CSC443

Database System Technology

Last updated October 1, 2025

Contents

1 Modelling SSD Write Amplification

Consider an SSD with logical address space of size L and physical capacity P. The usable capacity is a fraction of $\frac{L}{P} \leq 1$, the overall capacity.

Every page of an SSD can be in one of three states:

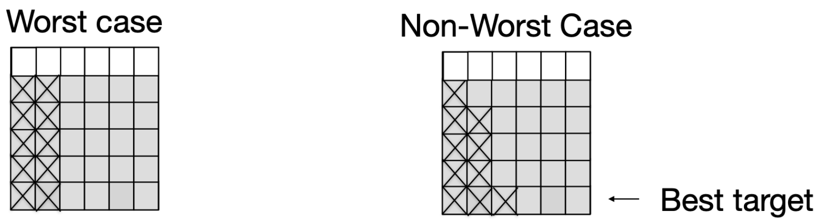
- Empty
  - I can write to at any time
- Invalid
  - I don't need it, and stays like this until I perform garbage collection
- Valid
  - Contains data that is in use

How do these states relate to each other?

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

Let's also note that the % of empty pages in an SSD are negligibly small as we do garbage collection. However you do need some empty pages for garbage collection.

1.1 When does the Worst Case Write Amplification Occur?



When invalid pages are uniform across erase units.

Consider an SSD with logical address of space size L and physical capacity P. The usable capacity is a fraction of $\frac{L}{P}$ of the overall capacity.

Worst-case write-amplification (WA) occurs when each erase-unit has the same number of invalid pages.

For the "worst case" SSD in the diagram, how do we add a block? To be able to garbage collect spaces for two invalid pages, I have to migrate 4 valid pages. In the worst case, $x = \frac{\text{valid}}{\text{total}} = \frac{4}{6}$ (all erase units have the same no. of valid vs. invalid units) $= \frac{L}{P}$

The SSD WA approx., where x is the average **fraction** of valid (in-use valid data) pages in the garbage-collected block, is

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

This worst-case is very unlikely to happen in practice. In practice, note that erase units that were written a longer time ago will usually have more invalid pages. Even if you are doing uniformly random writes in an SSD, you will still notice that older pages have more invalid pages. When we have an uneven distribution, we can find an erase unit with more invalid pages.

The cost model assuming uniformly randomly distributed writes is *approximately* but not exactly

$$1 + \left(\frac{1}{2}\right) \cdot \frac{L/P}{1 - L/P}$$

2 RAID

Remember: Writing to SSDs can be done at any time but deletions require a full block to be erased (which sets it to UNSET)

Redundant array of independent (inexpensive?) discs. Also for this section assume no bit flips happen since it's the drives' responsibility to fix these errors.

2.1 Motivation

What if...

- Database size exceeds one drive's limit, and we need more storage
- A drive fails and we need to recover its data given the inevitability of storage failure
- A disk drive and SSD has limited bandwidth. We want to overcome this limit.

2.2 What does it do?

RAID is a virtualization layer. It sits between the OS and the database and the disc, and it looks to the database and the OS as if it is just one storage drive.

Suppose I have two drives, where each one of them stores N bytes. RAID encapsulates and virtualizes them into one logical advice which seems to have $2N$ bytes. The OS and the database just see this one logical drive, no longer seeing that they consist of two physical drives.

With that, we can embed redundancy in drives and exploit the throughput of having multiple devices.

Looks like one drive to the OS, but actually multiple.

2.3 How do I implement it?

RAID can be implemented in:

- Hardware: many discs connected to one RAID device. The computer interacts the RAID device as if it is just one storage device.
- Software

2.4 RAID Designs

There are many RAID designs. Each design has trade-offs. In fact, a theme of this class is *making tradeoffs.* Here are the different RAID designs.

2.4.1 Raid 0

"Pure striping"

The idea is, and one implementation of that is:

- Suppose our logical address space is $0 \ldots 7$: sector addresses. Each number represents as 512-byte block address.
- We have 4 drives. Different drives manage different blocks. **Notice the order that optimizes sequential access such that many discs are spanned.**
- E.g.
  - Disk 1 maps to block 0&4, disk 2 maps to block 1&5, disk 3 maps to 2&6, disk 4 maps to 3&7

Because of that parallelism, having 4 disks:

- Sequential reads and writes at a combined bandwidth of all drives.
- Random reads / writes are faster due to load balancing (so no catches here)
- Failure tolerance? What happens if one drive fails? Can we recover from it?

  - **Nope, data is lost. If one disk fails, data is lost.**

Okay, that was easy, but you can't recover data. And the next one is still simple.

2.4.2 Raid 1

"One drive is mirrored across all other drives".

No benefit otherwise. This is just backing up. The idea is that **the same data is mirrored on as many drives as we have.**

So (again, think of parallelism, and I'm assuming that the controller has minimum overhead):

- Writing sequentially across the address space (~)
  - …writes in the bandwidth of a single drive. Whatever we write in the disk drive, we need to copy it.
- Reading (+)
  - …reads in the **combined** bandwidth of all drives. So you can exploit this redundancy to read in parallel.
- Storage (-)
  - You lose out on storage, due to the redundancy in question.

2.4.3 Raid 0+1

A combination of raid 0 and raid 1. Effectively copy raid 0 a ton of times. Combined **mirroring** and **striping.**

- Disk 1&2 maps to addresses 0, 2, 4, 6
- Disk 3&4 maps to addressees 1, 3, 5, 7

So, assume N drives and X mirrors per drive. Then:

1. Sequential write bandwidth speedup: $\frac{N}{X}$

   a. Can only be as fast as the number of *groups* there are

2. Sequential read bandwidth speedup: N
   a. No losses here
3. Storage capacity: $\frac{1}{X}$ of the total capacity across all drives

2.4.4 Raid 4

Goal: use less than 50% of the storage capacity to allow recovery. If we are using three drives per mirrored group, we are losing $\frac{2}{3}$rds of our storage capacity. The more mirroring we have, the more we can deal with concurrent failures. But the more of them we have, the more wasteful we are of capacity.

Suppose that we want to use a lot of drives to store useful data and just one drive to store redundancy information:

- Data
- Data
- Data
- Redundancy

This is what Raid 4 aims for, so how do we generate that redundancy. **How can we use one drive to recover any drive that fails (data loss)?**

By using the XOR operator in all the same stripe, we can generate redundancy. So:

$$DATA \oplus DATA \oplus DATA = REDUNDANCY$$

So, what type of magic does XOR do:

$$0011 \oplus 0101 = 0110$$

The result of XOR is called parity. Once we've generated the parity for two inputs, **we can reconstruct an input using the other and the parity.** Which means:

$$
\begin{aligned}
&A \oplus B = P \\
&\Rightarrow A \oplus P = B \\
&\Rightarrow B \oplus P = A
\end{aligned}
$$

As well: XOR is **commutative and associative.**

The point is:

- Given a number of inputs, if you XOR all of them to generate the parity, if you lose any individual input, you can XOR the remainder of the inputs with the parity to recover the input that you lost.

Which means:

$$
\begin{aligned}
&A \oplus B \oplus C \oplus D = P \\
&\Rightarrow A \oplus B \oplus P \oplus D = C
\end{aligned}
$$

Another point? What is $(A \oplus B) \oplus B$? You get back A. So if you XOR something with it twice, it cancels out. So:

$$(A \oplus B \oplus B) = A$$

So, here's how you construct RAID 4:

- Do your regular striping: for address space $0 \dots 8$ for all but your last drive:
  - Drive 1 gets 0, 3, 6
  - Drive 2 gets 1, 4, 7
  - Drive 3 gets 2, 5, 8
- You have a parity drive, storing $(0 \oplus 1 \oplus 2)$, $(3 \oplus 4 \oplus 5)$, $(6 \oplus 7 \oplus 8)$. So one drive stores all the parities.

Suppose you aren't updating another stripe, and you just want to update a single page, such as page 4. **What is the challenge?** This is the case of a random write. Updating is going to impact the parity. If you make the update without changing the parity, if you have a feature failure, you'll return the wrong answer. This means, **updating means, you'll have to change the parity.**

Questions:

1. How *could* I recompute the parity?
   a. Read all pages and regenerate the parity.
   b. Cost: $N - 2$ reads, 2 writes.
2. Is there a better way to do it?
   a. Read the original value and the parity
   b. Abuse the fact that XORing twice cancels out:
   c. $4_{new} \oplus 4_{old} \oplus (3 \oplus 4_{old} \oplus 5) = (3 \oplus 4_{new} \oplus 5)$
   d. For a total of only **2** reads and **2** writes

To make a logical random write, I just need two disc reads and 2 disk writes.

- One read from the drive the read is due, and one read from the parity drive
- Then you need to update your drive you updated and the parity drive

But *are* there any problems?

The parity drive becomes a **bottleneck** for random writes. If you happen to make 3 random writes to 3 different drives. This means you must make three updates to the parity drive. All the parities are in one drive, which means your random write bandwidth is just as good as a single drive. How do we fix this?

2.4.5 Raid 5

Striping and distributed parity

Now we're going to distribute parity in a round-robin way across the drives.

For example, if we have blocks 0...11, then these drives are responsible for

1. 0, 3, 6, $(9 \oplus 10 \oplus 11)$
2. 1, 4, 6, $(7 \oplus 8 \oplus 9)$
3. 2, $(3 \oplus 4 \oplus 5)$, 7, 10
4. $(0 \oplus 1 \oplus 2)$, 5, 8, 11

Each drive stores 3 sectors and one parity.

Because the parities are distributed, the random writes go to different disks. This solves the bottleneck. So, in summary:

- Does not waste most of the storage capacity
- Sequential writes at $\frac{N-1}{N}$ of max bandwidth (so speedup of $N - 1$)
- **Random reads have less flexibility than with mirroring**
- Fast sequential reads since striping
- Random write requires 2 reads and 2 writes
- **Random write loads are evenly distributed on all drives (new for raid 5)**

So far, we've covered designs where only one drive fails at a time. No, RAID 4 and 5 cannot solve this. But, there are more advanced types of RAID designs that can fix 2 or more drives failing simultaneously, RAID 6 being one of them (not covered in this class).

So, how do I recover a drive with this setup?

1. 0, 3, 6, $(9 \oplus 10 \oplus 11)$
2. 1, 4, 6, $(7 \oplus 8 \oplus 9)$
3. 2, $(3 \oplus 4 \oplus 5)$, 7, 10
4. $(0 \oplus 1 \oplus 2)$, 5, 8, 11

To recover drive no. 2:

- Read 0, 2, and $0 \oplus 1 \oplus 2$ to recover block 1
- To recover block 4, read 3, 4, and $3 \oplus 4 \oplus 5$
- Continue…

3 Buffer Pools, Row Stores, Storing Data Efficiently

3.1 The Goal For Databases

We want to efficiently support:

1. Scans (read, i.e. `select`)
2. Deletes
3. Updates
4. Insertions

This course is really to optimize for data movement. You learned how to model CPU performance of an algorithm, big-O notations, compare to how many instructions the CPU is running. We still do CPU analyses, but we also want to add the idea of analyzing data movement across the storage hierarchy, **as storage is orders of magnitude slower than the CPU.**

We know very well that **reading/writing from storage of units less than 4KiB does not pay off.** Why:

- **Disks:** moving takes forever! And then after the needle has moved you can read that region for free
- **SSD:** you read an entire page or don't

But also, we don't want to read units that are too large, as that leads to a lot of weird memory management. If all you care is a small fraction of data within a chunk, don't read that much.

Hence, to make the unit of I/O not too large or not to small, we use the notion of a database page. This aligns with an SSD page. Hence, data pages use **4KiB** as the page size.

3.2 The Disk Access Model (DAM Model)

A table has:

- N entries (or rows).
- B entries per DB page
  - B rows fit in a DB page.
- The whole table consists of $\frac{N}{B}$ pages

Therefore we will count the worst-case IO operations with respect to N and B.

This model is imperfect. It ignores many characteristics of storage – as:

- Sequential disk access is faster than random disk access.
- Asynchronous SSD I/Os are faster than synchronous SSD I/O s
- Ignores SSD garbage-collection due to random writes

However this model is useful due to its simplicity. You usually want to avoid complexity, as the more complex a model is, the less useful it is. Still, you want to use a simple model and still appreciate the shortcomings of the model so you can at least see all the nuances of it.

3.3 Scans

A scan operation here is **reading every row:**

```sql
SELECT * FROM CUSTOMERS
```

How do we structure scans?

3.3.1 Ways to Structure Scans

Each row here represents an attempt to design how data is laid out in disk for a scan.

What	Cost
Mix table rows within the same pages. Entries are jumbled everywhere.	Worst case $O(N)$ I/O reads. At most N pages have to be accessed when they're scatted everywhere.
Separate table rows into different sets of DB pages (make entries as **contiguous**). Same page, same table.	$O(N/B)$ I/O reads. (B is the number of rows that fit in a page)

3.3.2 How to Keep Track of Which Disk Page belongs to What Table?

How does a database keep track of which disk page belongs to which table?

Potential solutions?

What	Problems
Linked list. PAGE $\longrightarrow$ PAGE $\longrightarrow$ PAGE	Entails **synchronous I/Os,** which do not exploit SSD parallelism. I must read once, check result, then read again.
Directories, pointing to many singular pages. $\text{DIR}_1 \longrightarrow (\text{PAGE}_1, \text{PAGE}_2, \ldots)$ $\text{DIR}_2 \longrightarrow (\text{PAGE}_{11}, \text{PAGE}_{12}, \ldots)$	Small I/Os, which do not saturate a disk's sequential bandwidth (for hard disks only). For more than one page, you may have to access other pages far away.
Directory with extents (contiguous 8-64 pages). $\text{DIR}_1 \longrightarrow (\text{PAGE\_EXT}_1, \ldots)$ $\text{DIR}_2 \longrightarrow (\text{PAGE\_EXT}_{11}, \ldots)$ *It can grow into a tree if needed.*	

Every database has some sort of catalog that stores the most important metadata of the core data objects.

For example, each table might store:

- Data type
- Size
- **Start address**

The catalogue stores the root address of the table.

How does the database keep track of free pages or extents?

- (Do not use this!) Linked lists? Each extent points to the next free extent. No, not going to work.
- **Use a bitmap. This takes up space, but at least you know which extents free and which ones are not. It costs a bit more space, but the space used is quite moderate.**

The cost of scanning the entire table in this case takes O (N/B) I/O reads.

3.4 Deletes (Without Indexing)

```
DELETE FROM customers WHERE NAME = "..."
```

Attempts to support deletes:

- Scan the table.
- Create holes.
- Cost: $O(1)$ writes, $O\left(\frac{N}{B}\right)$ reads (the cost of scanning the table).

3.5 Supporting Updates (Without Indexing)

```
UPDATE customers SET email = "..." WHERE name = "..."
```

Attempts to support updates:

- Scan the table.

- Update in place.
- Cost: $O(1)$ writes, $O(N/B)$ reads.

For tables with non-constant size elements, if the newer version doesn't fit, delete and re-insert.

3.6 Supporting Insertions (Without Indexing)

```sql
INSERT into CUSTOMERS VALUES(...)
```

There are multiply ways to approach this, though here it's going to get successfully better:

Method	Description	Cost
Scan and find space (no bitmap)	The most trivial way to do this but requires a full scan.	Reads: $O(N/B)$ Writes: $O(1)$
Store extents that are full and extents that still have free space (maybe in a linked list)	Knowing where to write to does not require any reads	**For fixed-sized entries** Reads: $O(1)$ (accessing the directory table) Writes: $O(1)$ **For variable-sized entries** Reads: $O(N/B)$ Writes: $O(1)$
Buffer insertions in memories and append to extent	When the buffer fills up, flush it to an extent with free space	**Per insertion** Reads: None Writes: $O(1/B)$ (Every B writes perform a flush)

3.7 Internal Page Organization

Recall that each page is a unit of 4KB. Suppose that rows are fixed-sized. How do I organize rows within a page?

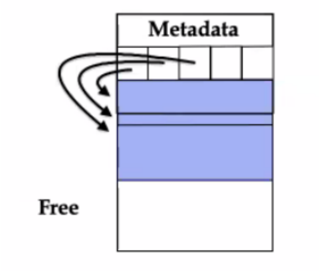
3.7.1 Organization With Fixed-Size Rows

Recall each page is 4KB and suppose rows are fixed-sized. How can rows be organized?

- **Option 1:** Similar to a Python list. On deletion, move everything after back by one. This is probably going to be very expensive, often requiring the entire page to be re-written.
  - $[A, B, C, D, E, F, \ldots] \to [A,\ B,\ \times, D, E, F, \ldots] \to [A, B, \overleftarrow{D}, \overleftarrow{E}, \overleftarrow{F}, \ldots]$
  - Requires re-organization due to deletes
- **Option 2:** Use a bitmap at the start of the page to detect which slots within a page are free, and which slots are not.
  - No reorganization but requires more space.

3.7.2 Organization With Variable-Size Rows

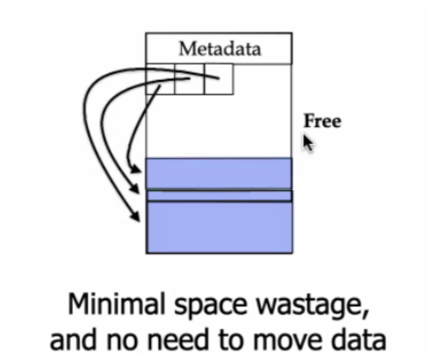
If we tried using the bitmap approach like before, unfortunately, you're going to get a lot of internal fragmentation when the metadata section of the page, which stores where contents of the page are, runs out.



Even in the absence of deletes, if the rows all end up being smaller than expected, then we end up wasting space in the page. We can only store so many pointers, so **if you run out of pointers you can't use the rest of the page.**

So, how do we solve this? The solution is: **Data grows from the back, metadata grows from the front**

This results in minimal space wastage, and no need to move data.



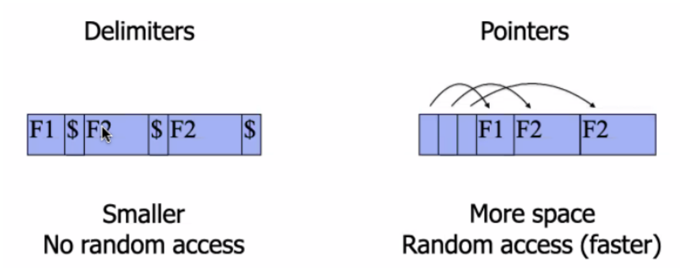
How do we store an individual row? A row consists of multiple fields. Some fields might be fixed-size, other fields might be variable size. If columns are fixed size, it's all easy. If they are variable sized, then we might need delimiters.

3.7.3 Delimiters with pointers

When storing variable-sized entries, we can either use delimiters to distinguish between different elements, or we can use pointers at the start.

While delimiters can be smaller, since they don't need to be as long as an address, you lose out on random access, requiring you to traverse up to the entire page to find where an entry is.

On the other hand, when using pointers, you get random access so you don't need to do what delimiters would require. However, they take more space.

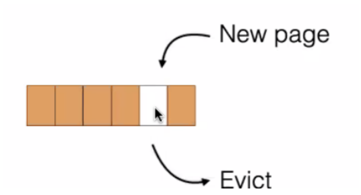


4 Buffer Management

A database is reading and writing aligned 4K storage pages. Suppose some pages are frequently accessed (hot). Retrieving these pages over and over is very wasteful. Every time a hot page is accessed, it is very wasteful to do an entire storage IO to retrieve it.

Hence, you should be keeping copies of hot pages in memory. You do not have enough memory to store the entire database, but you at least have memory to store the hot data.

A buffer pool consists of frames, each containing one page of data. As more pages are filled up, pages must be evicted to clear space.



Every page must store some metadata:

- Pin count: how many users are using the page
- Dirty flag: indicates whether the page has been updated

Which pages should be evicted when space is ran out?

- Avoid evicting pages likely to be used again soon. We want to save I/O cost. If a page is going to be written again, you'll going to have to do an entire I/O to read it again.

- Avoid using excessive metadata or CPU overheads.

The eviction policy has a lot of impact on I/O and CPU efficiency. So, we'll cover them.

4.1 Eviction Policies

THIS OCCURS IN MEMORY. So no disk accesses.

4.1.1 Random

When a new page comes in, which ever page it collides with in the hash table, the existing page gets evicted.

- Adv:
  - Easy to implement
- Disadvantage:
  - Could evict a page that needs to be used again

4.1.2 FIFO

Evict the page that was inserted the longest time ago.

Rationale:

- Less likely to be used again???? Not necessarily. It might be true for some workloads but it is not generally true. **The page that was read a long time ago to the buffer pool might as well be a hot page. This makes FIFO bad.**
- FIFO is implemented with a ring buffer, an array with front and rear pointers.

Unlike the random policy, **pages we evict now have different frames than the page we insert.** Therefore, we need a hash collision resolution algorithm, either by using linear probing or chaining. Too many hash collisions hurt performance, so we leave aside extra capacity in the table to reduce the likelihood of collisions.

So this means there will be extra space and CPU cost, but this is necessary for all policies but the random one. Whenever there are chances for a collision, never fill up the hash table entirely.

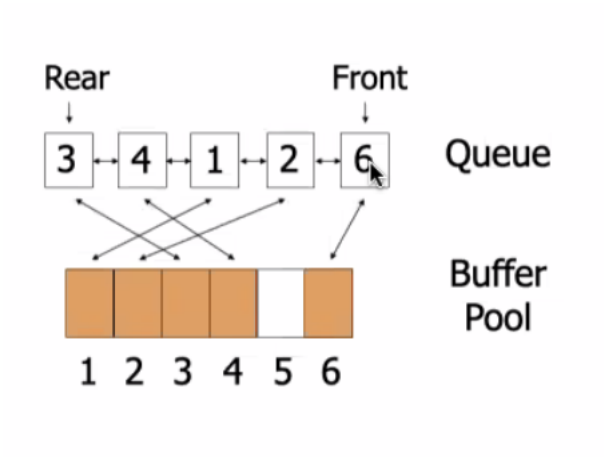
4.1.3 LRU

Evict the page that was *used* the longest time ago.

Implementation? Can I use an implementation similar to the ring buffer like FIFO or should I…

Linked list. A priority queue would be possible but is overkill for this purpose. A random-access queue, where we must be able to move an element at a random location back to the front.

Use a doubly linked list as the queue with pointer linking nodes to buckets.



When making an eviction, clear frame 6 and look for a free spot and add it to the back of the queue. When using something, move it to the back of the queue.

Problems:

1. CPU overhead to update queue for each access
2. Linked lists are less efficient than arrays due to pointer chasing
3. Metadata overhead for pointers

Good otherwise. Thrashing just sucks

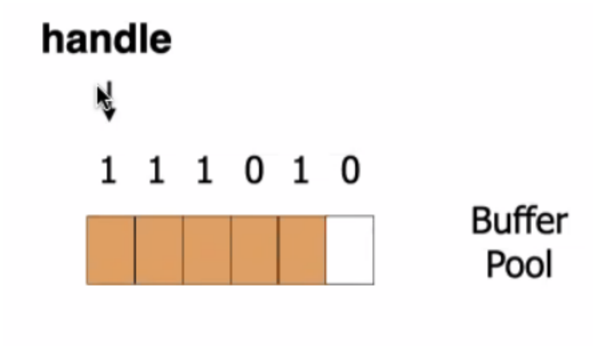
4.1.4 Clock

Traverse the hash table circularly as a clock.

Evict any entry not used since the last traversal.

Each page has an access bit.

Have a clock handle.



The handle only evicts pages where the corresponding page is set to 0. **The handle starts moving when an eviction is needed and runs until it evicts something.**

When a handle passes a bit, it sets it to 0.

When a page is accessed or inserted, the bit is set to 1.

Advantages:

- Lower overheads as there is no queue
- Bitmaps take little extra space

Disadvantages

- Can evict hotter pages than LRU but is still better than FIFO.

You evict some page that is not least recently used but it may not be the least recently used.

4.1.5 Summary

What	Eviction effectiveness	CPU
Random	Worst	Best
FIFO	Moderate	Good
LRU	Best	Worst
Clock	Close to Best	Good

Clocks are used all over the place. Postgres uses clock.

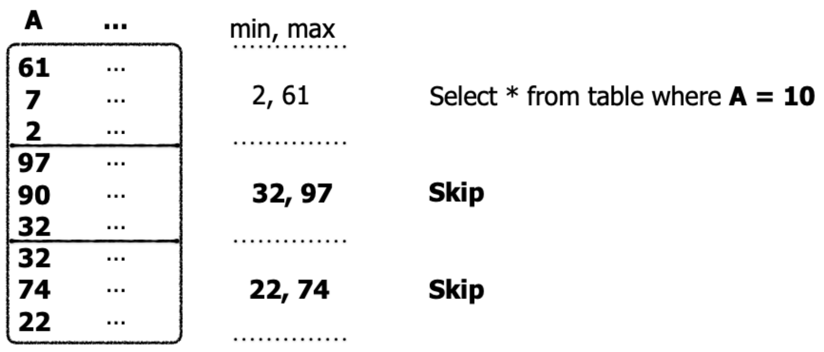
4.2 Trade-offs

- Eviction effectiveness
- CPU efficiency (thrashing is bad)
- Memory (the need for more space)

5 Indexing Databases

Consider `select * from table where A = "v"`. This can scan in $O(N/B)$ I/Os, but that's also a bit of a waste. So what better can we do?

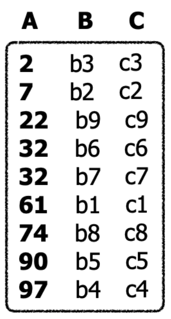
5.1 Zone Maps



Nothing but a scan. This might be ok for analytics, but it is not okay for many applications. Worst case $O(N/B)$ I/Os.

5.2 Sort Based on a Column of Interest

So, sort everything based on column A.



A	B	C
2	b3	c3
7	b2	c2
22	b9	c9
32	b6	c6
32	b7	c7
61	b1	c1
74	b8	c8
90	b5	c5
97	b4	c4

Search cost: $O\left(\log_2\left(\frac{N}{B}\right)\right)$ I/O, since we search over $\frac{N}{B}$ blocks. **Binary search is only done over the blocks for I/O.**

Every step, you are partitioning the data size by half. If you are on the last page, any additional searching in the last page only costs more CPU and not I/O. Constant work when looking through all pages but the last one.

The CPU cost for scans is $O\left(\log_2\left(\frac{N}{B}\right) + \log_2(B)\right) + O\left(\log_2(N)\right)$

The cost of updating: you have to sort everything. Move everything after the insertion by one slot, for a cost of $O(N/B)$.

The other disadvantage? You can only do this for one column. Not a great solution, we want to do better than this.

5.3 Binary Tree

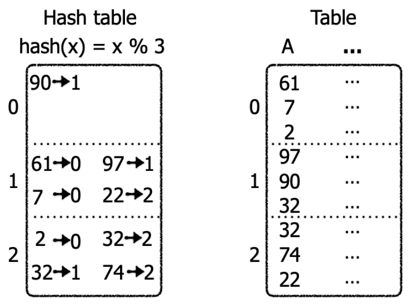
Use a binary tree to map each key to the page it resides in. Tree must be stored in storage due to size and for persistence.

I/O cost: $O\left(\log_2(N)\right)$. (Where the tree is stored in memory, but too large that it could be page swapped) Each node of the tree might be allocated in a random place in your memory space. Therefore, this many pages can be swapped in the worst case.

CPU cost: $O\left(\log_2(N)\right)$. In storage, having to do $\log_2(N)$ I/Os is a lot, and disk accesses are substantially worse than CPU. This latency can be seen by the user.

5.4 Hash Tables

Give up on sortedness and map entries to page, using a hash table. Chain on collision



Double when reaching capacity.

I can expect $O(1)$ I/Os on average assuming a good hash function and no key repetitions. But what are the downsides?

Downsides?

- Bad space efficiency after expansion
- Only equality comparisons

- Expansion leads to performance slumps

5.5 Extendible Hashing

How do we expand the hash table? Anyways:

- Less performance slumps due to expansion
- Avoid wasting 50% space after expansion
- Directory consumes memory if it is cached, or it requires one extra I/O per operation if in storage. **At least the directory is most likely to fit in memory since** it's based on how many pages there are.
- Range queries still not supported.

5.6 B-Tree

For every node that is read, it prunes the space by B instead of 2, where B corresponds to the number of entries that fit on a page.

Search I/O cost:

$$\log_B\left(\frac{N}{B}\right) = \log_B(N) - 1$$

So, you have $O\left(\log_B(N)\right)$ I/O. The fan-out is larger, which corresponds to the base of the log.

A block size is 4KB. In practice, B is going to be 10 or 100 or 1000. Suppose that $N = 2^{30}$ and $B = 2^{10}$, then $\log_B(N) = 3$. Only 3 disk I/Os need to be done.

How much CPU does this cost?

$$O\left(\underbrace{\log_B(N)}_{\text{pages searched}} \cdot \log_2 \overbrace{(B)}^{\text{Binary search}}\right) = O\left(\log_2 N\right)$$

The I/O cost is the bottleneck, but we care about both costs.

5.7 How Do I Insert?

Replay the slides.

Read I/Os per insertion: $O\left(\log_B(N)\right)$

(Amortized) Write I/Os per insertions: most insertions will not propagate upwards. It is a pretty rare occurrence that a split happens and the insertion propagates upwards.

- Every insertion updates 1 leaf
- One in every $O\left(1/B\right)$ insertions triggers leaf split
- One in every $O\left(1/B^2\right)$ insertions trigger a parent split
- And so on

These things do eventually happen, but most of the time they don't happen. If we add up the probabilities of these things happening, we end up with constant.

$$1 + B^{-1} + B^{-2} + B^{-3} + \cdots = O(1)$$

You may note:

On disk, read/write costs are symmetric.

On SSDs, they are different, so it's important to differentiate.

Supposing internal nodes reside in the buffer pool reside inside the buffer pool, how is that going to change the model?

Only one read is needed to read a leaf per lookup. If the B-tree contains $O(N)$ keys, then $O\left(\frac{N}{B}\right)$ are stored in memory. So, realistically it might as well be:

- $O(1)$ for both reads and write
- Where $O\left(\frac{N}{B}\right)$ keys are stored in memory
  - If I can't hold that much, I lose the $O(1)$ read speed

5.8 Index Tradeoffs

What is the downside of using indexes?

For every insertion, you need to make sure that every relevant index on the table is up to date. The more indexes you have, the more insertion to the indexes you need to do. It is good to use indexes, but you don't want to use them freely. Whether you should index or not depends on the workload.

The cost of looking up a range, the scan cost is $O\left(\log_B N + S\right)$ I/Os, where S is the entries in the range. For every entry in the range, you may need to make one random I/O into my table per entry.

So is there a solution?

You can optimize the S by storing the data in the b-tree. Of course, this means you can only index one column. This is called a clustered index. But you can have at most one clustered index per table, and I would not want to duplicate the entire table.

Now the cost for a scan is:

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

At least we can have multiple unclustered indexes in addition to one clustered index.

So at least, insertion, update, and delete costs at least $O(1)$ write I/O. Can we reduce this more towards $O\left(\frac{1}{B}\right)$?

6 Log Structured Merge Tree

B-trees were invented in 1970, and LSM trees were invented in 1996. What took so long? With SSDs, writes are more expensive than reads. The declining costs of storage allow us to store more data cheaply, hence application workloads become more write-intensive.

Lowering the cost of making an insertion comes with an expense.

6.1 Basic LSM Tree

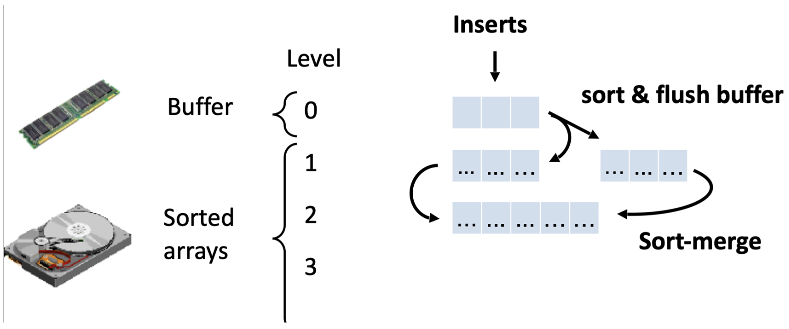
You have a buffer in memory, where all modifications to the structure happen.

The buffer is fixed sized, so when the buffer is filled above, it is flushed as a sorted file.

Then sort-merge multiple flushed buffers to eliminate duplicates.

Design principles:

1. Optimize for insertions by buffering
2. Optimize for lookups by sort-merging SSTs



Runs at level 1 start off as the size of the buffer.

Two files in a given level, sort-merge them and put them in the next larger level.

An update is made out-of-place through an insertion into the buffer.

Only an entry's most recent version should be returned to the user during a query.

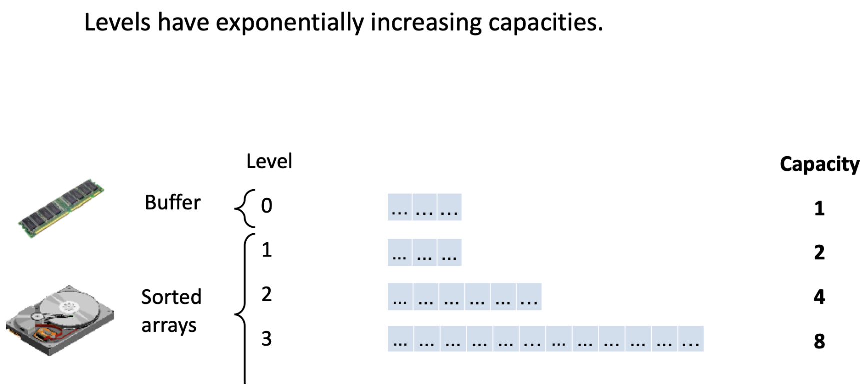
This means there may be multiple entries across different levels.

Going through each of them is going to be expensive.

Can we be a bit smarter?

A point query searches the LSM-tree from smaller to larger levels. It stops when it finds the first entry with a matching key. Entries with a matching key at larger levels are older and thus superseded.

Capacity – upper bound for run size



6.2 Insert Tombstone (Support Deletes)

All entries will have a bit that is a tombstone.

If the most recent entry is a tombstone then return nothing.

Compaction accounts for tombstone entries.

When a tombstone reaches the bottom level of the tree, you can remove the tombstone since it won't be masking anything else.

If the most recent version of an entry is a tombstone, it is considered deleted.

Are we going to accumulate too many tombstones? No. Compactions will eventually span the entire data and reach the largest level, and all the garbage will be removed.

6.3 Point Query Costs

What is the number of runs we have to query?

What are the number of levels and the cost querying these levels?

How many levels to search: $O\left(\lg\left(\frac{N}{P}\right)\right)$, where P is the no. of entries in the buffer.

Cost of searching each level? Note that each level has different capacities. The cost of searching the largest level is $O\left(\lg\left(\frac{N}{B}\right)\right)$

So, the total search cost is, in the worst case (we have to go all levels):

$$O\left(\log_2\left(\frac{N}{P}\right) \cdot \log_2\left(\frac{N}{B}\right)\right)$$

Such as looking for an entry that has never been added in the first place, which means you'll have to search through all levels to prove that it doesn't exist.

If $P = B$, then $O\left(\log_2\left(\frac{N}{B}\right)^2\right)$

We can do slightly better by structuring each file as a static B-tree. Since this is really a tree, we can just turn it into a B tree and the total search cost decreases to:

$$P\left(\log_2\left(\frac{N}{P}\right) \cdot \log_B(N)\right)$$

In which $\log_B(N)$ realistically never exceeds 4.

6.4 Scan (Range) Queries

Looking for all keys between (4, 6), range inclusive. Returns the most recent version of each entry in the range across the entire tree.

Scans cost

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

6.5 Modifications to LSM Trees

- **Leveled**: faster reads, slower writes. As $T \rightarrow \frac{N}{B}$ we get a sorted array.
- **Tiered**: faster writes, slower reads. As $T \rightarrow \frac{N}{B}$ we get a write-only log.

6.6 Runtime Table / Summary

Cost Metric	Append only	Sorted list	B-tree
Query	N/B	$\log\left(\frac{N}{B}\right)$	1
Ins – R	0	N/B	1
Ins – W	$1/B$	N/B	$1 + GC$
Memory	B		N/B

Cost Metric	Basic LSM	Tiered LSM	Leveled LSM
Query	$\lg\left(\frac{N}{P}\right)$	LT	L
Ins – R	$\frac{\lg\left(\frac{N}{P}\right)}{B}$	$\frac{L}{B}$	$\frac{LT}{B}$
Ins – W	$\frac{\lg\left(\frac{N}{P}\right)}{B}$	$\frac{L}{B}$	$\frac{LT}{B}$
Memory	N/B	N/B	N/B

7 Bloom Filters, MONKEY, DOSTOEVSKY

Probabilistic set. Where:

- You can add items to the set. You cannot remove items.
- You can check for inclusions (`in`)
- Returns **negative** with full certainty or **positive** with some uncertainty (TBA – probably ε)

A bloom filter is a block of memory that has:

- m bits
- k hash functions

To add an item:

- For each hash function
  - Hash the item
  - Set bit at corresponding memory to 1 if not already

To query:

- Run all the hash function on the item and check if **all** the corresponding bits are set

If any bit is 0, return a negative.

If every bit is a 1, it's either because an element was in the set, or a combination of other elements happened to set those bits.

7.1 Bits Per Entry

M is the bits per entry. The total bits in the bloom filter, $m = MN$.

7.2 Using Bloom Filters within LSM Trees

For each "run" of data, there is a bloom filter. This helps speed up queries, especially negative ones. We note the following:

- The more memory for a bloom filter, the less likely false positives are going to happen

So, how many hash functions should we use?

- One hash function is too few: a false positive occurs every collision
- Adding more hash functions exponentially decreases the FPR
- There is a limit: too many hash functions will end up increasing the FPR
- There is an optimal number of hash functions to use depending on the no. of bits per entry, M.

$$\text{optimal \# hash functions} = \ln(2) \cdot M$$

Assuming the optimal number of hash functions:

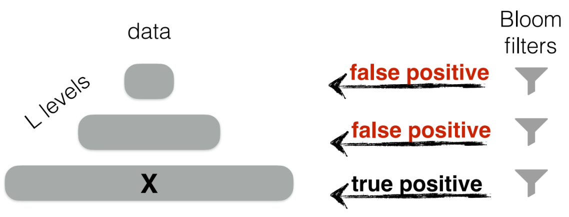
$$\text{FPR} = 2^{-M\ln(2)}$$

Therefore, operation costs, in terms of memory accesses, for a bloom filter, depending on M (bits per entry):

- **Insertions** = $M\ln(2)$
- **Positive Query** = $M\ln(2) = k$
- **Avg. Negative Query** = $1 + \frac{1}{2} + \frac{1}{4} + \ldots = 2$
  - Fraction of ones in filter is 0.5 with optimal number of hash functions
  - This goes for all negative queries (whether or not it's a TN or FP)
- **False positive rate** = $2^{-M\ln(2)}$

7.2.1 Filter Overhead for Basic Accesses

The **absolute** worst case is when all the bloom filters return false positives (to simplify things, let's just use the fact that $k \in O(M)$):



Worst case: $O(ML)$

Average worst case: All filters except for the last one return negative costs $O(M + L)$, where the M is for the positive query in the last bloom filter

7.3 Construction Contract – Choosing M

Known specifications in advance:

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

If you want this, allocate a filter with

$$N \ln(2) \log_2\left(\frac{1}{\varepsilon}\right) \text{ bits}$$

Where N elements are inserted using $-\frac{\ln(\varepsilon)}{\ln(2)}$ hash functions

This **guarantees** a FPR of ε

7.4 Handling Exponential Growth

No. of rows scale logarithmically: $L = O\left(\log(N)\right)$

Row count grows exponentially: $N \in O\left(2^{\text{time}}\right)$

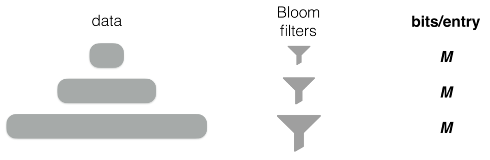
Results in linear scaling: $O\,(\text{time})$

Can this be done better?

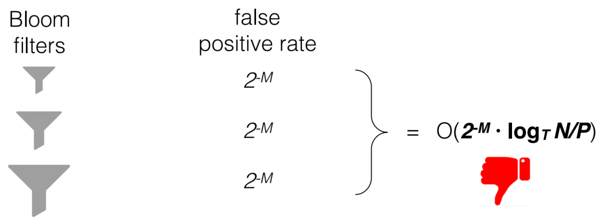
7.5 MONKEY (Optimizing GETS)

Stands for optimal navigable key-value store.

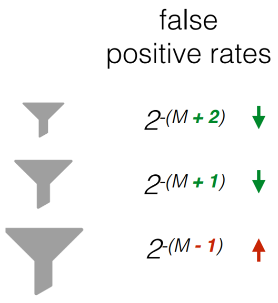
When dealing with data in an LSM tree, since the runs are longer the lower the level, the bloom filters scale up as well.

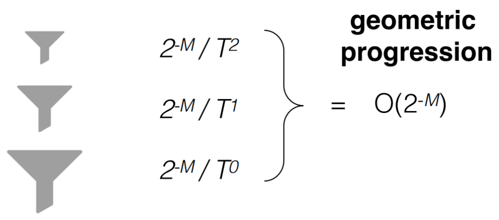


Each bloom filter has a FPR of $O\left(2^{-M}\right)$. **The same FPR per bloom filter.** In the worst case, when we have to search through every bloom filter, the overhead is $O\left(2^{-M}\log_T\left(\frac{N}{P}\right)\right) = O\left(2^{-M}\cdot L\right)$. **Not good.** False positives result in an additional I/O.



What we can do instead: reallocate and increase the bits per entry for the smaller bloom filters. This changes the false positive rates exponentially.





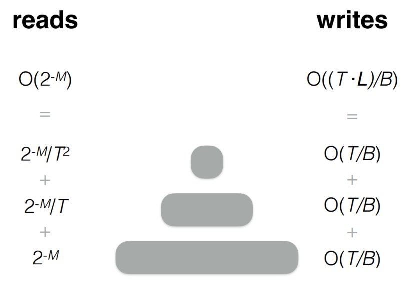
This results in a faster worst-case, $O\left(2^{-M}\right)$ instead of $O\left(2^{-M}\log_T\frac{N}{P}\right)$.

This does change the absolute worst case (all positives) to $O\left(ML + L^2\right)$ but the average worst-case remains the same: $O(M + L)$.

Summary: MONKEY reduces the GET I/O cost from $O\left(2^{-M}\cdot L\right)$ to $O\left(2^{-M}\right)$ – remember **false positives result in a wasted I/O**

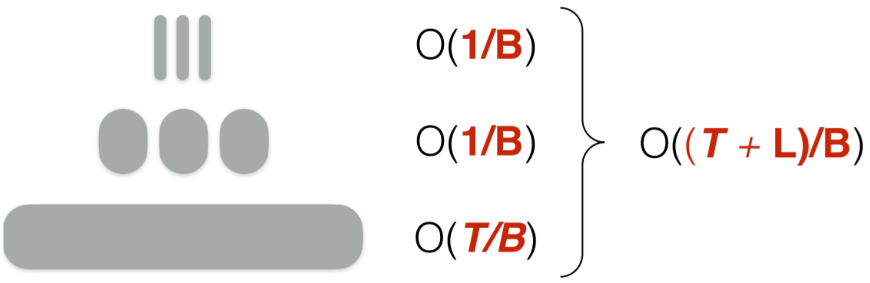
7.6 Dostoevsky (Optimizing Writes)

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.



Therefore, to speed this up, we want to make the writes lazy on the upper levels. Therefore:

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



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

7.7 Summary

	Before	After	Using
Get I/O cost	$O\left(2^{-M} \cdot L\right)$	$O\left(2^{-M}\right)$	MONKEY
Insert I/O cost	$O\left(\frac{TL}{B}\right)$	$O\left(\frac{T+L}{B}\right)$	DOSTOEVSKY

Costs assuming leveling

8 LEGACY USELESS INFO (TODO: REMOVE)

Many DB operations are selective and random. We've looked at B-trees, and how if I'm issuing queries and insertions to a B-tree, I end up with random reads and writes to my storage. It's not great:

Why random R/W are bad:

- Hard drives: mechanical latency
- SSD: 4KB access required for R/W & garbage collection

We then looked at LSM trees. It's a widely used data structure, and many large DBs use it.

A buffer takes in all modifications you make. When the buffer fills up, it gets flushed into storage, and merge-sorts happen. Each run has exponentially increasing capacities. We search, always, from the smaller to larger levels, in order from latest to oldest. If a tombstone is found, the key DNE. The number lf levels is $\log_T\left(\frac{N}{P}\right)$, where T is the size ratio, P is the no. of entries that fit in each buffer, and N is the no. of rows.

- Tiering builds multiple sorted components until capacity
  - Better for writing
- Leveling builds up the next sorted component
  - Better for reading

If you set the size ratio to 2, these designs converge in terms of their behaviors and performance cost.

If you set the size ratio to $T = \frac{N}{P}$ that the first tier never runs out:

- Tiering ends up being a write-only store
- Leveling ends up being a sorted array

To query each one, you structure each one of them as a static b-tree. You can store the internal nodes of the B-tree in memory, so most of the search costs will happen in memory. So, not a lot of I/Os.

An additional optimization that can be done are with bloom filters.

8.1 What is a Filter?

If you have a bunch of data in storage, what if you have a filter in memory that will tell you if the key exists? This will help avoid the need of going to storage. It does a lossy compression of the keys in storage to answer the set membership queries.

Since bloom filters are lossy:

- **False positives are possible**

So:

- If querying for a key that DNE, false positive probability is ε and true negative is $1 - \varepsilon$.

8.2 What is a bloom filter?

A bloom filter is just a bitmap with k hash functions. k is a parameter.

In order to insert a key, you hash it using the k functions to k random bits. This can flip a 0 to 1 or keep a 1.

When checking: you run the hash again. If the key is inserted, all the keys corresponding to the bits will be a 1 as well.

However, if you query for a key that doesn't exist. If at least one of the bits that the key maps to is a 0, you know that the key does not exist, for sure.

You can also have a situation where you query for a non-existing key that doesn't exist, and they all map to all bits that map to 1., This is effectively a hash collision, and in this case, you get a false positive.

How do we use this in the context of LSM tree, for every run of the LSM tree in storage, we will use a bloom filter in memory, one for each run.

Before we query each sorted run, we check the bloom filter: does this key exist?

- Negative
  - Search halted
- False positive
  - Search continues
- True positive
  - We find what we're looking for and the query terminates

Tradeoff:

- More memory $\longrightarrow$ fewer false positives.

How do we control that tradeoff?

Memory vs. I/O cost?

8.3 What should I set k?

What happens when I only use one hash function? What could be said about the false positive rate?

- Every time a collision occurs, you get a false positive. What happens when you start increasing the number of hash functions? Probabilistically, it becomes less likely that for every bit that I check, all of them will be mapped to a 1.
- As I check more bits, the probability that all of them will be set to 1, due to hash collisions goes down.
- Chance of bad coincidences being 1? Likely. 2 at the same time? The FP drops as you increase the number of hash functions…

What if you use too many hash functions?

- Too many bits get flipped to 1
- Again, too many false positive

In summary:

- One hash function is too few
- As I increase the number of hash functions, the FP rate will initially drop
- But doing that too much causes the FP rate goes up too much.

This is an optimization problem that you can solve.

THE OPTIMAL NUMBER OF HASH FUNCTIONS IS

$$k = \ln(2) \cdot M$$

Where M is the number of bits per entry you are using for the filter.

If you have a filter of 100 entries, and M is set to 10, then the bloom filter will hold 1,000 entries. Therefore M is in terms of bits per key. The total number of bits you end up using is $M \cdot N$.

Assuming you want to vary the bits per entry, you can achieve different false positive rates.

FPR = $2^{-M \ln(2)}$

8.4 Operation Costs

In terms of random locations I have to access. Or in this case how many times do I have to call the hash function?

How many bits do I have to flip for an insertion? That depends on the number of hash functions. This means that

$$\text{Insertion cost} = M \ln(2) = k$$

What is the cost of a positive query?

$$\text{Positive query} = M\ln(2) = k$$

Suppose that I query for a key that doesn't exist. On average, how much does a negative query cost?

The fraction of one sin the filter is 0.5 with the optimal number of hash functions. So, if you allocate a bloom filter with the optimal number of hash functions, then a half of the bits in your filter will be 0 and half of them will be 1s.

$$\text{Average negative query} = 1 + \frac{1}{2}\left(1 + \frac{1}{2} \cdot \ldots\right) = 2$$

The false positive rate is $2^{-M\ln(2)}$

Now, let's apply the costs.

8.5 Worst Case Overall

For a **basic** LSM tree, how many memory accesses do I have to make? Why is the worst-case too pessimistic?

The worst worst-case? If all the filters positives. And the target key is in the largest level. Then:

- You pay the cost of all positive queries, which is L times since you have L levels.

Hence worst case is $O(M \cdot L)$

So, what is the average worst case (where we all get negatives)? Most of the time, bloom filters will return negative. So, if the key is at the last function and the bloom filters behave correctly, average worst case is, in terms of memory access count just for accessing the bloom filters:

$$O(M + L)$$

8.6 Construction Contract

I want to construct the bloom filter.

- How many entries am I going to insert? I have to know that up front
- What is the false positive rate I want to have? I have to know that before I allocate it

For N and ε I'd allocate a filter with

$$N \ln(2) \log_2 \left(\frac{1}{\varepsilon}\right) = N \cdot M \text{ bits}$$

I will then insert N elements using

$$-\frac{\ln(\varepsilon)}{\ln(2)}$$

We guarantee that FPR is ε. Note that setting M also sets ε since they are dependent on one another.

8.7 Exponential Data Increase?

If data increases exponentially with time, we probably can't handle this.

8.8 Monkey (Optimal Navigable Key Value Store)

We want as many negatives as possible

False positives are bad

True positives just occur. You can't do anything about that.

Hence we want to minimize the no. of false positives.

The simplest ways of allocating the filters is to use the same number of bits per entry.

If there are M bits per entry, then the size of the filter increases with proportion to the size of the data run.

The false positive rate is $2^{-M\ln(2)}$

We can now add all of these up, giving me the worst case number of I/Os, that on average, don't do any useful work.

$$O\left(2^{-M\ln(2)}\log_T \frac{N}{P}\right)$$

The cost of a false positive is 1 I/O. The average number of I/Os I do, are the number of false positives. Unsurprisingly, it's the false positive rate. And each level you check, you will have to do the I/O there at least once (internal nodes are stored in memory).

Apparently, having a memory footprint of M bits of entry, this is not the best thing you could have hoped for.

If you set the same number of bits per entry for all the filters, the filter at the largest level is going to take up the memory footprint. And yet, I use it the least.

So, instead, for higher levels, I assign more bits of entry there. It decreases the false positive rates for the higher levels.

It increases for the lower levels, but the FPR decreases exponentially as M goes up. Therefore, overall FPR goes down. By a lot.

FPR proportional to the size of each run.

Doing this shaves off some of the cost: $O\left(2^{-M}\right) < O\left(2^{-M}\log_T\left(\frac{N}{P}\right)\right)$

So, as the data size grows, query costs stay stable. As there are more levels, you have more false positives.

8.9 Dostoevsky (Space time optimized evolvable scalable key value store)

To optimize write costs.

What are the costs of reads and writes?

For point queries, we want FPR to decrease the higher the levels.

Most of the costs in the WC is derived from the largest level.

Assuming queries to non-existing entries.

Now, about write costs: how is WA derived from across all different levels? Let's see how much each entry participates across compaction.

You can cont how many compaction operations happen in a level until it reaches capacity.

The number of times each entry participates in a merge corresponds to $O\left(\frac{T}{B}\right)$. So, the contribution to I/O costs for each element is $O\left(\frac{TL}{B}\right)$

So the problem, is write cost are derived equally from all other levels. What can we do to fix this?

All of the compaction work that is done on the smaller level is only improving our query costs by a small amount. So why don't you make compaction lazier at smaller levels? That's the core of the opitimization.

Let's have lazy merging at smaller levels and do greedy merging at the larger level.

So:

- Tier at smaller levels
- Level at larger levels

Hence, at largest levels, compact everything as something comes in.

Why is that good?

This is good, because we can otimize bloo filters for this purpose. The FPR will still be exponentially ecreasing for smaller levels, but

We get a larger win in writes.

Smaller levels have a write cost of $O\left(\frac{1}{B}\right)$

Larger levels have the cost of $O\left(\frac{T}{B}\right)$

Hence the cost in total is $O\left(\frac{T+L}{B}\right)$

Query cost about the same.

8.10 So

Applying MONKEY and the other one can improve read costs and write costs. There is no tradeoff here.

With these two schemes, we can better scale the cost of reads and writes. We can tackle the log filter.