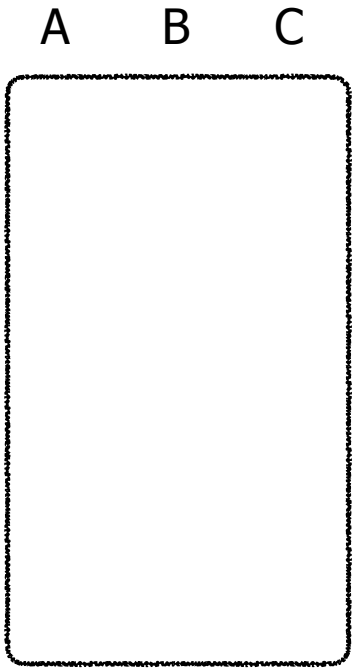
Write-Optimized Indexing

Niv Dayan

31 January, 2023

We will begin at 2:10 pm

We are trying to optimize the following kinds of queries



select * from table where A = “…”

select * from table where B > “…” and B < “…”

Let’s reconsider four baselines

Append-Only Table

Sorted Table

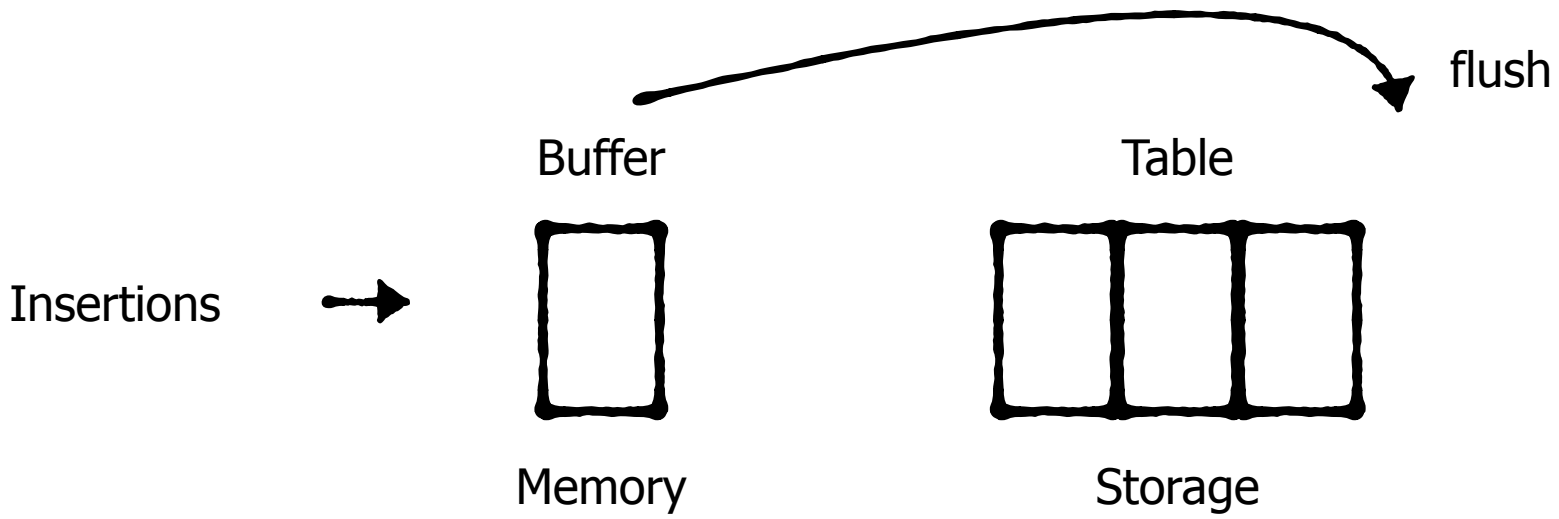
Extendible Hashing

B-tree

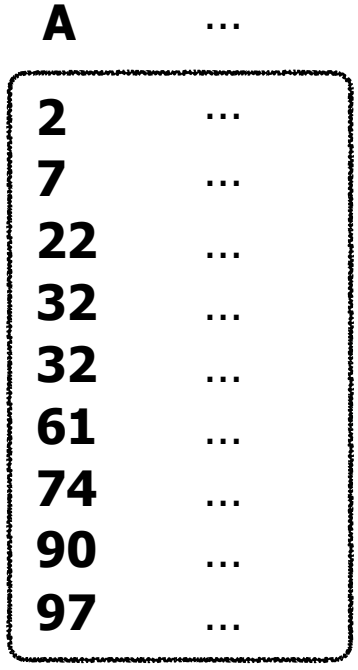
Append-Only Table

O(N/B) for any selection query

O(1/B) for insertions



Sorted Table



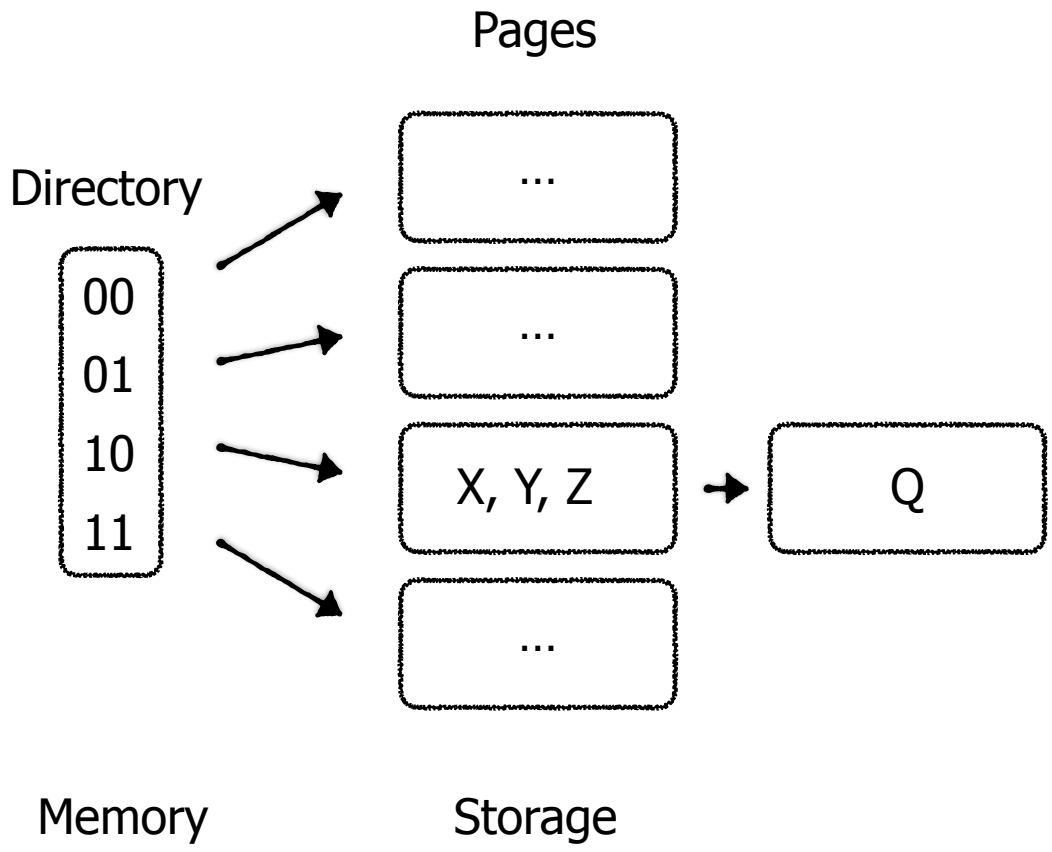
Binary search: $O(\log_2 N/B)$ I/O

Update/insert/delete: O(N/B) read & write I/O

Extendible Hashing

A directory maps pages in storage with a given hash prefix

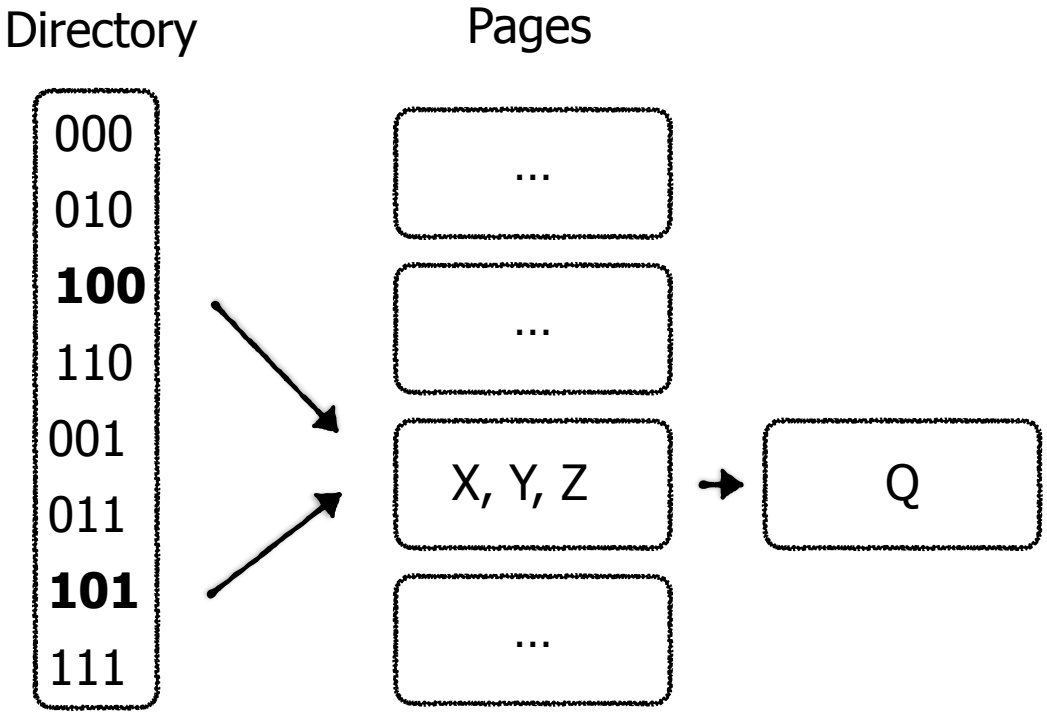
Handle overflows via chaining



Extendible Hashing

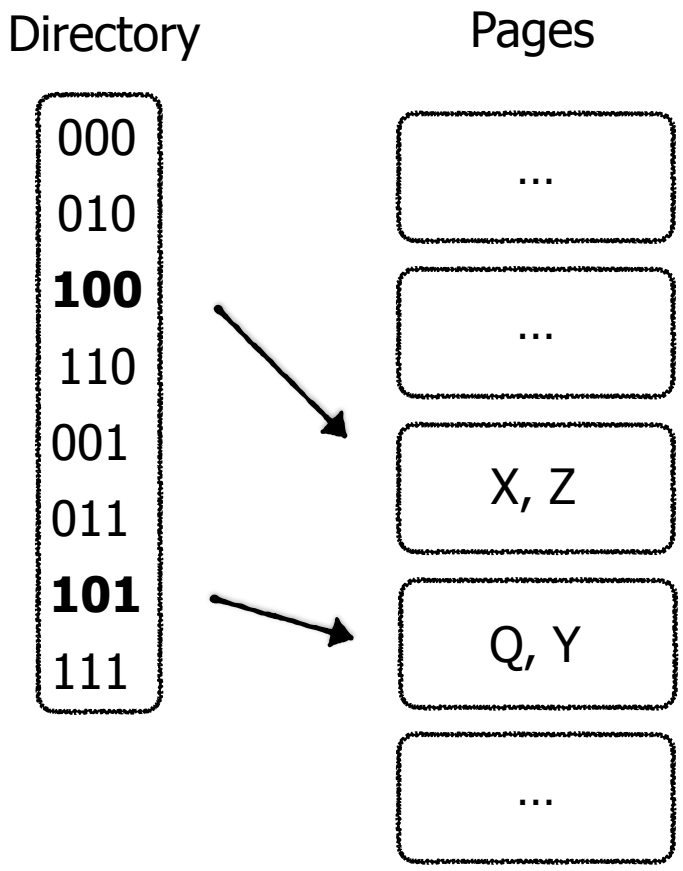
When we reach capacity, double directory size.

New directory slots still point to previous pages



Extendible Hashing

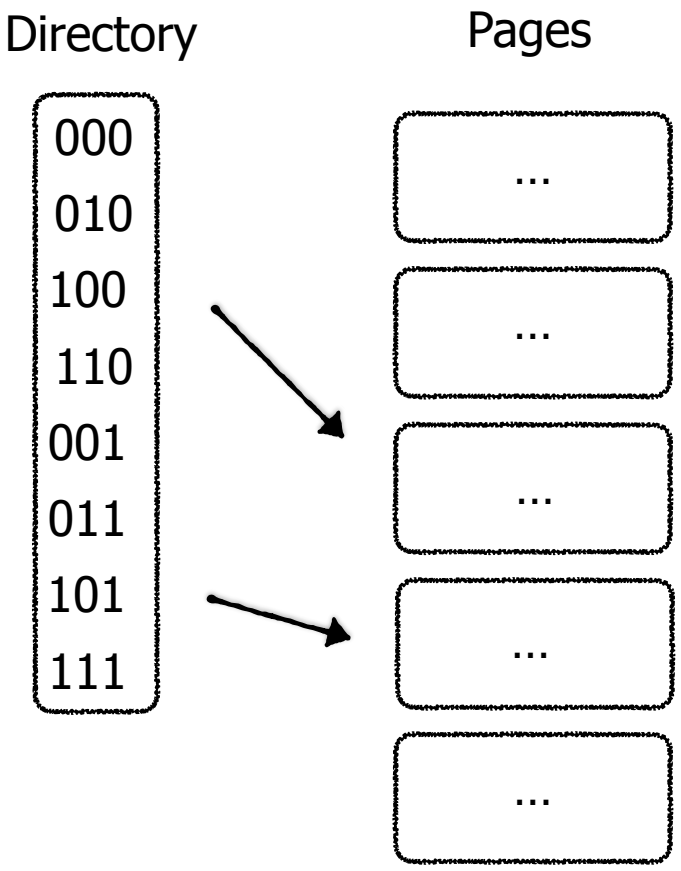
Split one overflowing bucket at a time by rehashing entries.



Extendible Hashing

Query cost: O(1) read I/O

Insertion cost:
- O(1) read I/O
- O(1) write I/O

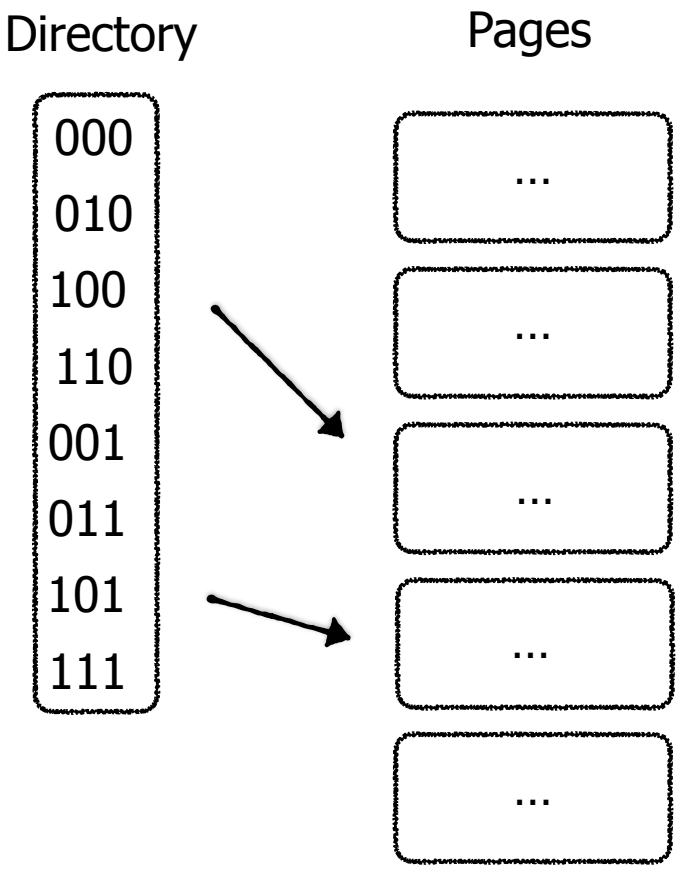


Extendible Hashing

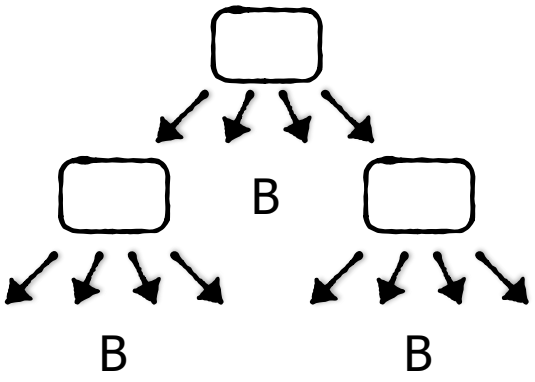
Query cost: O(1) read I/O

Insertion cost:
- O(1) read I/O
- O(1) * **GC** write I/O

Random writes to storage entail SSD garbage-collection



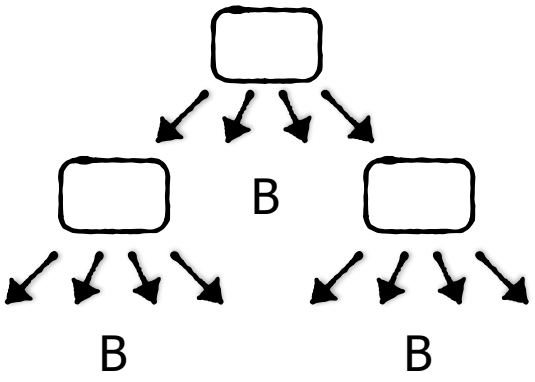
B-Tree



Each node has B children. This allows pruning by a factor of B in each level

Can issue random writes to storage leading to SSD garbage-collection

B-Tree



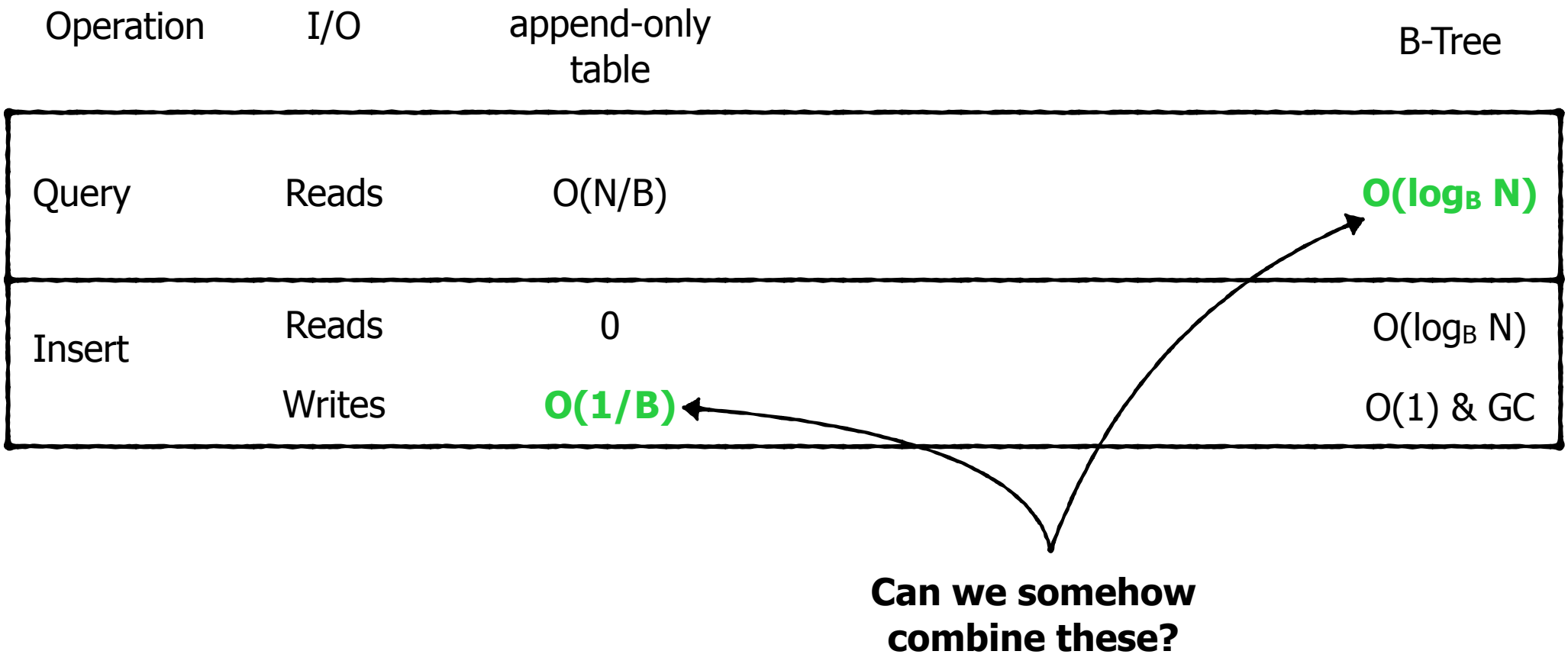
Each node has B children. This allows pruning by a factor of B in each level

Can issue random writes to storage leading to SSD garbage-collection

Query: $O(\log_B N)$ read I/O

Update/insert/query: $O(\log_B N)$ read I/O
& $O(1)$ * GC write I/O

Operation	I/O	append-only table	Sorted File	Extendible Hashing	B-Tree
Query	Reads	O(N/B)	$O(\log_2 N/B)$	O(1)	$O(\log_B N)$
Insert	Reads	0	O(N/B)	O(1)	$O(\log_B N)$
	Writes	O(1/B)	O(N/B)	O(1) & GC	O(1) & GC


Operation
I/O
append-only table
B-Tree
Query
Reads
O(N/B)
O(log_B N)
Insert
Reads
0
O(log_B N)
Writes
O(1/B)
O(1) & GC
Can we somehow combine these?

The Log-Structured Merge-Tree

B-tree is invented — 1970

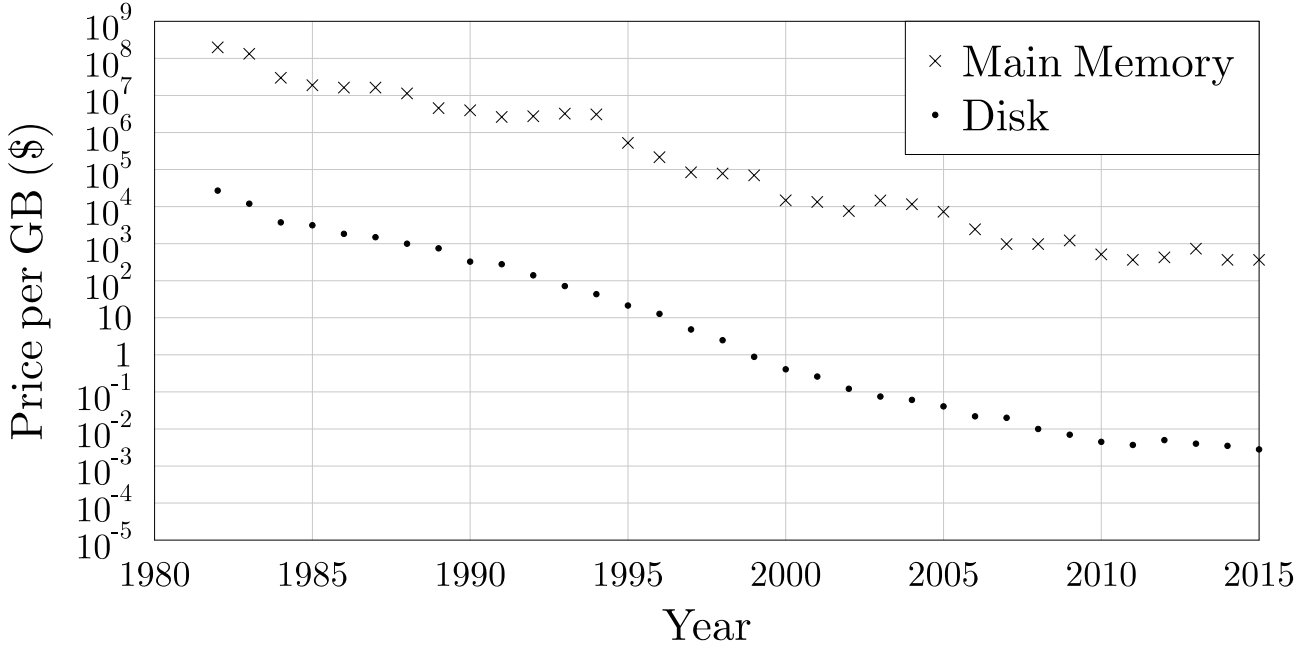
LSM-tree is invented — 1996

Google's BigTable adopts LSM-tree — 2005

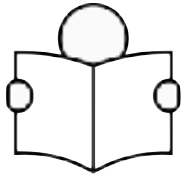
LSM-tree is widely used — Today

Why was it not invented and used sooner?

The declining costs of storage allow us to store more data cheaply.
Hence, application workloads are becoming more write-intensive.
This drives a need to optimize for data ingestion.



Leveled LSM-tree



Basic LSM-tree

Tiered LSM-tree

Basic LSM-tree

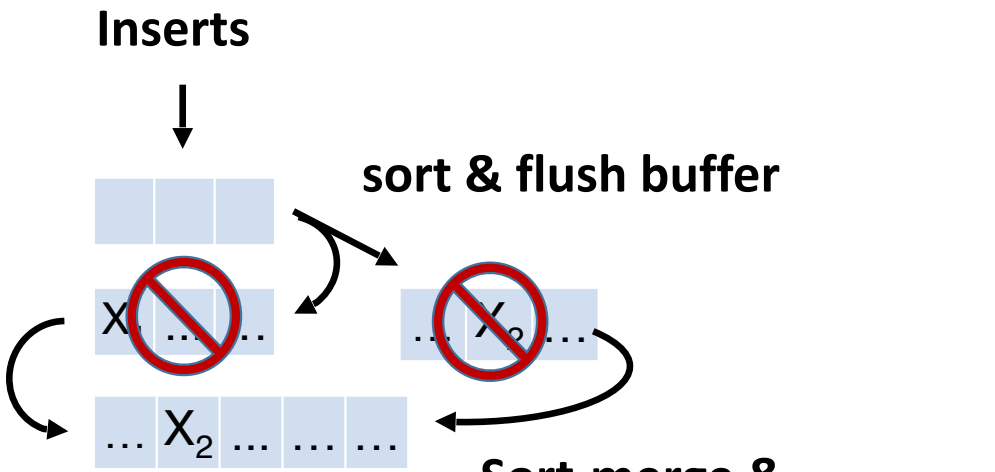
Design principle #1: optimize for insertions by buffering

Design principle #2: optimize for lookups by sort-merging SSTs

Level

Buffer 0

Sorted arrays 1 2 3



Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

2

3

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

2

3

inserts

4 6 9

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

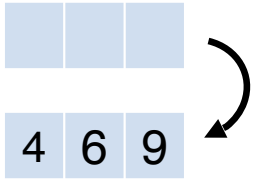
0

1

2

3

inserts



sort & flush buffer

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

2

3

inserts

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

2

3

inserts

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

1

2

3

inserts

sort & flush buffer

4 6 9

3 4 8

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

0

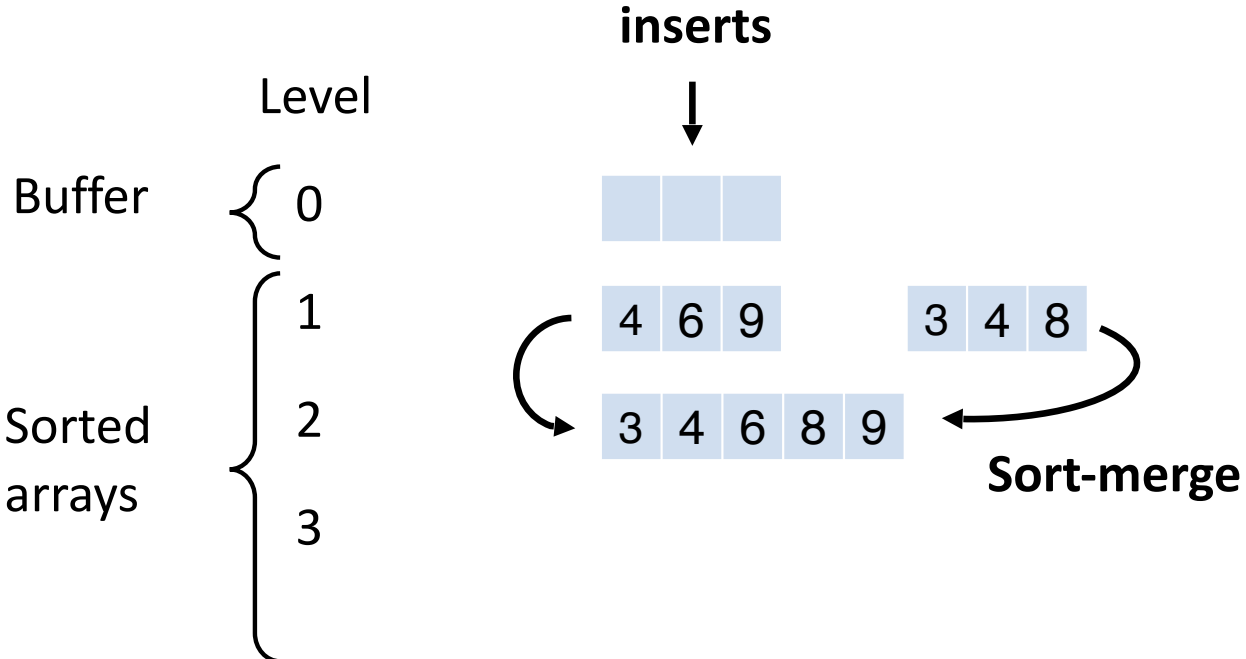
1

2

3

inserts

Basic LSM-tree – Example



Basic LSM-tree – Example

Buffer

Sorted arrays

Level

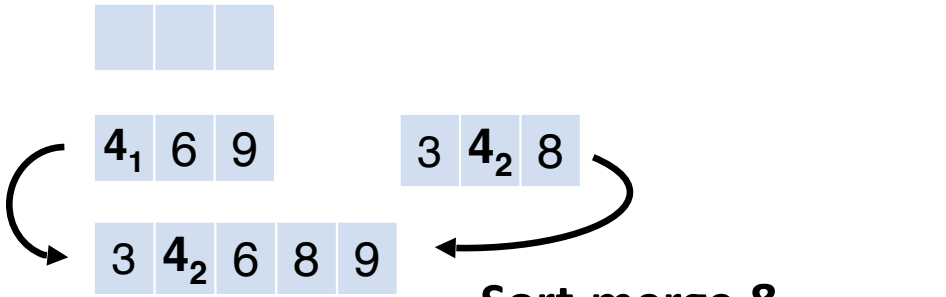
0

1

2

3

inserts



Sort-merge &
Eliminate duplicates

Basic LSM-tree – Example

Buffer

Sorted arrays

Level

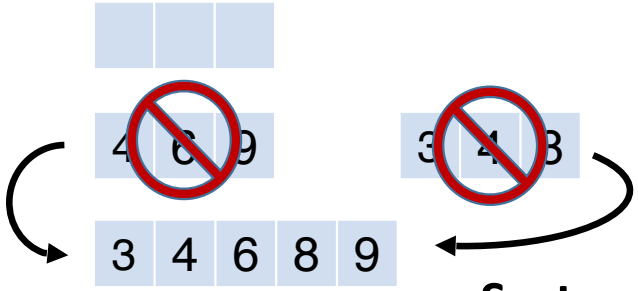
0

1

2

3

inserts



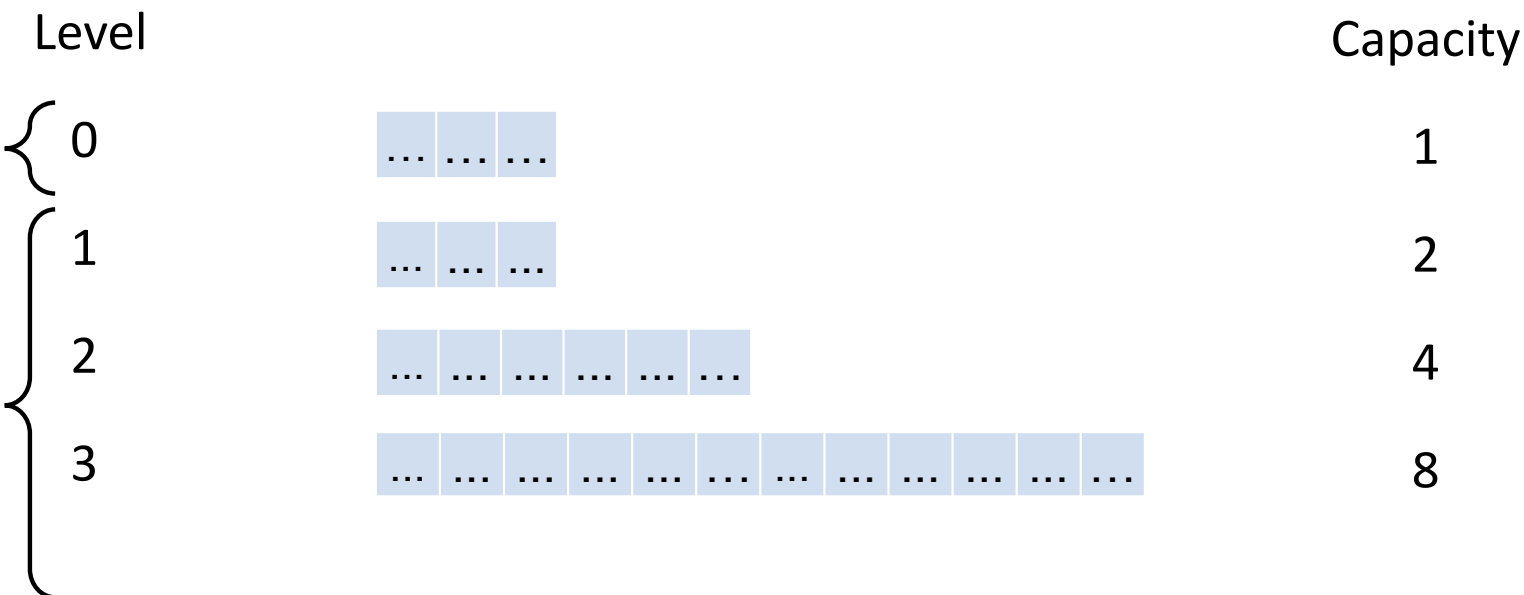
Sort-merge &
Eliminate duplicates &
Discard original arrays

Basic LSM-tree

Levels have exponentially increasing capacities.

Buffer

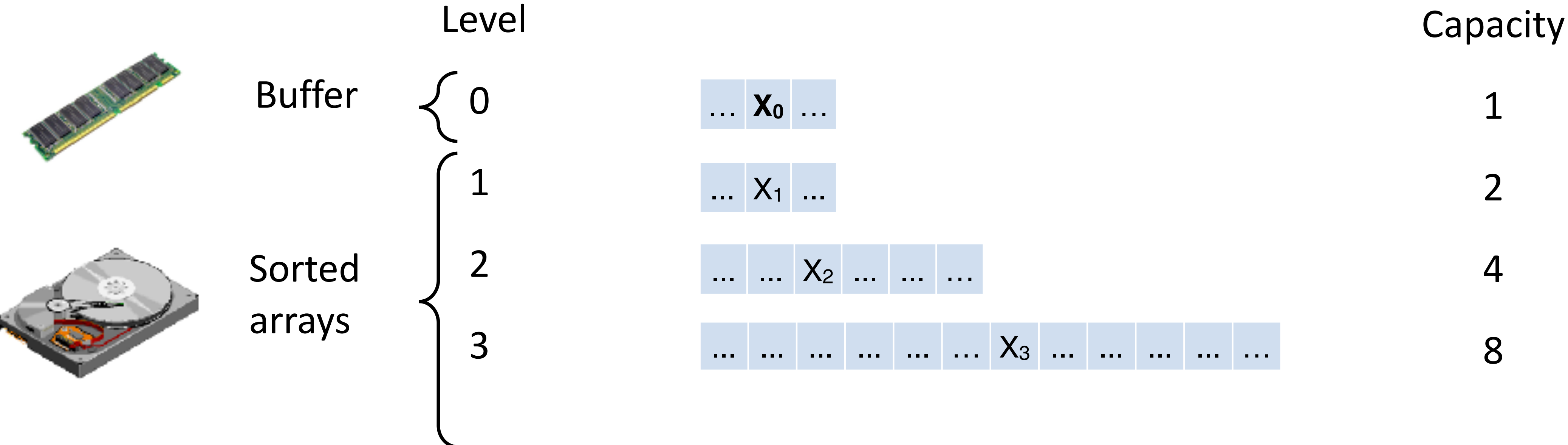
Sorted arrays



Basic LSM-tree - Updates

Can be made out-of-place through an insertion into the buffer

Only the most recent version counts and should be returned to users during queries.



Basic LSM-tree - Deletes

Delete an entry by inserting a tombstone

If the most recent version of an entry is a tombstone, it's considered deleted. (X is deleted by Y isn't in this example)

Basic LSM-tree – Get Queries

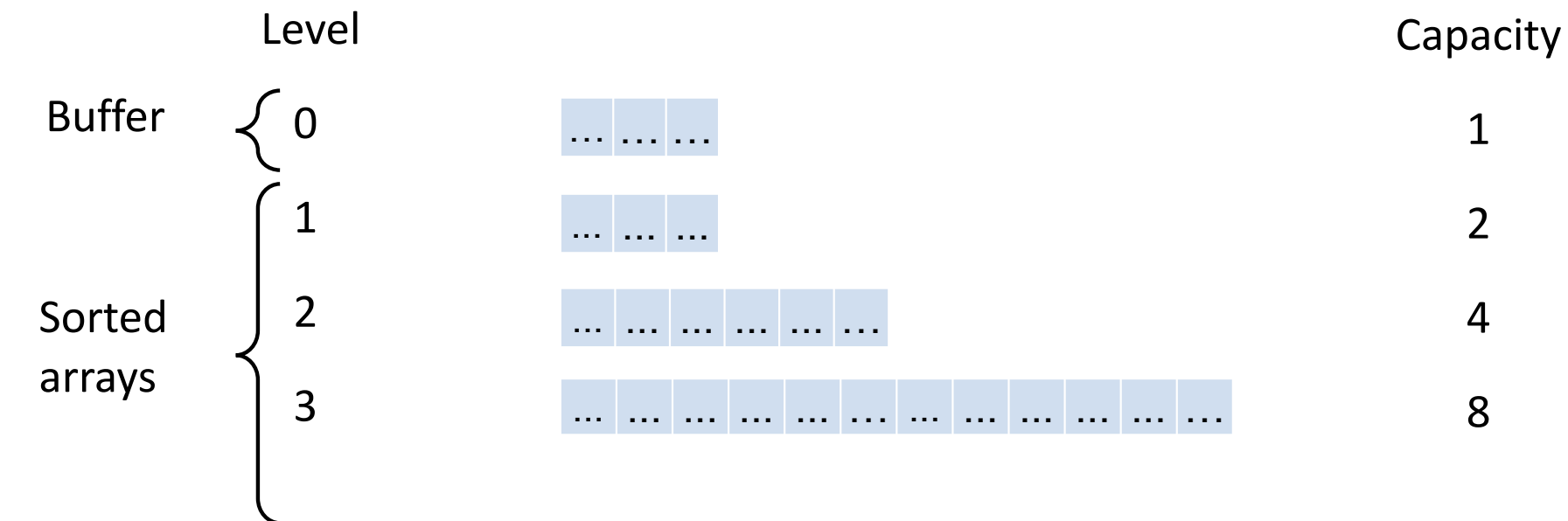
A get query searches the LSM-tree from smaller to larger levels. They stop when they find the first matching entry, as entries at larger levels are older and thus superseded.

Basic LSM-tree – Get Queries Cost

How many levels to search? $O(\log_2 N/B)$

Cost per searching each level? $O(\log_2 N/B)$

Total search cost: **$O(\log_2(N/B)^2)$**



Basic LSM-tree – Get Queries Cost

We can do slightly better by structuring each file (SST) as a static B-tree.
How would this impact search cost?

Total search cost: $O(\log_2(N/B) * \mathbf{\log_B N})$

SST static B-tree structure

Basic LSM-tree – Scan Queries

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

Basic LSM-tree – Scan Queries

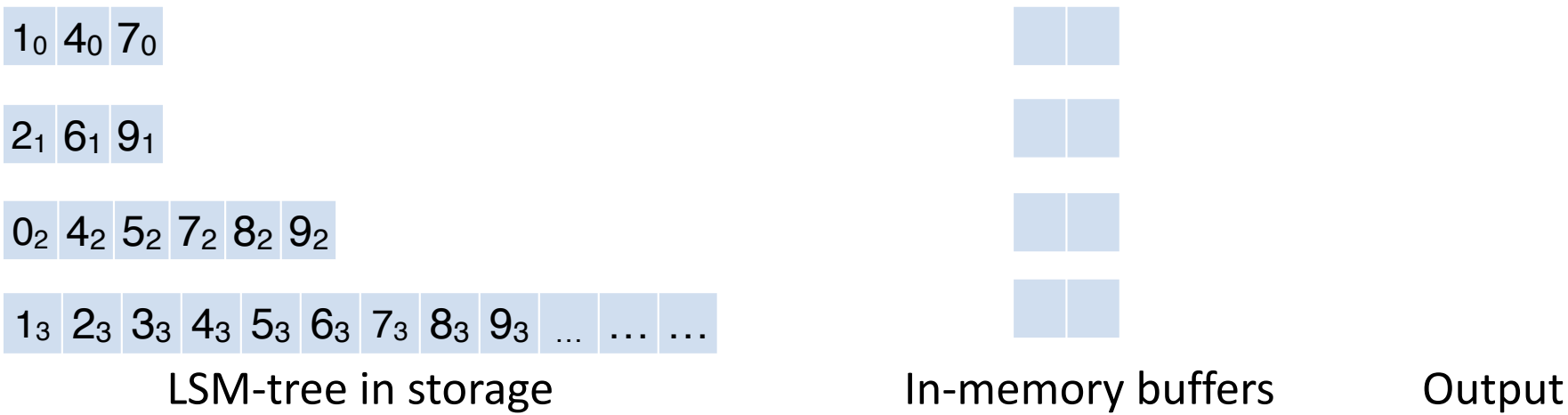
Return most recent version of each entry in the range across entire tree.

e.g., Scan(4, 6)

Expected output: $4_0\ 5_2\ 6_1$

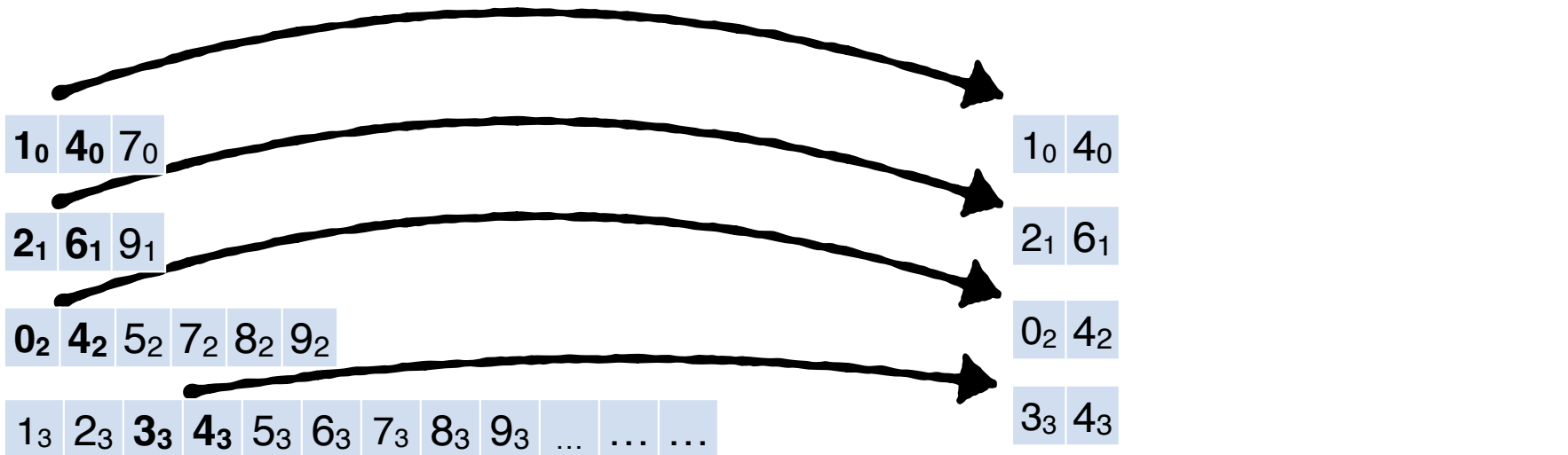
Basic LSM-tree – Scan Queries

1. Allocate an in-memory buffer (>1 page) for each level



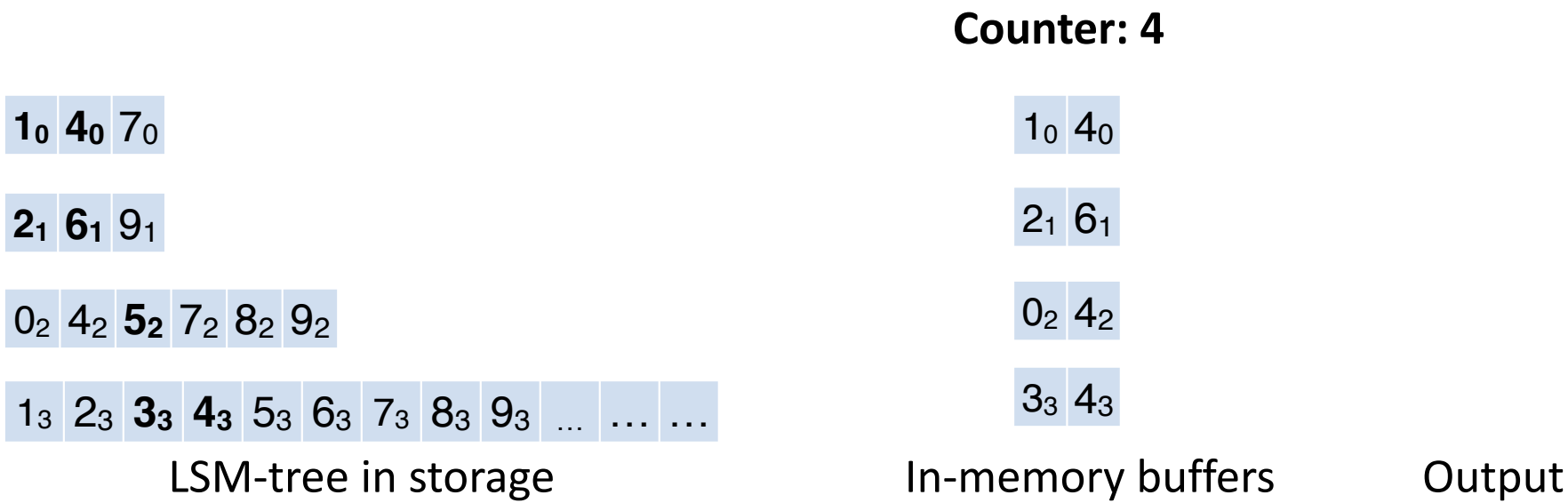
Basic LSM-tree – Scan Queries

1. Allocate an in-memory buffer (>1 page) for each level
2. Search for start of key range at each level



Basic LSM-tree – Scan Queries

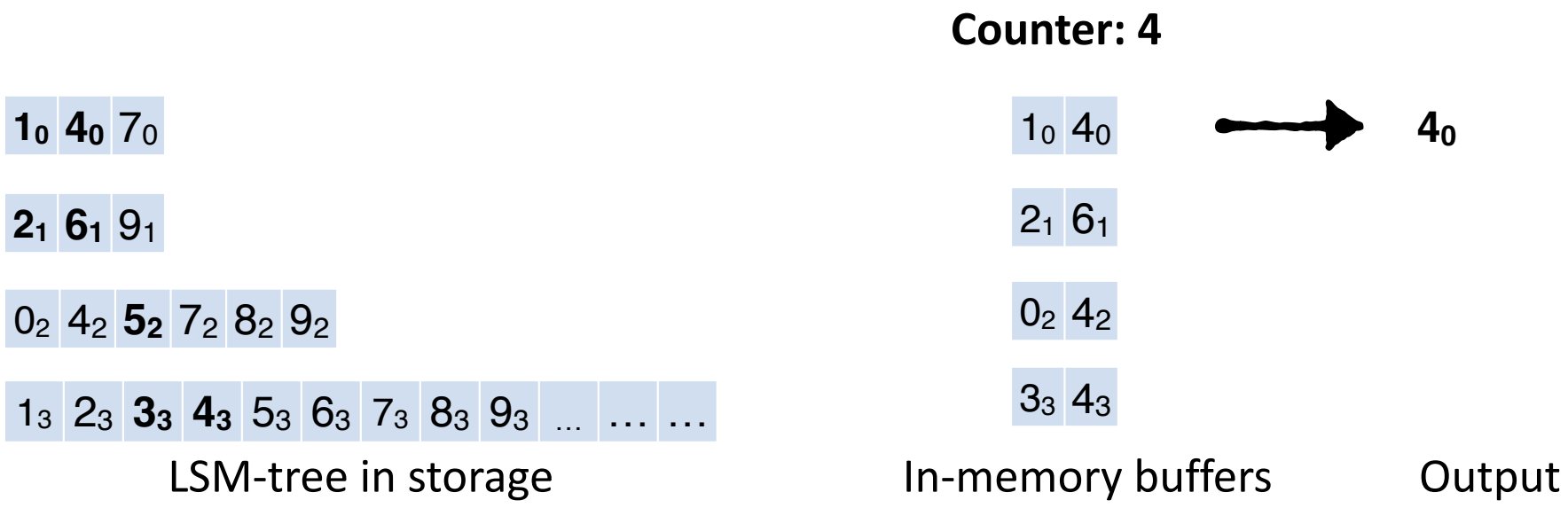
1. Allocate an in-memory buffer (>1 page) for each level
2. Search for start of key range at each level
3. Initialize counter to smallest key in range



Basic LSM-tree – Scan Queries

Loop until counter passes end of range

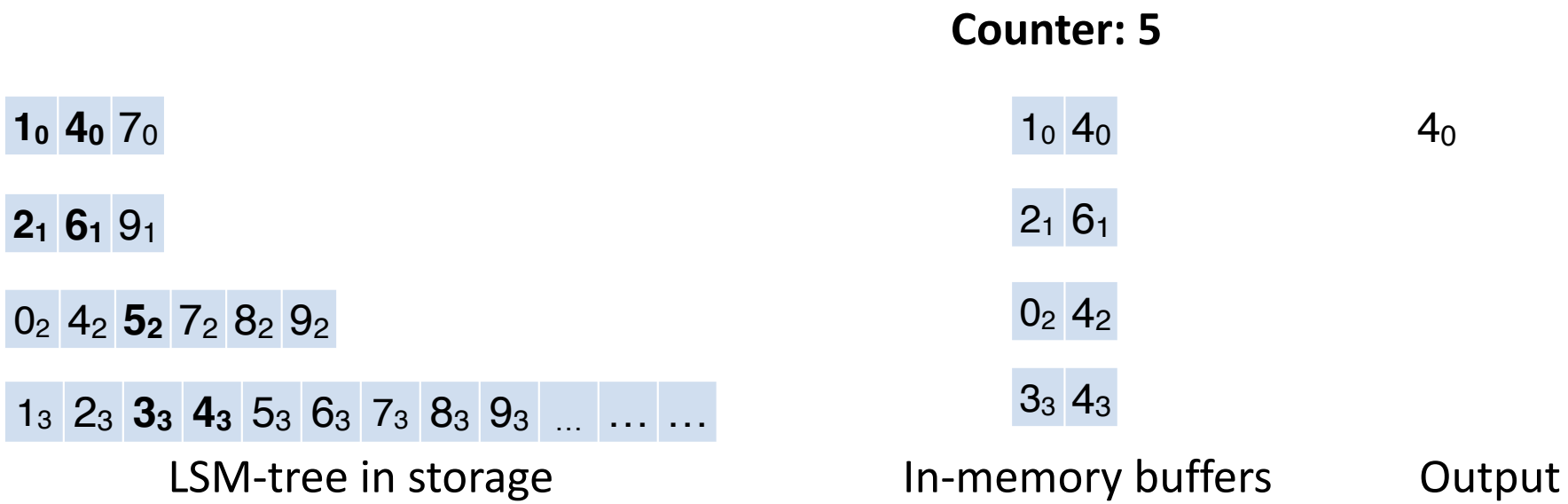
4. Bring youngest matching entry to output



Basic LSM-tree – Scan Queries

Loop until counter passes end of range

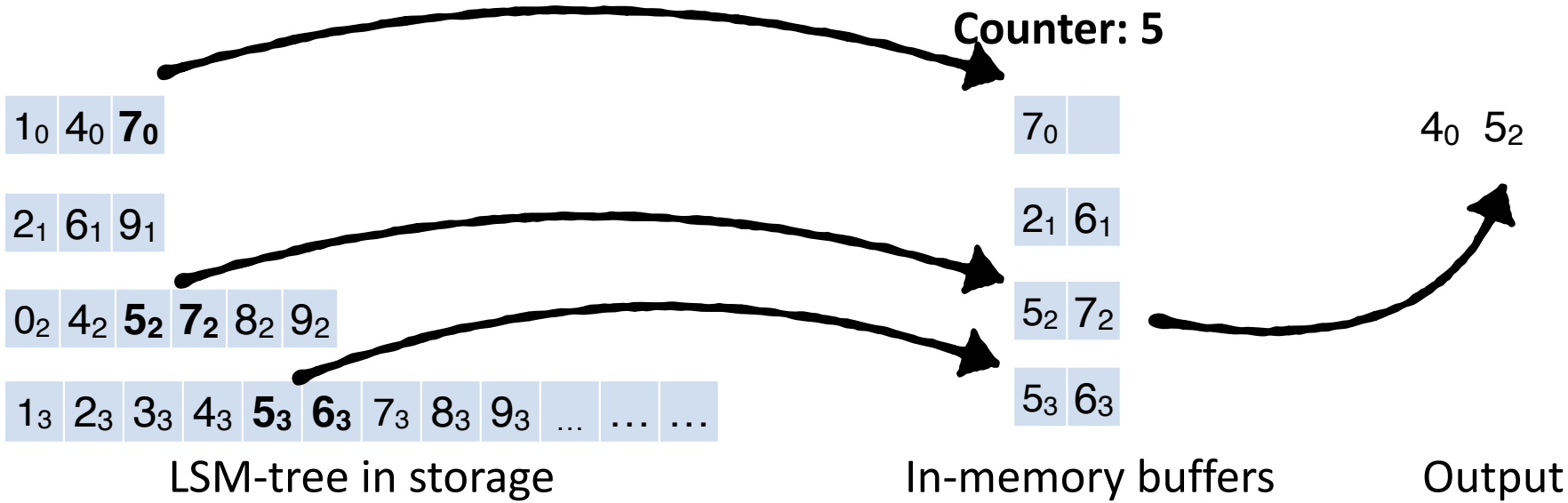
4. Bring youngest matching entry to output
5. Increment counter



Basic LSM-tree – Scan Queries

Loop until counter passes end of range

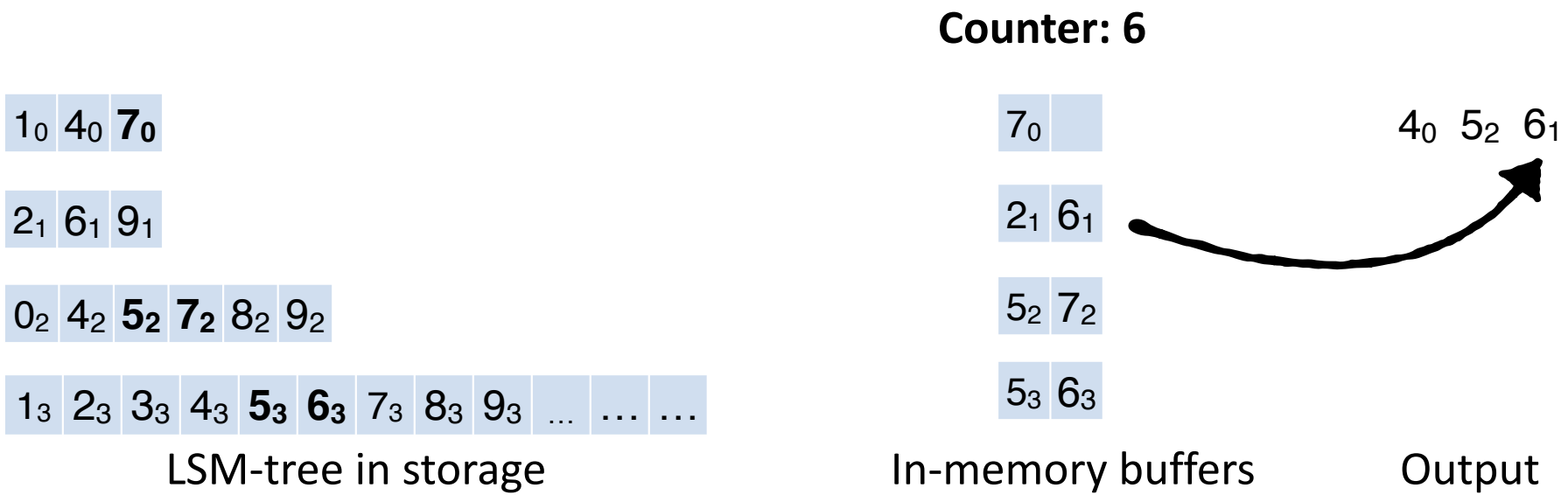
4. Bring youngest matching entry to output
5. Increment counter
6. Bring more pages to buffers if needed



Basic LSM-tree – Scan Queries

Loop until counter passes end of range

4. Bring youngest matching entry to output
5. Increment counter
6. Bring more pages to buffers if needed



Basic LSM-tree – Scan Queries I/O Cost

lvls

 # output size

$O(\log_2(N/B) * \log_B(N) + \mathbf{S/B})$

B-tree search cost

LSM-tree in storage

In-memory buffers

Basic LSM-tree – Insertion/Update/Delete cost

How many times is each entry copied?

Basic LSM-tree – Insertion/Update/Delete cost

How many times is each entry copied? $O(\log_2 N/B)$

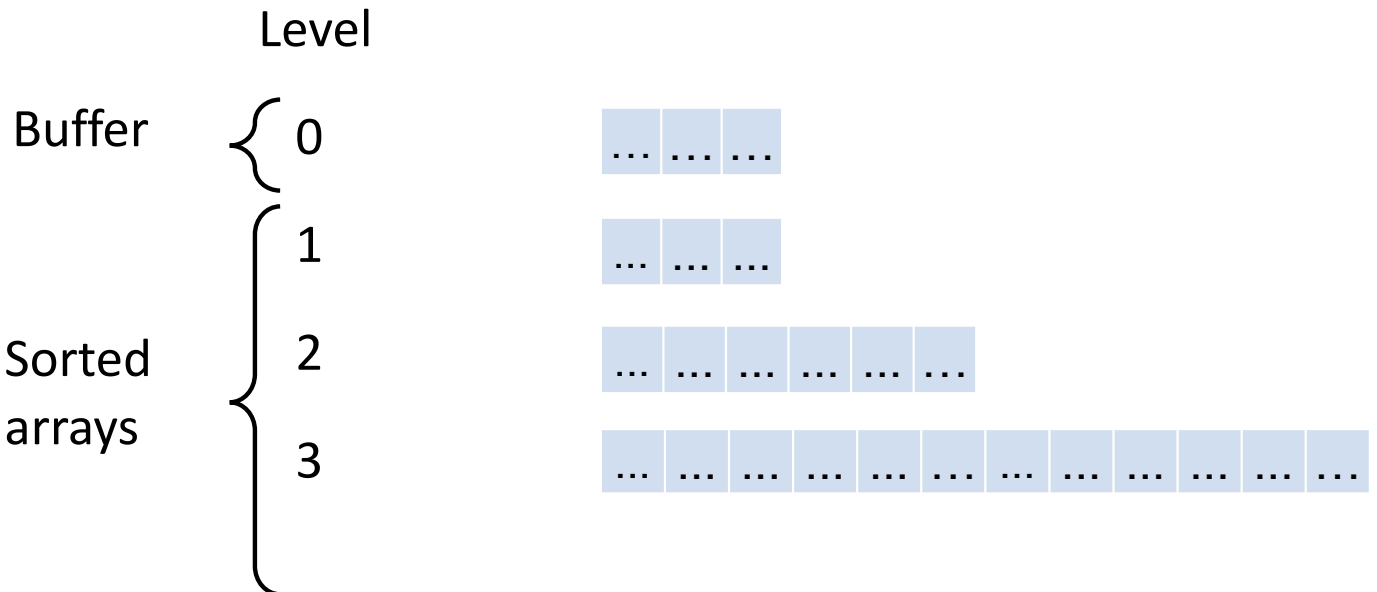
Price of each copy? $O(1/B)$ reads & writes

Total cost: $O((\log_2 N/B)/B)$ read & write I/Os

Basic LSM-tree – Insertion/Update/Delete cost

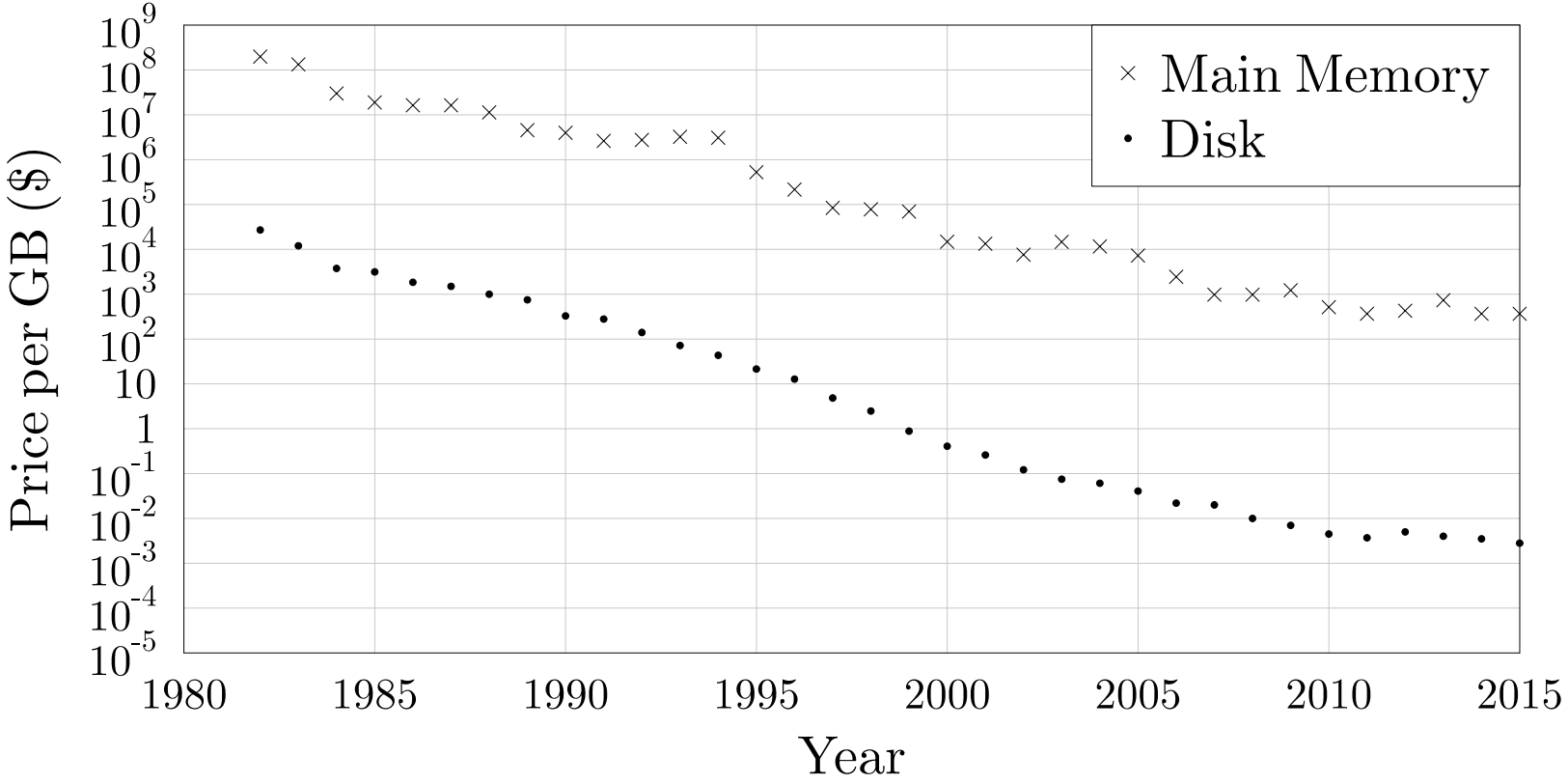
Total cost: $O((\log_2 N/B)/B)$ read & write I/Os

As all writes are large & sequential (rather than random), there is less SSD garbage-collection



Operation	I/O	append-only table	Basic LSM-tree table	B-Tree
Query	Reads	O(N/B)	**$O(\log_2(N/B) * \log_B N)$**	$O(\log_B N)$
Insert	Reads	0	**$O((\log_2 N/B)/B)$**	$O(\log_B N)$
	Writes	O(1/B)	**$O((\log_2 N/B)/B)$**	O(1) & GC

Declining Main Memory Cost

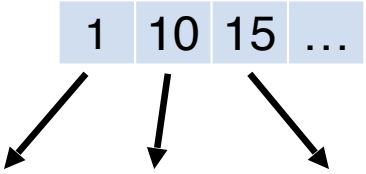


Declining Main Memory Cost

It’s viable to pin the internal nodes of B-trees in memory

Fence pointers

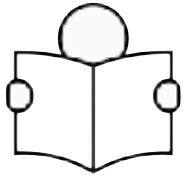
array



Block 1	Block 2	Block 3	…
1	10	15	…
3	11	16	…
6	13	18	…

Operation	I/O	append-only table	**Basic LSM-tree table**	**B-Tree**
Query	Reads	O(N/B)	O($\log_2$ N/B)	O(1)
Insert	Reads	0	O(($\log_2$ N/B)/B)	O(1)
	Writes	O(1/B)	O(($\log_2$ N/B)/B)	O(1) & GC
Memory		O(B)	O(N/B)	O(N/B)

Leveled LSM-tree



Basic LSM-tree

Tiered LSM-tree

Leveled LSM-tree

 Lookup cost

 Update cost

Leveled LSM-tree

Lookup cost?

$O(\log_T(N/B))$

Insertion cost?

$O(T/B \bullet \log_T(N/B))$

What happens as we increase the size ratio T?

What happens when size ratio T is set to be N/B?

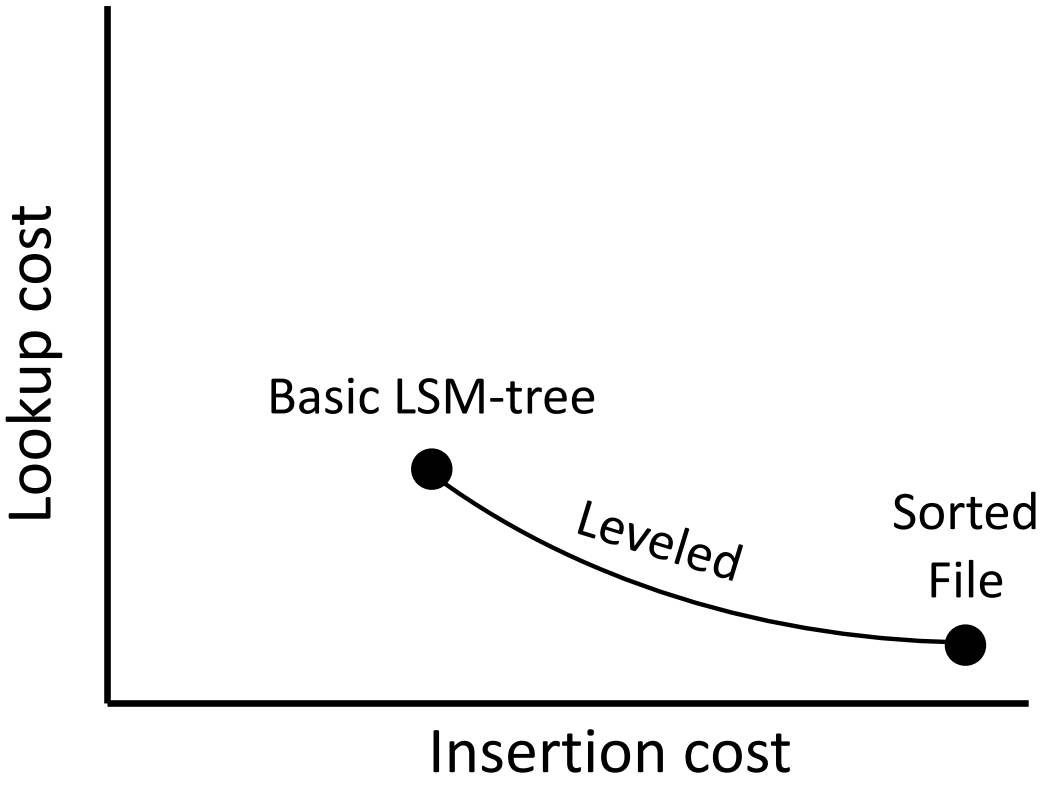
Lookup cost becomes:

$O(1)$

Insert cost becomes:

$O(N/B^2)$

The LSM-tree becomes a sorted file!


Lookup cost
Basic LSM-tree
Leveled
Sorted
File
Insertion cost

Tiered LSM-tree

Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.
Do not merge within a level.
E.g. size ratio of 4

Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.
Do not merge within a level.
E.g. size ratio of 4

Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.
Do not merge within a level.
E.g. size ratio of 4

Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.
Do not merge within a level.
E.g. size ratio of 4

Tiered LSM-tree

Reduce the number of levels by increasing the size ratio.
Do not merge within a level.
E.g. size ratio of 4

Tiered LSM-tree

Lookup cost?

$O(T \bullet \log_T(N/B))$

Insertion cost?

$O(1/B \bullet \log_T(N/B))$

What happens as we increase the size ratio T?

What happens when size ratio T is set to be N/B?

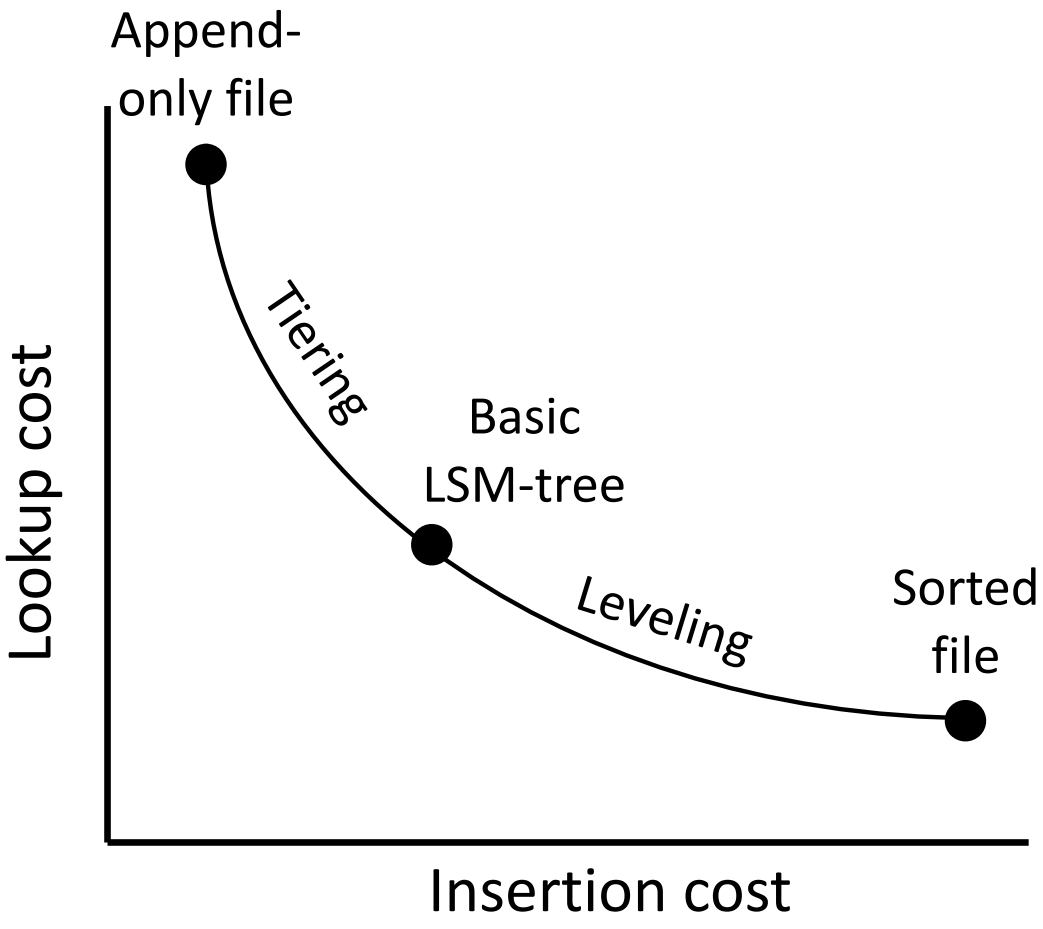
Lookup cost becomes:

$O(N/B)$

Insert cost becomes:

$O(1/B)$

The tiered LSM-tree becomes an append-only file!

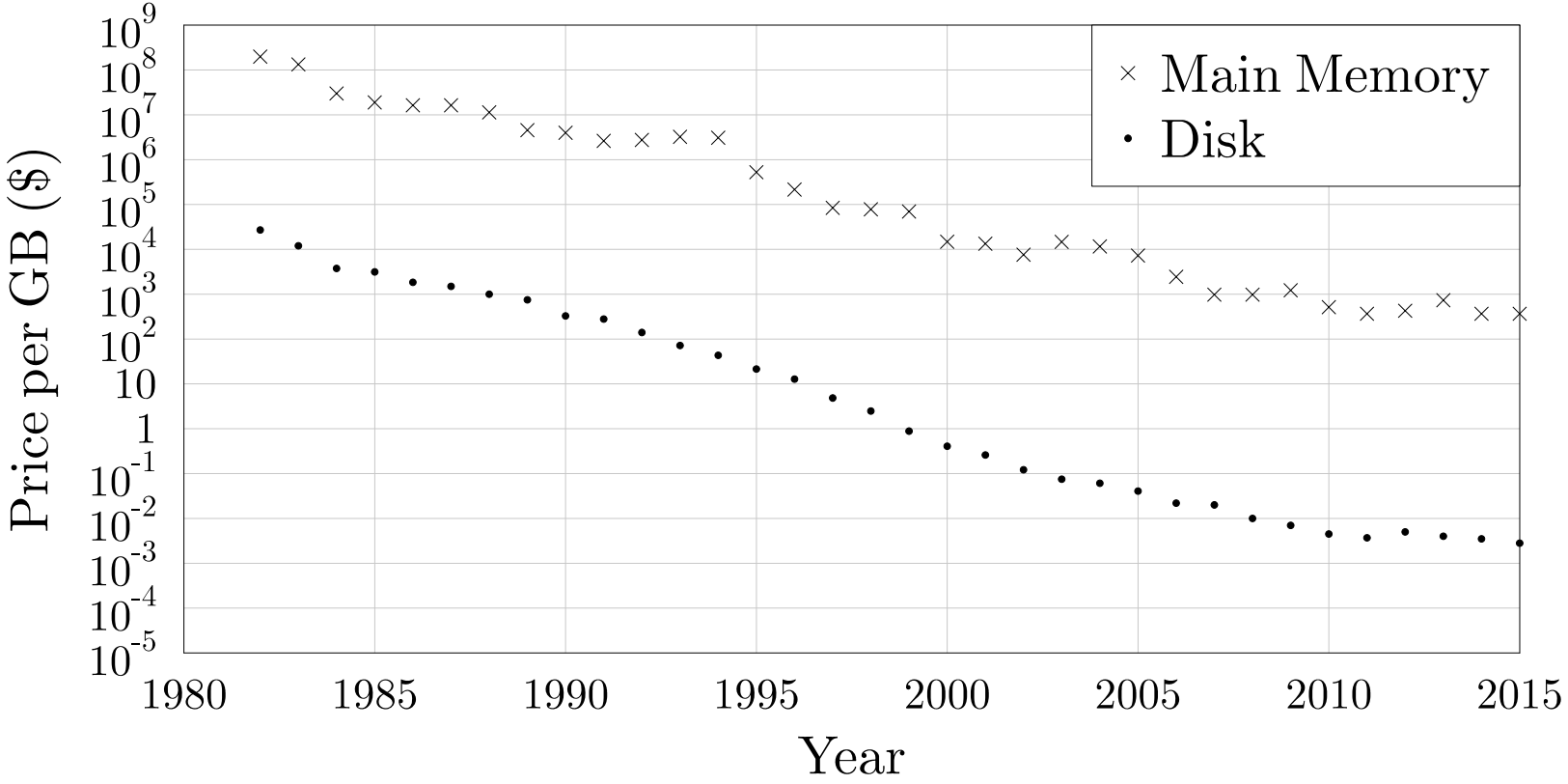

Append-
only file
Tiering
Basic
LSM-tree
Leveling
Sorted
file
Lookup cost
Insertion cost

Operation		**Unordered File**	**Leveled LSM-tree**	**Tiered LSM-tree**	**B-Tree**
Query	Reads	O(N/B)	O(L)	O(L * T)	O(1)
Insert	Reads	0	O((L * T)/B)	O(L/B)	O(1)
	Writes	O(1/B)	O((L * T)/B)	O(L/B)	O(1) & GC
Memory		O(B)	O(N/B)	O(N/B)	O(N/B)

We let $L = \log_T(N/B)$

Bloom filters

Declining Main Memory Cost

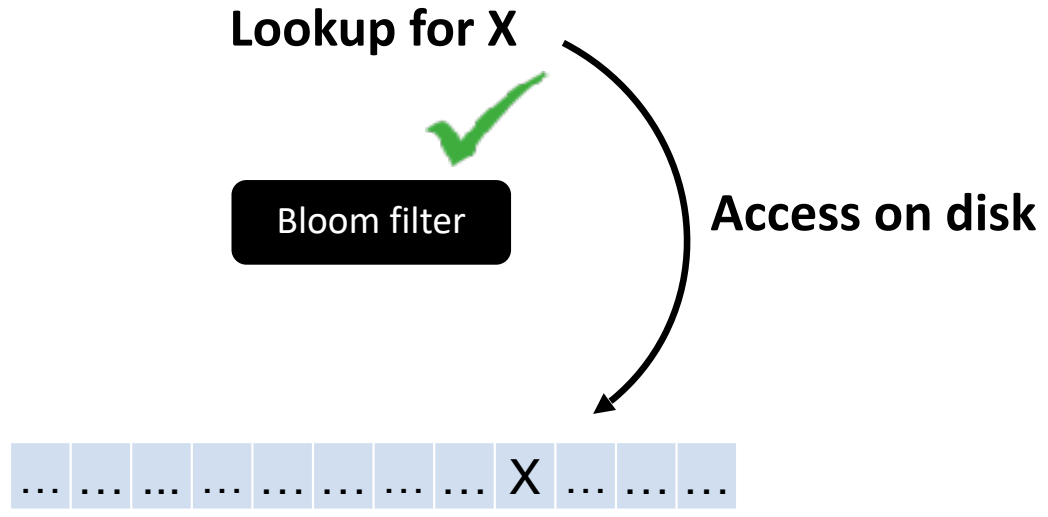


Bloom Filters

Answers set-membership queries
Smaller than data it represents
Purpose: avoid accessing storage if entry is not present
Subtlety: may return false positives.

filters

array

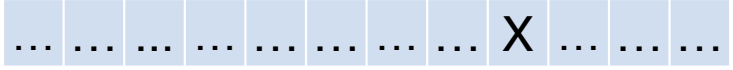


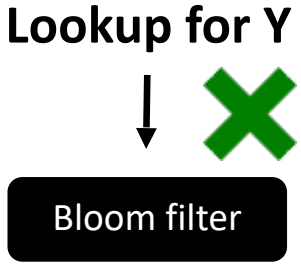
Bloom Filters

Answers set-membership queries
Smaller than data it represents
Purpose: avoid accessing storage if entry is not present
Subtlety: may return false positives.

filters

array



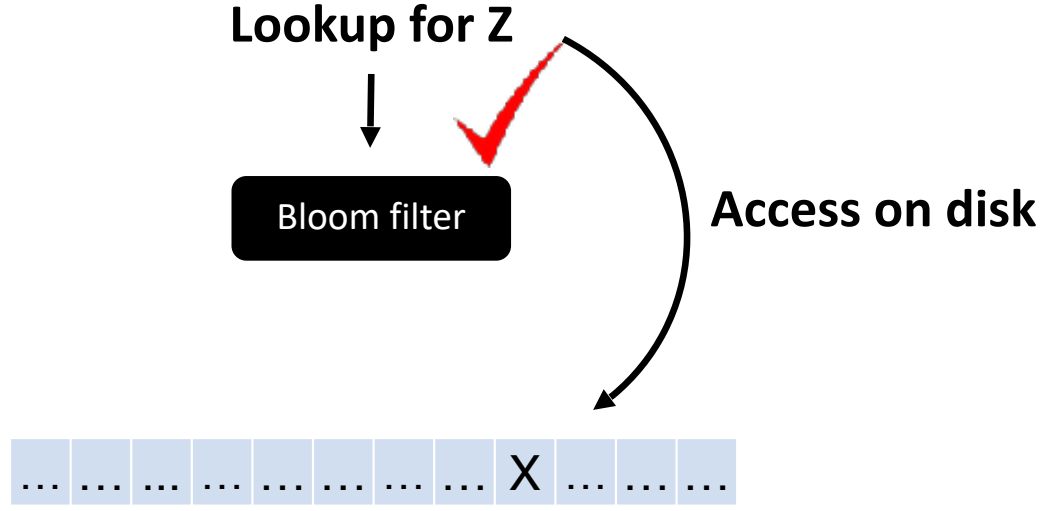


Bloom Filters

Answers set-membership queries
Smaller than data it represents
Purpose: avoid accessing storage if entry is not present
Subtlety: may return false positives.

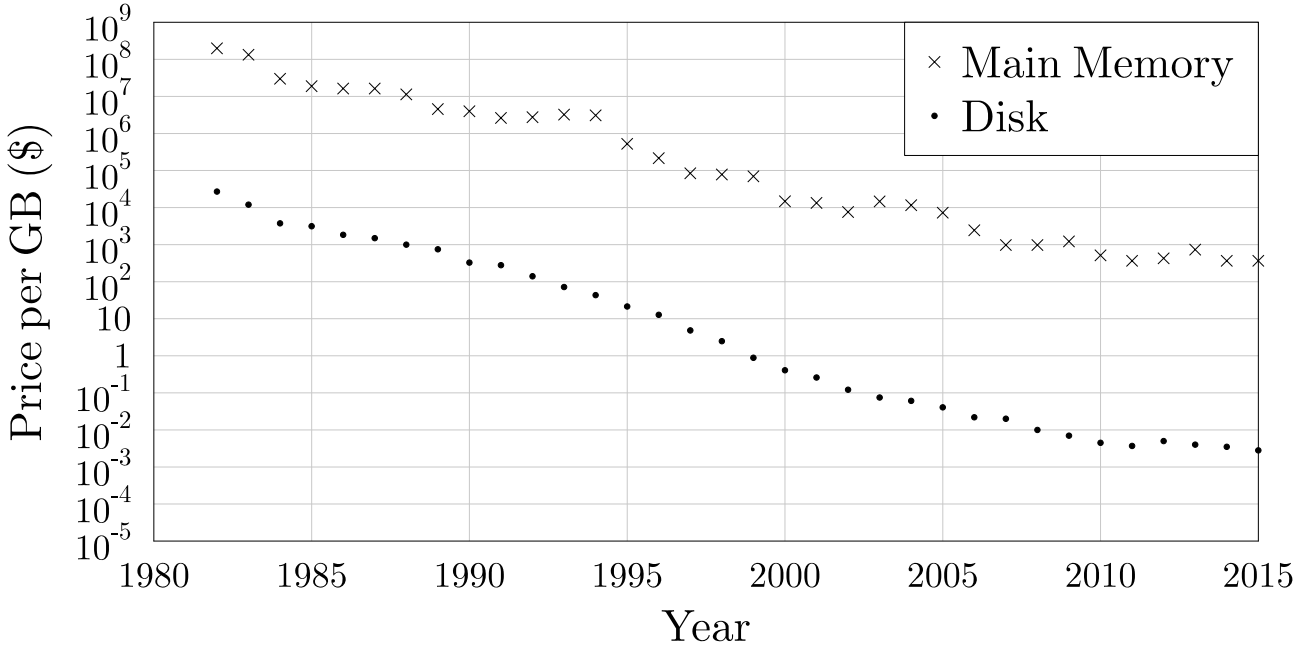
filters

array



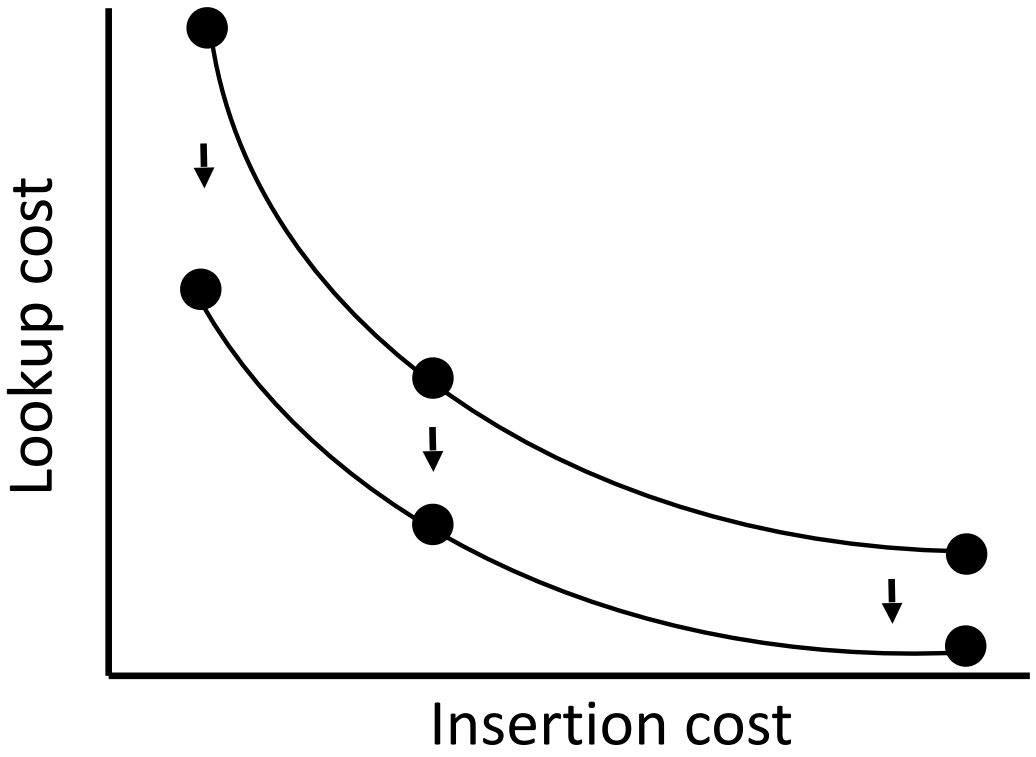
Bloom Filters

The more main memory, the less false positives ⇒ cheaper lookups



Bloom Filters

The more main memory, the less false positives ⟹ cheaper lookups



Conclusions - LSM-trees are:

Write-optimized

Highly tunable

Backbone of many modern systems

Trade-off between lookup and insert cost (tiering/leveling, size ratio)

Trade main memory for lookup cost (fence pointers, Bloom filters)