Fast Clone Deletion

Sara Hartse | Delphix

Outline

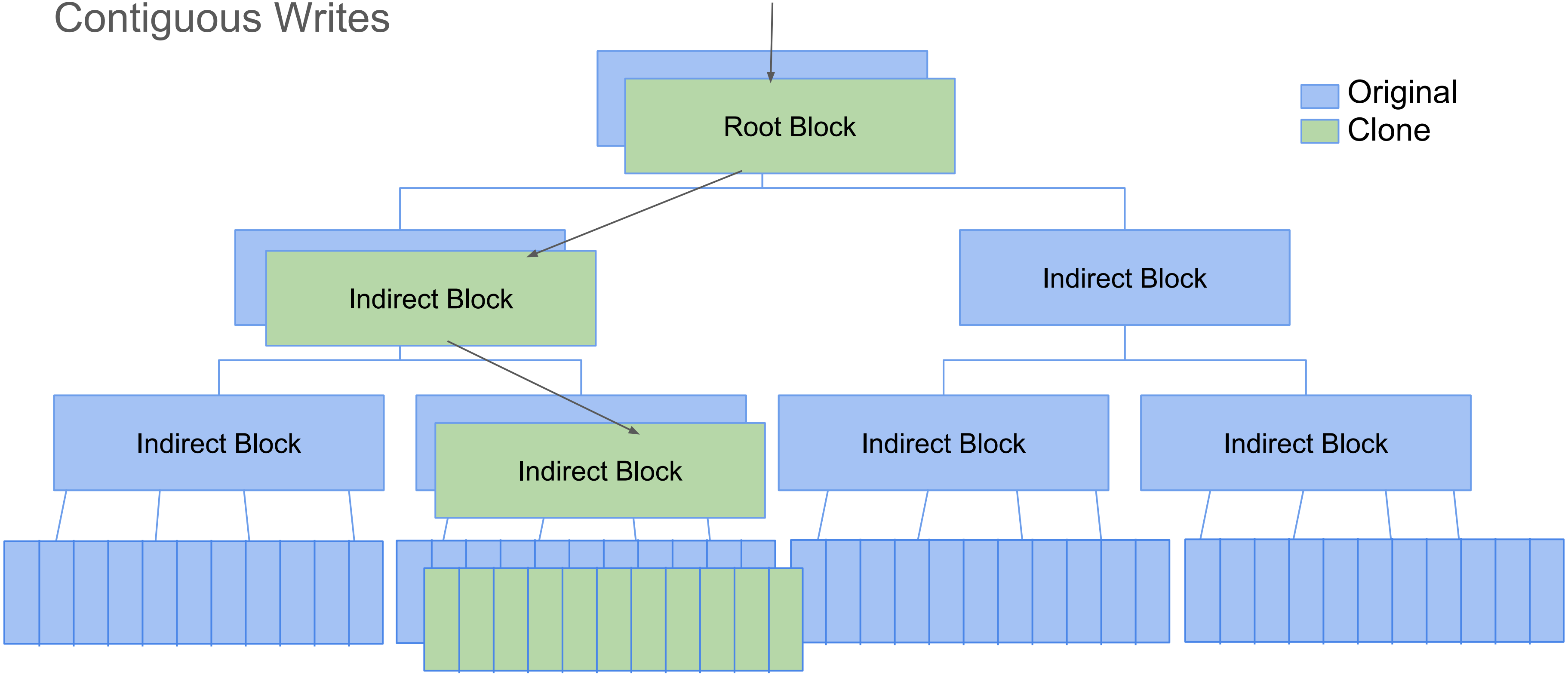
- Clone deletion now
- Fast deletion algorithm
- Algorithm scalability
- Performance gains

Clone Deletion

- Clones are mutable copies of existing datasets
- Copy on write means that **creating a clone** is as simple as pointing to the root of a given snapshot
- Throughout the course of the clone’s lifetime it diverges from the original
- **Deleting a clone** requires determining which blocks are still shared with the snapshot and which blocks are unique to the clone
    - Iterate over on-disk tree, ignore sections based on birth time

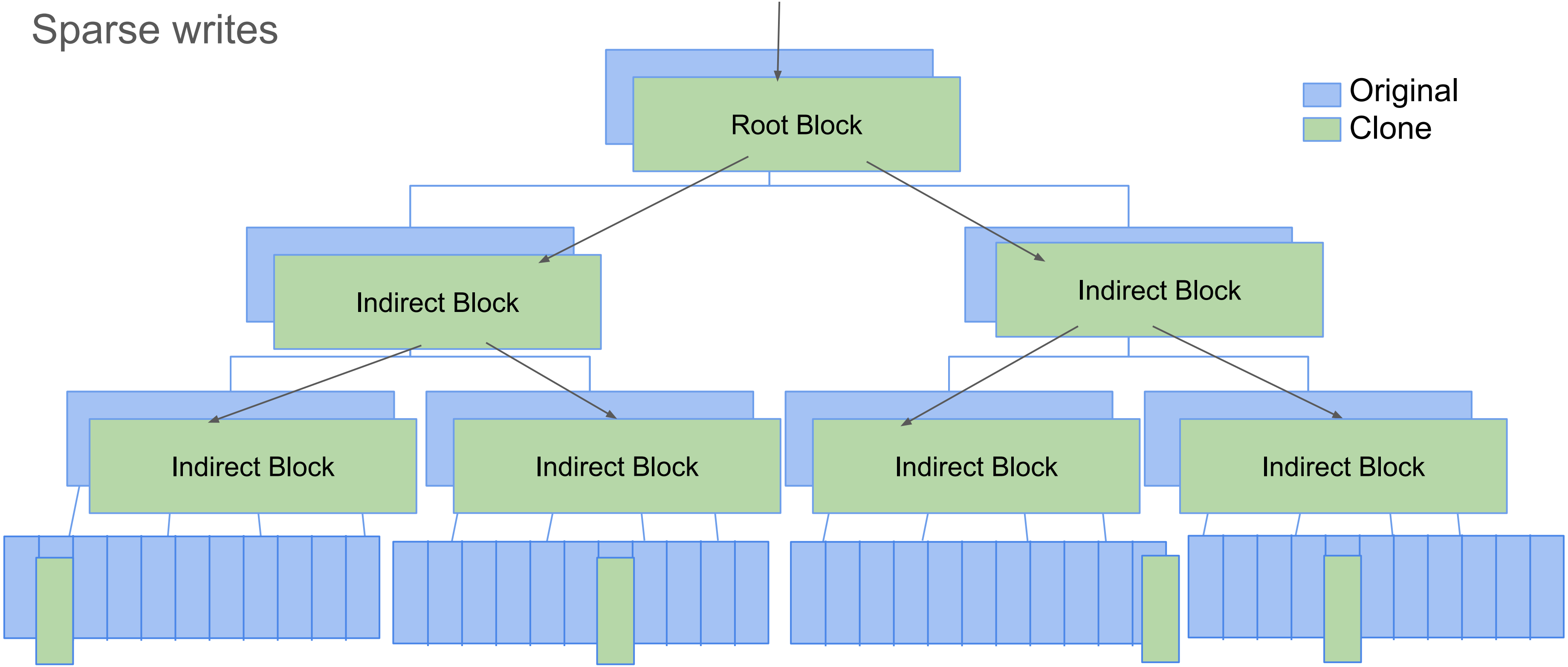
Best Case

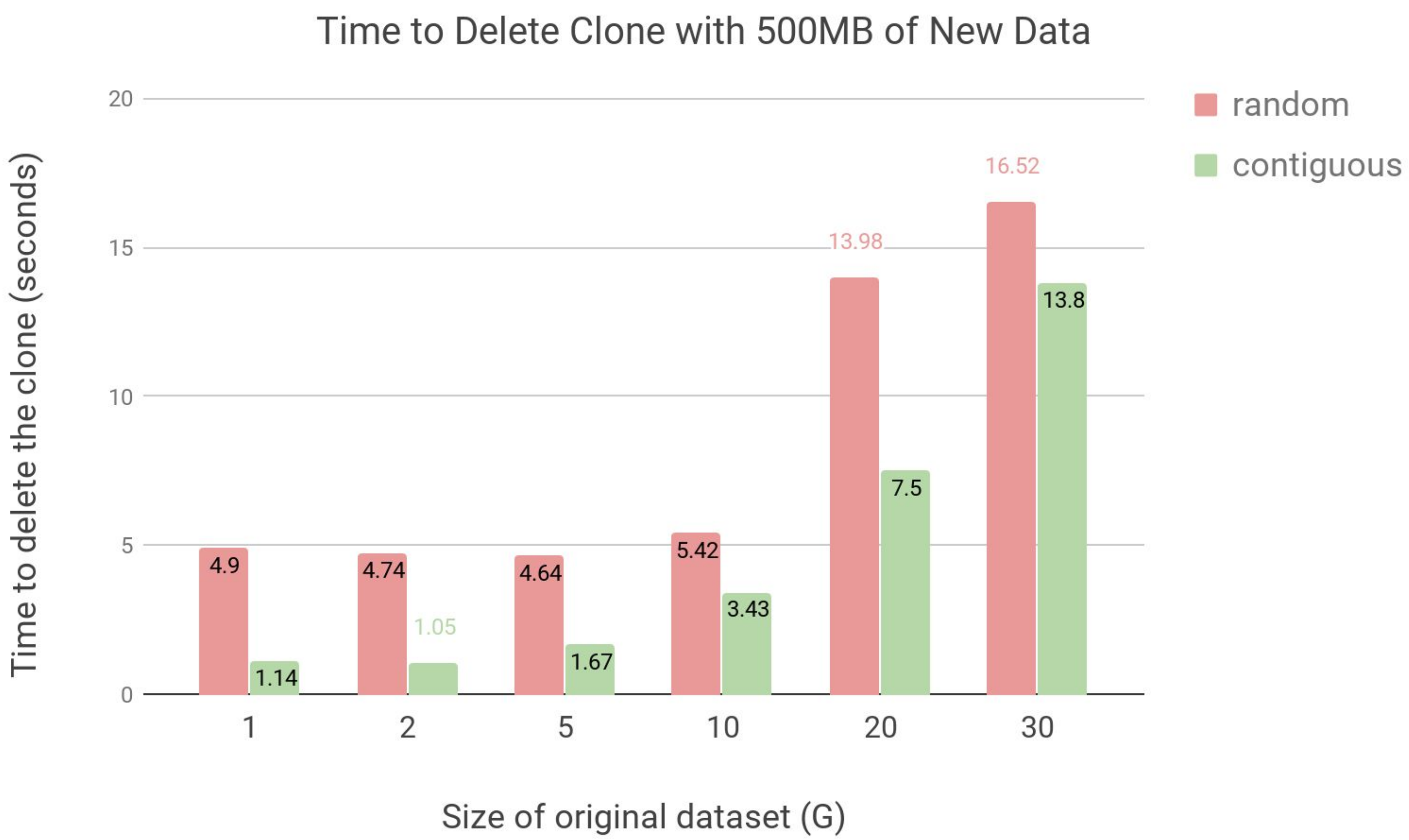
Contiguous Writes



Worst Case

Sparse writes



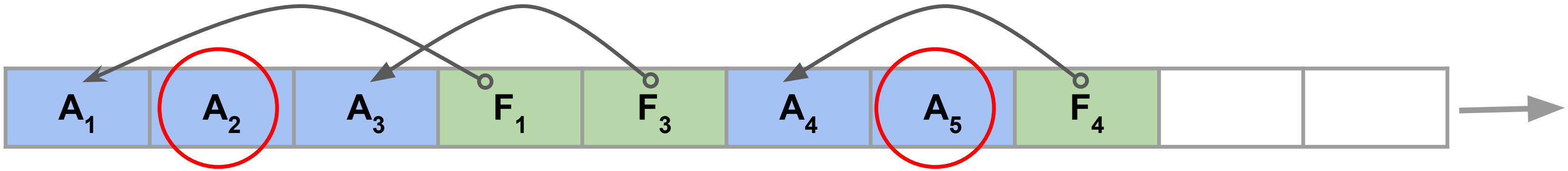

Time to Delete Clone with 500MB of New Data
random
contiguous
Time to delete the clone (seconds)
Size of original dataset (G)
0
5
10
15
20
1
2
5
10
20
30
4.9
1.14
4.74
1.05
4.64
1.67
5.42
3.43
13.98
7.5
16.52
13.8

Fast Delete

- Keep track of clone specific writes and deletes as they occur
- Store them in a **livelist**
- To delete the clone, just have to process each element in the livelist
- Work is proportional to the number of writes to the clone

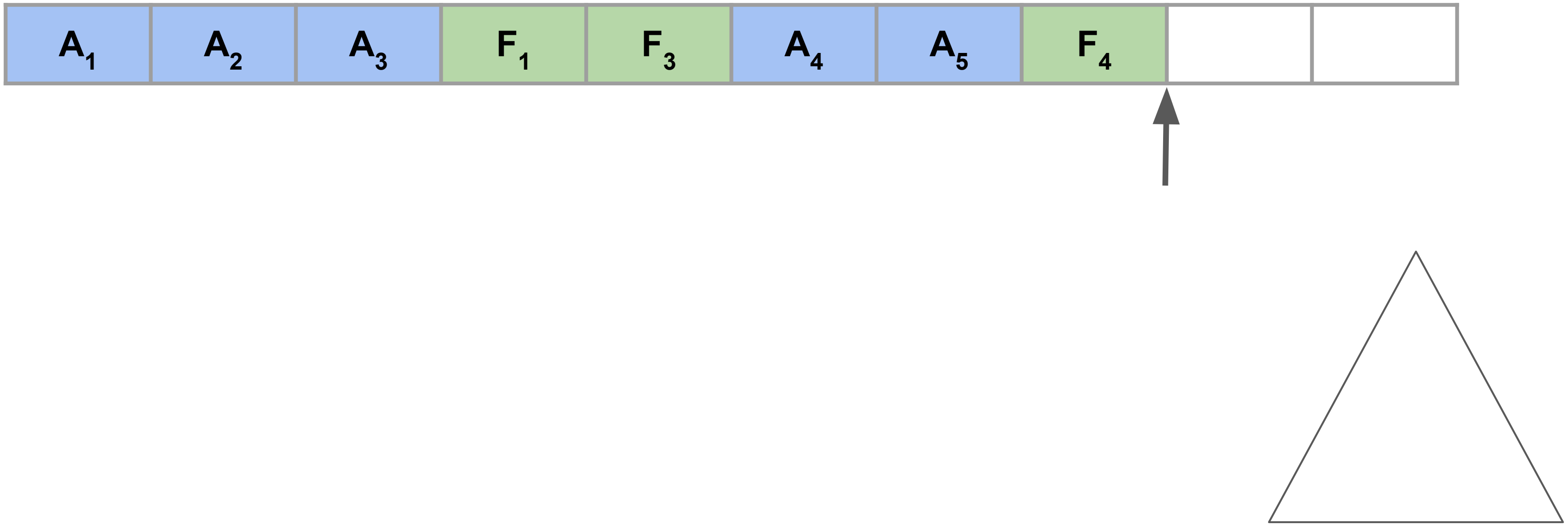
Livelist algorithm

- Enqueue blockpointers **allocated** and **freed** on the clone as the writes occur

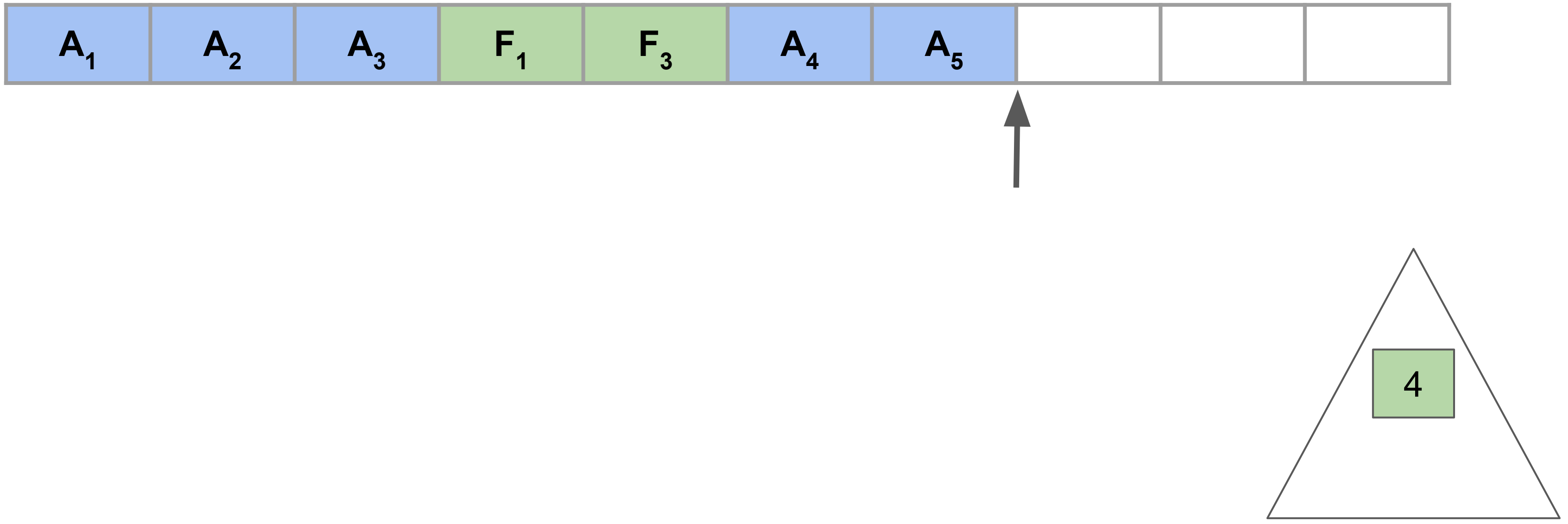


- When it’s time to delete the clone, determine the not yet freed blocks and free them
    - Step backwards through the livelist: insert frees into an AVL tree, check for membership of allocs in the AVL tree.

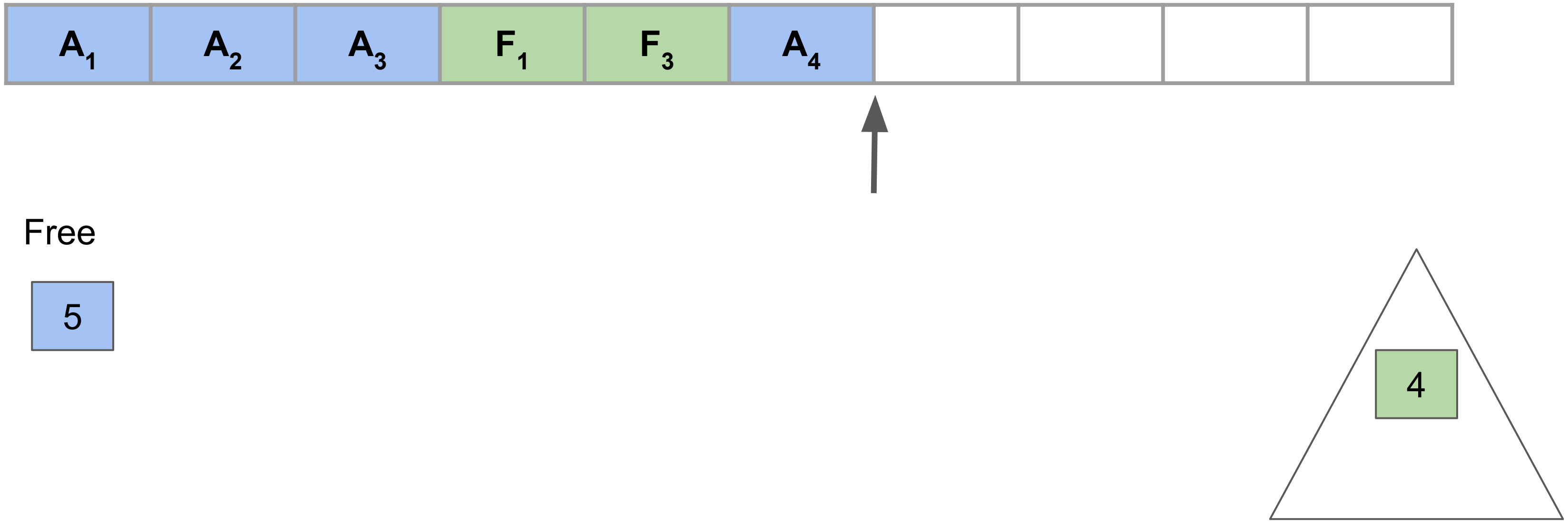
Start at the end of the Livelist



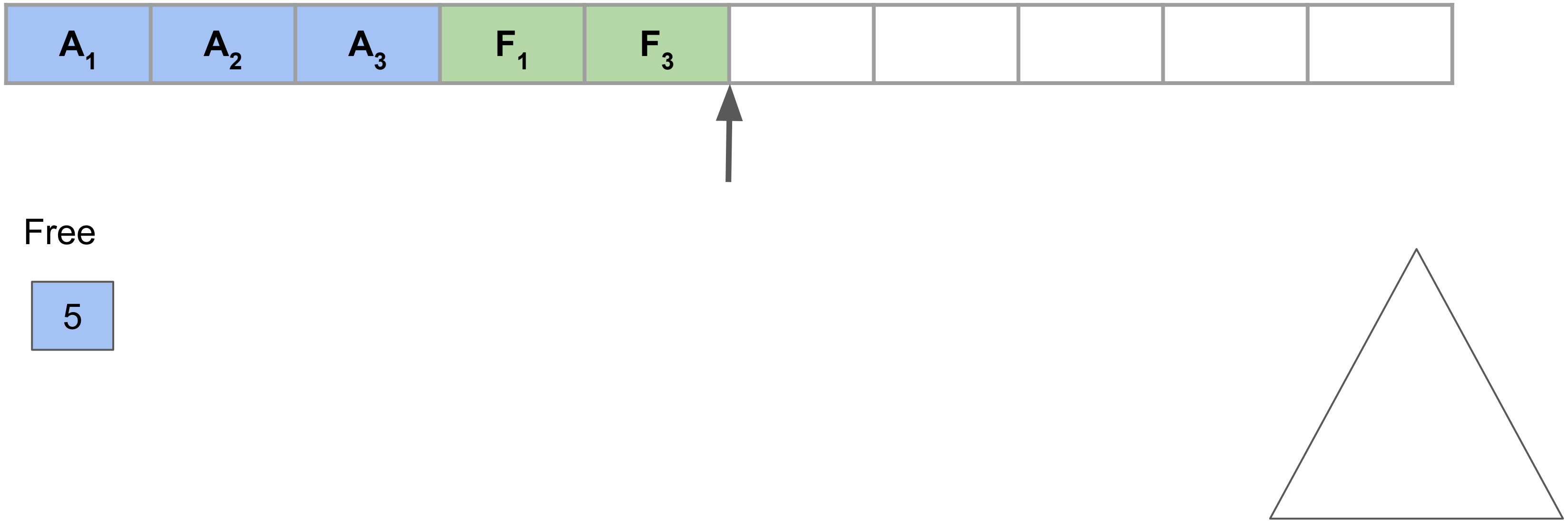
Insert block 4 in AVL tree



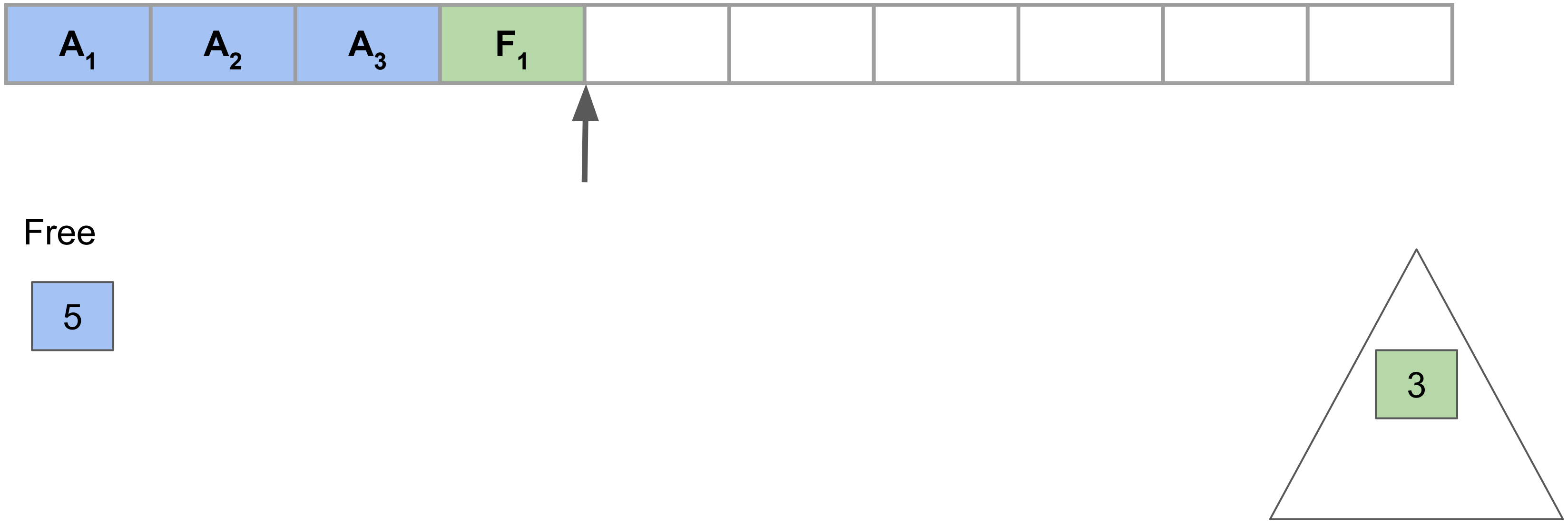
Check for block 5 in AVL tree. Free it.



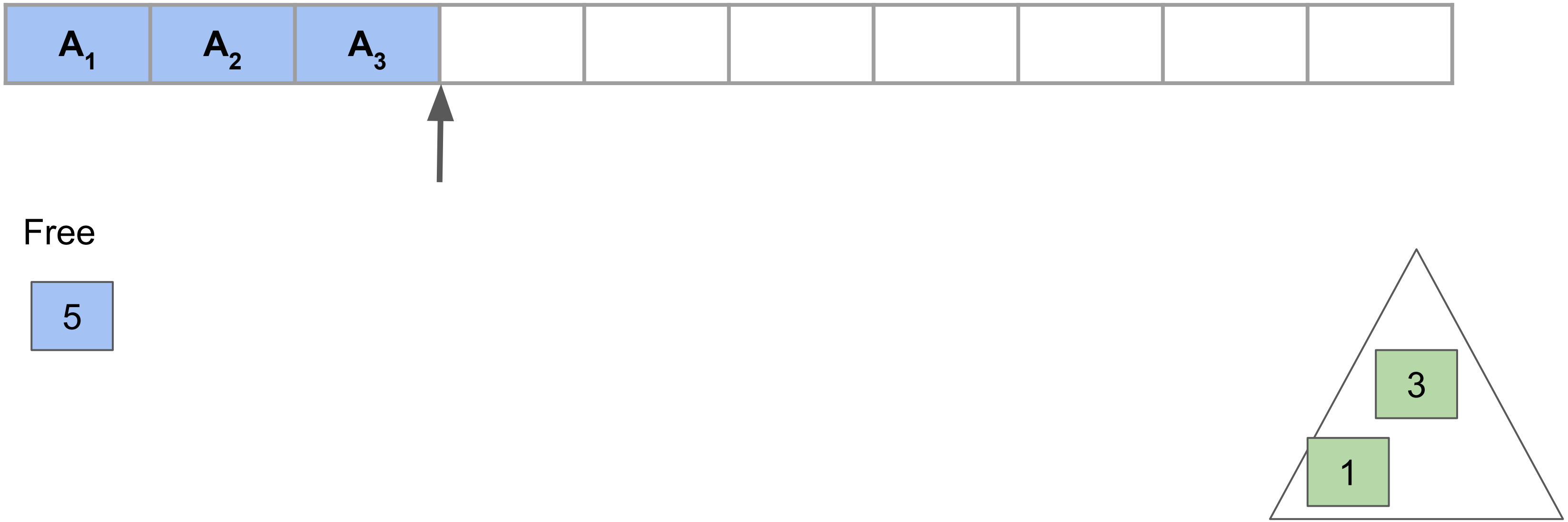
Check for block 4 in AVL tree. Ignore it.



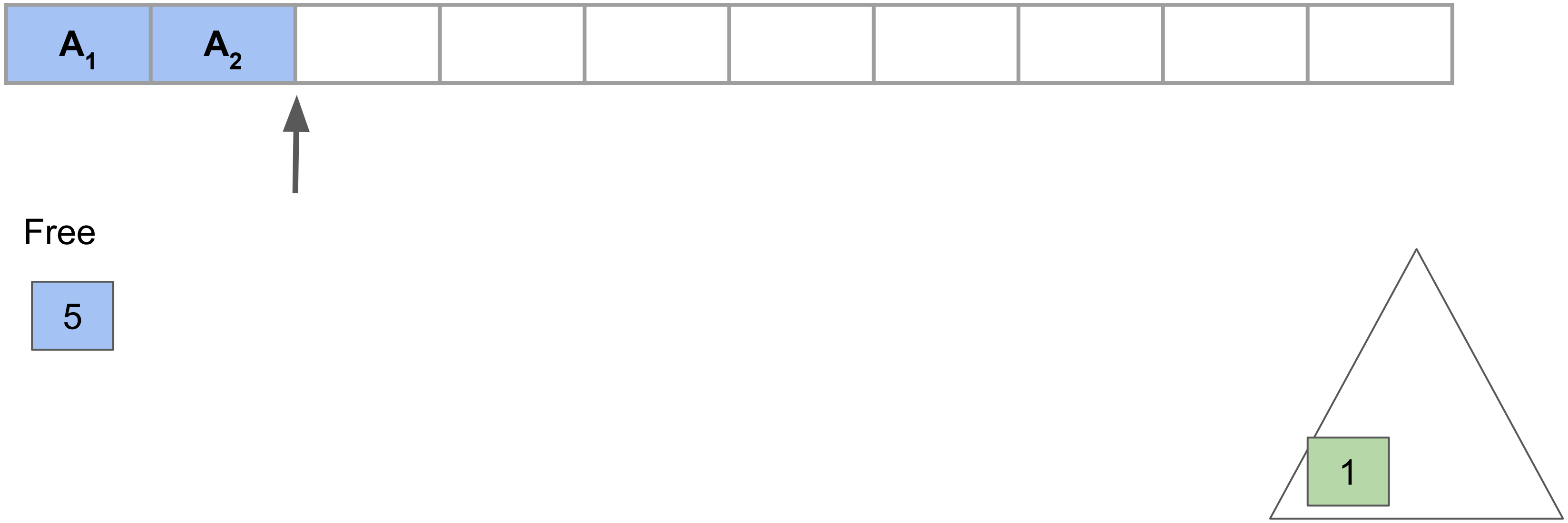
Insert block 3 into AVL tree



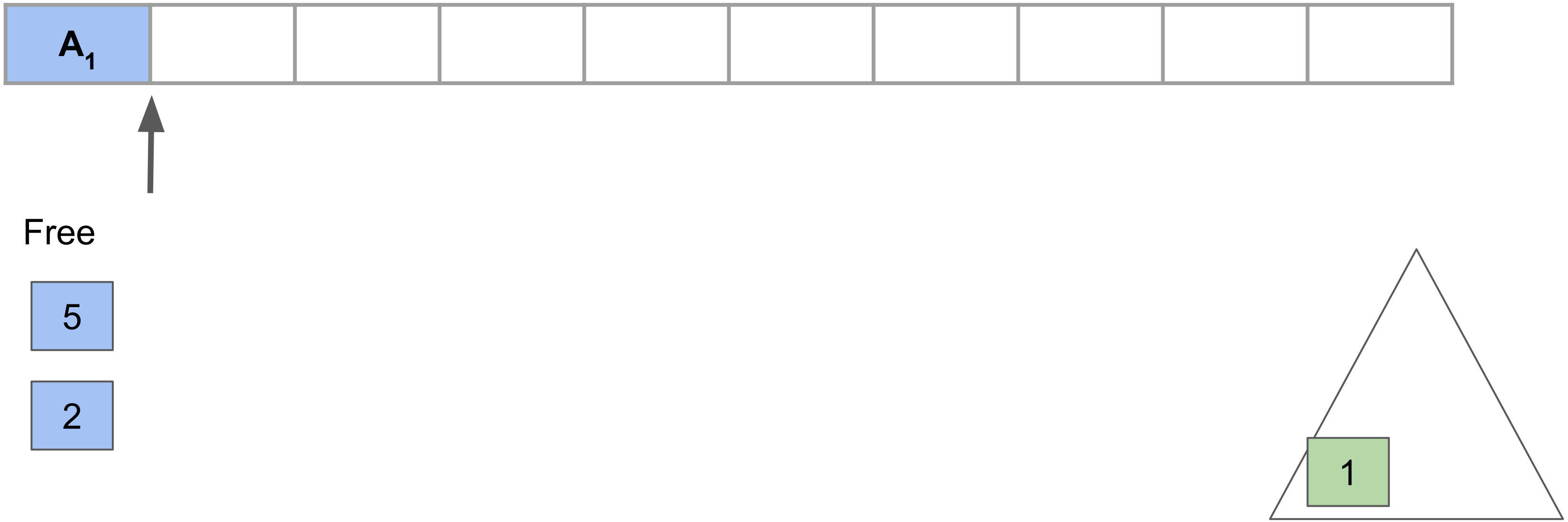
Insert block 1 into AVL tree



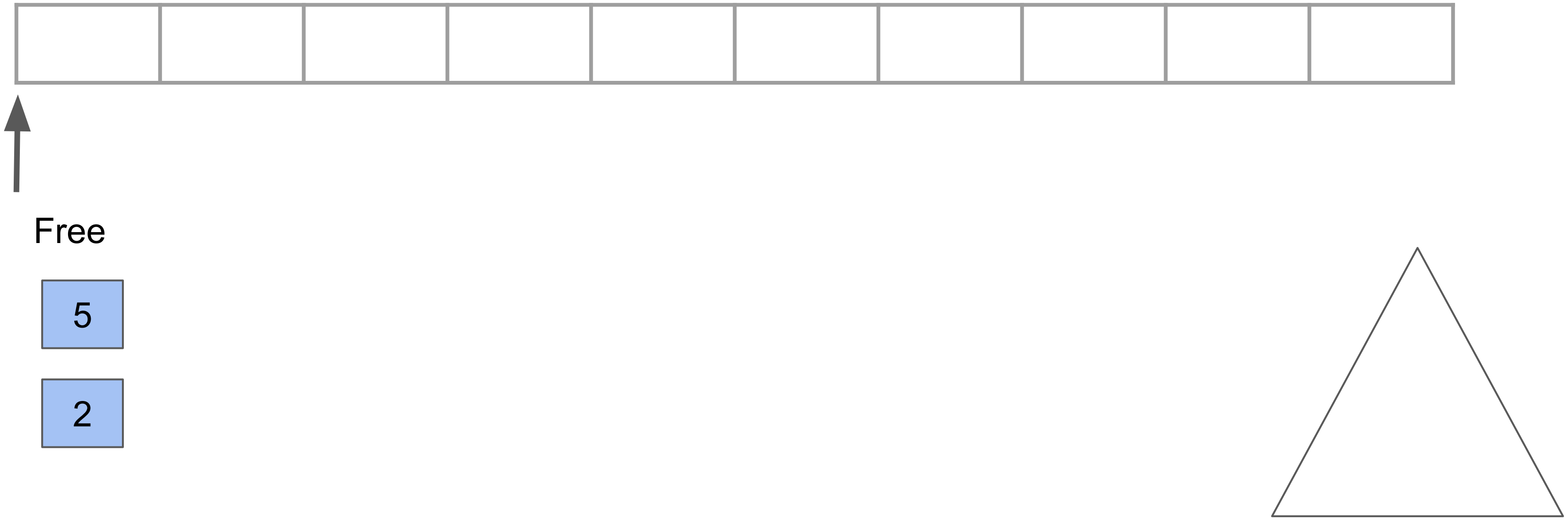
Check for block 3 in AVL tree. Ignore it.



Check for block 2 in AVL tree. Free it.



Check for block 1 in AVL tree. Ignore it.



Pros

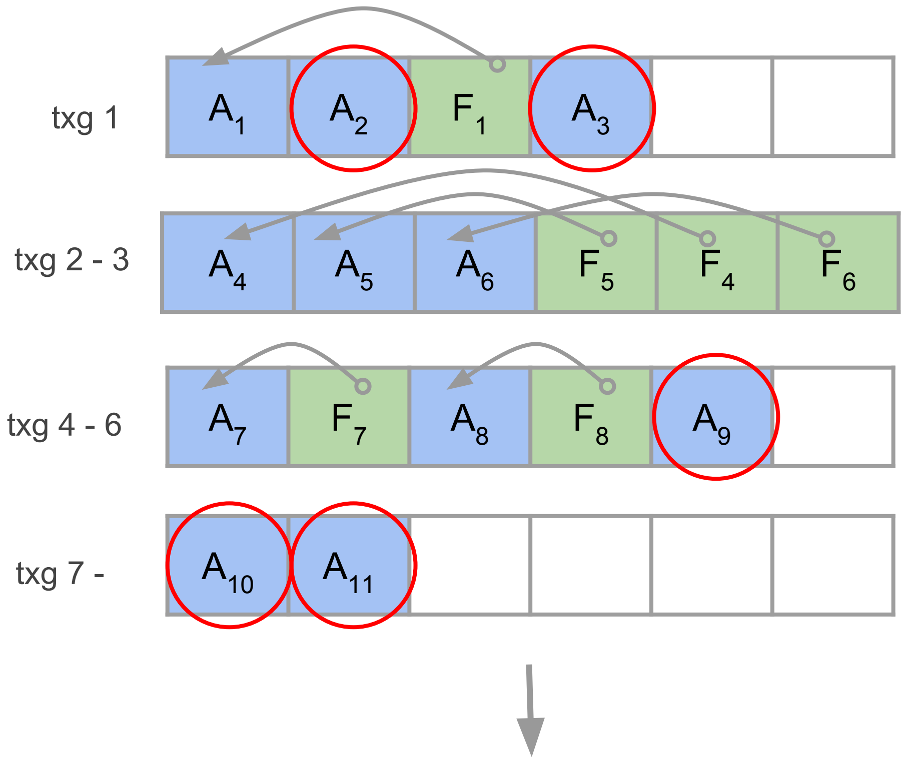
- Deletion work is now proportional to the number of writes to the clone
- Low insertion cost - we know exactly where to put the block pointers

Cons

- Livelist can grow arbitrarily large and we'll have to load the whole thing into memory to delete the clone
- Tricky to destroy incrementally

Sublists

- Break livelist into smaller sublists
- Decide which sublist to insert into based on birth time
- How big should they be?
- Natural way to implement incremental destroy



Asynchronous Destroy

- Want to limit the amount of work we do per **sync**
  - Only destroy one sublist each transaction group
- Loading a sublist into memory could be very expensive
- Some delete work must be synchronous and some can be in the background

```
> zfs destroy clone

Store livelist id in pool
Signal thread
>

                          Load livelist into memory
                       Determine blkptrs to delete
                                     Call synctask

Free blkptrs
Update livelist info in pool
```

Pros

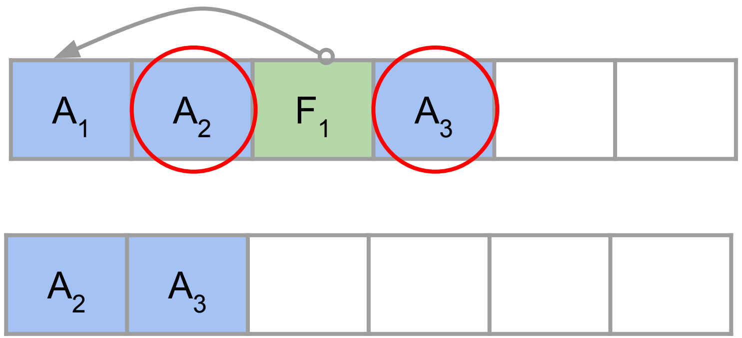
- Limited how much memory is loaded in at once
- Can delete quickly and incrementally

Cons

- Number of sublists can grow arbitrarily large
  - The more sublists we have, the more costly insertion is
  - Disk space

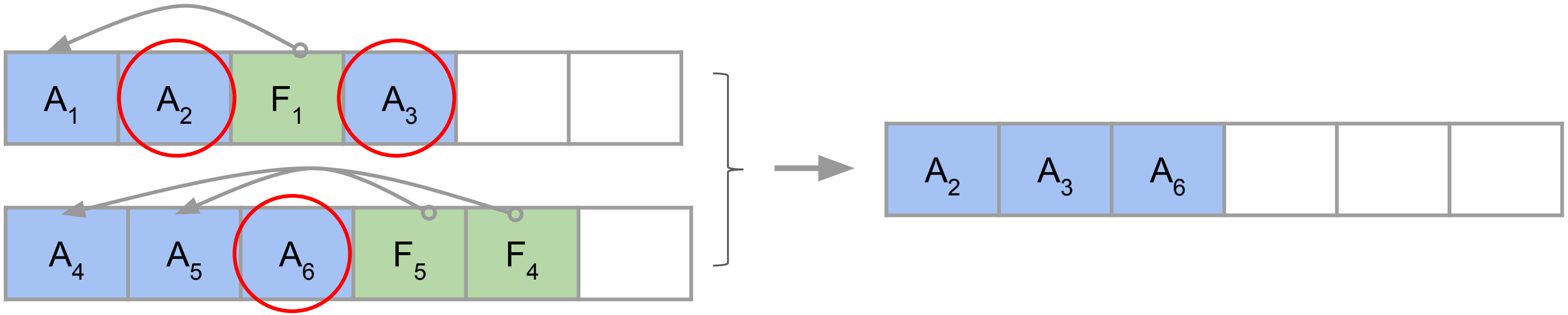
Condensing sublists

- After a block is freed, the livelist contains irrelevant information
- We can condense the list to store only what we need



Merging sublists

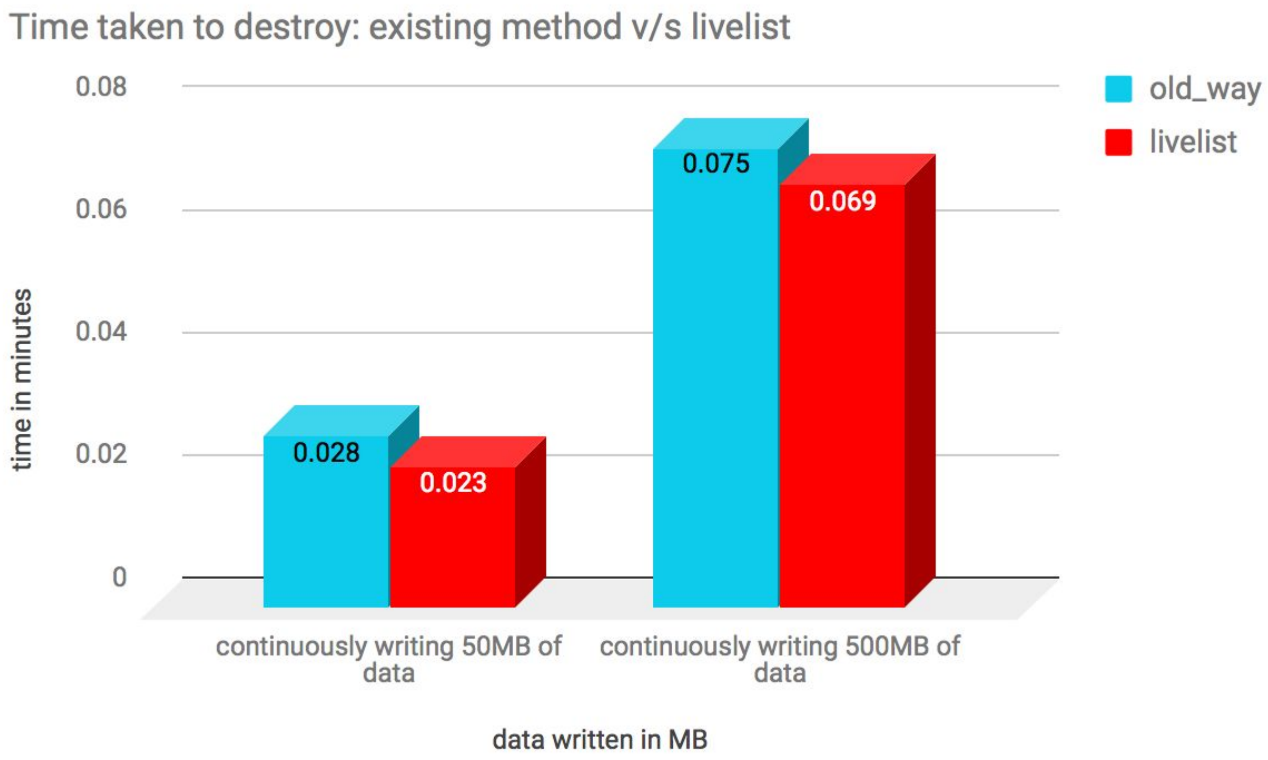
- Now we can merge smaller sublists and reduce their overall number

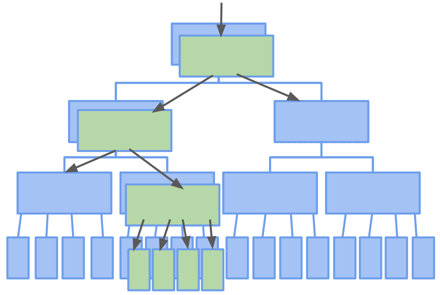


In Summary

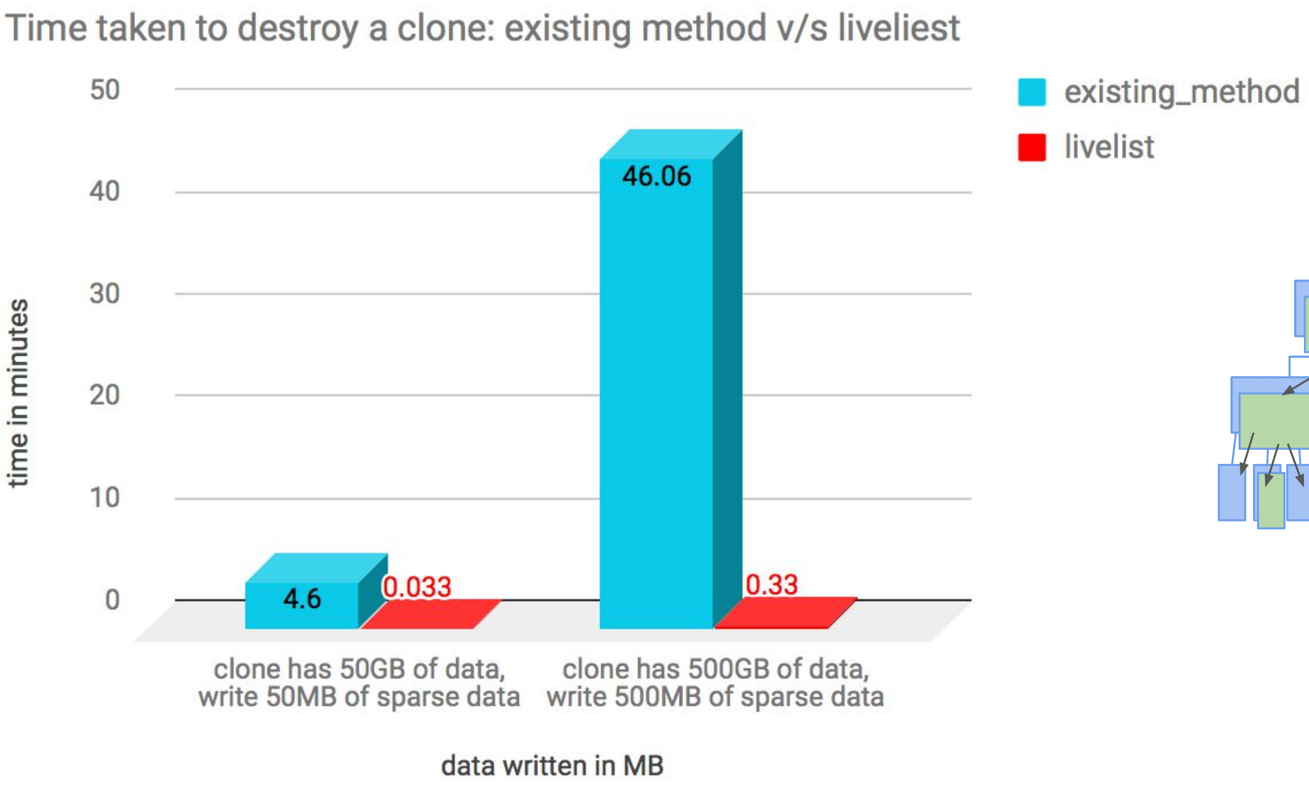
- Made the work of deleting a clone proportional to the number of writes to that clone using a **livelist**
- Limited memory loaded at once using **sublists**
  - Makes it easier to delete **incrementally** and **asynchronously**
- Slowed the growth of the number of sublists by periodically **condensing** the sublists

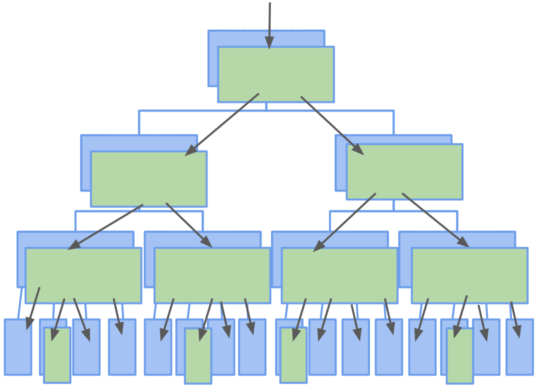
Least Improvement: contiguous writes





Most Improvement: sparse writes





Conclusion

- Livelist method of clone deletion gives dramatic performance improvements in the worst case scenarios
  - Gains in the best case as well
- Tweaks were needed to make the algorithm scalable for production use
  - Balancing space and efficiency
- Coming soon!

Thank you!

Sowrabha Gopal

Matt Ahrens

Serapheim Dimitropoulos

Questions?