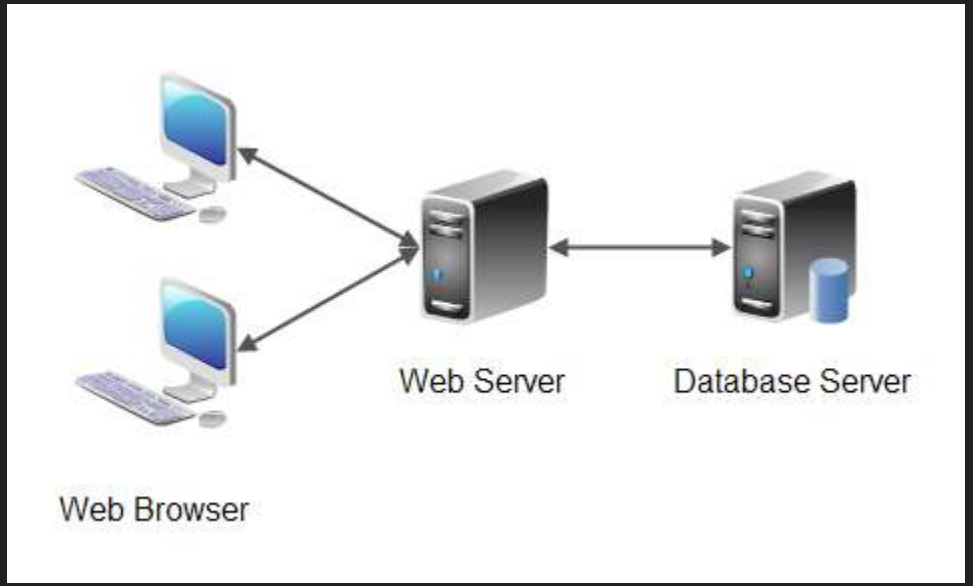
Database Design

What is a database?



Objectives of a database

- Fast reads
- Fast writes
- Persistent data

Quick comment about disks

Generally, we are using hard drives

Results in slow random reads

We should always aim for sequential operations

Much cheaper than SSDs but slower

Naive Database Implementation

Literally just a list, O(n) reads and updates

Id	Name	Favorite Color
1	Alan Turing	Red
2	Mark Zuckerberg	LightBlue
3	Tech Lead	Doesn’t see color
4	Mitch McConnell	White

Naive Database Implementation

Literally just a list, O(n) reads and writes

Id	Name	Favorite Color
1	Alan Turing	Red
2	Mark Zuckerberg	LightBlue
3	Tech Lead	Doesn’t see color
4	Mitch McConnell	White

Naive Database Implementation

Literally just a list, O(n) reads and writes

Id	Name	Favorite Color
1	Alan Turing	Red
2	Mark Zuckerberg	LightBlue
3	Tech Lead	Pink
4	Mitch McConnell	White

Slightly Better Database Implementation

Append only log on disk to take advantage of sequential logs

Id	Name	Favorite Color
1	Alan Turing	Red
2	Mark Zuckerberg	LightBlue
3	Tech Lead	Doesn’t see color
4	Mitch McConnell	White

Slightly Better Database Implementation

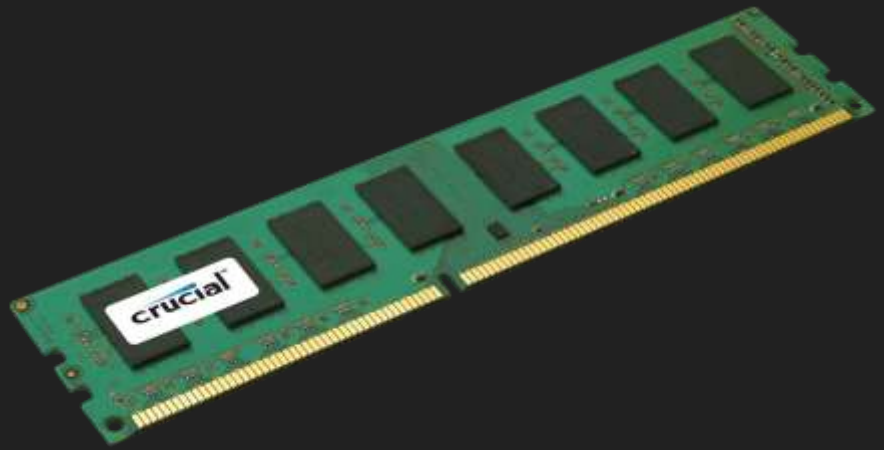
Append only log on disk to take advantage of sequential logs

Id	Name	Favorite Color
1	Alan Turing	Red
2	Mark Zuckerberg	LightBlue
3	Tech Lead	Doesn’t see color
4	Mitch McConnell	White
3	Tech Lead	Pink

Better database implementation

Hashmap, O(1) reads and writes

However, this does not scale because the second there is too much data we are in trouble, hashmap has to go on disk which becomes slow

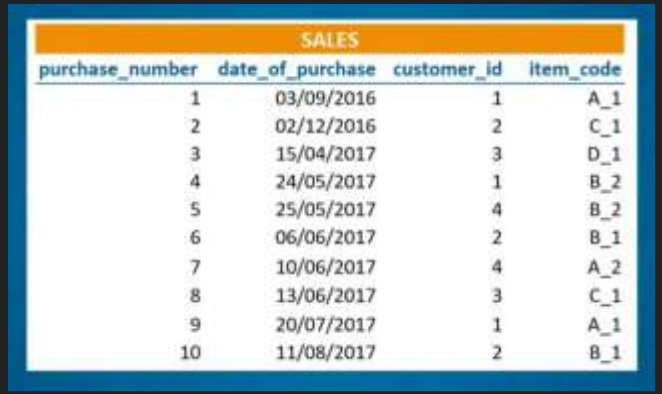


Indexes - making read times much faster

Keep extra data on each write to improve database read times

Pro: Faster reads

Con: Slower writes (only use indexes if you need them, do not declare an index for every field)



SALES

purchase_number	date_of_purchase	customer_id	item_code
1	03/09/2016	1	A_1
2	02/12/2016	2	C_1
3	15/04/2017	3	D_1
4	24/05/2017	1	B_2
5	25/05/2017	4	B_2
6	06/06/2017	2	B_1
7	10/06/2017	4	A_2
8	13/06/2017	3	C_1
9	20/07/2017	1	A_1
10	11/08/2017	2	B_1

Types of index implementations

- Hash Indexes
- LSM trees + SSTables
- B Trees

Hash Index

Keep an in memory hash table of the key mapped to the memory location of the corresponding data, occasionally write to disk for persistence

Pros: Easy to implement and very fast (disks are slow, RAM is fast)

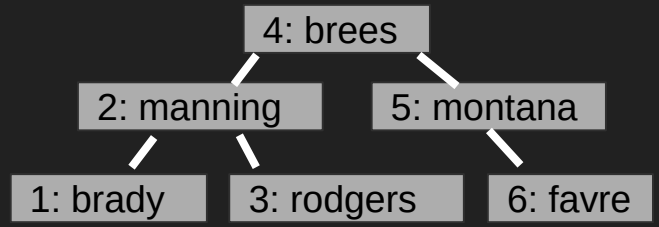
Cons: All of the keys must fit in memory, bad for range queries

Key	Offset on Disk
jordan	0x04444511
kobe	0x01112365
lebron	0x06128989

SSTables and LSM trees

Write first goes to an in-memory balanced binary search tree (memtable), eventually written to disk

When tree becomes too large, write the contents of it (sorted by key name) to an SSTable file

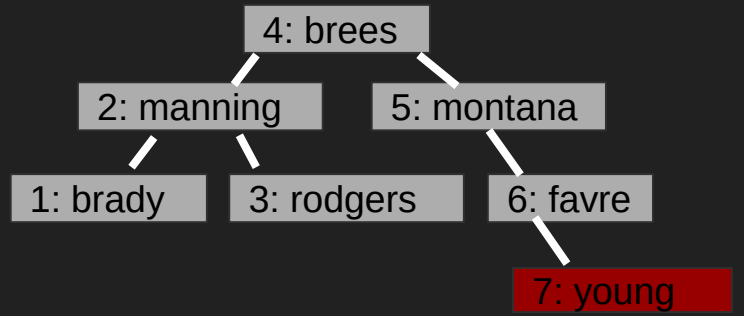


To increase persistence, keep log on disk of memtable writes to restore it in the event of a crash.

SSTables and LSM trees

Write first goes to an in-memory balanced binary search tree (memtable), eventually written to disk

When tree becomes too large, write the contents of it (sorted by key name) to an SSTable file



To increase persistence, keep log on disk of memtable writes to restore it in the event of a crash.

SSTables and LSM trees continued

Recall: Tree gets written to SSTable files, where the keys are sorted

SSTable file 1

0	Westbrook
7	Anthony
9	Wade
23	Jordan
33	Jabbar
34	Olajuwon

SSTable file 2

3	Iverson
5	Garnett
13	Harden
23	Lebron
33	Pippen
34	Giannis

SSTables and LSM trees continued

Recall: Since we are only using append only logs, there will be duplicate keys

SSTable file 1

0	Westbrook
7	Anthony
9	Wade
23	Jordan
33	Jabbar
34	Olajuwon

SSTable file 2

3	Iverson
5	Garnett
13	Harden
23	Lebron
33	Pippen
34	Giannis

SSTables and LSM trees continued

Recall: Since we are only using append only logs, there will be duplicate keys

SSTable file 1

0	Westbrook
7	Anthony
9	Wade
23	Jordan
33	Jabbar
34	Olajuwon

SSTable file 2

3	Iverson
5	Garnett
13	Harden
23	Lebron
33	Pippen
34	Giannis

Compacted SSTable

0	Westbrook
3	Iverson
5	Garnett
7	Anthony
9	Wade
13	Harden
23	Lebron
33	Pippen
34	Giannis

SSTables and LSM trees continued

Can be merged in O(n) time, in case of duplicate key take the more recent value

SSTable file 1

0	Westbrook
7	Anthony
9	Wade
23	Jordan
33	Jabbar
34	Olajuwon

SSTable file 2

3	Iverson
5	Garnett
13	Harden
23	Lebron
33	Pippen
34	Giannis

Compacted SSTable

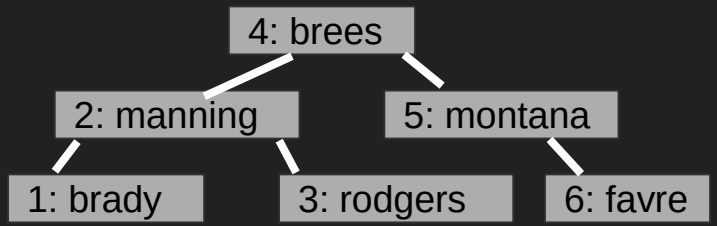
0	Westbrook
3	Iverson
5	Garnett
7	Anthony
9	Wade
13	Harden
23	Lebron
33	Pippen
34	Giannis

SSTables and LSM trees continued

Let’s discuss how to quickly read a value by its index!

First

MemTable



Second

SSTable n

28	Peterson
44	Bradshaw
81	Moss

Third

SSTable n-1

28	Taylor
80	Rice
85	Gates

Keep going through SSTables until you either find the key or run out!

SSTables and LSM trees continued

For each SSTable, have a sparse in-memory hashmap of keys with their value in memory. Since each table is sorted, we can quickly binary search the SSTable to find the value of a key!

In memory hash table

Alice	0x00000000
Bob	0x00000080
Charlie	0x000000f0

SSTable

Alice	22
Andy	61
Anna	40
Bob	80
Brian	15
Charlie	35

SSTables and LSM trees summarized

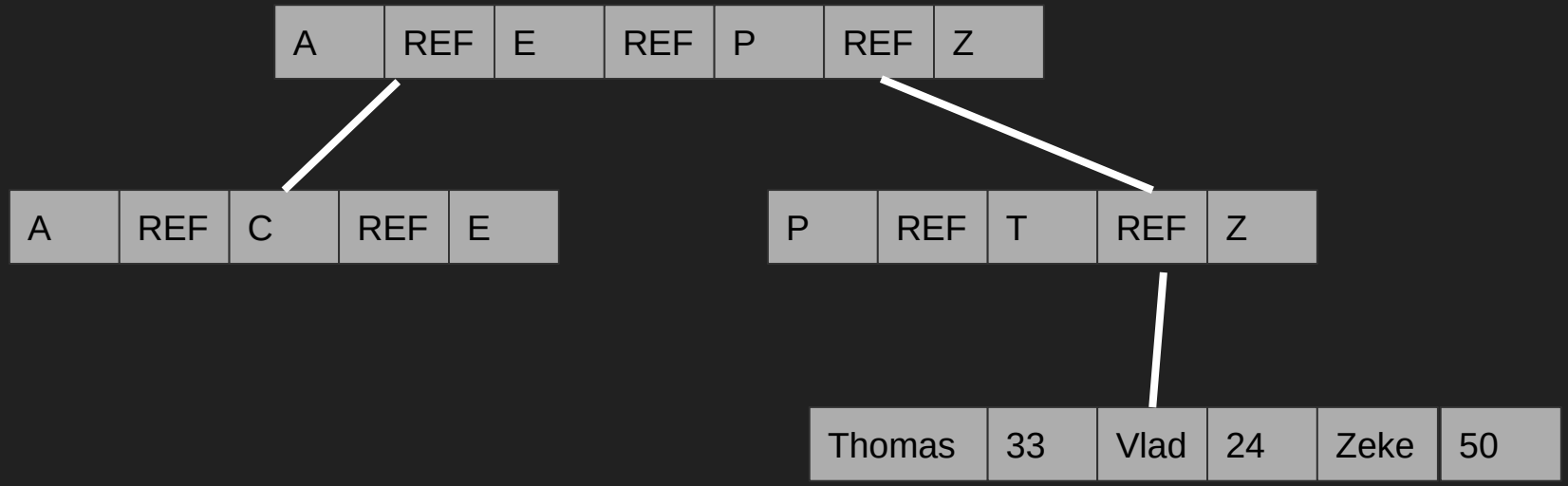
Pros:

- High write throughput due to writes going to in memory buffer
- Good for range queries due to internal sorting of data in the index

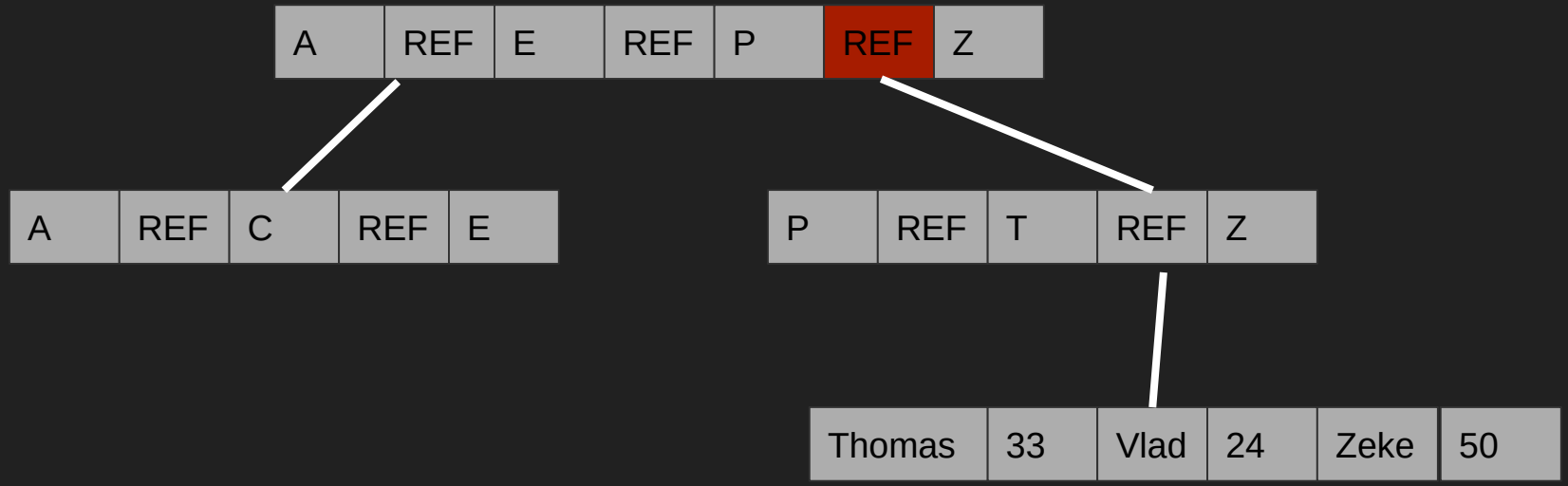
Cons:

- Slow reads, especially if the key we are looking for is old or does not exist
- Merging process of log segments can take up background resources

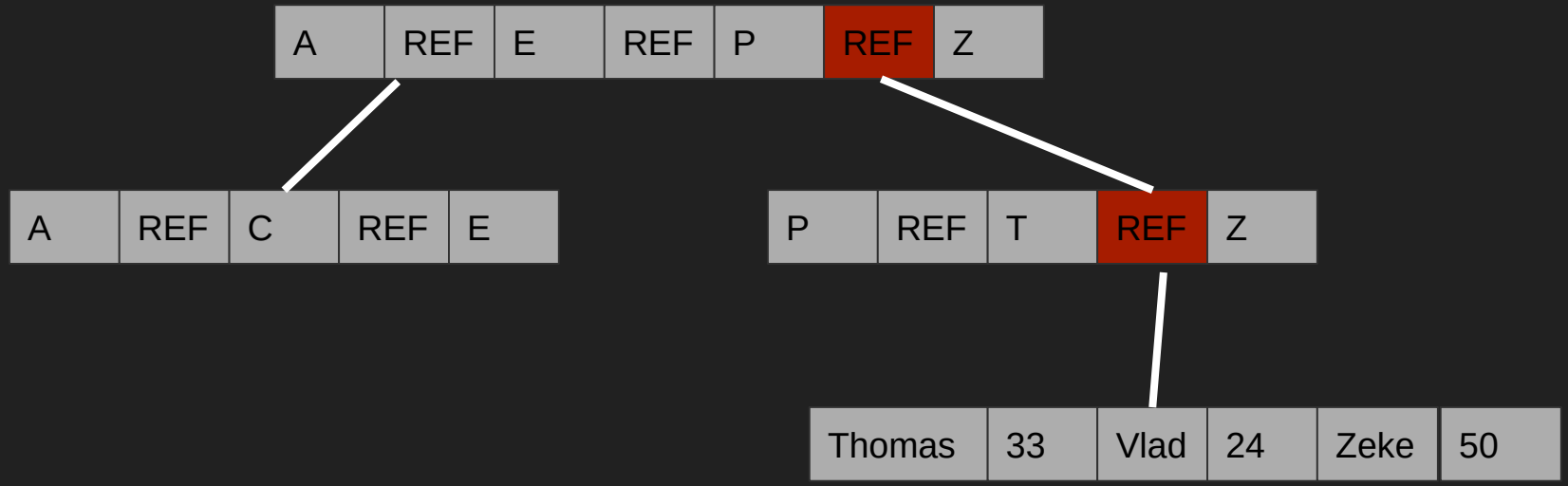
B-Trees


A
REF
E
REF
P
REF
Z
A
REF
C
REF
E
P
REF
T
REF
Z
Thomas
33
Vlad
24
Zeke
50

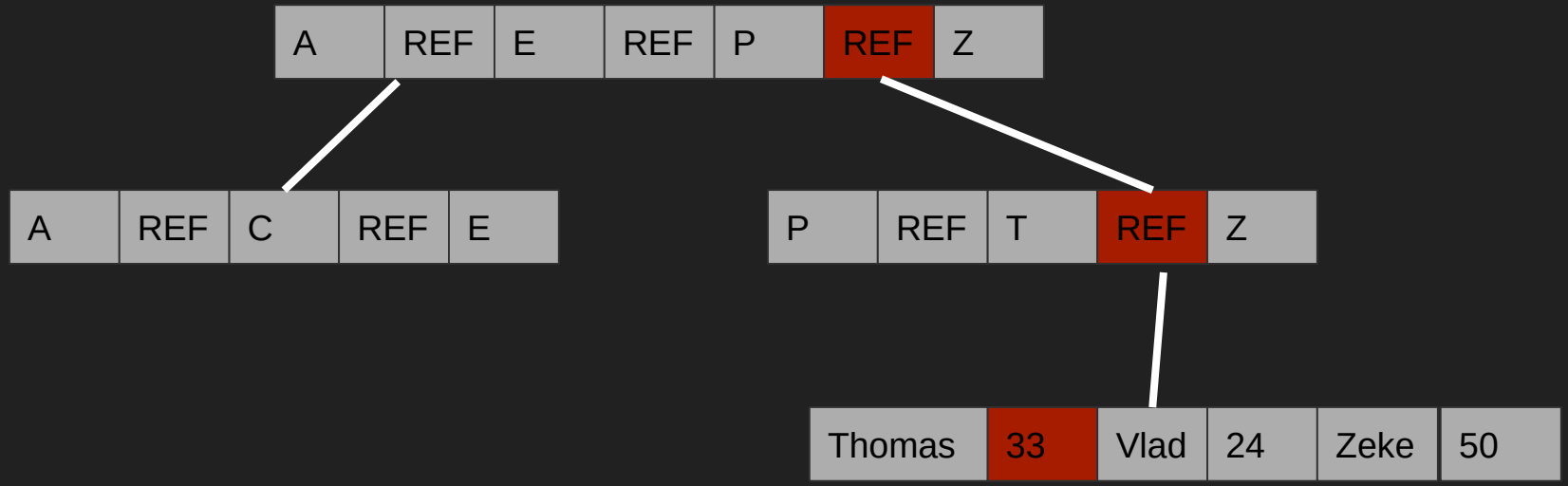
B-Trees continued



B-Trees continued



B-Trees continued



B-Trees continued

To read: traverse through the tree and find the value

To update: traverse through the tree and change the value

To write: traverse through the tree, if there is extra space in the block where the value belongs, add the key, otherwise you have to split the location block in two, add the key, and then update the parent block to reflect this action. Can be made durable in the event of crashes using a write ahead log.

B-trees summarized

Pros:

- Relatively fast reads, most B-trees can be stored in only 3 or 4 levels
- Good for range queries as data is kept internally sorted

Cons:

- Relatively slow writes, have to write to disk as opposed to memory

Conclusion

In a system, it is important to know what type of database engine/design you are using so that you can optimize for writes or reads.

Hash indexes: fast but only useful on small datasets

SSTables and LSM-Trees: better for writing, slower for reading

B-Trees: better for reading, slower for writing