How does FlashGraph minimize its memory usage?

Vertices - User-defined vertex state
Vertex status

Graph index to access edge lists (on SSD)

Msg queues

Optimizations:
- Calculate edge list locations instead of storing locations
- 8 bits - 255 degree (undirected) edges

Fig 5 - If ≥ 255 (need more than 8 bits) store degree in hash table

Don't store vertex ID, compute based on address

Should they be obsessed? Yes

6. How does FlashGraph minimize accesses to SSD?

Does not scan all of edge list like Graph-Chi + X-stream

Issue many requests in parallel

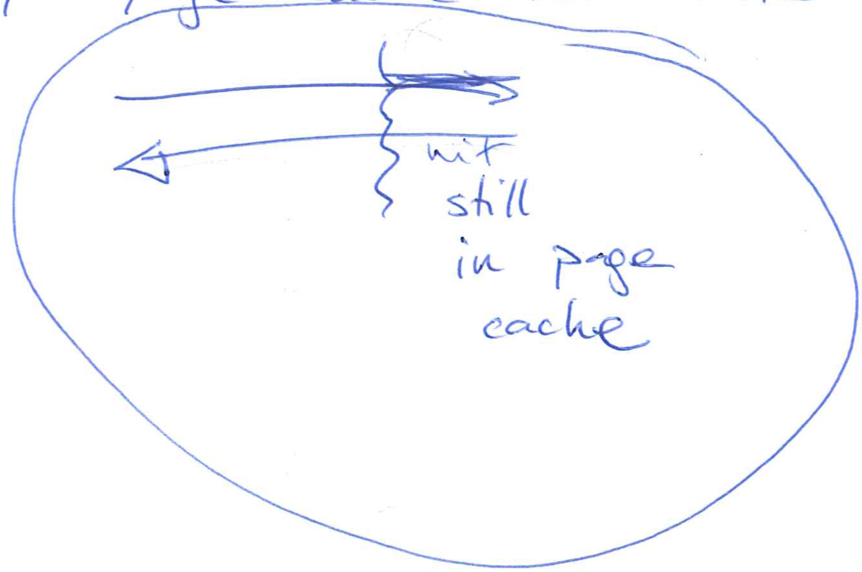
Merge I/O requests - on same page or adjacent

More merge opportunities when process in order

7. To optimize performance, in what order should vertices be scheduled? Why?

- In order (with neighboring edges) helps to merge edge requests

- Alternating order of vertices can help w/ page cache hit rate



Can split vertices w/ many edges
- helps w/ cache hit rate to work on shared edges

8. In the performance evaluation, is FlashGraph compared to reasonable configurations of GraphChi and X-stream? Is FlashGraph always better?

Why not all in-memory X-Stream?

X-stream does require significantly less memory

Maybe FlashGraph isn't really as semi-external as it could be...

- Function of app
  - Need all edges - X stream do better?
- Optimized FS

9. Are there any experiments you wish they would have shown? Metrics?

- Preprocessing overheads?

- Graphs that change over time?

- Amount of write traffic (wearout goal)

10. At a high level, how are the techniques used by FlashGraph and WiscKey similar?

~~For reads~~:

Both try to perform many parallel (random) ops ~~reads~~ instead of sequential ops that aren't fully needed

Multiple threads for prefetching + overlap

(key, value) $\approx$ (vertices, edges)

Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau,Remzi H. Arpaci-Dusseau [WiscKey: Separating Keys from Values in SSD-conscious Storage]
Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)

1. Why do LSM-trees perform well on HDDs? What different characteristics about SSDs encourage a change in LSM design?

HDDs - huge diff btwn random + seq. perf.

LSM-trees perform all sequential ops.

SSDs - not as sig. diff. between seq + rand.

2. Why do LSMs incur write amplification? Why do they incur read amplification?

Write:

- when fill one level of tree, merge w/ others at same level (compaction) and move down a level.

- as tree fills, writes performed many times

Read:

- Must search through levels to find key

- Significant metadata to help find

3. How can one achieve nearly sequential bandwidth out of an SSD for random operations?

- Many concurrent ops
  - 32 threads
  - 64KB or greater does as well as sequential

4. Why might one think that separating keys from values would hurt performance?
   Why does separating keys and values improve performance?

- Extra random read to fetch value (key, value not contiguous)

- Keeping values out of LSM-tree means levels do not fill as quickly — much lower compaction
  - less write + read amplification

- Keys are small, values are large (100B - 4KB)

5. Why is it challenging to handle range queries in WiscKey? How are they handled efficiently?

LevelDB: keys are usually sorted + contiguous w/ values

→ read many keys/values in one sequential operation

WiscKey: values are scattered

→ many random reads.

→ need to carefully prefetch values ~~when~~ after read keys

(hard for small ranges)

6. How does WiscKey determine a value is garbage and should be reclaimed? What is similar about this compared to LFS?

- Add backpointer to value in vLog. (must ~~know value~~ know key)

- Read ~~value~~ key in LSM-tree: does it point to this value?

LFS: data blocks - added backpointer to inode + offset

Live/dead by looking up inode in imap + seeing if data block points here

7. How is the performance of small put() operations improved?

- Batch many together + write out

group

8. Does WiscKey performance relative to LevelDB increase or decrease as the value size increases? Why?

- Better w/ larger values since amplification is more significant for LevelDB

- Larger values → better bw when accessing values

- Less buffering

9. As shown in the experiments, what hurts/helps the performance of range queries in WiscKey? Do you have any ideas for how to help the difficult situation?

Fig. 12

- Better bw when keys were loaded sequentially (sorted) instead of random order (small values) (leveldb doesn't do well w/ random order either)

-

Idea: For workloads w/ range queries + small values, sort keys in background

10. Are there any experiments you wish they would have shown?

More analysis of RocksDB?