Dissecting the humble LSM Tree and SSTable

Suhail Patel | @suhailpatel | https://suhailpatel.com

SRECon EMEA 2022

Wait, this is SRECon?

A core understanding of the data structures that powers stateful applications we operate helps us make better decisions

👋

Suhail Patel

Staff Engineer at Monzo

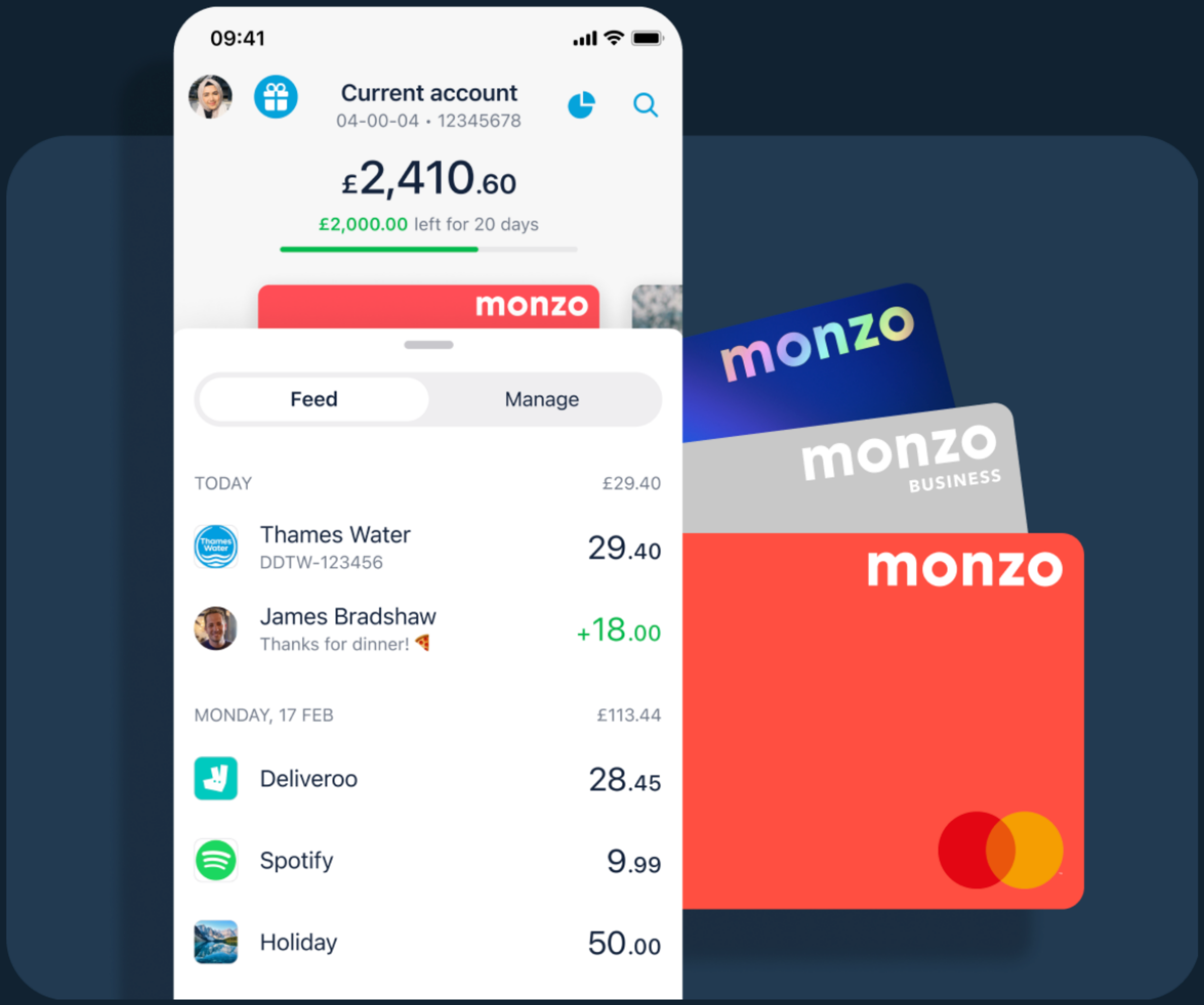
@suhailpatel

Banking made easy

Spend, save and manage your money, all in one place. Open a full UK bank account from your phone, for free.

Open a Monzo account

British Bank Awards BEST BRITISH BANK WINNER 2022



09:41

Current account
04-00-04 • 12345678

£2,410.60

£2,000.00 left for 20 days

Feed | Manage

TODAY £29.40

Thames Water 29.40
DDTW-123456

James Bradshaw +18.00
Thanks for dinner! 🍕

MONDAY, 17 FEB £113.44

Deliveroo 28.45

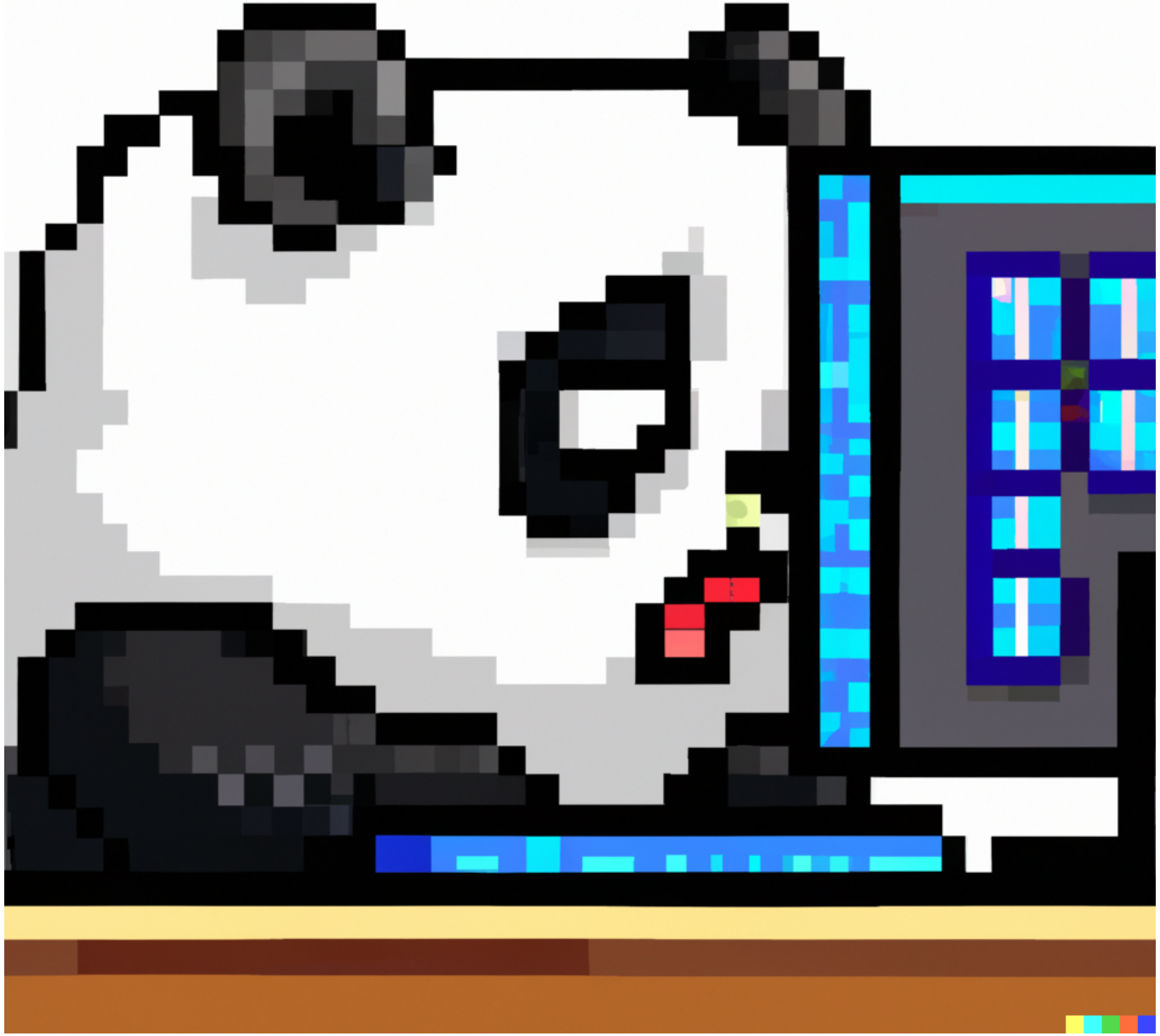
Spotify 9.99

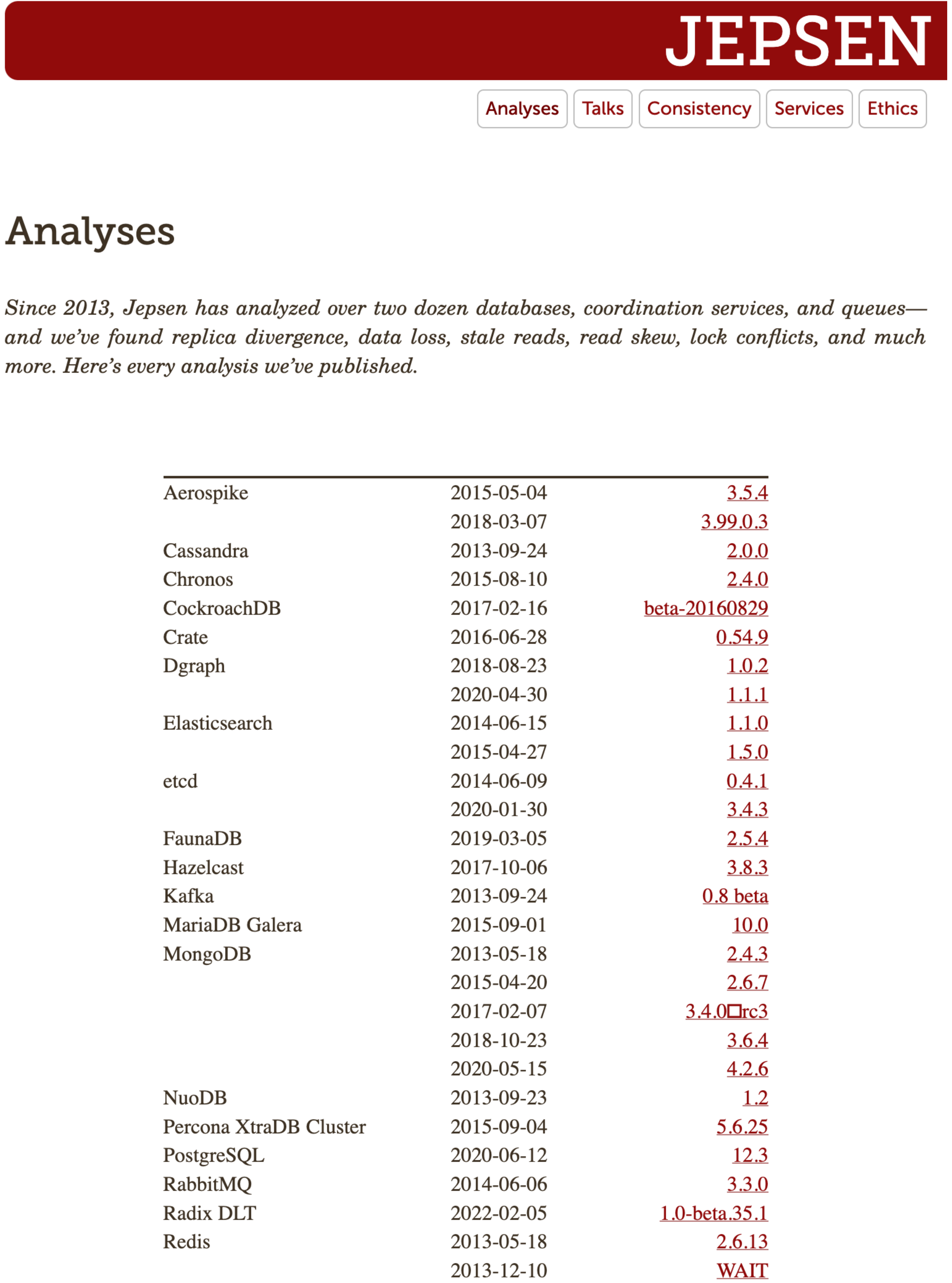
Holiday 50.00

Liability Disclaimer

I absolve myself of all responsibility if you use the example code shown and you lose your data or have an incident

https://gist.github.com/suhailpatel/331ffa65f434a9743dfb1db893931361





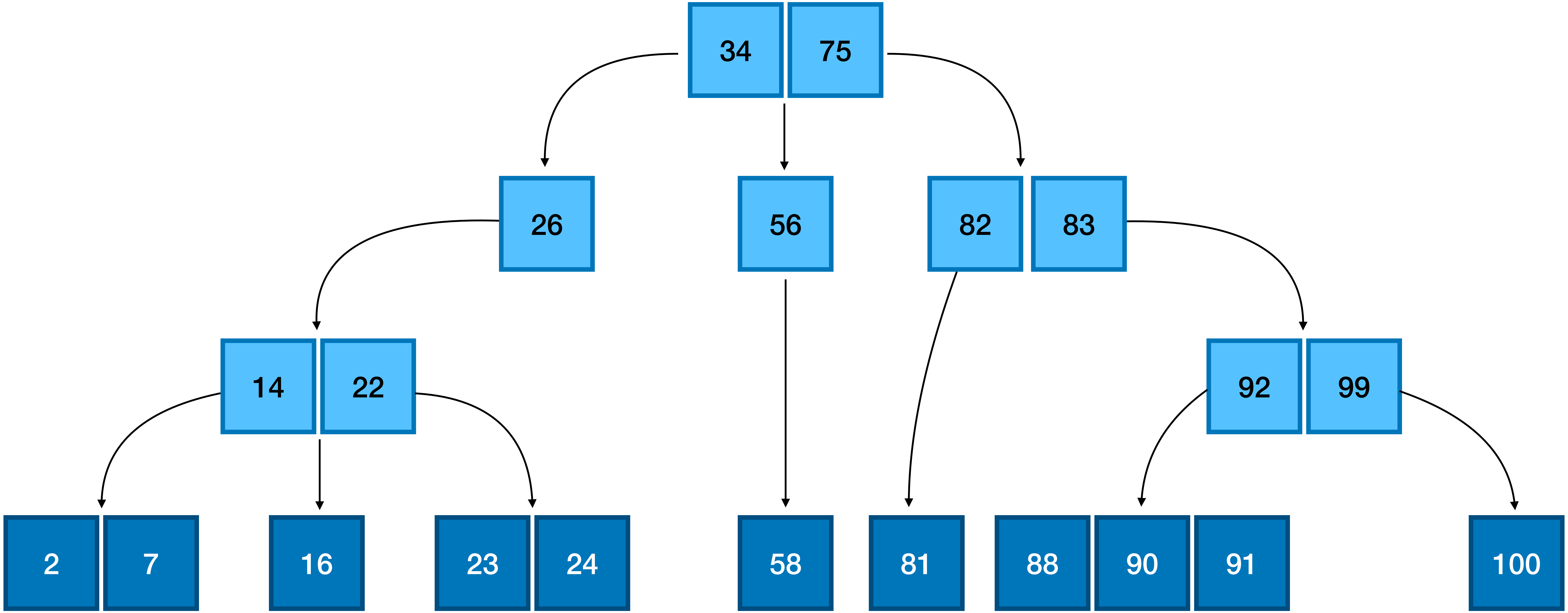
Analyses

Since 2013, Jepsen has analyzed over two dozen databases, coordination services, and queues—and we've found replica divergence, data loss, stale reads, read skew, lock conflicts, and much more. Here's every analysis we've published.

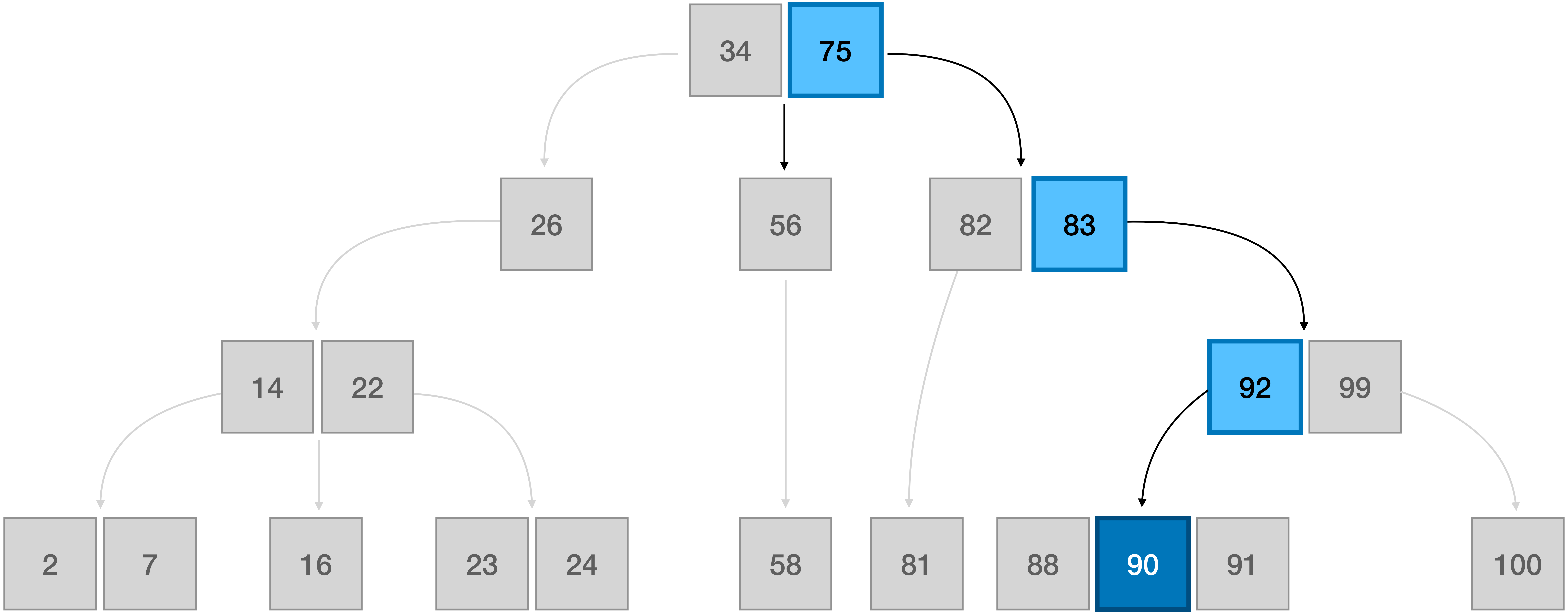
Aerospike	2015-05-04	3.5.4
	2018-03-07	3.99.0.3
Cassandra	2013-09-24	2.0.0
Chronos	2015-08-10	2.4.0
CockroachDB	2017-02-16	beta-20160829
Crate	2016-06-28	0.54.9
Dgraph	2018-08-23	1.0.2
	2020-04-30	1.1.1
Elasticsearch	2014-06-15	1.1.0
	2015-04-27	1.5.0
etcd	2014-06-09	0.4.1
	2020-01-30	3.4.3
FaunaDB	2019-03-05	2.5.4
Hazelcast	2017-10-06	3.8.3
Kafka	2013-09-24	0.8 beta
MariaDB Galera	2015-09-01	10.0
MongoDB	2013-05-18	2.4.3
	2015-04-20	2.6.7
	2017-02-07	3.4.0□rc3
	2018-10-23	3.6.4
	2020-05-15	4.2.6
NuoDB	2013-09-23	1.2
Percona XtraDB Cluster	2015-09-04	5.6.25
PostgreSQL	2020-06-12	12.3
RabbitMQ	2014-06-06	3.3.0
Radix DLT	2022-02-05	1.0-beta.35.1
Redis	2013-05-18	2.6.13
	2013-12-10	WAIT

https://jepsen.io

B-Tree



Searching within a B-Tree



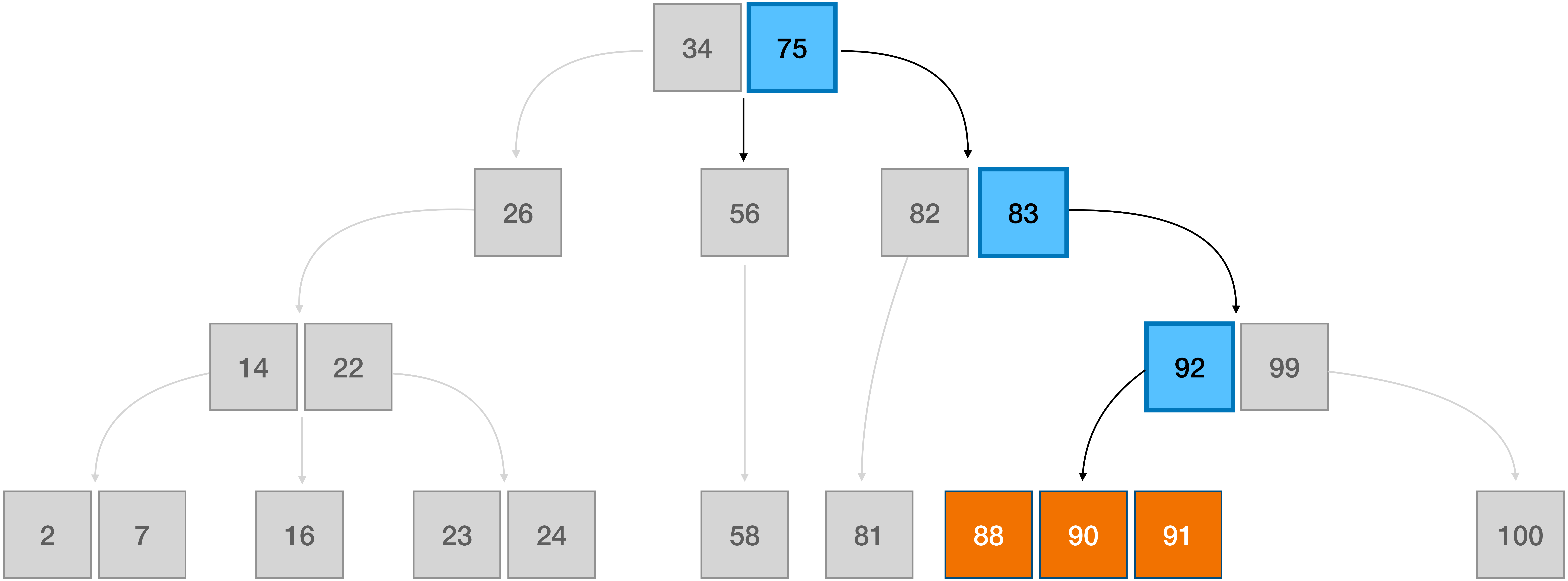
B-Trees are used in many index implementations

MySQL®

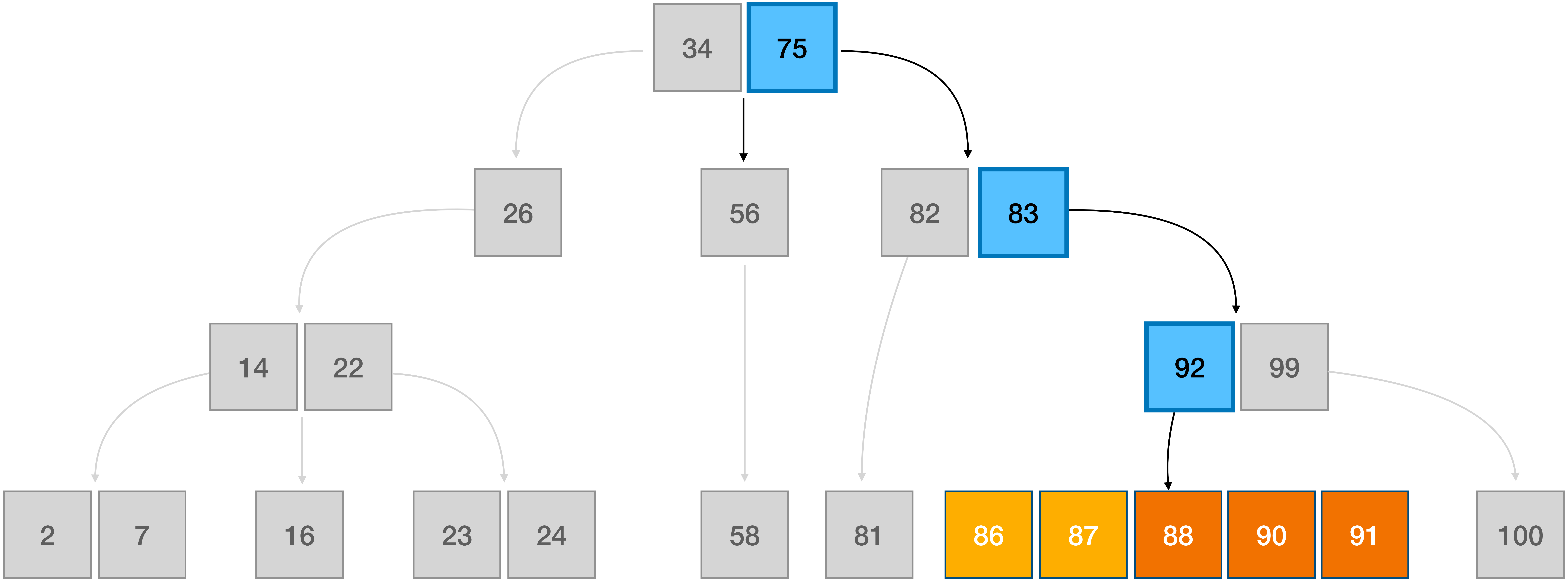
SQLite

PostgreSQL

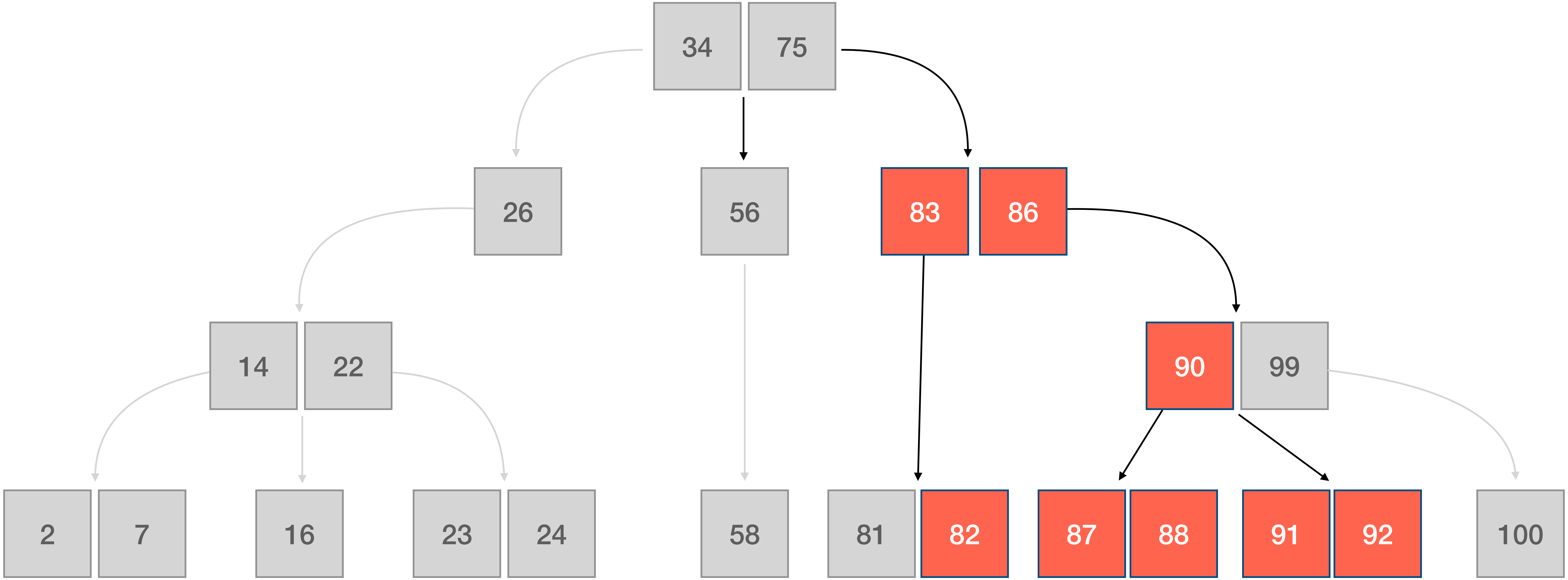
Inserting within a B-Tree

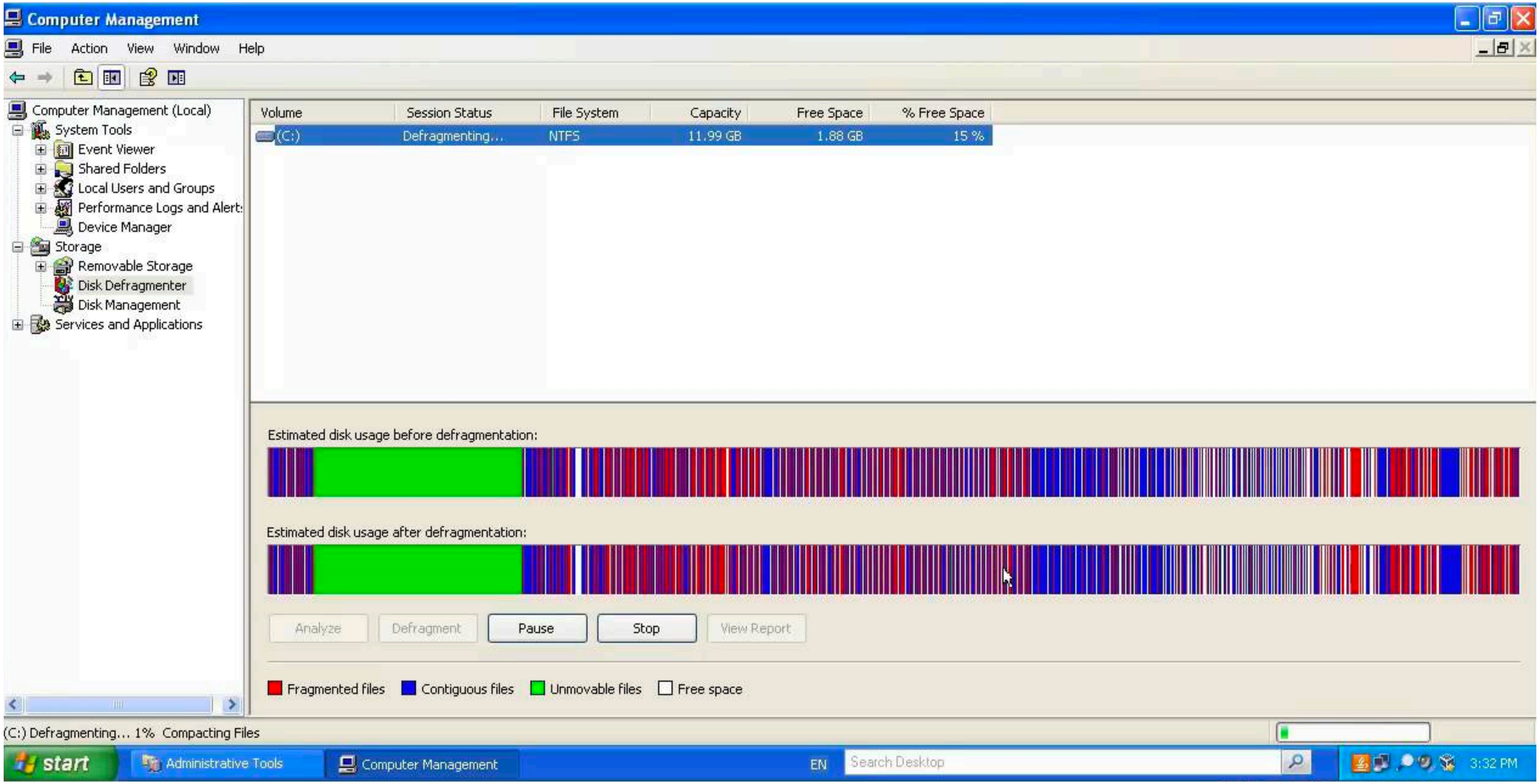


Inserting within a B-Tree



Rebalancing





Volume	Session Status	File System	Capacity	Free Space	% Free Space
(C:)	Defragmenting...	NTFS	11.99 GB	1.88 GB	15 %

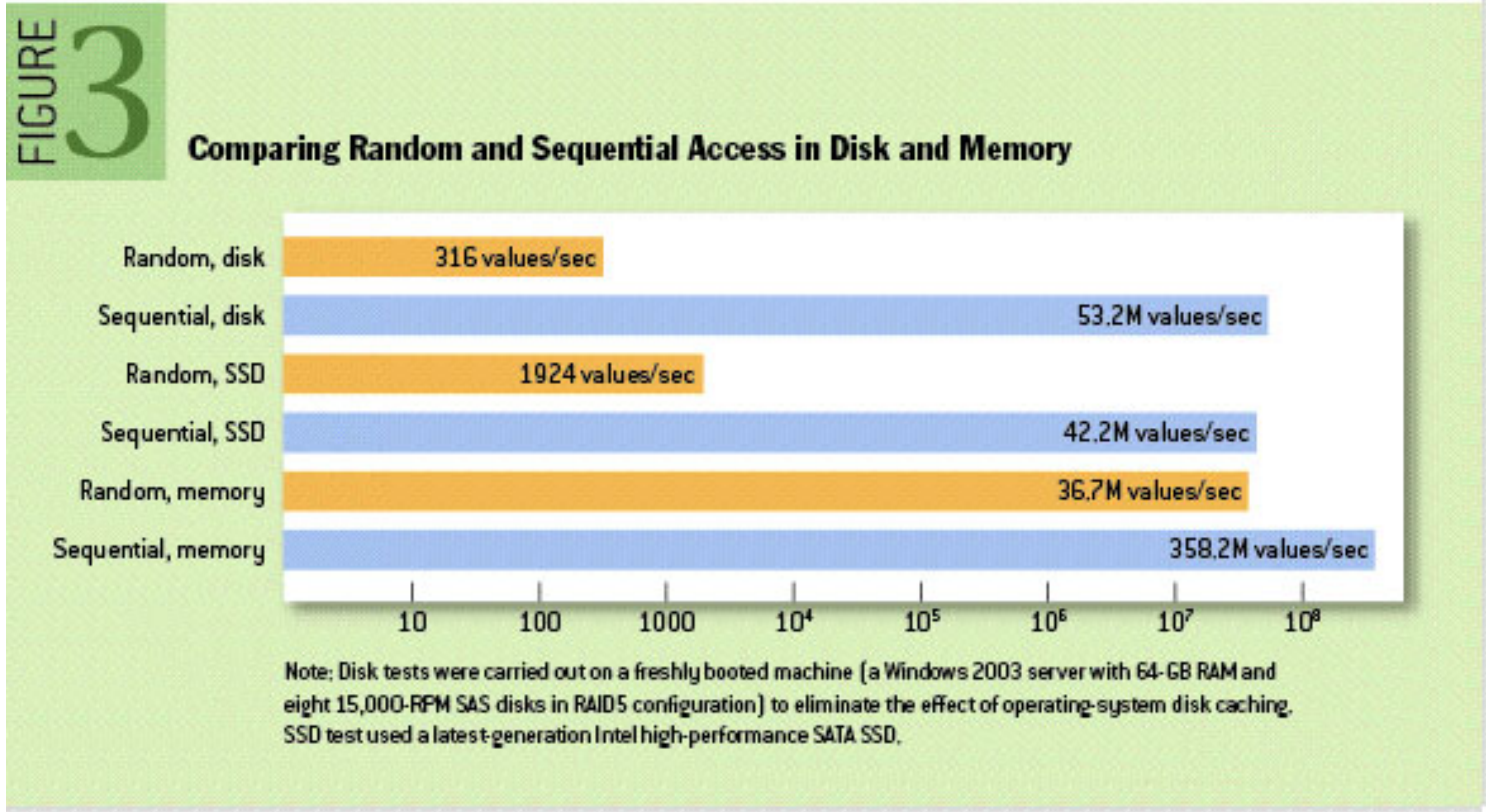
Estimated disk usage before defragmentation:

Estimated disk usage after defragmentation:

🟥 Fragmented files 🟦 Contiguous files 🟩 Unmovable files □ Free space

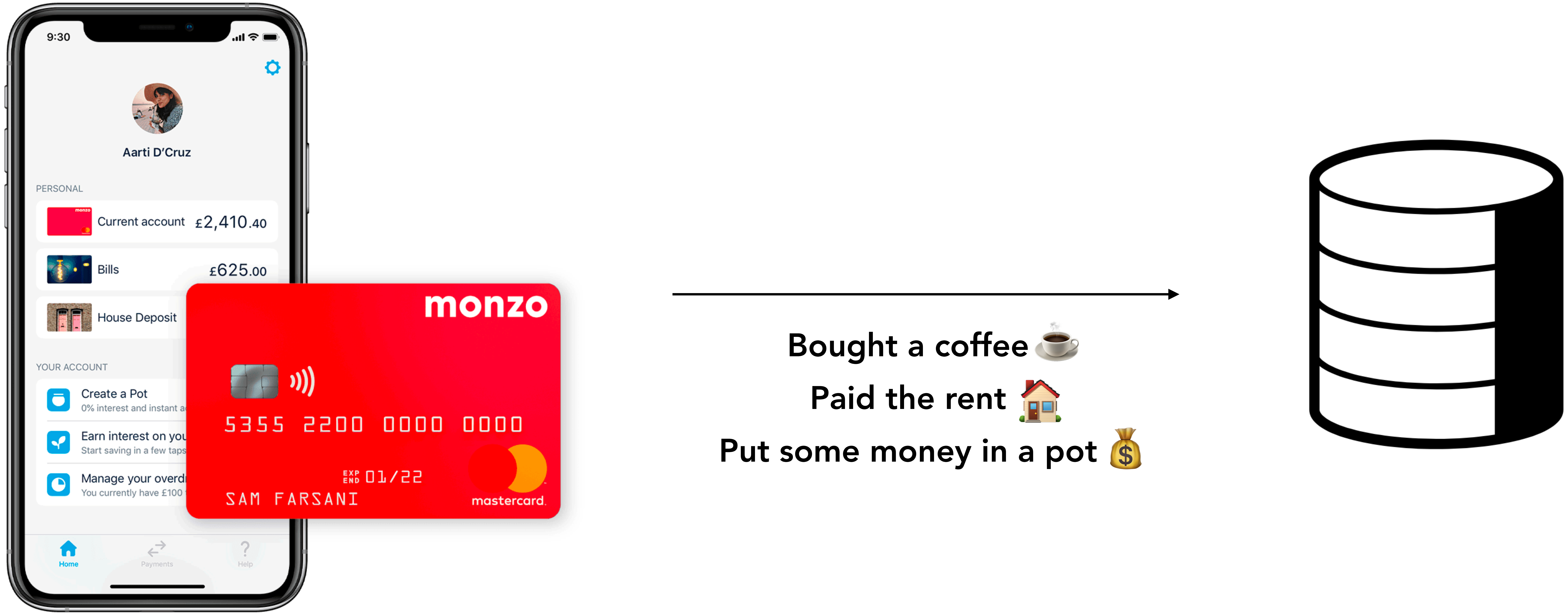
(C:) Defragmenting... 1% Compacting Files

Random & Sequential I/O



Source: The Pathologies of Big Data by Adam Jacobs (2009)
https://queue.acm.org/detail.cfm?id=1563874

Event based data



Bought a coffee ☕

Paid the rent 🏠

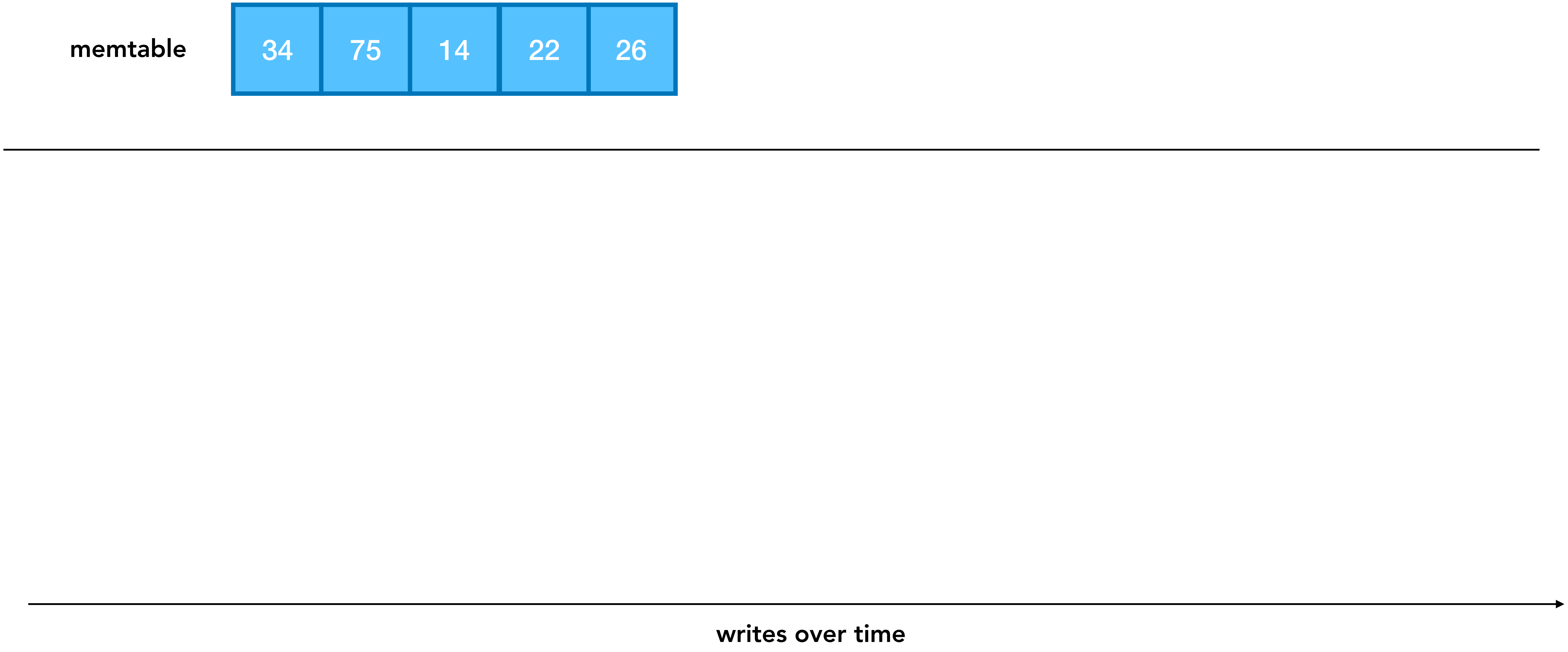
Put some money in a pot 💰

Log-Structured Merge-Tree (LSM Tree)

Make inserting data really efficient by leveraging sequential disk operations rather than lots of random disk operations

Database systems built on the LSM Tree

Inserting items

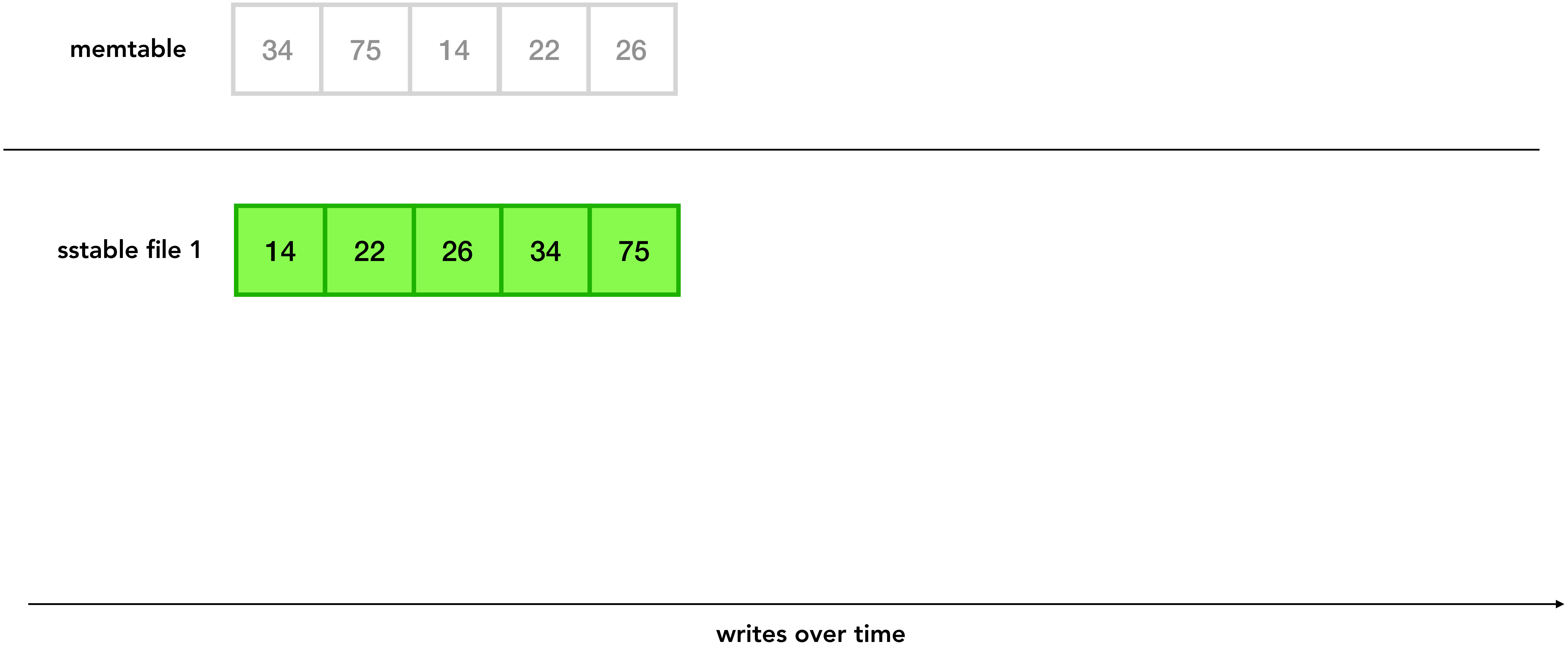


memtable	34	75	14	22	26

writes over time

```python
@dataclass
class Item(object):
    key: str
    value: str

class SRECon2022Database(object):
    memtable = []

    def insert(self, key: str, value: str):
        self.memtable.append(Item(key=key, value=value))
```

Inserting items



memtable

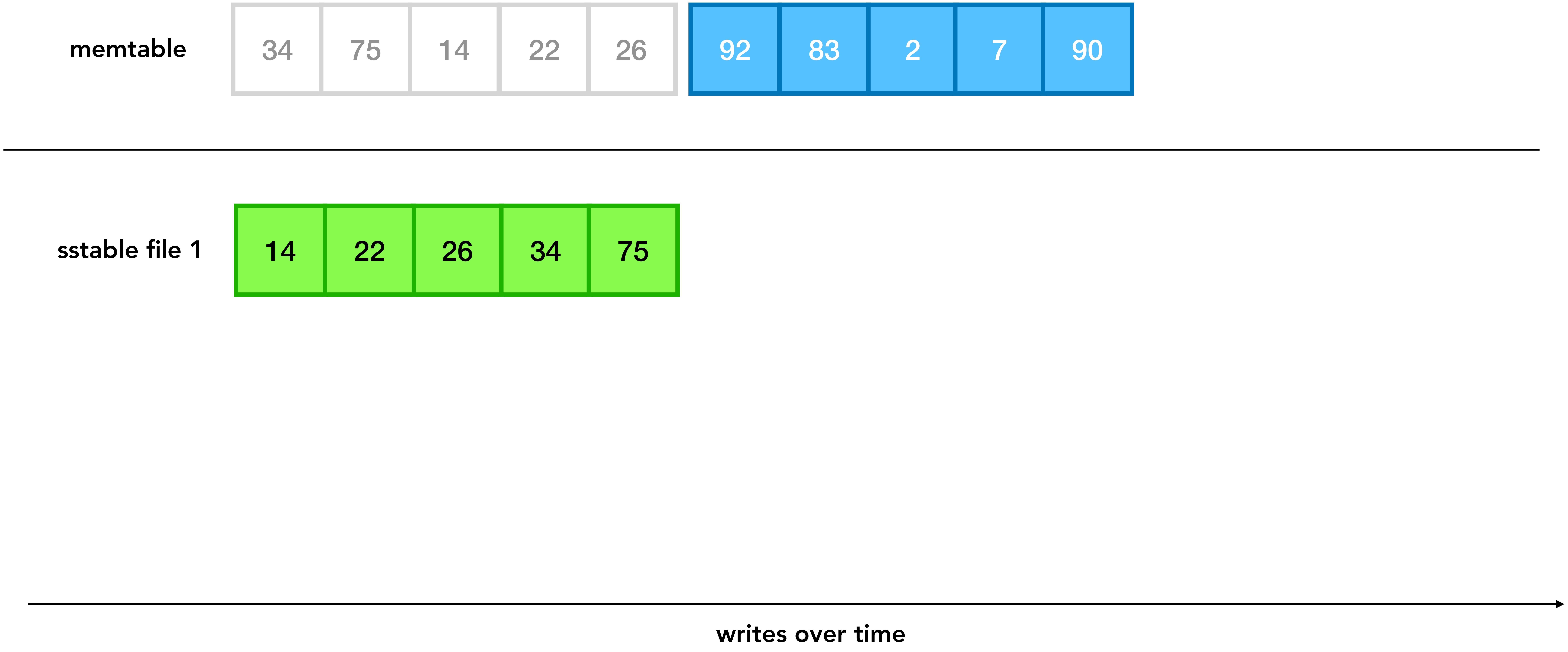
34	75	14	22	26

sstable file 1

14	22	26	34	75

writes over time

Inserting items

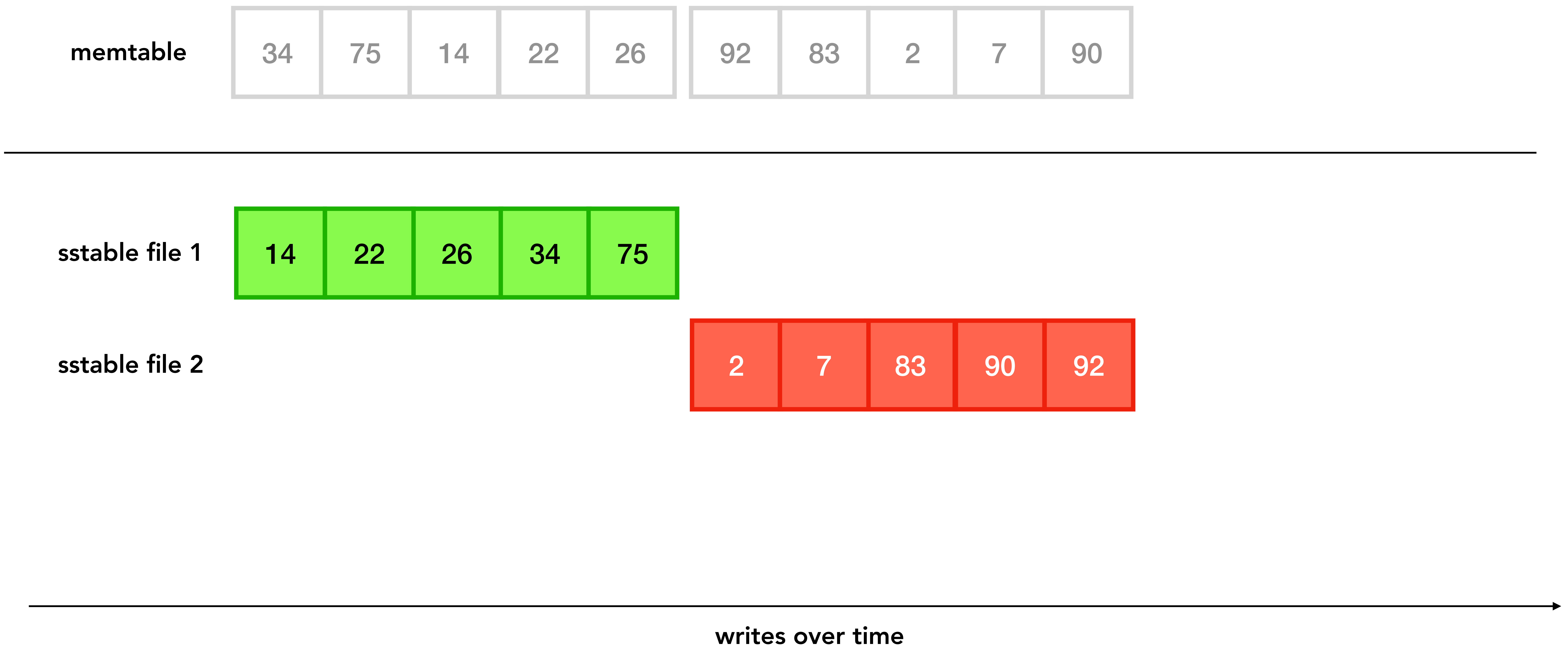


memtable 34 75 14 22 26 92 83 2 7 90

sstable file 1 14 22 26 34 75

writes over time

Inserting items



memtable 34 75 14 22 26 92 83 2 7 90

sstable file 1 14 22 26 34 75

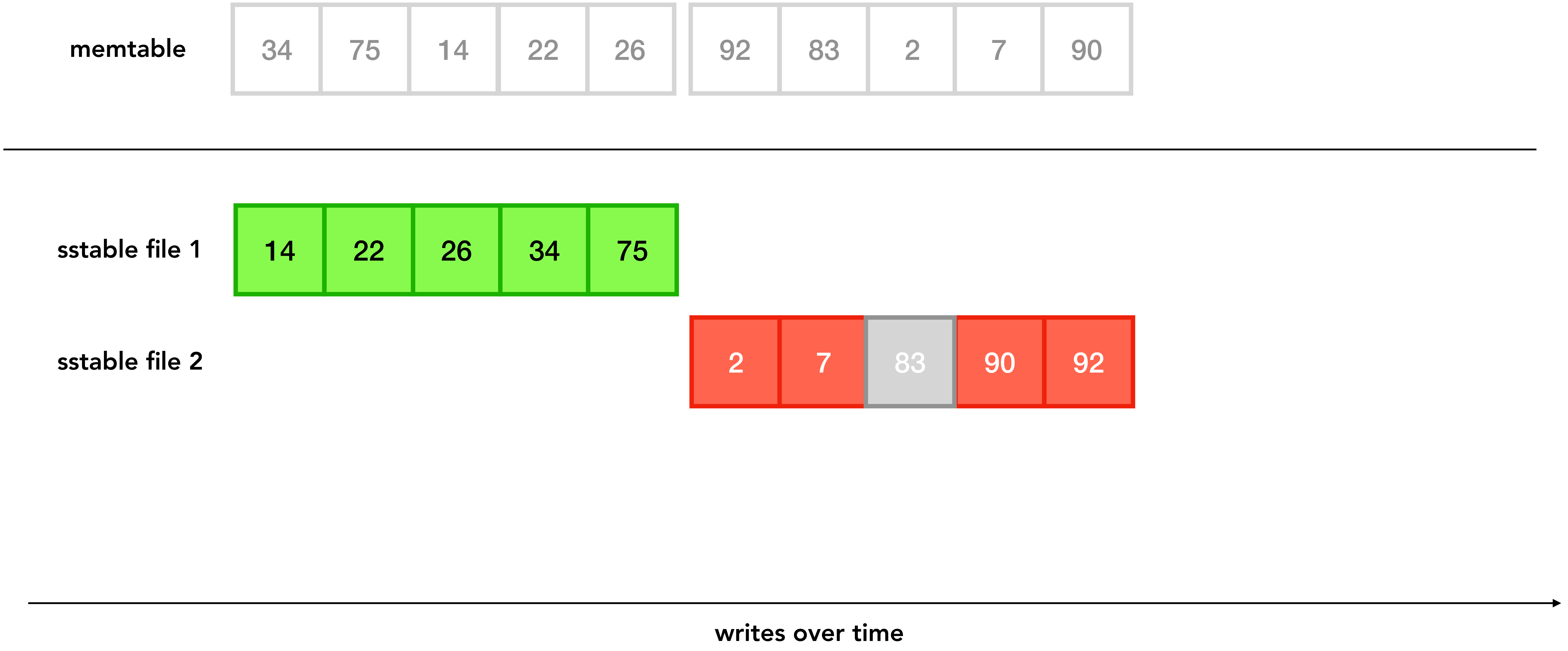
sstable file 2 2 7 83 90 92

writes over time

```
@dataclass
class Item(object):
    key: str
    value: str

    def sort_key(self):
        return (self.key)

    def to_line(self) → str:
        return f"{self.key}\n"

class SRECon2022Database(object):
    memtable, sstables = [], []

    def insert(self, key: str, value: str):
        if len(self.memtable) ≥ 5:
            self.flush()
        self.memtable.append(Item(key=key, value=value))

    def flush(self):
        filename = f”fancy-{int(time.time_ns())}-sstable.db"
        with open(filename, "a") as f:
            for item in sorted(self.memtable, key=lambda x: x.sort_key()):
                f.write(item.to_line())
                f.write('\n')

        self.memtable = []
        self.sstables.append(filename)
```

```
@dataclass
class Item(object):
    key: str
    value: str

    def sort_key(self):
        return (self.key)

    def to_line(self) → str:
        return f"{self.key}\n"

class SRECon2022Database(object):
    memtable, sstables = [], []

    def insert(self, key: str, value: str):
        if len(self.memtable) ≥ 5:
            self.flush()
        self.memtable.append(Item(key=key, value=value))

    def flush(self):
        filename = f”fancy-{int(time.time_ns())}-sstable.db”
        with open(filename, "a") as f:
            for item in sorted(self.memtable, key=lambda x: x.sort_key()):
                f.write(item.to_line())
                f.write('\n')

        self.memtable = []
        self.sstables.append(filename)
```

Deleting an item



memtable

34 75 14 22 26 92 83 2 7 90

sstable file 1

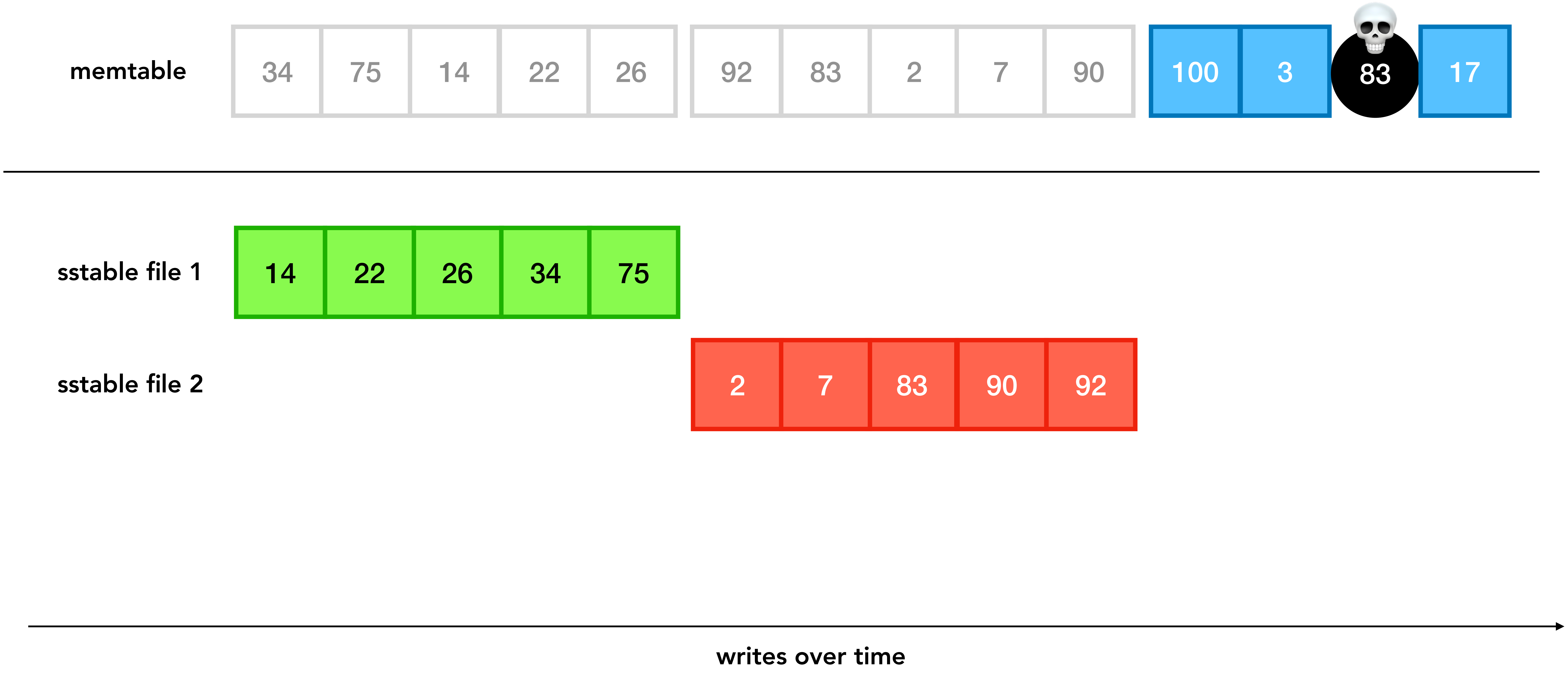
14 22 26 34 75

sstable file 2

2 7 83 90 92

writes over time

Deleting an item



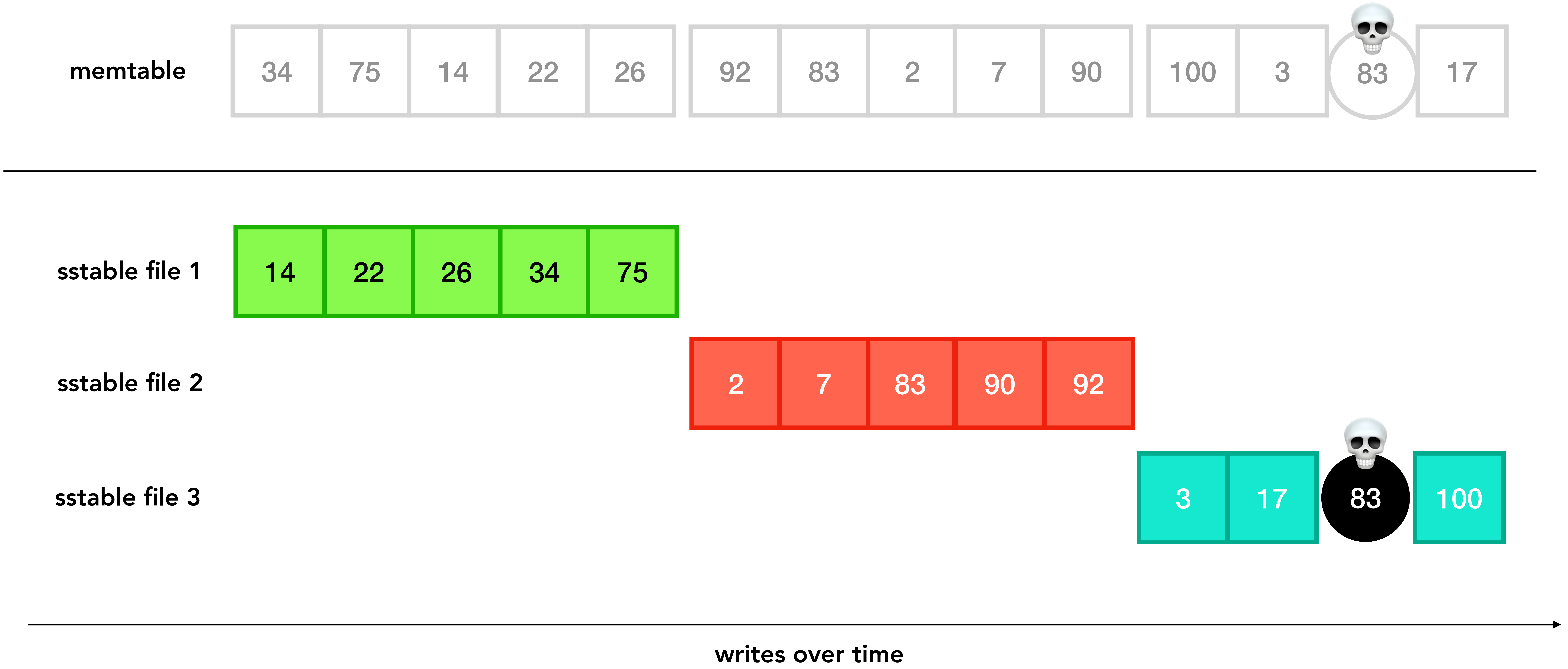
memtable: 34 75 14 22 26 | 92 83 2 7 90 | 100 3 💀 83 17

sstable file 1: 14 22 26 34 75

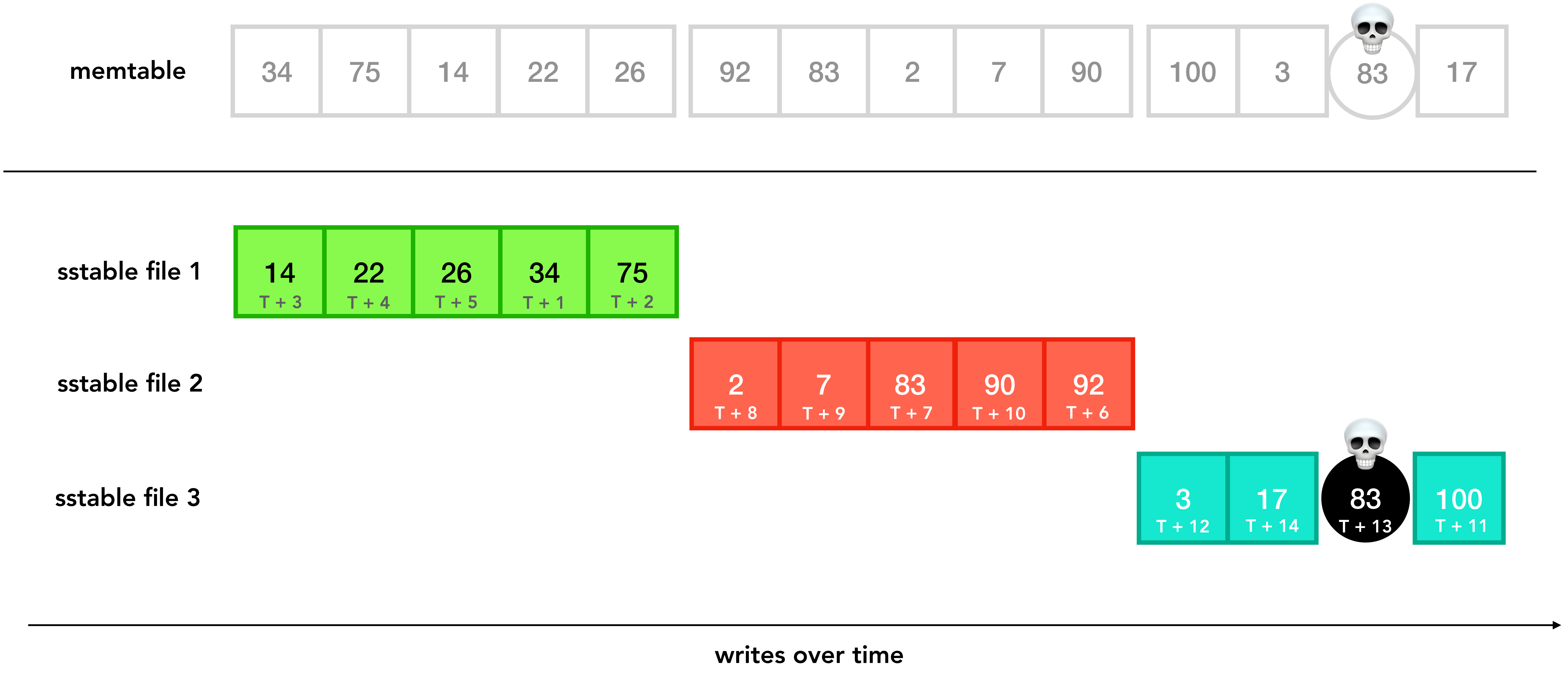
sstable file 2: 2 7 83 90 92

writes over time

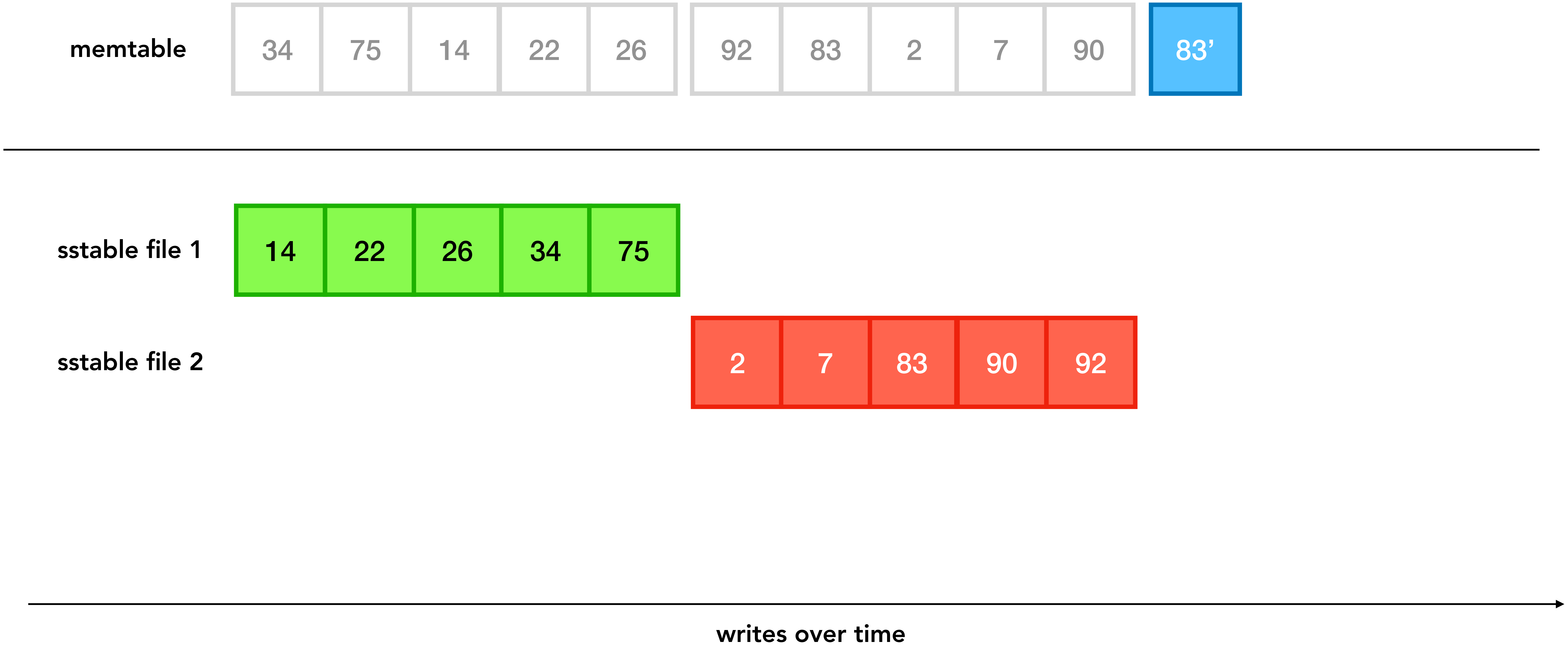
Deleting an item



Deleting an item



Updating an item



memtable: 34 | 75 | 14 | 22 | 26 | 92 | 83 | 2 | 7 | 90 | 83’

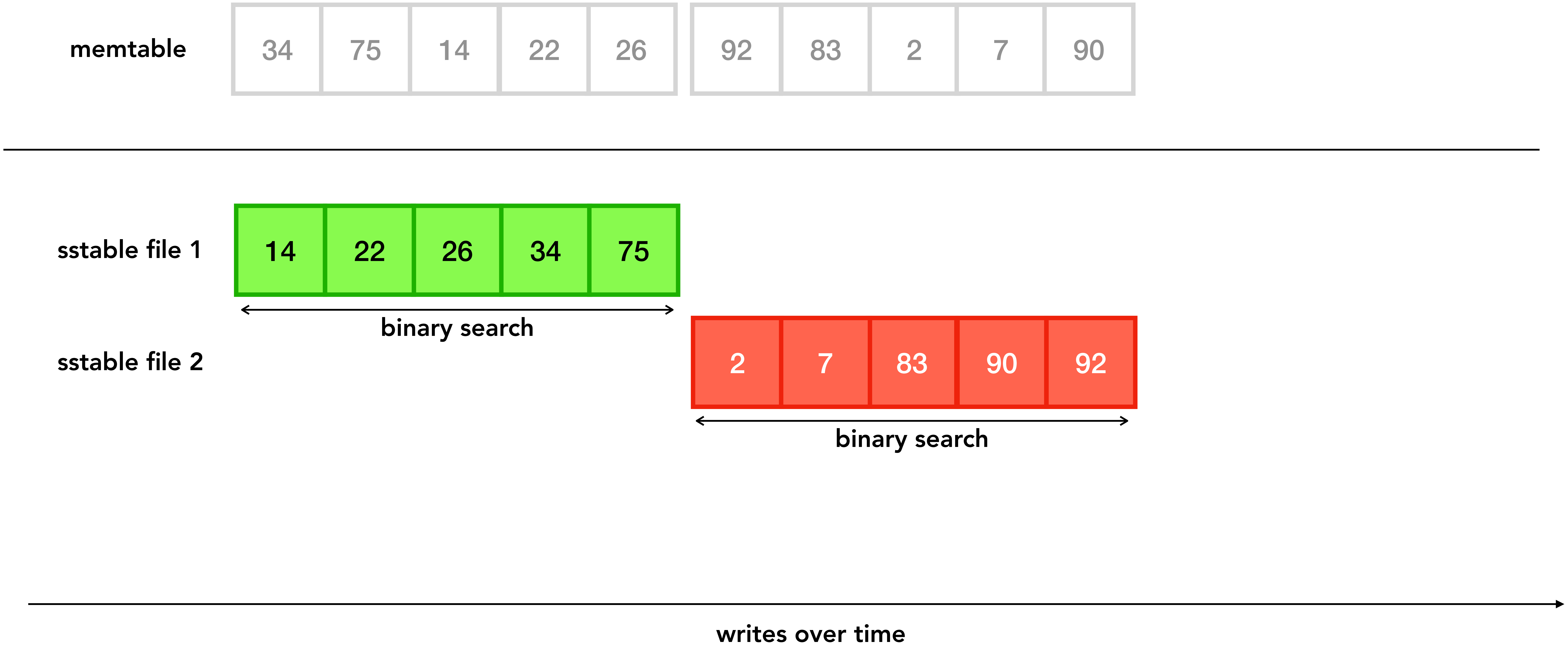
sstable file 1: 14 | 22 | 26 | 34 | 75

sstable file 2: 2 | 7 | 83 | 90 | 92

writes over time

```
class SRECon2022Database(object):
    ...

    def insert(self, key: str, value: str):
        if len(self.memtable) ≥ 5:
            self.flush()
        self.memtable.append(Item(key=key, value=value, timestamp=time.time_ns()))

    def update(self, key: str, new_value: str):
        self.insert(key, new_value)

    def delete(self, key: str):
        if len(self.memtable) ≥ 5:
            self.flush()
        self.memtable.append(Item(key=key, timestamp=time.time_ns(), is_deleted=True))

    ...
```

Searching for an item



memtable

34	75	14	22	26	92	83	2	7	90

sstable file 1

14	22	26	34	75

binary search

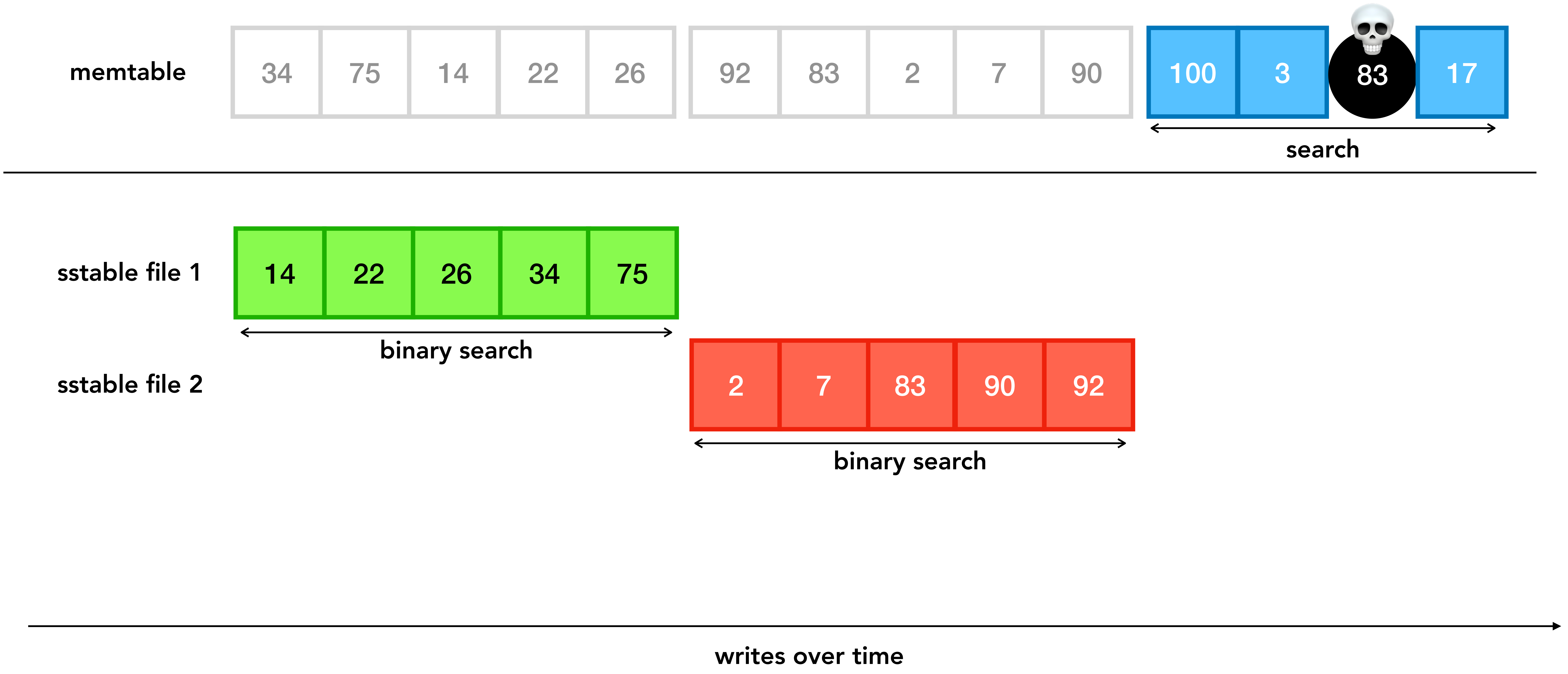
sstable file 2

2	7	83	90	92

binary search

writes over time

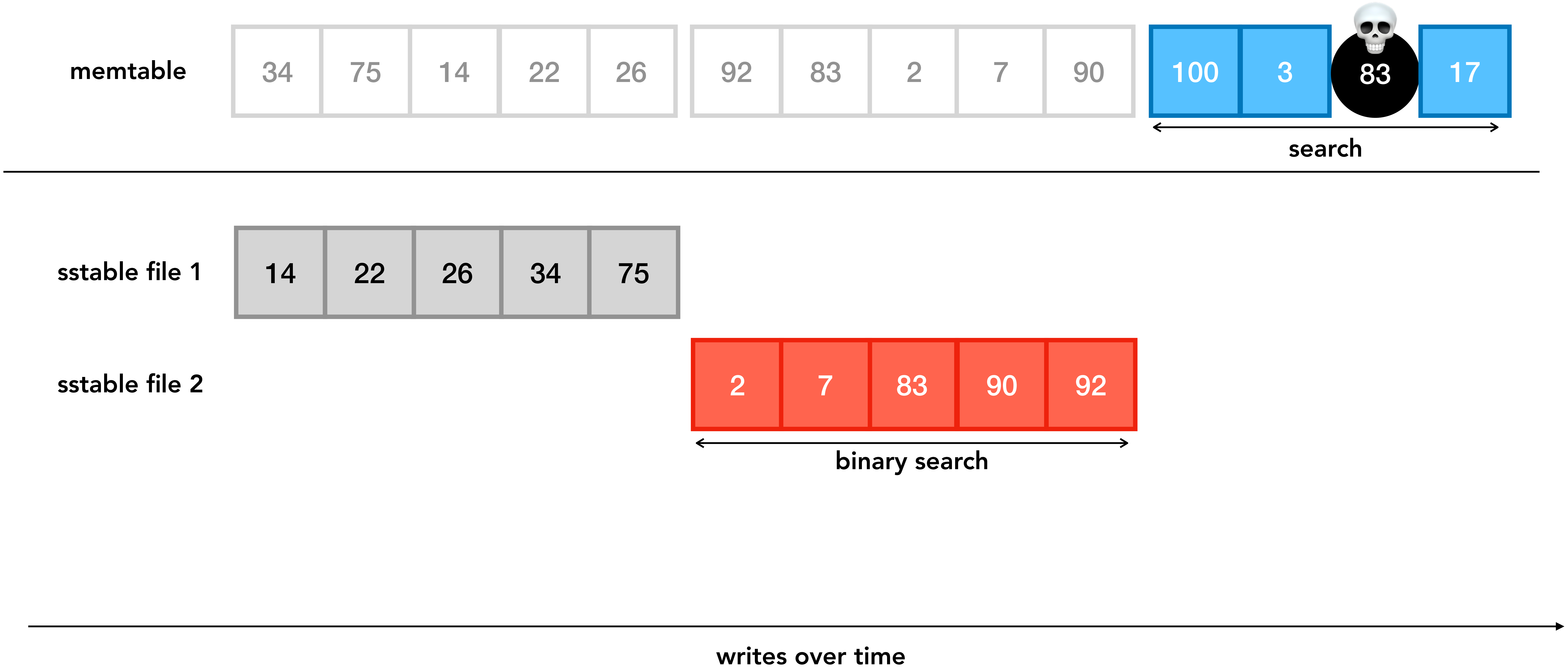
Searching for an item



```python
class SRECon2022Database(object):
    ...

    def search_in_sstable(self, sstable, key) -> List[Item]:
        records = []

        with open(sstable) as f:
           for line in f.readlines():
               if not line.strip():
                   continue

               item = Item.from_line(line.strip())
               if item.key == key:
                   records.append(item)

        return records

    ...
```

```
class SRECon2022Database(object):
    ...

    def search(self, key) -> Optional[Item]:
        records = []

        # Read from our SSTables and find any that match this particular key
        for sstable in self.sstables:
            records.extend(self.search_in_sstable(sstable, key))

        # Read from our Memtable and find any that match this particular key
        records.extend(filter(lambda x: x.key == key, self.memtable))

        # Sort by timestamp ascending pick the most recent record (last timestamp wins)
        records = sorted(records, key=lambda x: x.timestamp)

        # Apply some logic to see what we return:
        # - If we found no matches, return nothing
        # - If our last item was a deletion event, return nothing
        # - Otherwise, return the most recent result
        if not records:
            return None
        elif records[-1].is_deleted:
            return None
        else:
            return records[-1]
```

Avoid looking in unnecessary files



memtable | 34 | 75 | 14 | 22 | 26 | 92 | 83 | 2 | 7 | 90 | 100 | 3 | 💀 83 | 17

search

sstable file 1 | 14 | 22 | 26 | 34 | 75

sstable file 2 | 2 | 7 | 83 | 90 | 92

binary search

writes over time

Bloom filters

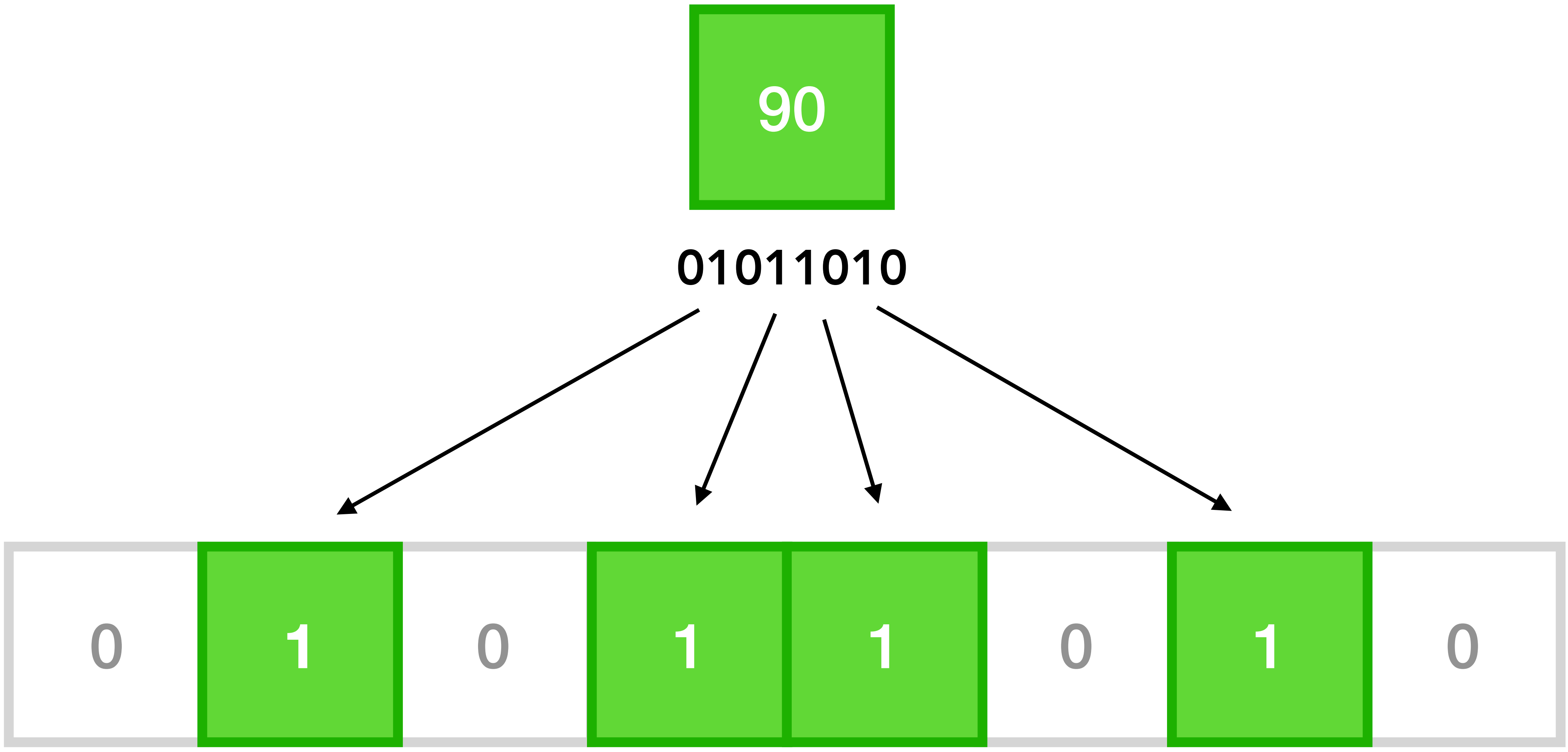
A probabilistic data structure that can be used to determine whether an item is **potentially in a set** or **definitely not in a set**

Bloom filters

0	0	0	0	0	0	0	0

bit array

Bloom filters



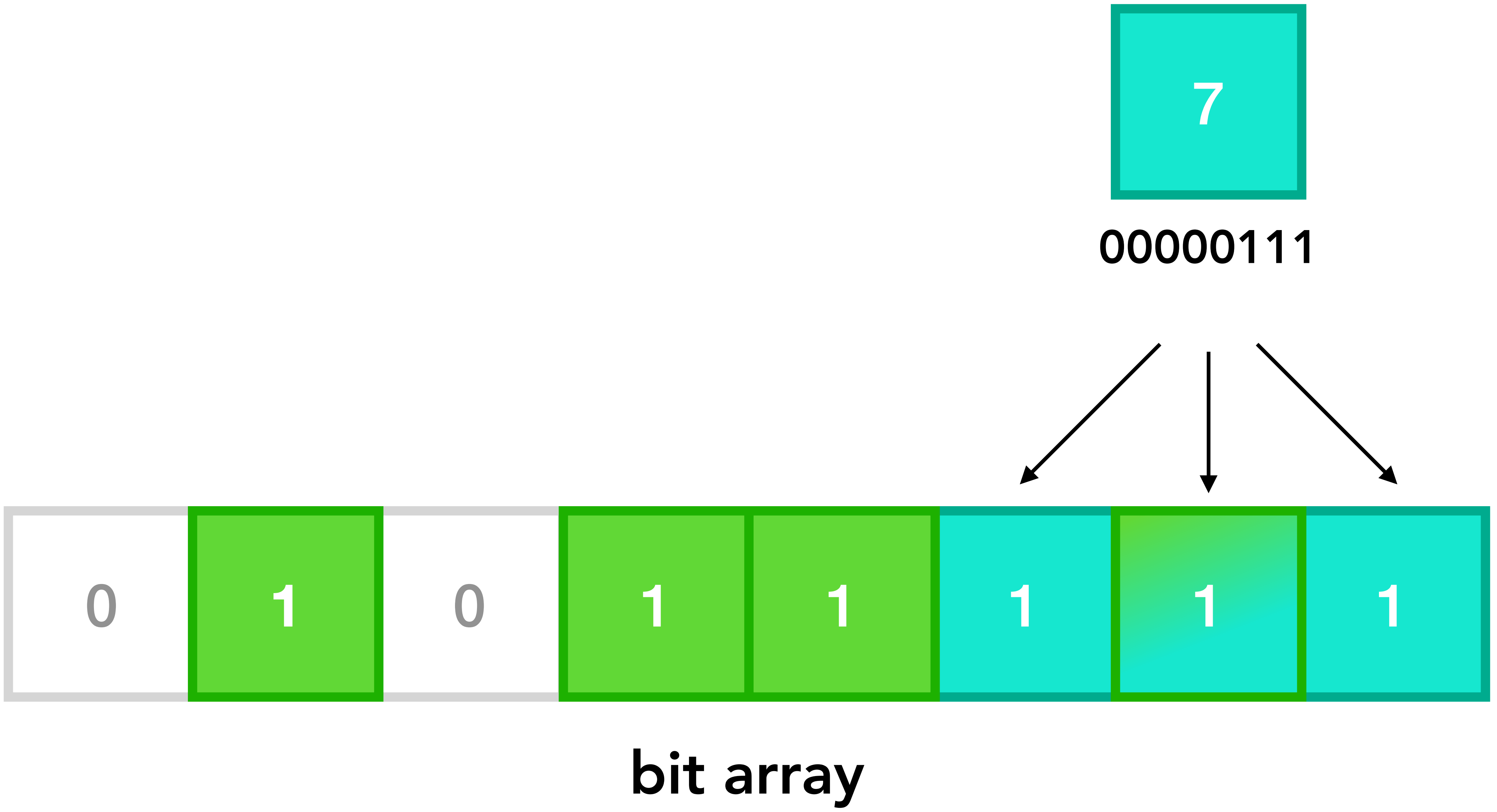
90

01011010

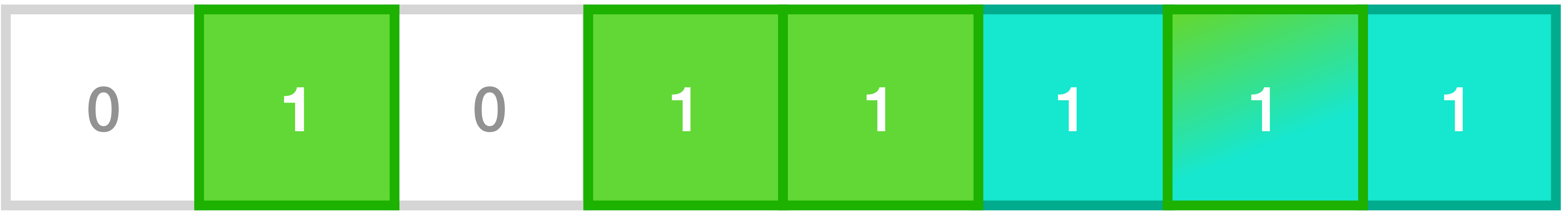
0	1	0	1	1	0	1	0

bit array

Bloom filters



Is an item in the set?

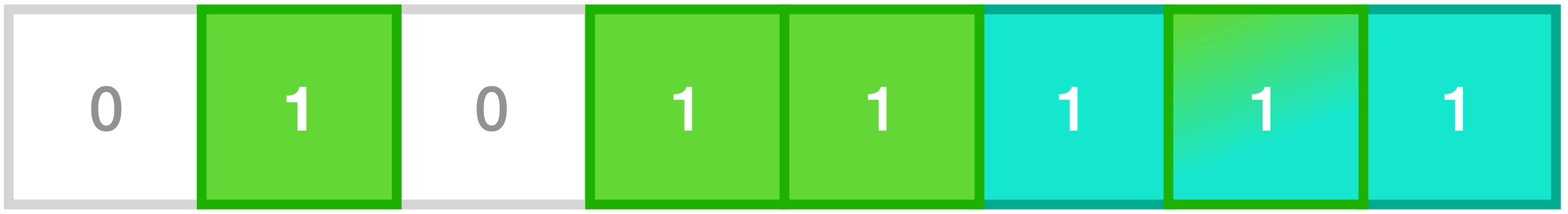


bit array

is **255** in the set? = **definitely not**

11111111

Is an item in the set?



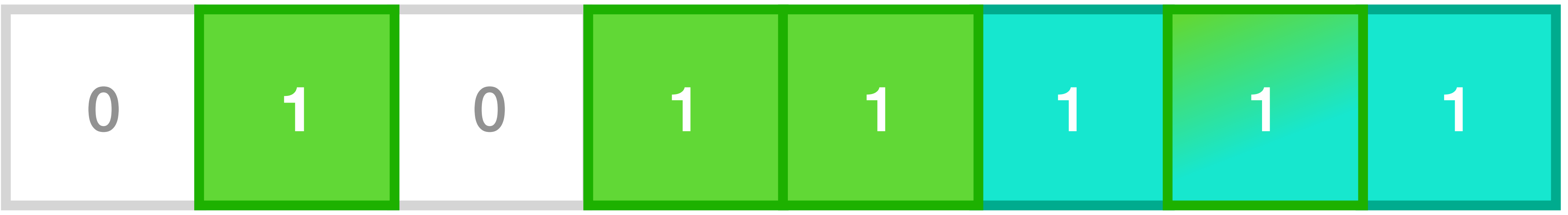
0	1	0	1	1	1	1	1

bit array

is 2 in the set? = **possibly yes**

00000010

Is an item in the set?

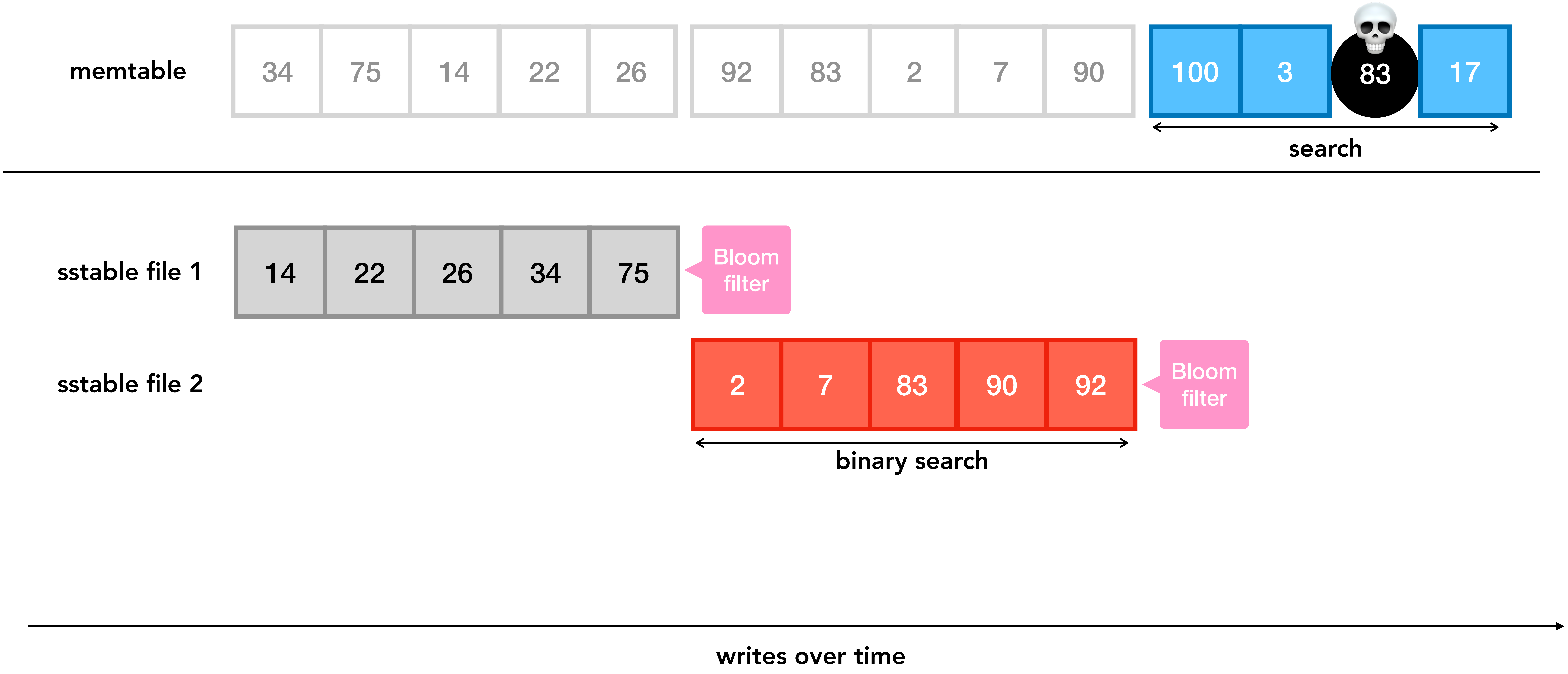


bit array

is 83 in the set? = **possibly yes**

01010011

Bloom filters to the rescue



memtable

34 75 14 22 26 92 83 2 7 90 100 3 💀 83 17

search

sstable file 1

14 22 26 34 75

Bloom filter

sstable file 2

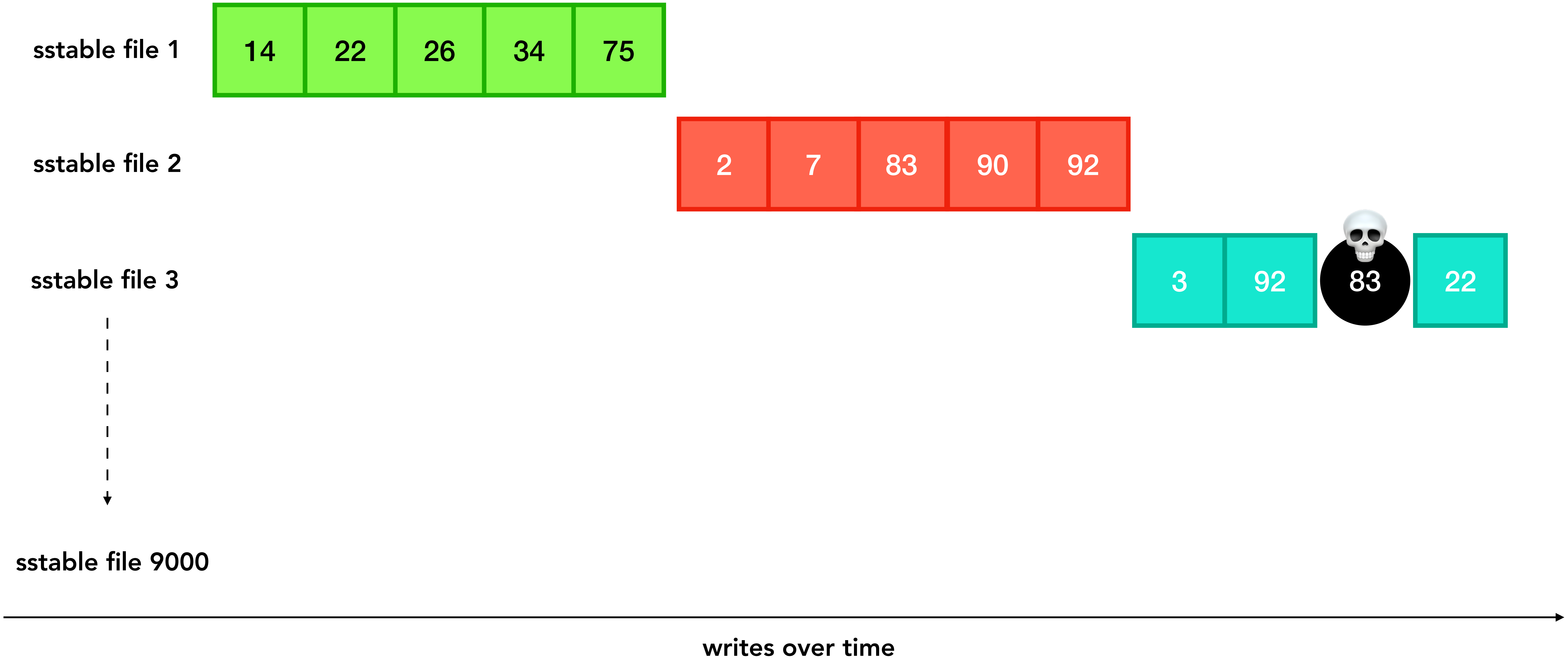
2 7 83 90 92

Bloom filter

binary search

writes over time

Compacting data together



sstable file 1: 14 22 26 34 75

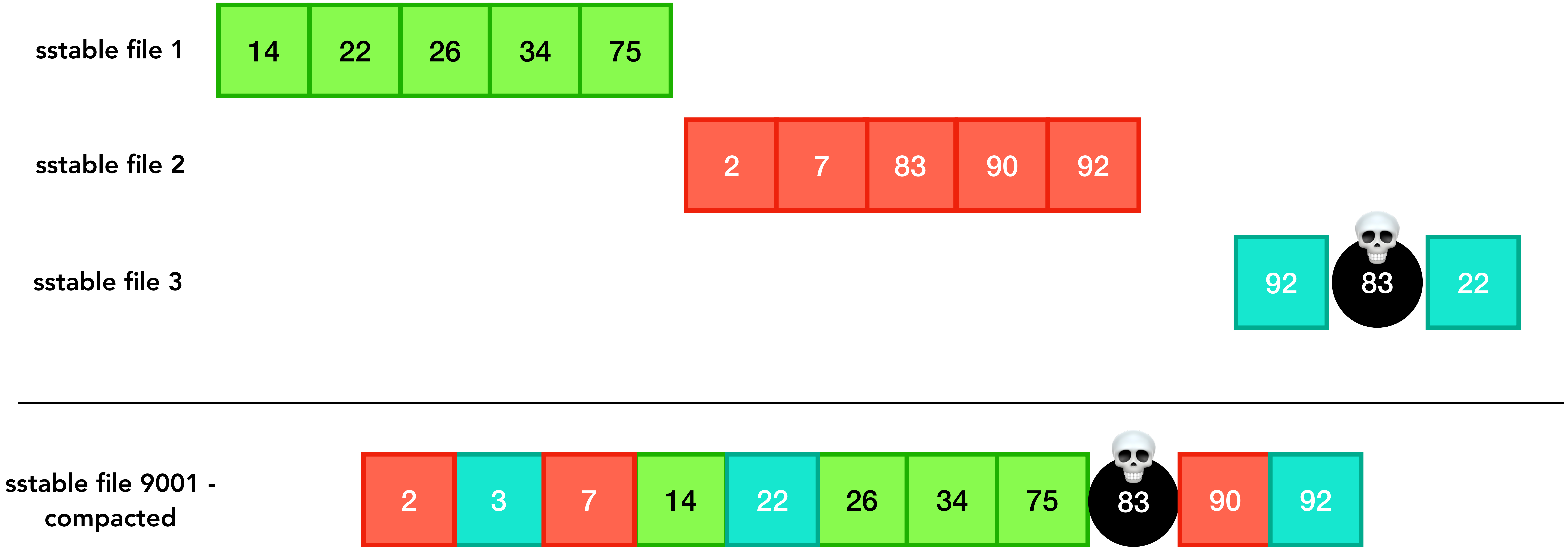
sstable file 2: 2 7 83 90 92

sstable file 3: 3 92 💀 83 22

sstable file 9000

writes over time

Compacting data together



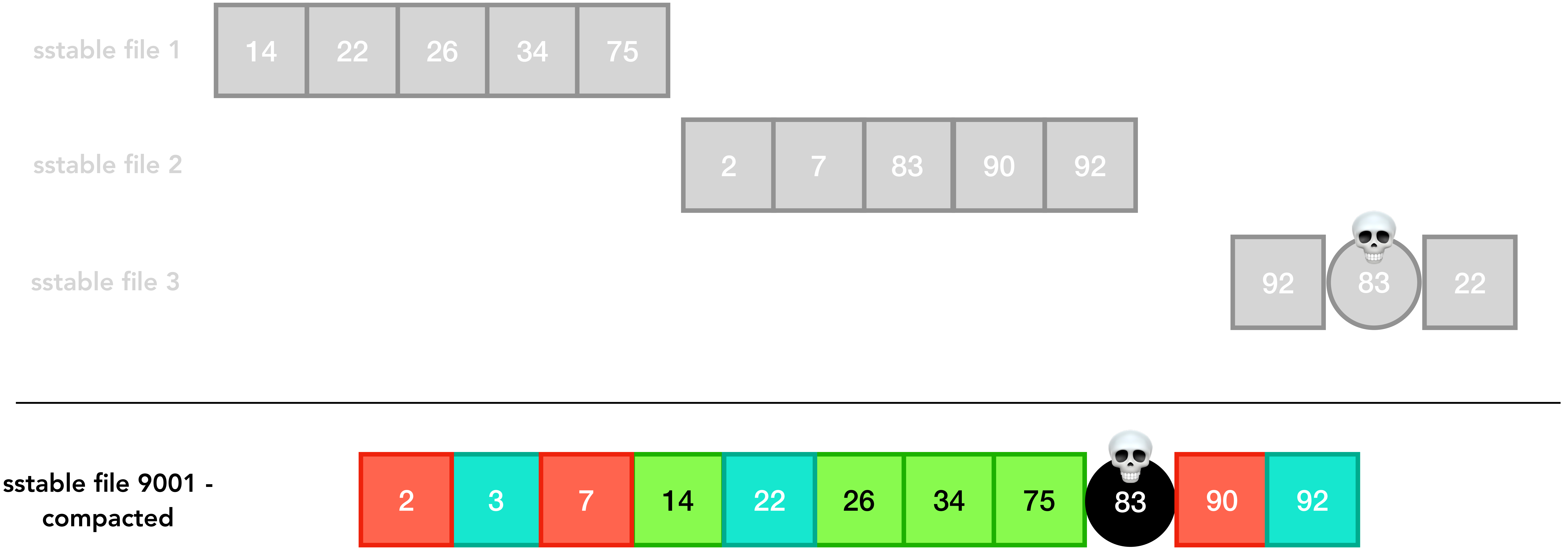
sstable file 1: 14 | 22 | 26 | 34 | 75

sstable file 2: 2 | 7 | 83 | 90 | 92

sstable file 3: 92 | 💀 83 | 22

sstable file 9001 - compacted: 2 | 3 | 7 | 14 | 22 | 26 | 34 | 75 | 💀 83 | 90 | 92

Compacting data together



sstable file 1: 14 | 22 | 26 | 34 | 75

sstable file 2: 2 | 7 | 83 | 90 | 92

sstable file 3: 92 | 💀 83 | 22

sstable file 9001 - compacted: 2 | 3 | 7 | 14 | 22 | 26 | 34 | 75 | 💀 83 | 90 | 92

Compaction strategies

- **Size Tiered** - Aim to merge together SSTables of a similar size into larger SSTable files
- **Levelled** - Aim to group SSTables in a way that keys are not spread across multiple files, making reads more efficient
- **Time Window** - Aim to group data that has a similar timestamp within a particular time window

sqlite.org/src4/doc/trunk/www/lsmusr.wiki

2. Using LSM in Applications

LSM is not currently built or distributed independently. Instead, it is part of the SQLite4 library. To use LSM in an application, the application links against libsqlite4 and includes the header file "lsm.h" in any files that access the LSM API.

Pointer to build instructions for sqlite4

3. Basic Usage

3.1. Opening and Closing Database Connections

Opening a connection to a database is a two-step process. The lsm_new() function is used to create a new database handle, and the lsm_open() function is used to connect an existing database handle to a database on disk. This is because some database connection properties may only be configured before the database is opened. In that case, one or more calls to the lsm_config() method are made between the calls to lsm_new() and lsm_open().

The functions are defined as follows:

```
int lsm_new(lsm_env *env, lsm_db **pDb);
int lsm_open(lsm_db *db, const char *zFile);
```

Like most lsm_xxx() functions that return type int (the exception is lsm_csr_valid()), both of the above return LSM_OK (0) if successful, or an LSM error code otherwise. The first argument to lsm_new() may be passed either a pointer to a database environment object or NULL. Almost all applications should pass NULL. A database environment object allows the application to supply custom implementations of the various operating system calls that LSM uses to read and write files, allocate heap memory, and coordinate between multiple application threads and processes. This is normally only required if LSM is being used on a platform that is not supported by default. Passing NULL instructs the library to use the default implementations of all these things. The second argument to lsm_new() is an output variable. Assuming the call is successful, *pDb is set to point to the new database handle before returning.

The first argument passed to lsm_open() must be an existing database handle. The second is the name of the database file to connect to. Once lsm_open() has been successfully called on a database handle, it can not be called again on the same handle. Attempting to do so is an LSM_MISUSE error.

For example, to create a new handle and connect it to database "test.db" on disk:

```
int rc;
lsm_db *db;

/* Allocate a new database handle */
rc = lsm_new(0, &db);
if( rc!=LSM_OK ) exit(1);

/* Connect the database handle to database "test.db" */
rc = lsm_open(db, "test.db");
if( rc!=LSM_OK ) exit(1);
```

A database connection can be closed using the lsm_close() function. Calling lsm_close() disconnects from the database (assuming lsm_open() has been successfully called) and deletes the handle itself. Attempting to use a database handle after it has been passed to lsm_close() results in undefined behaviour (likely a segfault).

```
rc = lsm_close(db);
```

It is important that lsm_close() is called to close all database handles created with lsm_new(), particularly if the connection has written to the database. If an application writes to the database and then exits without closing its database connection, then subsequent clients may have to run "database recovery" when they open the database, slowing down the lsm_open() call. Additionally, not matching each successful lsm_new() call with a call to lsm_close() is a resource leak.

Counter-intuitively, an lsm_close() call may fail. In this case the database handle is not closed, so if the application exits it invites the "database recovery" performance problem mentioned above. The usual reason for an lsm_close() call failing is that the database handle has been used to create database cursors that have not been closed. Unless all database cursors are closed before lsm_close() is called, it fails with an LSM_BUSY error and the database handle is not closed.

3.2. Writing to a Database

What did we cover?

- **The choice of data structure matters**, understand the relationship between hardware and software

What did we cover?

- **The choice of data structure matters**, understand the relationship between hardware and software
- **LSM Trees and SSTables can be simple**, and for many programming languages, you don’t need to re-implement from scratch

What did we cover?

- **The choice of data structure matters**, understand the relationship between hardware and software
- **LSM Trees and SSTables can be simple**, and for many programming languages, you don't need to re-implement from scratch
- **You can add optimisations** like bloom filters and compaction and tune these based on resource consumption prioritisation

Thank you!

Suhail Patel | @suhailpatel | https://suhailpatel.com