Today 🍕🍕🍕🍕 is on us

Before we begin

WeTransfer reserves the right to investigate, provide to third parties, (temporarily) block and/or remove from its servers, without warning, any transfers, files, wallpapers and/or Accounts or to block anyone from accessing any part of the Website or Service, when WeTransfer ascertains, at their own discretion or after receiving substantiated and valid complaints, that You violate these Terms or act in violation of any applicable law or regulation.

https://wetransfer.com/legal/terms

heresy |ˈhɛrɪsi|

belief or opinion contrary to orthodox religious (especially Christian) doctrine

We have a problem.

With spam.

Best practice of spam filters and fraud detection systems

- You never give talks about them
- You never tell anyone how they work
- You never admit they do exist

So, spam. Something has to be done.

We need a high-speed classifier that works at roughly 5K items **per minute.**

This:

```
if Spamo.advisory(transfer).spam?
  StrictMeasures.take! # ™©®
end
```

For every single transfer.

Best practice: design microservices for availability

We have a luxury that our service is optional. It provides advisories, not orders – it can have a sensible failure default

```
# In the calling application...
module Spamo
  HAM = {ok: true, advisory: 'ham'}
  def advisory(for_transfer)
    perform_request(service_url, for_transfer.to_json)
  rescue => e # Includes Patron timeout, which we set very low (300 millis)
    Appsignal.add_exception(e) # We should know something happened
    HAM # Pretend the default result is "ham"
  end
end
```

If we have a catastrophic failure, nothing breaks.

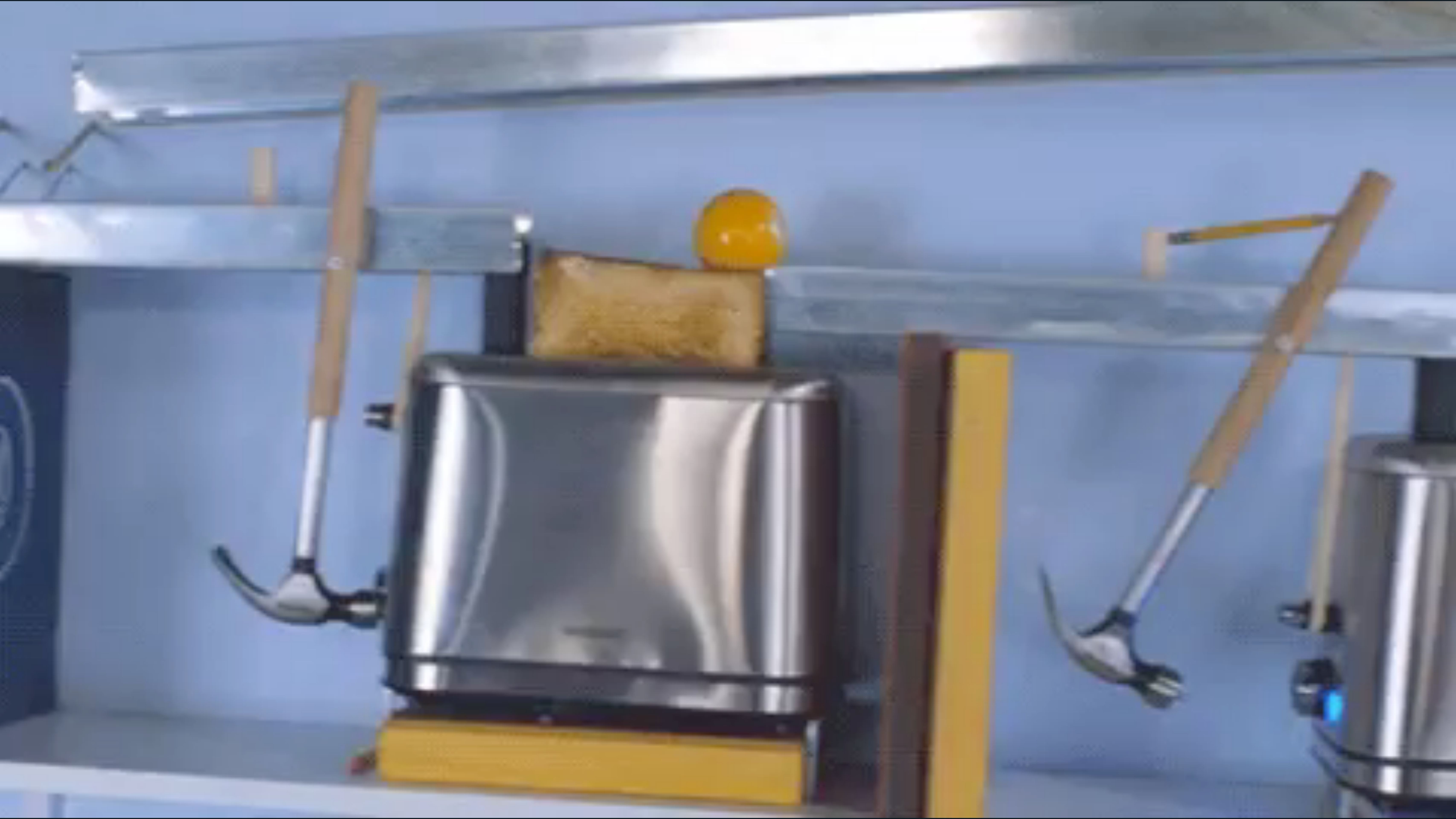
It's this basically:

```
try {
  ...
} catch(e) {
  // nevermind
}
```

We had Spamo broken in production. For a few days.
Nobody noticed.

Best practice of AWS high-ingress systems

- An endpoint in AWS API Gateway that...
- Talks to a Lambda function which...
- Submits the payload to Kinesis Firehose, which...
- Submits the payload to an SQS queue which...
- Writes the entries to DynamoDB which...
- then get bulkified to S3 from which...
- you read them into Redshift where...



Best practice of non-AWS high-ingress systems

— Kafkasparksamzazookeeperomgbbq.

This stuff is not free

But most importantly, it is not free in terms of operator workload. Services have to be provisioned. Terraform modules have to be written. Ansible playbooks have to played. It is more stuff.

And worst of all, you wanted **one** more stuff, but you are getting the kitchensink, the kitchen and the house.

Here's what we are going to do instead.

- One database
- One server
- No containers
- No AWS-provided, closed source, async databases [1]
- No Lambda
- No VPCs
- No `org.apache.commons.bigdata.factory.provider`

[1] That for some reason are only decently usable from Java

THE DATABASE IS A DETAIL!

And which database shall we choose?

— PostgreSQL?
— MySQL?
— Mongo?
— CockroachDB?
— Influx?
— Redis?
— ElasticSearch

And which one shall we choose... hmm...

- PostgreSQL? - requires a daemon, lots of config
- MySQL? - requires a daemon, lots of config
- Mongo? - don't even!
- CockroachDB? - A notch too experimental
- Influx? - It's not time-series, mostly
- Redis? - I want to query. Conveniently.
- ElasticSearch - costly deletes

Desired perf

- Extremely fast reply to a classifier query (may read, should not need to write right now)
- Writes may be deferred (eventual consistency)
- Writes may be dropped in overflow situations

Consistency SQL-style

```
commit_txn(&txn)
fsync_pages(&pages_touched)
really_really_fsync_pages(&pages_touched)
really_totally_oncemore_fsync_pages(&pages_touched)
```

Consistency MongoDB style

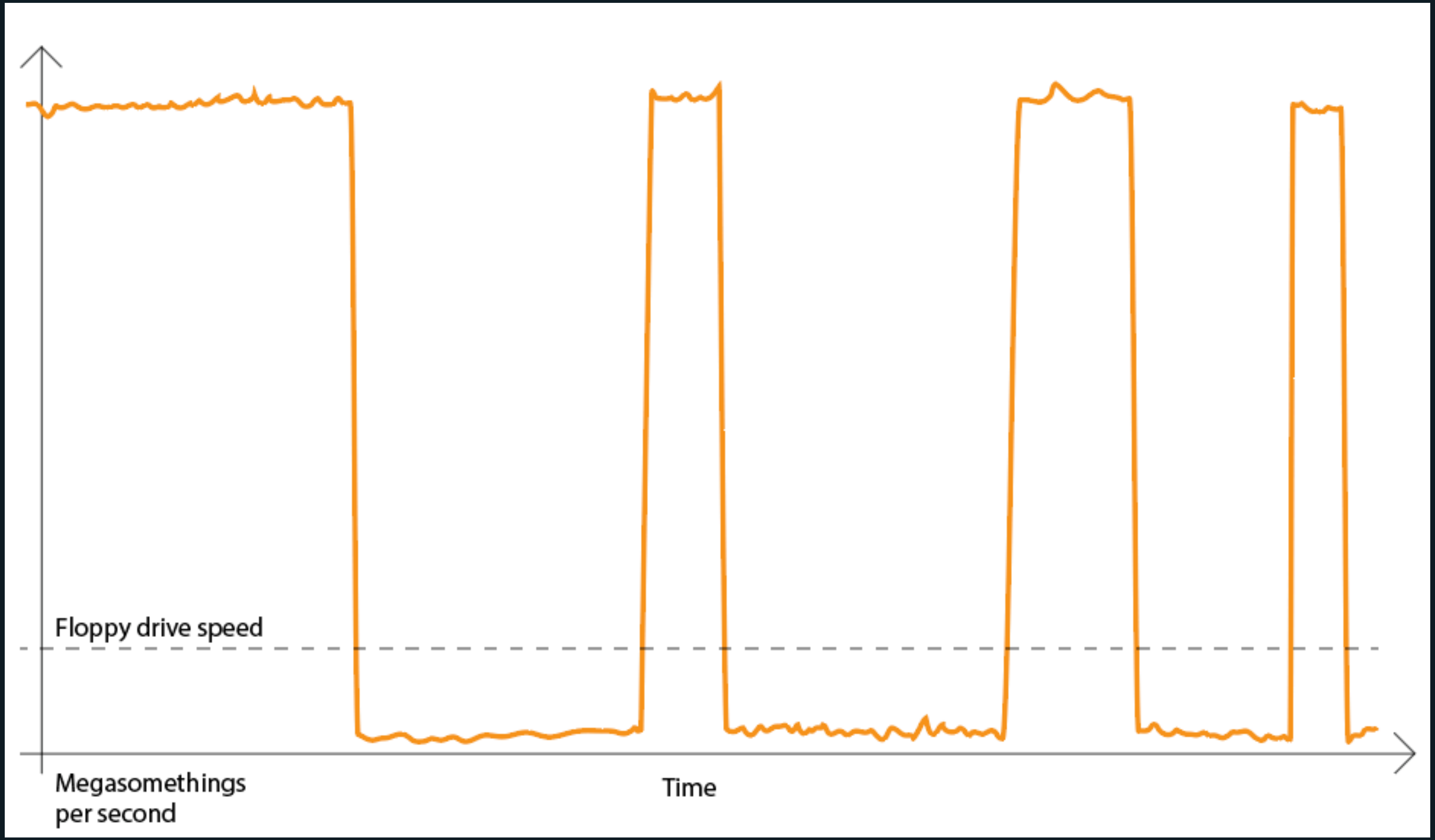
```
commit_txn(&txn)
yolo(NULL)
```

Databases store data. On the filesystem.

I/O Credits and Burst Performance

The more credits your storage has for I/O, the more time it can burst beyond its base performance level…

. Your IO performance on a typical EBS volume on AWS:



Database in memory

Because cloud providers do not meter MIOPS yet. They will. One day.

We are going to use the filesystem. A very special one.
We call it `/run/shm`

https://askubuntu.com/a/169526

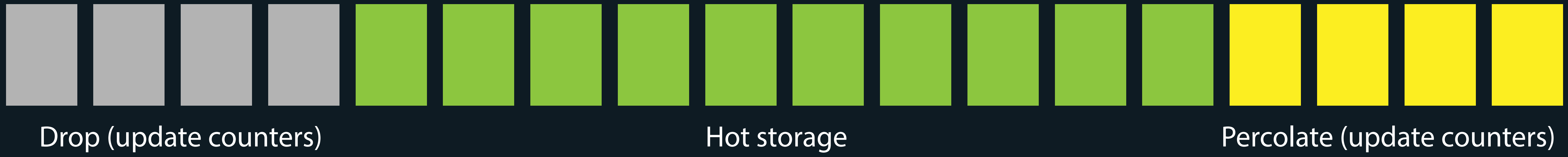
`/run` is, in general, a temporary filesystem (tmpfs) residing in RAM (aka "ramdisk"); its meant for storing "temporary" system or state files which may be critical but do not require persistence across reboots.

Database best practice
for write-heavy environments:
Use a client-server database solution!

SQLite

SQLite does not compete with client/server databases. SQLite competes with fopen().

Our data model will be this



```
# Evaluation
rating_module.rating_for(transfer, recently_seen: log)

# Ingress
log.append(transfer)
```

- We need an event log (items incoming on the stream)
- We never make it sparse (never delete from the middle)
- We never persist very old items out-of-sequence
- A very big LIFO stack
- Kind of like…. a **Kafka** topic sans the multiple consumers with offsets

Let's append

```
CREATE TABLE event_log (monotonic_id SERIAL PRIMARY KEY, payload TEXT)

BEGIN -- DEFERRED is the SQLite default, no need to mention it
SELECT MAX(monotonic_id) FROM event_log -- Where are we currently?
INSERT OR IGNORE INTO event_log (payload) VALUES (?) -- a few thousands, in bulk
SELECT MAX(monotonic_id) FROM event_log -- How many did we insert, roughly?
COMMIT -- fsync!
```

Let's process the freshly inserted events (our **segment**):

```
SELECT COUNT(id) AS hits, advisory FROM event_log GROUP BY advisory WHERE id BETWEEN ? AND ?
```

Aggregate queries are slow as long as there is a lot to **sort** through. But if we can reduce it to a per-segment set of operations...

That is, to the slice we just written. This we call percolation and is a concept somewhat similar to ElasticSearch.

Map-reduce all the things!
…without Hadoop.

It is very cheap to delete

```
db.execute("DELETE FROM event_log WHERE monotonic_id < ?", cutoff_id)
```

As well as to update counters

```
num_spams_in_segment = db.get_first_value(
  "SELECT COUNT(id) FROM event_log
   WHERE rating='spam' AND id < ?")
db.execute(
  "UPDATE counters SET value = value + ? WHERE counter_name = 'spams_in_event_log'", num_spams_in_segment)
```

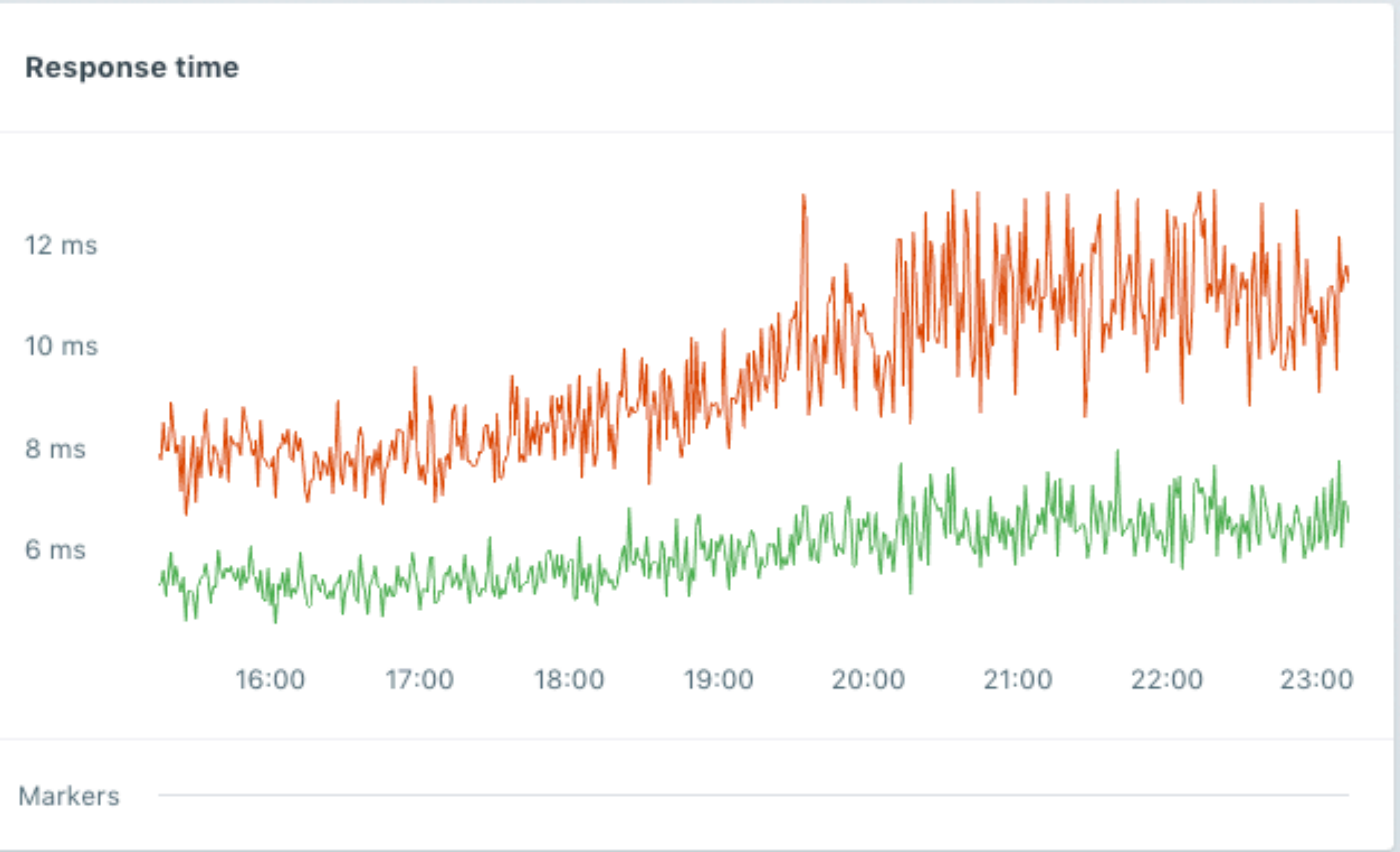
The magic bit is this:

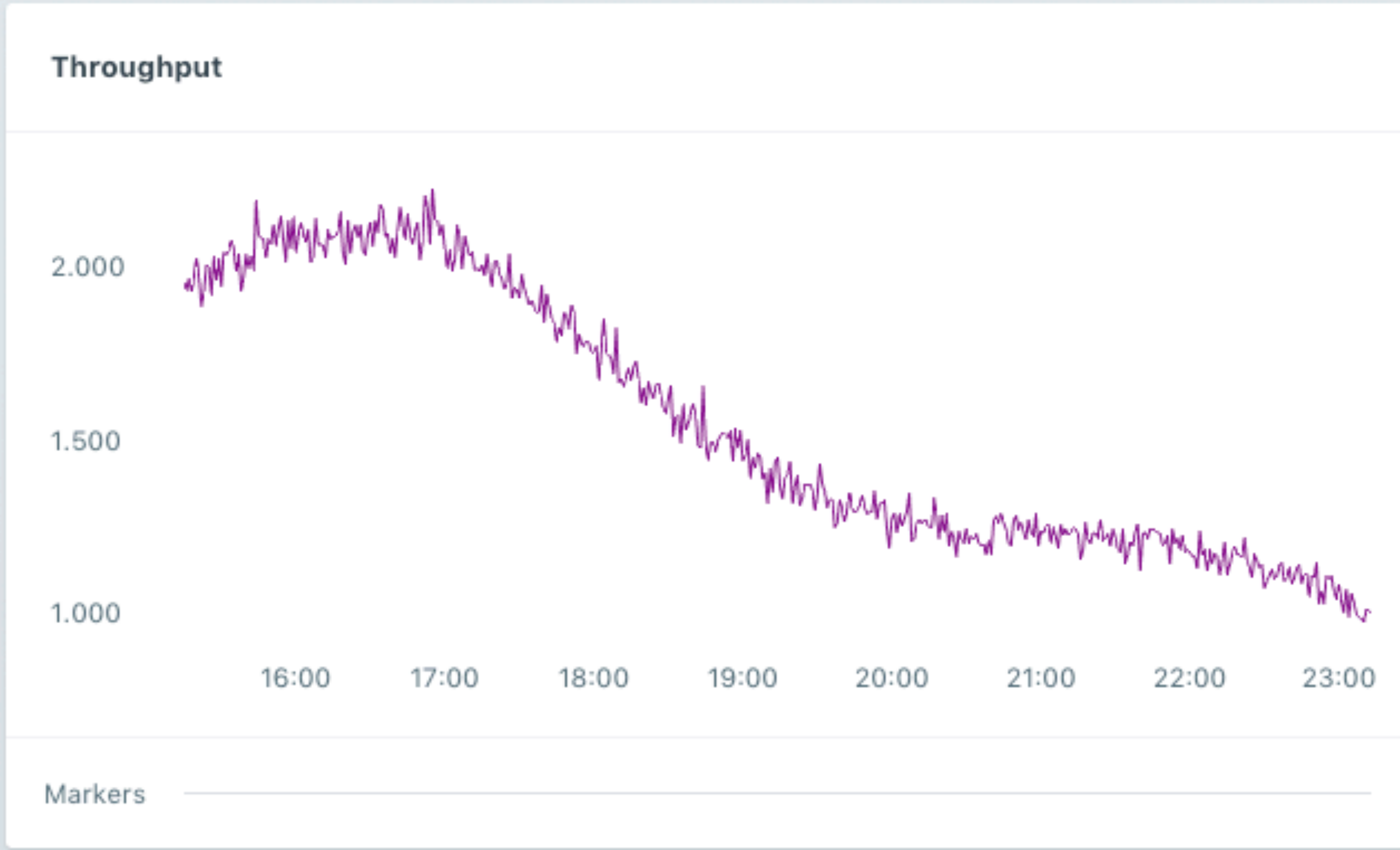
```
SELECT ... FROM event_log WHERE monotonic_id BETWEEN x AND y
```

No LIMIT, no OFFSET

Recap: to give you a `LIMIT` and `OFFSET`, the SQL engine has to sort the entire output first.

Best streaming service practice:
Use $fastlang!..


Response time
12 ms
10 ms
8 ms
6 ms
16:00
17:00
18:00
19:00
20:00
21:00
22:00
23:00
Markers


Throughput
2.000
1.500
1.000
16:00
17:00
18:00
19:00
20:00
21:00
22:00
23:00
Markers

How even?

- 1 server (lots of RAM for ramfs)
- 2 Pumas, 16 threads each
- Each Puma worker has a backround writer thread

A writer thread?

```ruby
class IntermediateFlushes < Struct.new(:app)
  PROM = Mutex.new

  def call(rack_env)
    PROM.synchronize do
      @worker ||= Thread.new do
        loop do
          # If this call raises something more severe than a StandardError,
          # this thread will terminate since the exception will propagate.
          perform_background_data_operations!
          sleep(SLEEP_SECONDS_BETWEEN_FLUSHES)
        end
      end
    end

    app.call(rack_env)
  end
end
```

And within perform_background_data_operations!:

```
Spamo.perform_deferred_db_writes! if rand(0...WRITE_ONCE_IN_N).zero?
Spamo.prune_stored_transfers! if rand(0...PRUNE_ONCE_IN).zero?
Spamo.vacuum! if rand(0...VACUUM_ONCE_IN).zero?
Spamo.send_database_stats_to_metrics!
```

Spamo.send_database_stats_to_metrics **and** .vacuum!

```
byte_size = db.get_first_value('PRAGMA page_size').to_i *
  db.get_first_value('PRAGMA page_count').to_i
```

We VACUUM every N flushes. In general, SQLite reuses pages very well.

`Spamo.perform_deferred_db_writes!`

When the service rates a transfer using transfers in the log we apply one of the best known memory saving practices: send a block to a Queue.

```
deferred_writes << ->(db) {
  db.execute("INSERT INTO stored_transfers ...", values)
}
```

It **doesn't leak.**

We then read everything from the queue buffered until now (kind of like a buffered Go channel, but unordered)

```
# Take all the pending writes from the queue
deferred_write_procs = []
while deferred_write = @queued_writes.pop(non_block=true) rescue nil
  deferred_write_procs << deferred_write
end
return if deferred_write_procs.empty?
```

```ruby
# Execute the writes
from, to = Spamo.with_writable_db do |db|
  db.execute("BEGIN TRANSACTION")

  # Note where the log is at at the moment
  last_id_before_writes = DataOperations.last_created_transfer_id(db)

  # Perform all the writes we retreived from the queue
  deferred_write_procs.each { |defp| defp.(db) }

  # Peek which range of IDs we have been allocated
  last_id_after_writes = DataOperations.last_created_transfer_id(db)
  db.execute("COMMIT")

  [last_id_before_writes, last_id_after_writes]
end
```

```ruby
# Call percolators (update stats for the slice)
percolators.each do |perc|
  Spamo.with_writable_db
    perc.after_insert(db, from, to)
  end
end


...
# ...and before delete (remove contributing stats for the slice)
percolators.each do |perc|
  Spamo.with_writable_db
    perc.before_delete(db, cutoff_id)
  end
end
```

Example percolator - maintains per minute stats. For a fast `UPSERT` we use `INSERT OR IGNORE ...` followed by `UPDATE ...`

```ruby
class PerMinuteRatingsPercolator
  def call(db, first_id, last_id)
    stmt = "SELECT created_at, advisory FROM stored_transfers WHERE id BETWEEN ? AND ?"
    buckets = db.execute(stmt, first_id, last_id).each_with_object({}) do |(created_at, advisory), minute_buckets|
      created_at = Time.parse(created_at)
      minute_ts = created_at.to_i - (created_at.to_i % 60)
      minute_buckets[minute_ts] ||= {"ham" => 0, "spam" => 0, "maybe" => 0}
      minute_buckets[minute_ts][advisory] += 1
    end

    buckets.each do |(minute_ts, advisories)|
      DatabaseUtils.increment_counters(db, :rating_datapoints, conditions: {ts: minute_ts}, increments: advisories)
    end
  end
end
```

What if we are above throughput? We will put the items back into the queue since there will be less throughput later in the day. Or in the hour.

```
  ...
rescue SQLite3::BusyException => e
  # E_TROUBLE_SO_HIGH. Let's try again later.
  deferred_write_procs.each {|p| @queued_writes << p }
  Appsignal.add_exception(e)
end
```

SQLite3::BusyException
SQLite3::BusyException
SQLite3::BusyException
SQLite3::BusyException
SQLite3::BusyException
SQLite3::BusyException
SQLite3::BusyException

Happens when processes access the same database concurrently. It is a concern. You also have this on the BigDBs (MySQL, PostgreSQL) which slow down to a crawl, without telling you. So you have to do this...

```
SELECT value FROM bizarro_innodb_performance_figures_nobody_knows_about
  WHERE parameter_name='locking_delay_threshold_semaphore_freeze_count_dynamic_differential'
```

SQLite just raises.

Which is good.

When you are threaded, though, you should release the GIL so that some other threads can do stuff while we wait:

```ruby
def open_db(retry_timeout: 0.5, retry_interval: 0.05, **sqlite_args)
  dbfile = ActiveRecord::Base.connection_config[:database]
  conn = SQLite3::Database.new(dbfile, **sqlite_args)
  conn.busy_handler do |times_retried_so_far|
    if (times_retried_so_far * retry_interval) > retry_timeout
      false
    else
      sleep(retry_interval) # Give other threads room to breathe
      true
    end
  end
  conn
end
```

But it's no fun when you have error rates of 30%. So we tried, for example, replica files that would be read from concurrently. You have to copy the entire database into another, and then make sure your readers use the copy, not the original. And we tried pooling them.

This was not pretty.

But, as it turns out there is a magic button.

`PRAGMA journal_mode=WAL;`

This is not the default, you have to explicitly set it.

It changes the way SQLite manages it's journal

- WAL provides more concurrency as readers do not block writers and a writer does not block readers.
- 16 reading Puma threads, 1 off-band writing Puma thread
- All processes using a database must be on the same host computer
- The built-in unix and windows VFSes support this but third-party extension VFSes for custom operating systems might not

I can't even! Where did this come from?

Beginning with version 3.7.0 (2010-07-21), a new “Write-Ahead Log” option (hereafter referred to as “WAL”) is available.

People who needed it moved on and never found out it has become available.

With WAL, the busy exceptions stop except for when two Puma processes attempt to do their flushing, at the same time. For which case see previous slide (requeue and forget).

WAL requires non-networked, high-performance local filesystem.

So what?

FINE!

We are nihilists. We don't believe in any magic NFS storage. No cloud stuff here.

A lot of the ranker modules do not touch the database at all. But if they do, the interface looks roughly like this:

```
r = ->(readonly_db, transfer) {
  rand() # We love living dangerously
}
```

We apply them with a map:

```
rankings = rankers.each_with_object({}) do |r, h|
  h[ranker_name(r)] = r.(db, transfer)
end
```

We also use some functional sauce for exception capture for instance

```
OptionalDB.new(ExceptionCapture.new(r)).(db, xfer)
```

…and then save into the evaluation log. Entries leave the evaluation log after a few hours (we currently can scale up into a few million entries, with no issues performance-wise).

Best practice - have a normalized dataset

None of that. Instead: the "FriendFeed approach"

```
document = StoredTransfer.from_payload({from: ..., message: ...})
```

We store all the given JSON properties denormalized in the `payload`, and only normalize the properties for which columns exist.

These columns get indexed by SQLite and are mostly used for stats/searching by the rankers.

Best practice - use ActiveRecord if you are onto SQL anyway

Things that have to commit immediately use ActiveRecord a bit. ActiveRecord migrations are awesome.

```
# Gemfile
gem 'otr-activerecord'

# Rakefile
load "tasks/otr-activerecord.rake"
OTR::ActiveRecord.configure_from_file! "db/config.yml"
```

All the things that have to run fast bypass it and use a ConnectionPool directly. There is a separate pool for reading only.

```
$read_pool = ConnectionPool.new(size: 3, timeout: 0.5) {
  SQLite3::Database.new(path, readonly: true)`
}
```

But the instance!

Who needs SQL dumps when you have the backups API? Backup API is a little-known SQLite feature to copy a database into another pagewise. It keeps everything binary. It is fast.

```ruby
Spamo.with_readonly_db do |sdb|
  ddb = SQLite3::Database.new(backup_db_path)

  backup = SQLite3::Backup.new(ddb, 'main', sdb, 'main')
  [backup.remaining, backup.pagecount] # Has to be triggered _once_ before starting the procedure
  begin
    backup.step(1) #=> OK or DONE
    if (backup.remaining % print_every).zero?
      $stderr.puts "Backup progress: %d pages remaining of %d" % [backup.remaining, backup.pagecount]
    end
  end while backup.remaining > 0

  backup.finish
end
```

If your SQLite is annoying you with slow disk access...

```ruby
def read_db_into_memory(path)
  sdb = SQLite3::Database.new(path, readonly: true)
  ddb = SQLite3::Database.new(':memory:')

  backup = SQLite3::Backup.new(ddb, 'main', sdb, 'main')
  [backup.remaining, backup.pagecount] # Has to be triggered _once_ before starting the procedure
  begin
    backup.step(1) #=> OK or DONE
  end while backup.remaining > 0
  ddb
ensure
  backup.finish if backup # Has to be called to release locks
  sdb.close if sdb
end

crazyfast_in_memory_copy = read_db_into_memory('/mnt/slownfs/mount/db.sqlite3')
```

By the way...

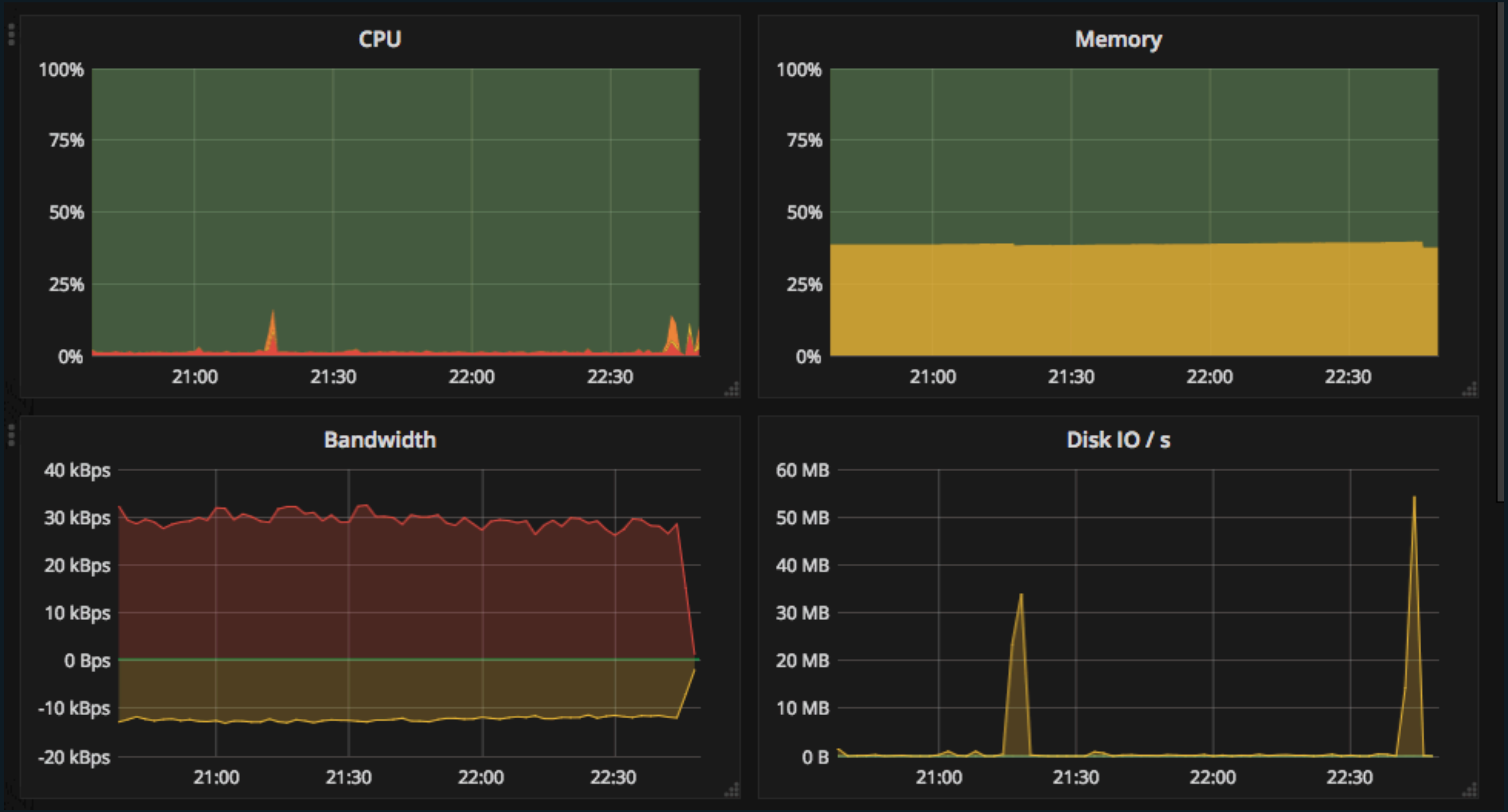
```
db = SQLite3::Database.new(':memory:')
....
db.close # Releases all used memory. Now.
```

So you can also use it for buffering items in memory and deallocating explicitly afterwards. Or use it like a huge, explicitly allocated data structure.

Testing

```
it "classifies from event log" do
  db = SQLite3::Database.new(':memory:')
  db.execute("CREATE TABLE event_log (ip_address TEXT)")
  1000.times { db.execute("INSERT INTO event_log (ip_address) VALUES ?", fake_ip_) }
  ...
  rating = subject.call(db, fake_data)
  expect(rating).to be_between(1).and(1.3)
end
```

RAM/CPU use


CPU
100%
75%
50%
25%
0%
21:00
21:30
22:00
22:30
Memory
100%
75%
50%
25%
0%
21:00
21:30
22:00
22:30
Bandwidth
40 kBps
30 kBps
20 kBps
10 kBps
0 Bps
-10 kBps
-20 kBps
21:00
21:30
22:00
22:30
Disk IO / s
60 MB
50 MB
40 MB
30 MB
20 MB
10 MB
0 B
21:00
21:30
22:00
22:30

Look at all the horror

- ☑ Non-redundant architecture with a single server
- ☑ Running without container isolation layers
- ☑ Threads
- ☑ Non-durable writes in the background
- ☑ Non-hosted, not-AWS-provided database
- ☑ Using ephemeral local filesystems
- ☑ Push-deploys to a hot EC2 instance
- ☑ Assuming the service is broken by default

Web scale.

Use the damn server. All of it.
Steal ideas, not jars.

- https://github.com/WeTransfer
- https://wetransfer.homerun.co/
- https://github.com/julik