1 Write Amp

- Logical address space L
- Physical capacity P
- Usable capacity $= \frac{L}{P} = 1$

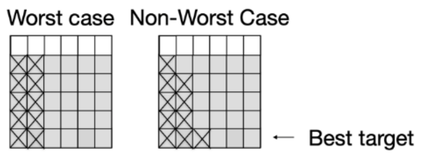
Pages in an SSD can be:
- Empty – writable
- Invalid – to be GC'd
- Valid – contains in-use data

How do states relate to each other:

$$\frac{L}{P} = \frac{\text{valid}}{\text{empty} + \text{valid} + \text{invalid}}$$

1.1 WC Write Amp

Occurs: when invalid pages are uniform across erase units.



Best erase unit to erase is the one with the most invalid pages.

SSD WA approximation: where x is % valid pages:

$$1 + \left(\frac{x}{1-x}\right)$$

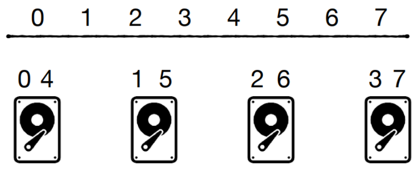
Cost model assuming uniformly randomly distributed writes:

$$1 + \frac{1}{2} \cdot \frac{L/P}{1 - L/P}$$

2 RAID

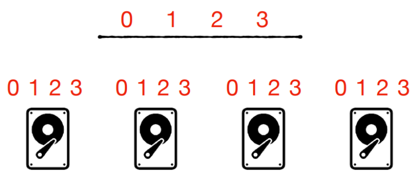
Virtualizes many drives at once to speed things up and provide fail-safes

2.1 Raid O



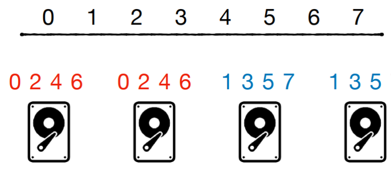
1. Seq R/W at combined bandwidth of all drives
2. Faster random reads/writes due to load balancing
3. No failure tolerance due to no redundancy

2.2 Raid 1



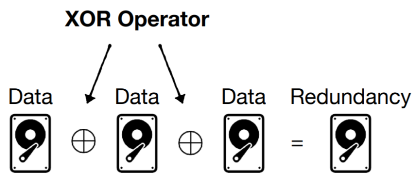
1. Writes as fast as bandwidth of single drive
2. Reads in combined bandwidth of all drives
3. Costs a lot of storage

2.3 Raid 0+1



Stripe and mirror at the same time
N drives, X drives per mirror group. Then:
1. Sequential writes at $\frac{N}{X}$ bandwidth
2. Reads at combined bandwidth of N
3. Capacity: $\frac{1}{X}$ of total

2.4 Raid 4



Raid 5 combines striping and distributed parity

3 Buffer Pools

The idea of buffers: keep hot pages in memory. Accessing the same pages over and over, given no writes in-between, are free

Why storing sequentially is better
Reading/writing from storage of units less than 4KB does not pay off:
- Disks: needles must move. For reading pages sequentially that stops becoming a problem
- SSDs: an entire page gets read or not at all

3.1 Tables

A table has:
- N entries
- B entries per DB page
- Whole table fits in $\frac{N}{B}$ pages

3.2 Continuous Scans

- Mixing table rows everywhere has WC $O(N)$ I/O reads
- Storing table rows contiguously has **WC** $O\left(\frac{N}{B}\right)$ I/O reads

3.3 Storing Pages

- Storing pages as a linked list, where previous pages point to the next requires synchronous I/Os, preventing SSD parallelism.
- Directories, each pointing to many singular pages does not saturate a disk's sequential bandwidth since each directory entry only points to one page
- Directories with extents (contiguous 8-64 pages) combine the best of both worlds and can grow into a tree

3.4 What each table stores

- Data type
- Size
- Start address

Databases keep track of free pages/extents by using a bitmap.

3.5 Supporting Indexless Deletes and Updates

Scan table, create holes for deletes and update in place for updates, $O(1)$ writes and $O\left(\frac{N}{B}\right)$ reads due to the need to read everything

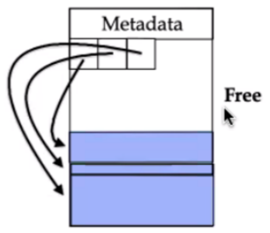
3.6 Optimizing Insertions

- **Scanning and looking** for free space costs $O\left(\frac{N}{B}\right)$ reads and $O(1)$ writes.
- **Storing extents and knowing which are full and which are not** reduces the read time to $O(1)$, cost of accessing directory table. Note: does not make variable-sized entry access faster
- **Buffering insertions** eliminates the cost of reads and reduces write cost to $O\left(\frac{1}{B}\right)$ (every B writes perform a flush). Duplicates are allowed so no overwriting occurs.

3.7 Internal Page Organization

1. Similar to a Python list (deletes take forever)
2. Bitmap (no reorganization but requires more space)

3.7.1 Variable-Sized Row Organization



Using pointers instead of delimiters allow for random access, which is faster

4 Indexing Options

Non-B-Tree versions of indexing. This is where you might want to quickly know the page number given indexable item.

4.1 Zone Maps

Each page stores the min and max. No, this doesn't make it any faster – WC $O(N/B)$ I/Os

4.2 Sort The Column

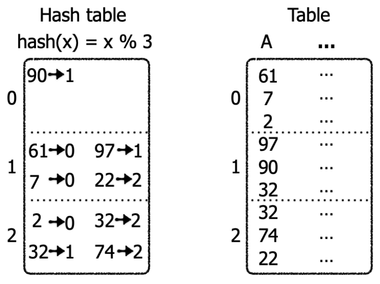
Can only sort one column at once, search costs $O\left(\log_2\left(\frac{N}{B}\right)\right)$ I/O for a total CPU cost of $O\left(\log_2\left(\frac{N}{B}\right)+\log_2(B)\right)$, but updates cost $O\left(\frac{N}{B}\right)$.

4.3 Binary Tree

I/O **and** CPU cost for reads and writes are $O(\log_2(N))$. They don't exploit page locality.

4.4 Hash Table

Give up on sortedness and scan query optimization for average case $O(1)$. Map entries to page. Key repetitions are not allowed.



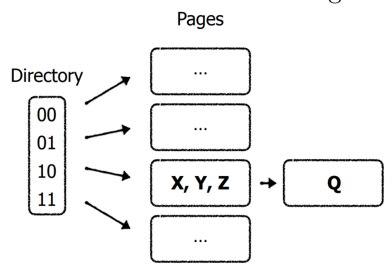
Downsides:

- Bad space efficiency after expansion (50%)
- Only == comparisons
- Expansion leads to performance slumps

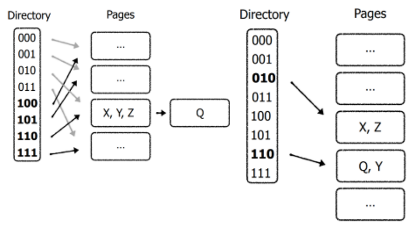
A fix for that?

4.5 Extendible Hashing

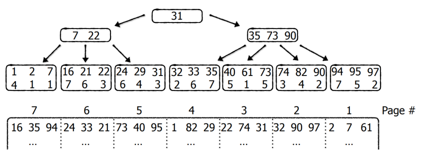
A directory maps pages in storage with a given hash suffix.
Overflows are handled through chaining.



On capacity, double directory size. **New directory slots still point to previous pages.** Now only one overflowing bucket needs to be expanded at a time.

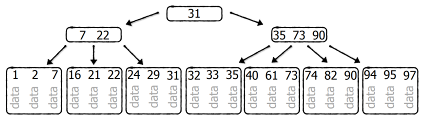


5 B-Tree



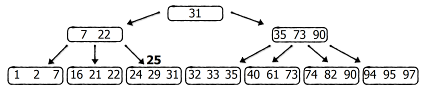
Where B is how many entries fit in a page:

- **Point search** costs $O(\log_B N)$, the height of the tree, since that's the pages it needs to search
  – $O(\log_2(N))$ CPU
- **Writes** cost $O(\log_B N)$ reads and $O(1)$ writes. $O(1)$ reads if internal nodes are stored in memory, where $O\left(\frac{N}{B}\right)$ memory is used for that
- **Scans** cost $O(\log_B N+S)$ reads. If the index is **clustered**, that is table data is stored within the index, that reduces to $O\left(\log_B(N)+\frac{S}{B}\right)$

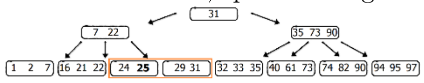


5.1 Going over Inserts (No Propagation)

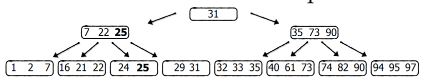
How to insert?
First, find the target node



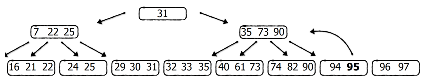
If the node is full, split the target node

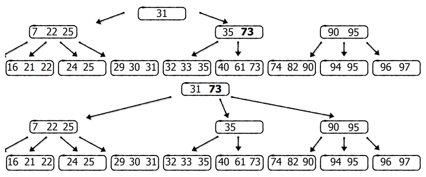


Connect new node to the parent



5.2 Inserts With Propagation





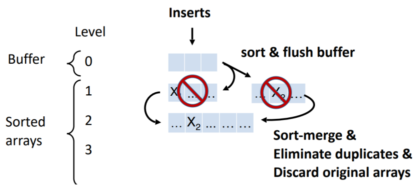
The read I/Os per insertion (assuming no storage in memory) remains $O(\log_B N)$, as we need to locate where to insert.
For writes:

- Every insertion updates 1 leaf
- One in $O\left(B^{-1}\right)$ insertions triggers leaf split
- One in $O\left(B^{-2}\right)$ insertions triggers parent split
- One in $O\left(B^{-3}\right)$ insertions triggers grandparent split

5.3 Why Can We Just Store Internal Nodes In Memory?

Moreover, why do we usually assume that internal nodes are in the buffer pool? Because we access them so often. If a B-tree has $O(N)$ keys, $O(N/B)$ of them are stored in memory. In practice, this happens all of the time, and there are fewer internal nodes than there are keys in total.

6 LSM Trees



Stored as a B-tree, each sorted array is
Deletes: insert a tombstone entry (the tombstone is a **value**).
Variables:

- $P=$ no. entries that fit in the buffer (level 0). We assume LSM trees are clustered by default, so the size of an entry directly impacts P if memory remains constant

Point searches cost

$$O\left(\underbrace{\log_2\left(\frac{N}{P}\right)}_{L}\cdot\underbrace{\log_B(N)}_{\text{B-tree search}}\right)$$

Since:

- $O\left(\log_2\left(\frac{N}{P}\right)\right)$ levels to search
- $O(\log_B(N))$ cost of searching level (if it's organized in a B-tree) (only one sorted run per level)

6.1 LSM Tree Scans

Return most recent version of each entry in the range across the entire tree.
Scans cost:

$$O\left(\log_2\left(\frac{N}{P}\right)\log_B(N)+\frac{S}{B}\right)$$

From this point onwards, we will assume LSM trees store internal nodes for each B-tree in memory.

6.2 LSM Analysis Internal Nodes Memory

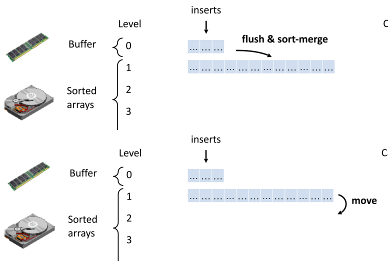
Queries (point)

$$\text{Searches} = O(L)$$
$$\text{Insertions} = O\left(\frac{L}{B}\right)$$
$$L = O\left(\log_2\left(\frac{N}{P}\right)\right)$$

Rationale: searches must go through each level. Insertions may result in a total of L write amplifications since each element can be merged a total of L times.

6.3 Leveled LSM Trees

Improves read performance at the expense of writes
Reduce number of levels by increasing size ratio T
Level count changes to $O\left(\log_T\left(\frac{N}{P}\right)\right)$
Example: size ratio of 4:



Why does this slow down writes? Because elements get sort-merged more often. In the worst case, each element is re-written T times per level. If every element makes it to the bottom, inserts have a WA of $T \cdot L$.
Changes in cost:

$$\text{Lookup} = O(L)$$
$$\text{Insertion} = O\left(\frac{\mathbf{T}L}{B}\right)$$
$$L = O\left(\log_T\left(\frac{N}{P}\right)\right)$$

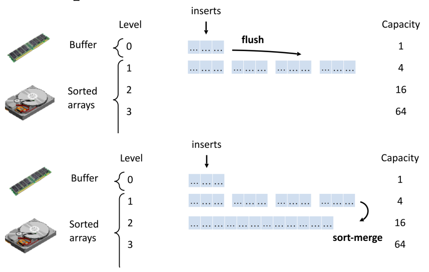
Rationale:

- Less levels than the basic model
- Since only one sorted run per level, and **level count reduced**, looks up are faster
- Writing is more expensive by a factor of T

In the extreme case, where $T = \frac{N}{P}$ (or at most one level), **LSM tree becomes a sorted file**

6.4 Tiered LSM Trees

Improve write performance at the expense of reads
Reduce number of levels by increasing the size ratio T
Example: size ratio of 4



Why does this slow down reads? Because now I need to perform multiple searches per level, because there are more runs.
Changes in cost:

$$\text{Lookup} = O(\mathbf{T}L)$$
$$\text{Insertion} = O\left(\frac{L}{B}\right)$$
$$L = O\left(\log_T\left(\frac{N}{P}\right)\right)$$

Increasing size ratio to $\frac{N}{P}$ turns the LSM tree into an append-only file

6.5 Conclusions I

LSM trees are:

- Write optimized
- Highly tunable
- Backbone of many modern systems
- Trade-off between lookup and insertion costs

6.6 Runtime Table

All internal nodes are stored in memory.

$$L = \log_T\left(\frac{N}{P}\right)$$

Basic: $T = 2$

\$	Apd. only	Sorted list	B-tree
Q	N/B	$\log\left(\frac{N}{B}\right)$	1
IR	0	N/B	1
IW	$1/B$	N/B	$1+GC$
M	B		N/B

\$	LSM-BSC	LSM-TRD	LSM-LVL
Q	$\lg\left(\frac{N}{P}\right)$	LT	L
IR	$\frac{\lg\left(\frac{N}{P}\right)}{B}$	$\frac{L}{B}$	$\frac{LT}{B}$
IW	$\frac{\lg\left(\frac{N}{P}\right)}{B}$	$\frac{L}{B}$	$\frac{LT}{B}$
M	N/B	N/B	N/B

Notice how the write costs correspond to write amplification. The write costs correspond to the no. of times an element is copied in the worst case (if we factor out the B).

7 Bloom Filters

Bloom filters are probabilistic sets with the possible operations:

- Add
- Check for inclusion, where:
  - Negative returned for sure, or
  - Positive with error rate (FPR) of ε

It consists of k different hash functions, where its total size can be tweaked with a knob called "bits per entry": M. With that, the total number of bits the filter takes up is $M \times N$.

7.1 Deciding no. of Hash Functions (K)

More memory $\Rightarrow$ lower FPR
No. of hash functions:

- 1 too low, FPR occurs every collision
- Adding more hash functions exponentially decreases the FPR
- Too many increases it again
- There's an optimal count depending on the bits per entry $M = \frac{m \text{ (total bits)}}{N}$

$$k^* = \ln(2)M$$
$$\Rightarrow \varepsilon = 2^{-M\ln(2)}$$

7.2 Construction Contract

If we know:

- N entries to insert
- ε = desired false positive rate (FPR)

Then a filter should be allocated with $N\ln(2)\log_2\left(\frac{1}{\varepsilon}\right)$ bits.

7.3 Bloom Filter Access Cost

$$\text{Insertions} = M\ln(2) = k = O(M)$$
$$\text{Positive query} = M\ln(2) = O(M)$$
$$\text{Avg. Negative query} = 2$$
$$\text{False Positive Rate} = 2^{-M\ln(2)}$$

Rationale:

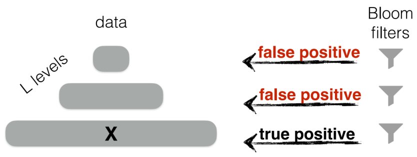
- Insertions: the number of hash functions that must run.

- Positive query: observed all the target bits were 1.
- Avg. negative query: observed one of the target bits was 0 and short circuited.

Note that "positive query" means querying something that's there, in this context.

7.4 Using Bloom Filters with LSM Trees

For a non-tiered LSM tree with L levels, each run of data needs a bloom filter. After all, it's meant to stop a potential I/O read to that level.



7.4.1 Filter Overhead for Basic Accesses

I want to perform a **point query.** The absolute worst case that could happen is **I search for an element that exists, it's on the bottom of the tree, and every bloom filter returns a positive.** Which means $L-1$ false positives and 1 true positive, for a total **worst-case** cost of

$$\underbrace{(L-1)}_{\text{no. FPs}}M + M =$$

$$\text{Worst case} = O(ML)$$

In the average case, false positives usually do not occur so the **average worst case** is:

$$2(L-1)+M$$

$$\text{Avg. worst case} = O(M+L)$$

7.5 Optimizing Leveling

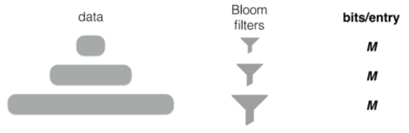
Let's optimize a leveled LSM. With bloom filters:

- Gets cost $O\left(2^{-M}\cdot L\right)$
  - Want $O\left(2^{-M}\right)$
  - *MONKEY* stores this
- Inserts cost $O\left(\frac{TL}{B}\right)$
  - Want $O\left(\frac{T+L}{B}\right)$
  - *DOSTOVESKY* solves this

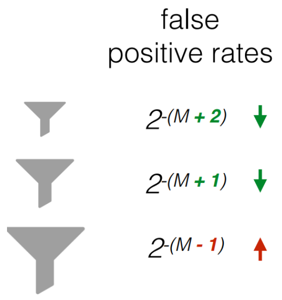
7.5.1 MONKEY (Get optimizer)

MONKEY OPTIMAL NAVIGABLE KEY-VALUE STORE

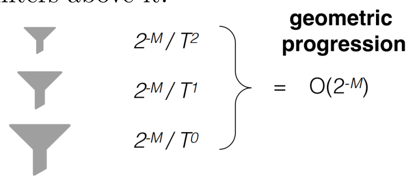
Bloom filters scale up the lower the level, **yet each bloom filter has the same FPR.**



If we're accessing our data on the bottom of the LSM, we want our bloom filters higher up to return true negatives, because the data higher is smaller, and we wouldn't want to waste a read. Hence, we can relocate bloom filter sizes, **considering the bottom one is the largest**:



This shrinks the bloom filter on the lowest level and expands the bloom filters above it.



This results in a faster worst-case, $O\left(2^{-M}\right)$ instead of $O\left(2^{-M}\cdot L\right)$. Of course this increases the absolute worst case up to $O\left(ML+L^2\right)$ but that never happens.

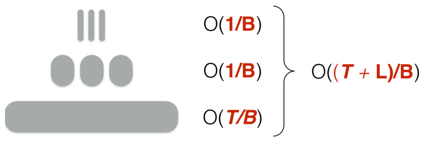
7.5.2 DOSTOVESKY (Write Optimizer)

DOSTOEVSKY SPACE TIME OPTIMIZED EVOLVABLE SCALABLE KEY-VALUE STORE

We want to fix the cost of writes being proportional to the number of levels there are. Can be done by making writes lazy on the upper levels. Therefore:

- Apply tiering on the upper levels
- Apply leveling on the lowest level

This reduces the insert I/O cost from $O\left(\frac{TL}{B}\right)$ to $O\left(\frac{T+L}{B}\right)$.

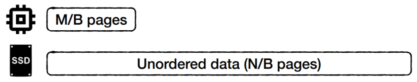


8 External Sorting

Mergesort is worst $O\left(n\log(n)\right)$ and takes 2x more space. Quicksort has average worst case $O\left(n\log(n)\right)$ and is done in-place. These only work in memory. So, **what is the optimal way to sort lots of data in storage?** Here are some attempts:

- B-tree
  - Scan unordered data, insert each into B-tree, append afterwards for $O\left(N\log_B N\right)$ R and $O(N)$ W
  - If internal nodes in mem: $O(N)$ R and W
- LSM-tree
  - $O\left(\frac{N}{B}\log_2\left(\frac{N}{P}\right)\right)$ reads and writes all the time

How can we efficiently sort data that doesn't fit in memory?

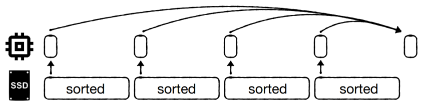


Setting up variables:

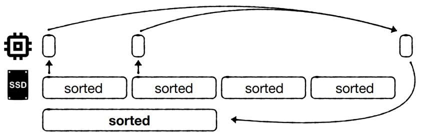
- M entries fit in memory
- N entries in total
- B entries per 4KB page
- $\Rightarrow M/B$ pages in mem, N/B pages to process in total

8.1 Multi-Way Merge Sort

For each chunk that fits into memory, sort them for N/M sorted temp. files for a cost of N/B I/Os



Allocate a buffer for each input file and merge into output stream.
Too little memory requires multiple passes (here: 2 iters).



There are $\frac{N}{M}$ partitions, and we have $\frac{M}{B}$ buffers (pages in memory). **Each insertion merges $\frac{M}{B}$ partitions.** Total iterations:

$$\text{Iterations} = \log_{M/B} N/M$$

$$= \log_{\text{pages in memory}} \text{pages in storage}$$

$$= \log_{\text{PG}_{\text{mem}}} \text{PG}_{\text{storage}}$$

Each iteration corresponds to how many levels of merging need to be done.
Each iteration does a full pass over data for a cost of $O\left(N/B\right)$
Merging costs $\frac{N}{B}\left\lceil\log_{\frac{M}{B}} N/M\right\rceil$

alongside N/B for partitioning. Total $O\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$.

Increasing memory count means merging more partitions per pass.

How much memory is required to merge **all** partitions in one pass? (I mean 2) – solve for M:

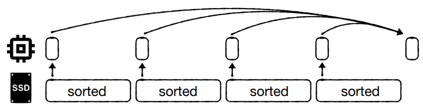
$$\log_{\frac{M}{B}} N/B = 2$$
$$\Rightarrow M = \sqrt{N \cdot B}$$

With this M, memory can accommodate $\sqrt{N/B}$ buffers.

With $M = \sqrt{N \cdot B}$ memory to partition data, this creates at most $N/M = \sqrt{N/B}$ sorted partitions. Then merging in one pass uses at most $\sqrt{N/B}$ input buffers.

For the cost of $O\left(N/B\right)$ I/Os overall.

The cost of this:



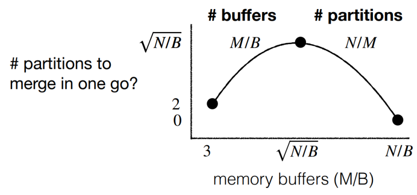
For all practical purposes, a 2-pass sorting-algorithm is practical.

8.2 Optimizing CPU cost

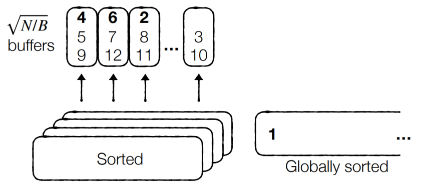
We expect $O\left(N\log_2(N)\right)$.

Partitioning and sorting each chunk for a total of N/M chunks cost $O\left(N\log_2(M)\right)$.

For merging:



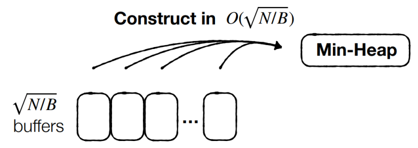
The idea is to traverse all $\frac{M}{B}$ buffers (e.g. $\sqrt{N/B}$ at the optimal) and pick minimum:



For a total of $O\left(\sqrt{N/B}\cdot N\right)$ comparisons.

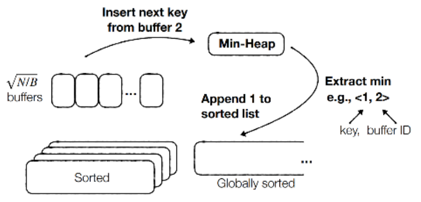
How do we sort this more efficiently?

For a min-heap with $\sqrt{N/B}$ items, one per buffer, we're able to **pop then insert** in log time.



So, for all buffers, the first elements of each buffer go in the min-heap. Each heap entry stores the buffer ID.

When item is extracted, insert next key from the heap entry's buffer ID.



Per entry, this costs $O\left(\log_2\sqrt{N/B}\right)$.

For N items that must be processed, this costs $O\left(N\log_2\left(N/B\right)\right)$.

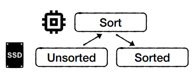
In all:

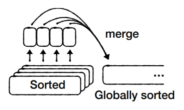
- **Partitioning** costs $O\left(N\log_2(M)\right) = O\left(N\lg\left(\sqrt{NB}\right)\right)$
- **Merging** costs $O\left(N\lg\left(\sqrt{N/B}\right)\right)$
- Total cost is $O\left(N\log_2 N\right)$

The same as in-memory algorithms.

Thus, the overall cost is:

$$O\left(N\log_2(N)\right) \quad \text{CPU}$$
$$\begin{cases} O\left(\frac{N}{B}\log_{\frac{M}{B}}\left(\frac{N}{B}\right)\right) & \text{I/O} \\ O\left(\frac{N}{B}\right) & M > \sqrt{MN} \end{cases}$$





9 Runtime Summary

If internal nodes are not in mem, B-tree Q&IR$\times\log_B(N)$, LSM Q&IR: $\times\log_T\left(\frac{N}{B}\right)$. LSM trees assumed clustered

\$	Apd. only	Sorted list	B-tree
Q	N/B	$\log\left(\frac{N}{B}\right)$	1
IR	0	N/B	1
IW	$1/B$	N/B	$1+GC$
S	N/B	$\log\left(\frac{N}{B}\right)+\frac{S}{B}$	S Clust: S/B
M	B		N/B

\$	LSM-BSC	LSM-TRD	LSM-LVL
Q	$\lg\left(\frac{N}{P}\right)$	LT	L
IR	$\frac{\lg\left(\frac{N}{P}\right)}{B}$	$\frac{L}{B}$	$\frac{LT}{B}$
IW	$\frac{\lg\left(\frac{N}{P}\right)}{B}$	$\frac{L}{B}$	$\frac{LT}{B}$
S	$Q+\frac{S}{B}$	$Q+\frac{S}{B}$	$Q+\frac{S}{B}$
M	N/B	N/B	N/B

\$	LSM-LVL	LSM-BLM	MKY+DV
Q	L	$2^{-M}L$	2^{-M}
IR	$\frac{LT}{B}$	$\leftarrow$	$\frac{L+T}{B}$
IW	$\frac{LT}{B}$	$\leftarrow$	$\frac{L+T}{B}$
S	$Q+\frac{S}{B}$	$Q+\frac{S}{B}$	$Q+\frac{S}{B}$
M	N/B	N/B	N/B

For bloom filters, CPU cost is:

- Insertions $= M\ln(2) = k,\ O(M)$
- Positive query $= M\ln(2),\ O(M)$
- Avg. Negative query $= 2$
- FPR $= 2^{-M\ln(2)}$

External sorting:

- $M \geq \sqrt{N\cdot B}$ for one-pass
- If so: $O\left(\frac{N}{B}\right)$ I/O R&W, $O\left(N\lg(N)\right)$ CPU