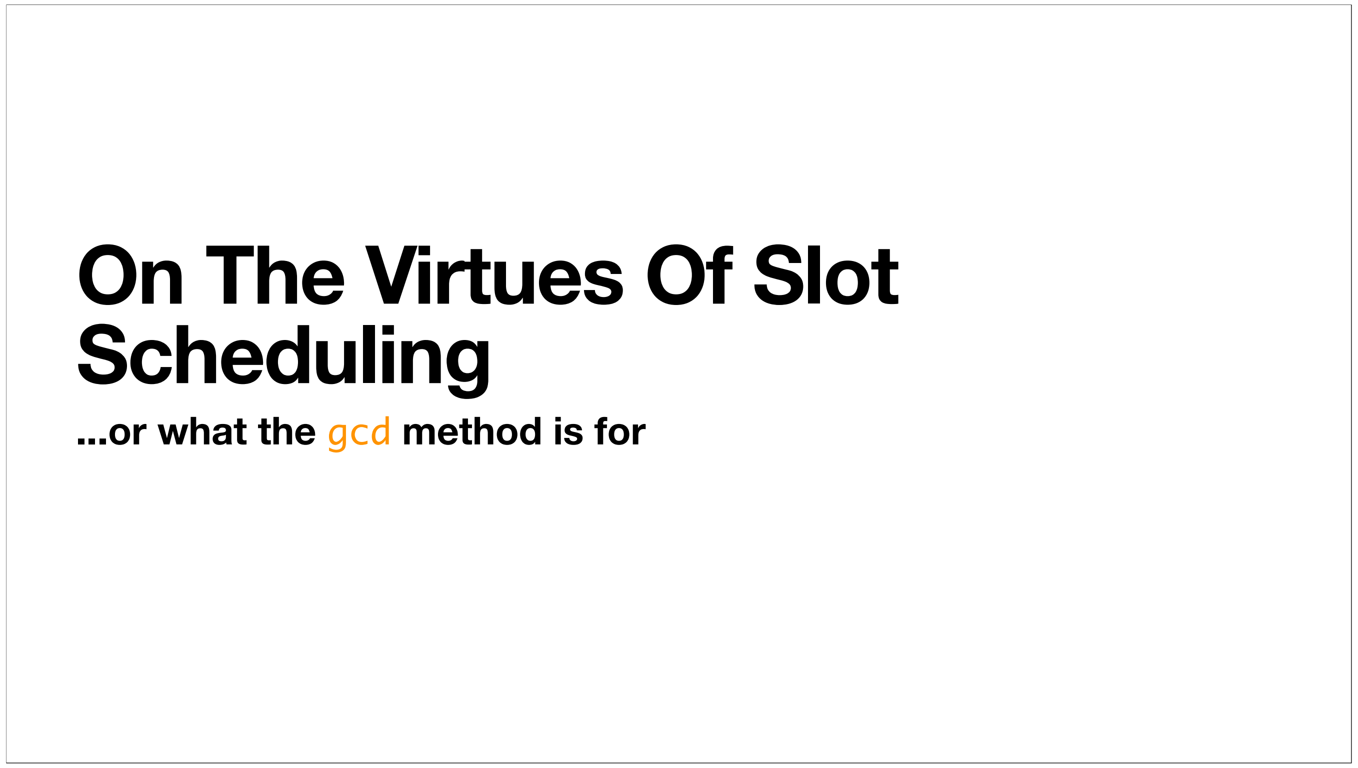


On The Virtues Of Slot Scheduling

...or what the `gcd` method is for

Hi all, it's great to see you all and it is great to be at Euroko. Last time we saw each other was in 2019, I believe - remember that Covid thing right after? and how everything collapsed? So, the talk... but first...

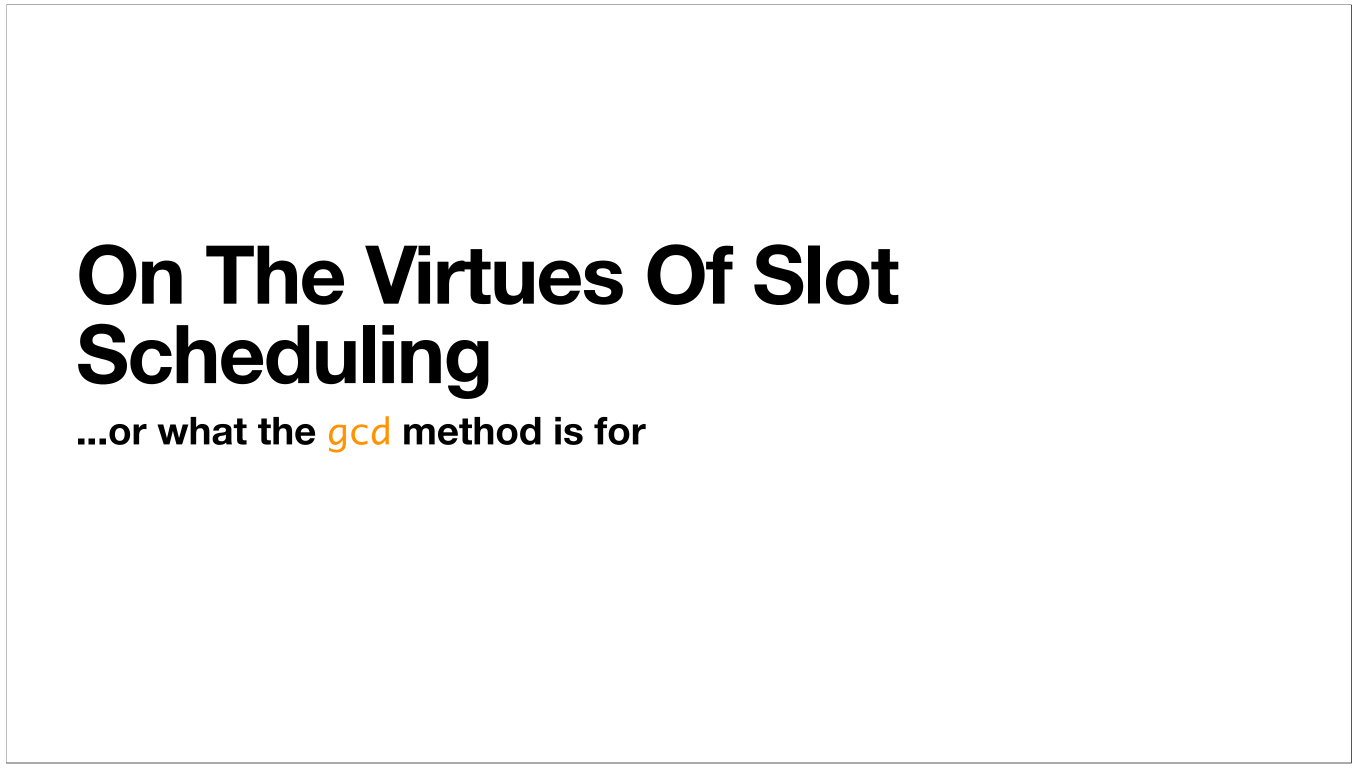
Now, scroll forward just a few years and we finally get to meet each other again. So, who am I?

My name is Julik, I have been a Ruby developer for almost 20 years now - and working professionally in the field for over a decade. I work for a company called Cheddar...

🇺🇦

https://www.freedombirds.help
https://zeilenvanvrijheid.nl

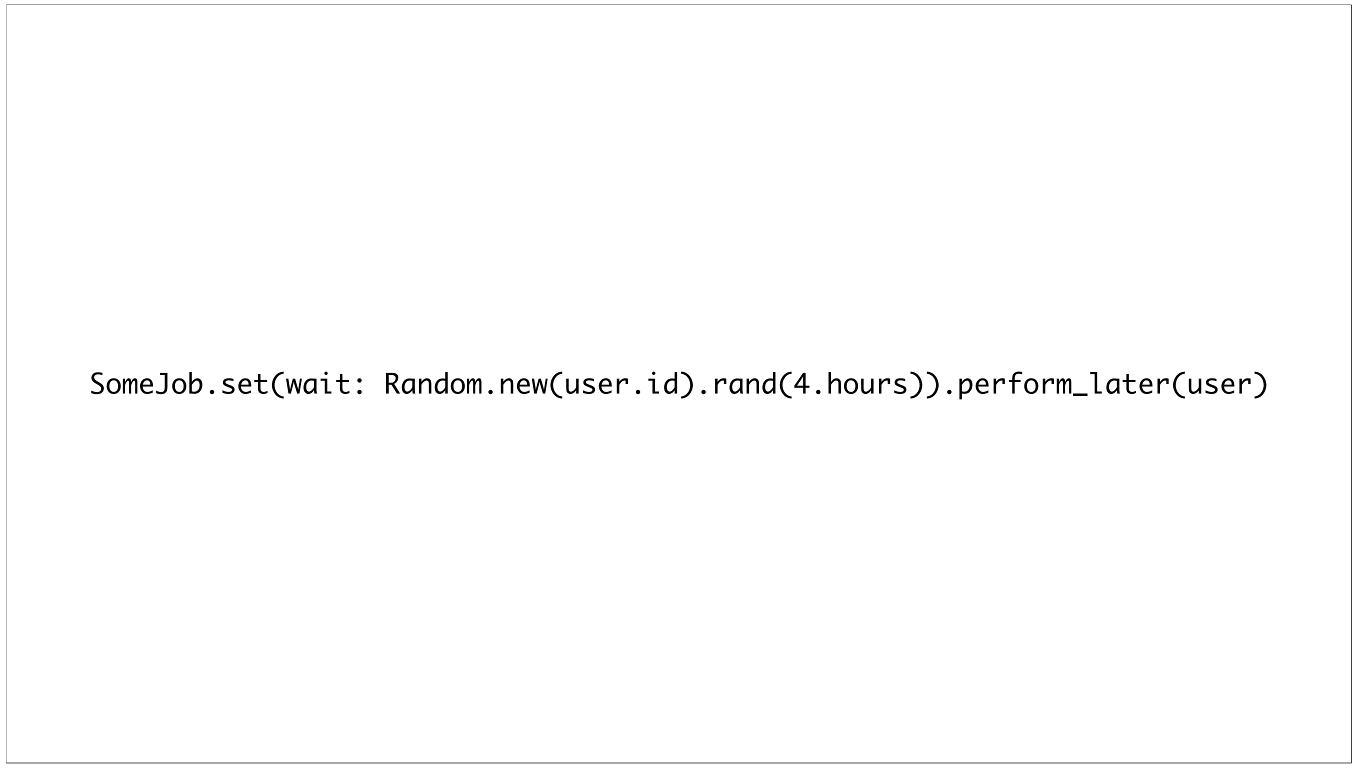
Currently there is a war going on in Europe, and there are few places where the memory of how bad it can be is so recent as it is here. The UKR Ruby community often say that people-with-russian-last-names stay silent. Ukraine must be free, and the war must end. These two URLs are for donations that I personally know go to good deeds and I endorse.



On The Virtues Of Slot Scheduling

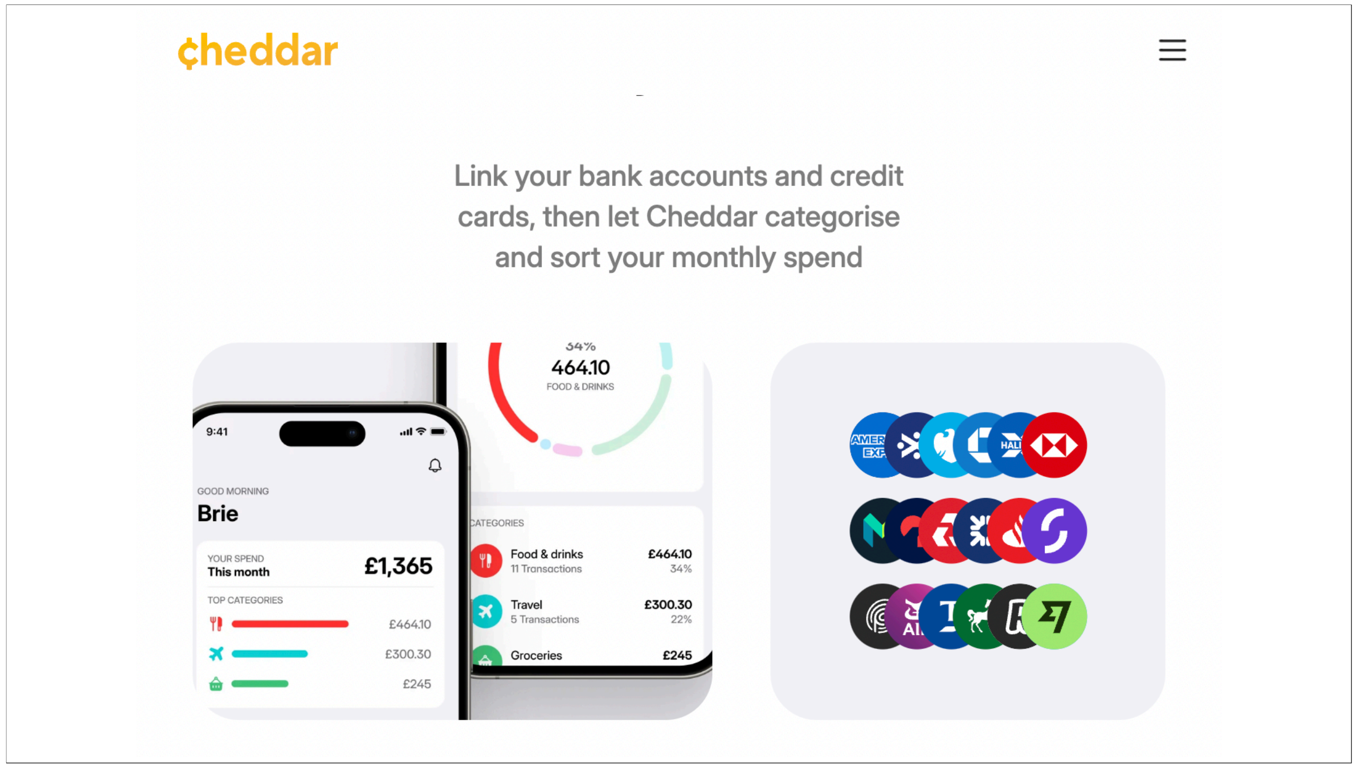
...or what the `gcd` method is for

My name is Julik, I have been a Ruby developer for almost 20 years now - and working professionally in the field for over a decade. I work for a company called Cheddar...

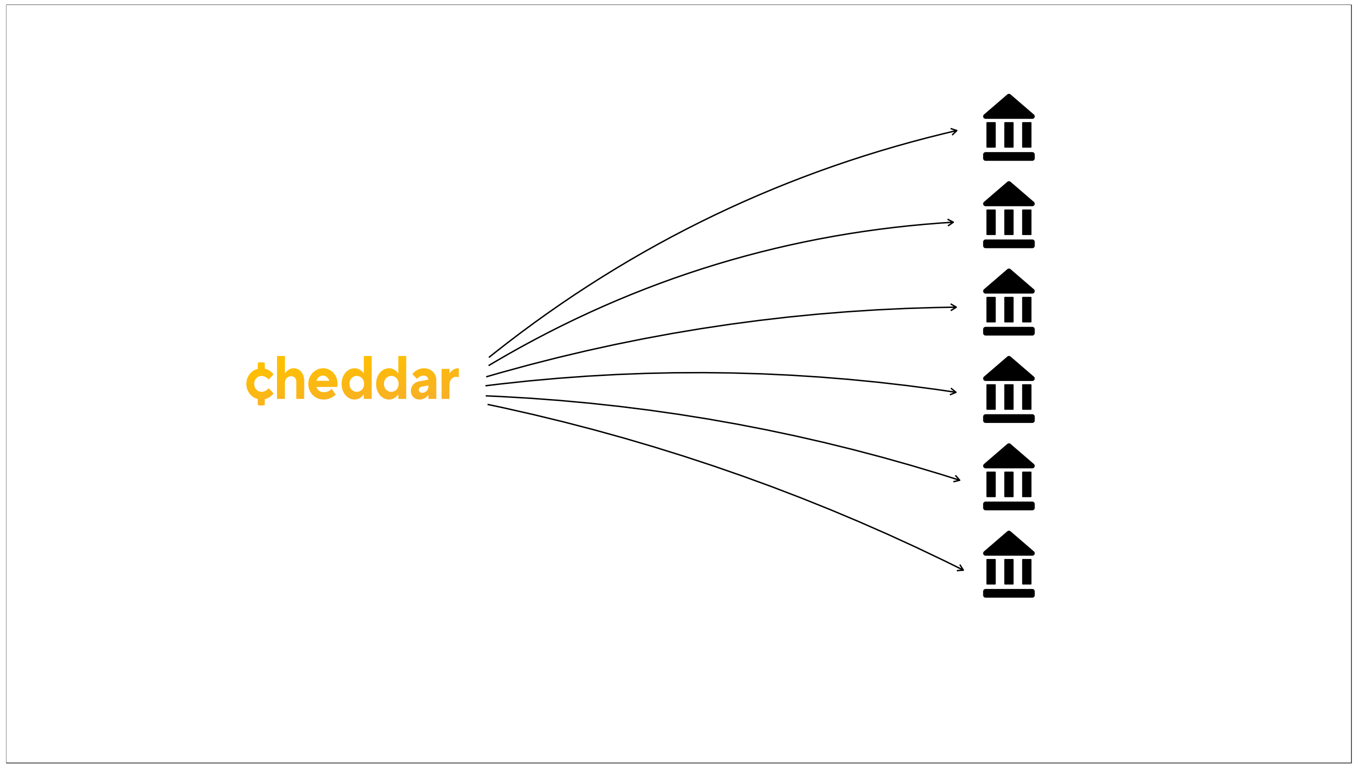


```
SomeJob.set(wait: Random.new(user.id).rand(4.hours)).perform_later(user)
```

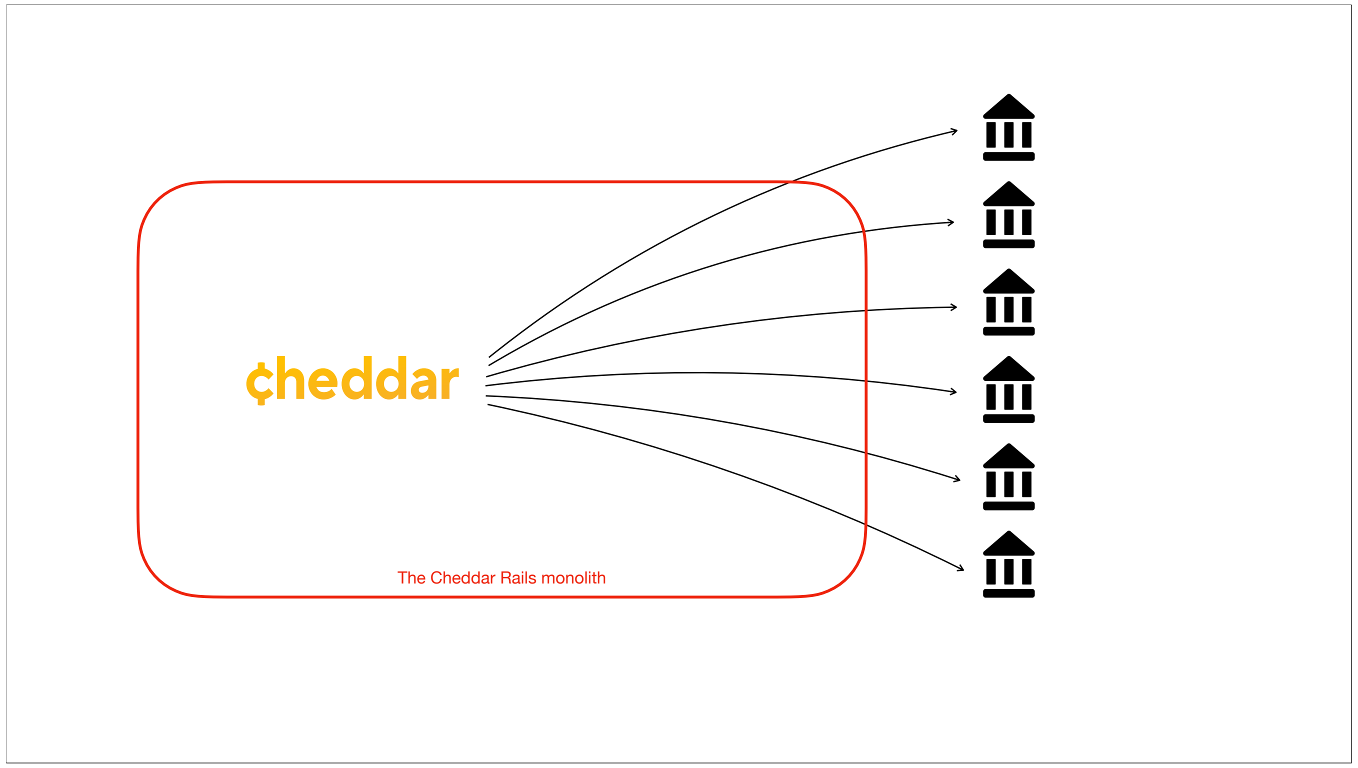
So, first of all: if you want to have a TL;DR of this talk - and zone out for the remaining 30 minutes - I won't deny you this opportunity, so if there is one thing you want to take from this talk it is this.



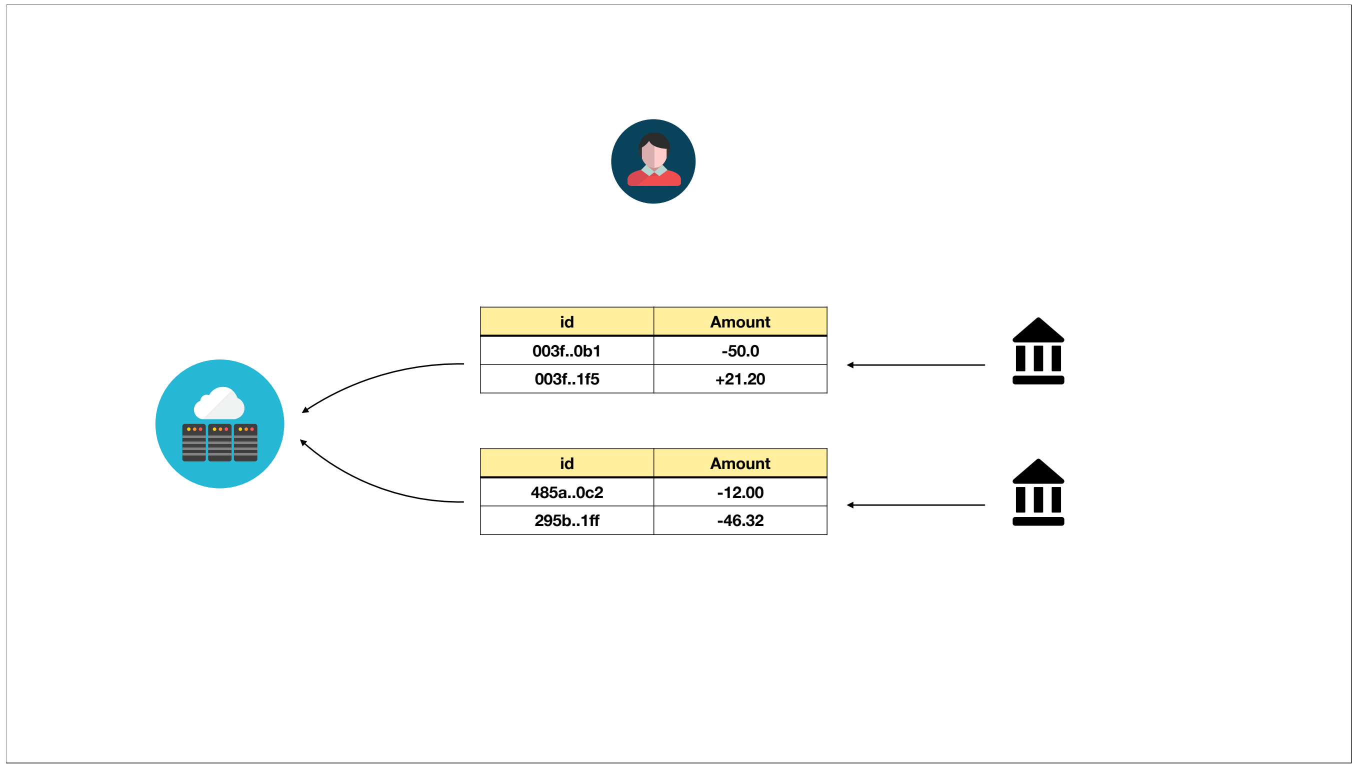
..and now that we got it out of the way. Cheddar aims to make it easier for people to rein in their personal finances in the UK. We are growing at a steady pace, and have won the Best newcomer British Bank award this year. The areas of focus for us are gift cards, personal finance management, and person-to-person payments. You can find out more at cheddar.me, and if you are in the UK here is a handy referral code you can use to join which will give you a small bonus ;-)



Cheddar integrates with a ton of UK banks through the OpenBanking ecosystem. I am not exaggerating: at the moment we support 23 banks, and this number keeps increasing quarter-to-quarter. OpenBanking works pretty well in the UK and we use it for both so-called "account information service" access (reading out bank transactions) and "payment initiation services" - you can initiate a payment using Cheddar, and it will bounce you to your banking application - with the payment going not to us, but directly to your counterparty.



Cheddar is actually unique in one very special way in the OpenBanking ecosystem. Most fintech apps choose to integrate with all those banks through an intermediary. It is a large company that allows you access, and it is not you but that company getting access to all the banking resources and APIs. The advantage of this is clear: you don't have to spend time figuring out all those snags we have to figure out (what is this particular way this particular bank in Scotland treats this particular JWT claim in this particular OAuth2 flow that differs from the other 22?). The downsides are also sizeable, though: you pay per accessed resource, and the fees are not insubstantial! Also, you actually do not own either the access to the data, or the user connection. It is that intermediary who does, which has business consequences.



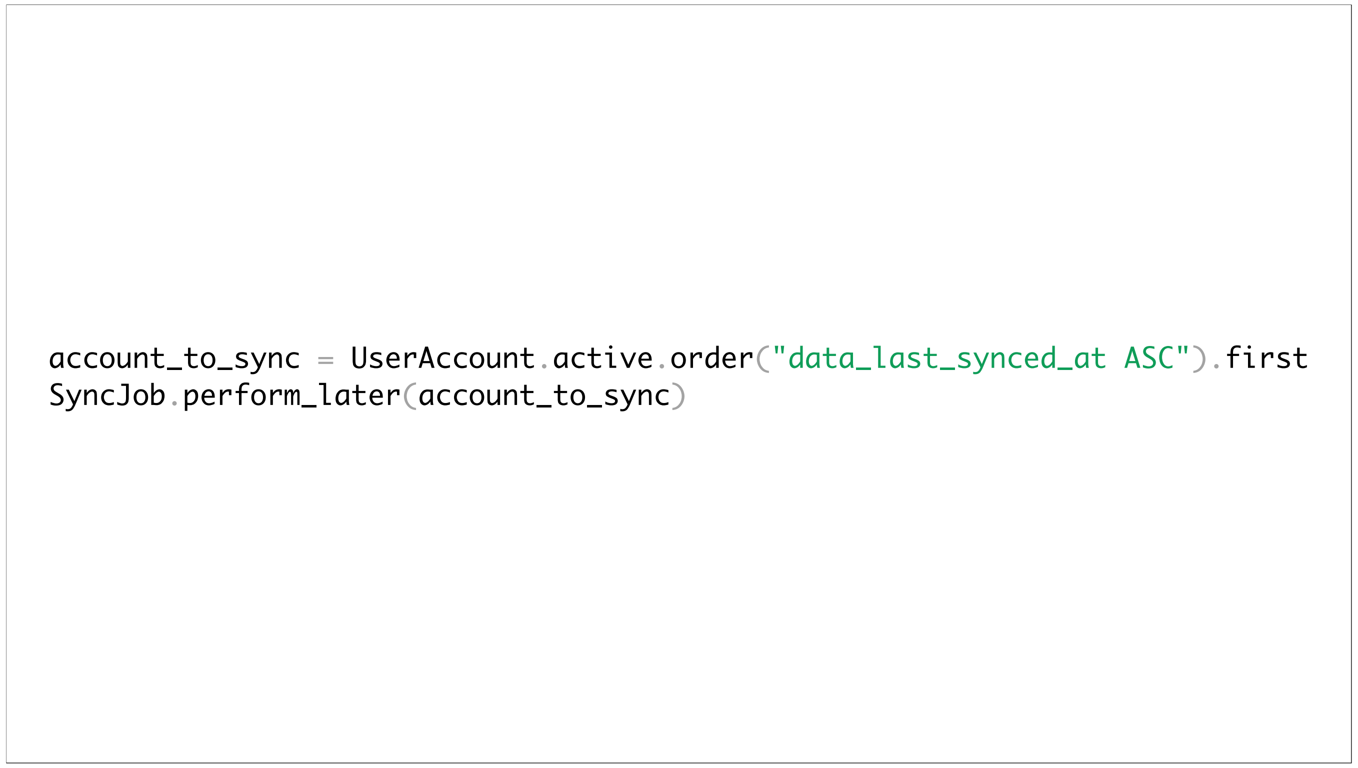
id	Amount
003f..0b1	-50.0
003f..1f5	+21.20

id	Amount
485a..0c2	-12.00
295b..1ff	-46.32

So: Cheddar does their own integrations. When you link your bank account with us, we set up a flow called "transaction sync" for you. This means that Cheddar will, at regular intervals, look at your bank accounts and download your recent transaction data - for instance, to show you this handy page here. This means a lot of API calls. Many, many, many API calls.

For each of the linked bank accounts, we need to perform this sync process. The sync process will potentially require OAuth token refreshes. The API calls themselves can be slow. The OAuth calls can be slow. A banking API can be down, so a circuit breaker may trip. A lot of different things can happen. But most importantly: _when_ do you sync?

Most banks do not update your bank transactions instantly as they come in. Most do that in bulk, at midnight - as they run a lot of batch processes themselves, so sycing transactions usually makes sense after midnight UK time. Then it is a matter of syncing the transactions as frequently as possible for any given user, and this is where we get presented with choices.

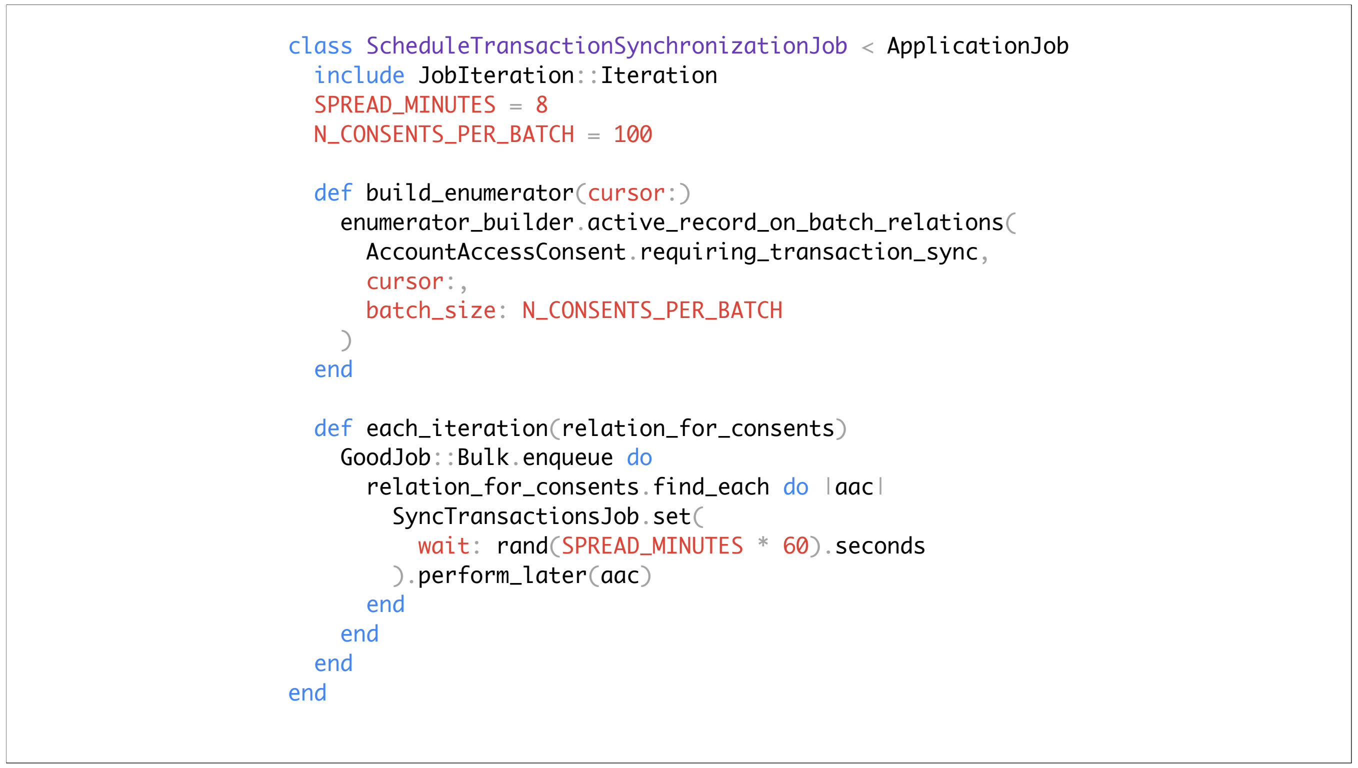


```ruby
account_to_sync = UserAccount.active.order("data_last_synced_at ASC").first
SyncJob.perform_later(account_to_sync)
```

Our initial approach was to do it on the "least recent" basis. Kind of like this.

And this is where we would run into a snag. For example: the `data_last_synced_at` would only get updated inside the `SyncJob`, and only at succesful completion. But most importantly, these jobs would all be enqueued in one big bulk.

This led to large jumps in our queue depth - at some point there would be close to 0 jobs, next thing you know - you end up with a few tens of thousands! Moreover, the enqueue rate there would be dependent on job-iteration from Shopify. So we would end up with a job like this:

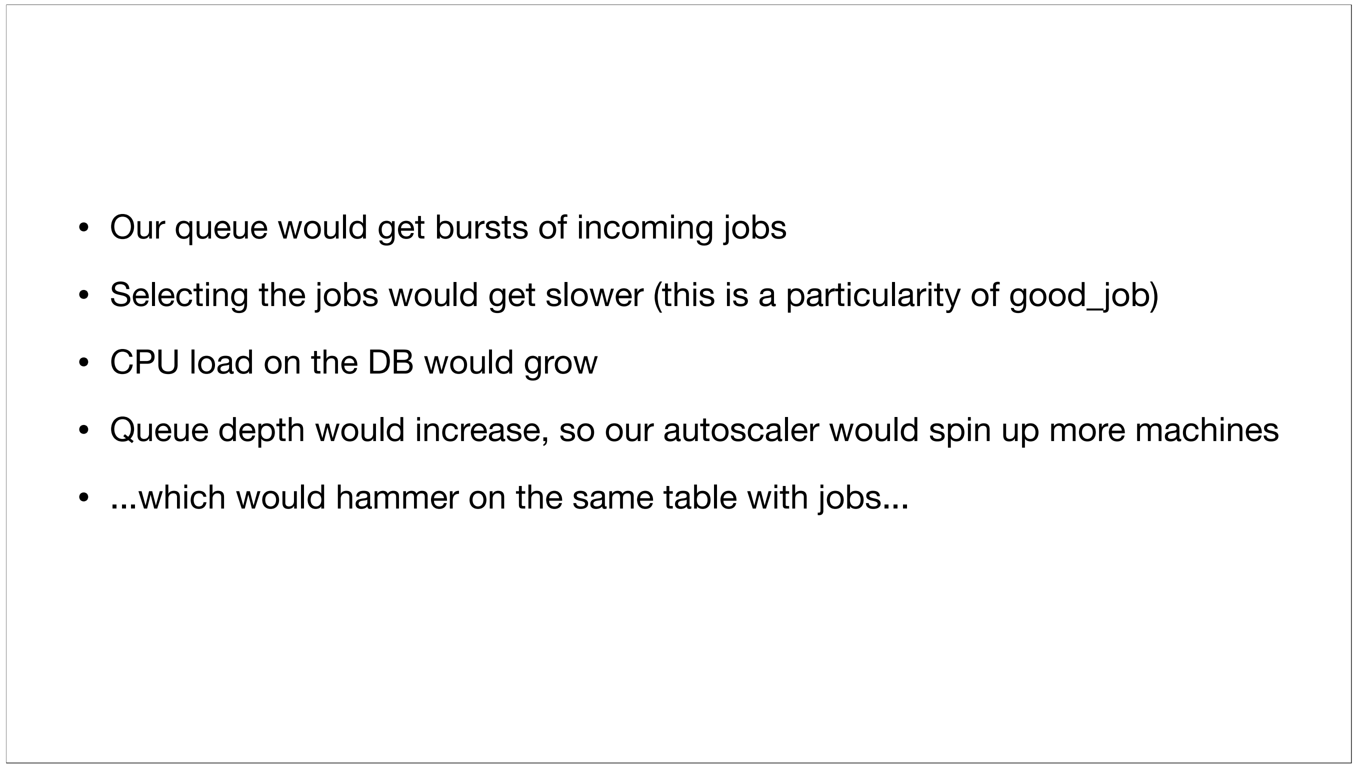


```ruby
class ScheduleTransactionSynchronizationJob < ApplicationJob
  include JobIteration::Iteration
  SPREAD_MINUTES = 8
  N_CONSENTS_PER_BATCH = 100

  def build_enumerator(cursor:)
    enumerator_builder.active_record_on_batch_relations(
      AccountAccessConsent.requiring_transaction_sync,
      cursor:,
      batch_size: N_CONSENTS_PER_BATCH
    )
  end

  def each_iteration(relation_for_consents)
    GoodJob::Bulk.enqueue do
      relation_for_consents.find_each do |aac|
        SyncTransactionsJob.set(
          wait: rand(SPREAD_MINUTES * 60).seconds
        ).perform_later(aac)
      end
    end
  end
end
```

At a certain point we would start this job, and it would inject those sync tasks in the queue and postpone further execution until the point when that `Schedule...` job would have been running for too long. job-iteration is great that way, see?..

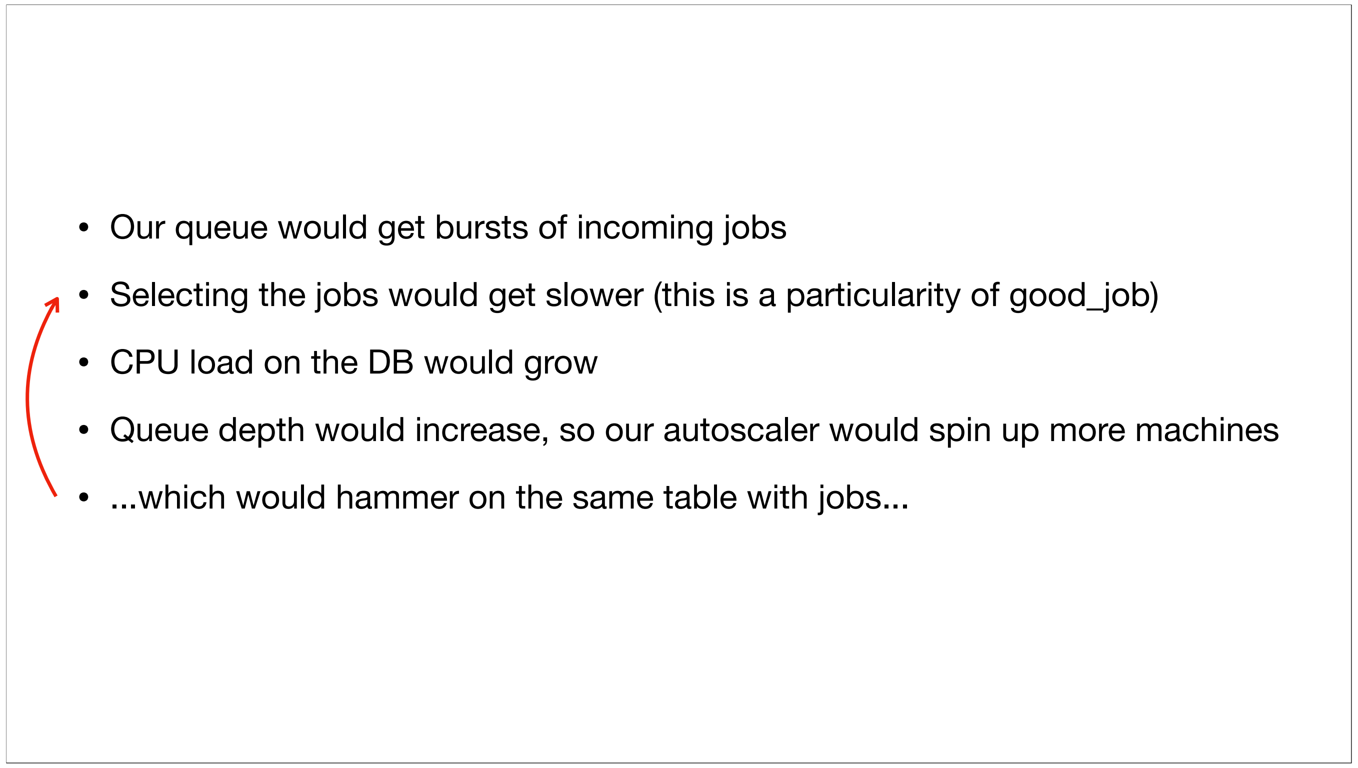
But we ended up with a problem. See, banks throttle. If you send them too many requests in a short period of time - they get really unhappy about it (and rightly so). Also, we have just one database ("Just Use Postgres For Everything") as we are proponents of the "Less Tech" movement - and proud members of the Boring Technology club https://boringtechnology.club/

Banks also go down. Have one bank go down and the `last_synced_at` value not updated for a day or more (it does happen)... and those accounts will always be enqueued first, as they are "least recently synced". This can cause clogging for other accounts in the system.



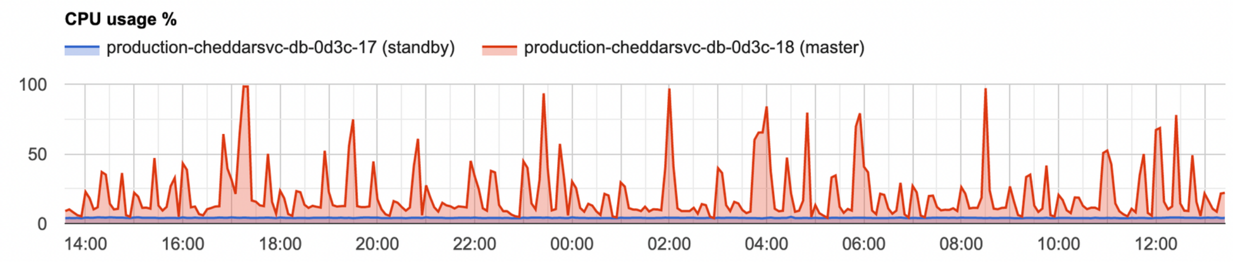
- Our queue would get bursts of incoming jobs
- Selecting the jobs would get slower (this is a particularity of good_job)
- CPU load on the DB would grow
- Queue depth would increase, so our autoscaler would spin up more machines
- ...which would hammer on the same table with jobs...

This kind of "least recent first", coupled with an effectively "throttled" job that enqueues those syncs, created a spiral of discontent. At the time, we were using good_job for our job queue, and this is what would transpire:

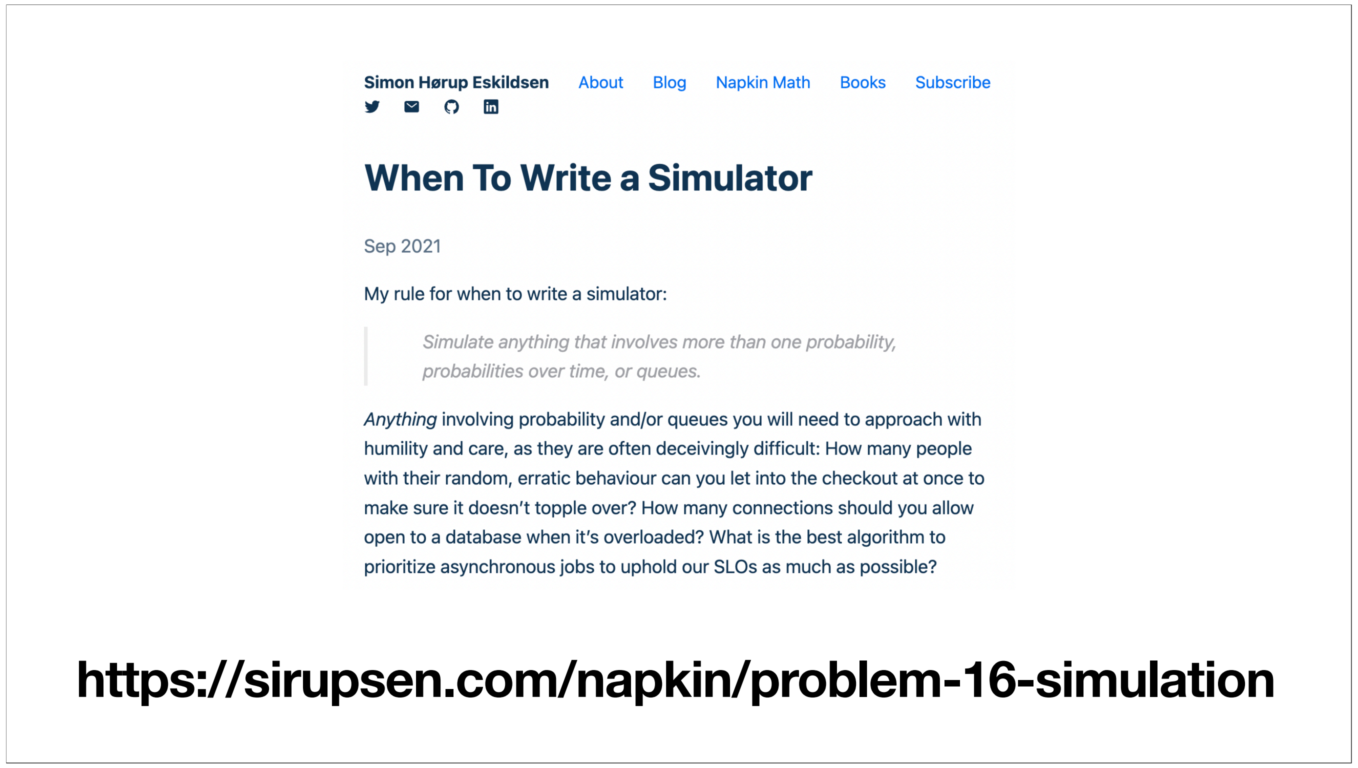


- Our queue would get bursts of incoming jobs
- Selecting the jobs would get slower (this is a particularity of good_job)
- CPU load on the DB would grow
- Queue depth would increase, so our autoscaler would spin up more machines
- ...which would hammer on the same table with jobs...

ending with this



So every day, day in and day out we would bring our own system down in this miserable way. And then something dawned on us: there is a solution for this, so stupid in fact, that people discourage each other from even thinking of using it. But what if that is exactly the solution we need? What if what we need is spreading our users' tasks throughout a fixed time interval - but in a deterministic way?


Simon Hørup Eskildsen About Blog Napkin Math Books Subscribe

When To Write a Simulator

Sep 2021

My rule for when to write a simulator:

> *Simulate anything that involves more than one probability, probabilities over time, or queues.*

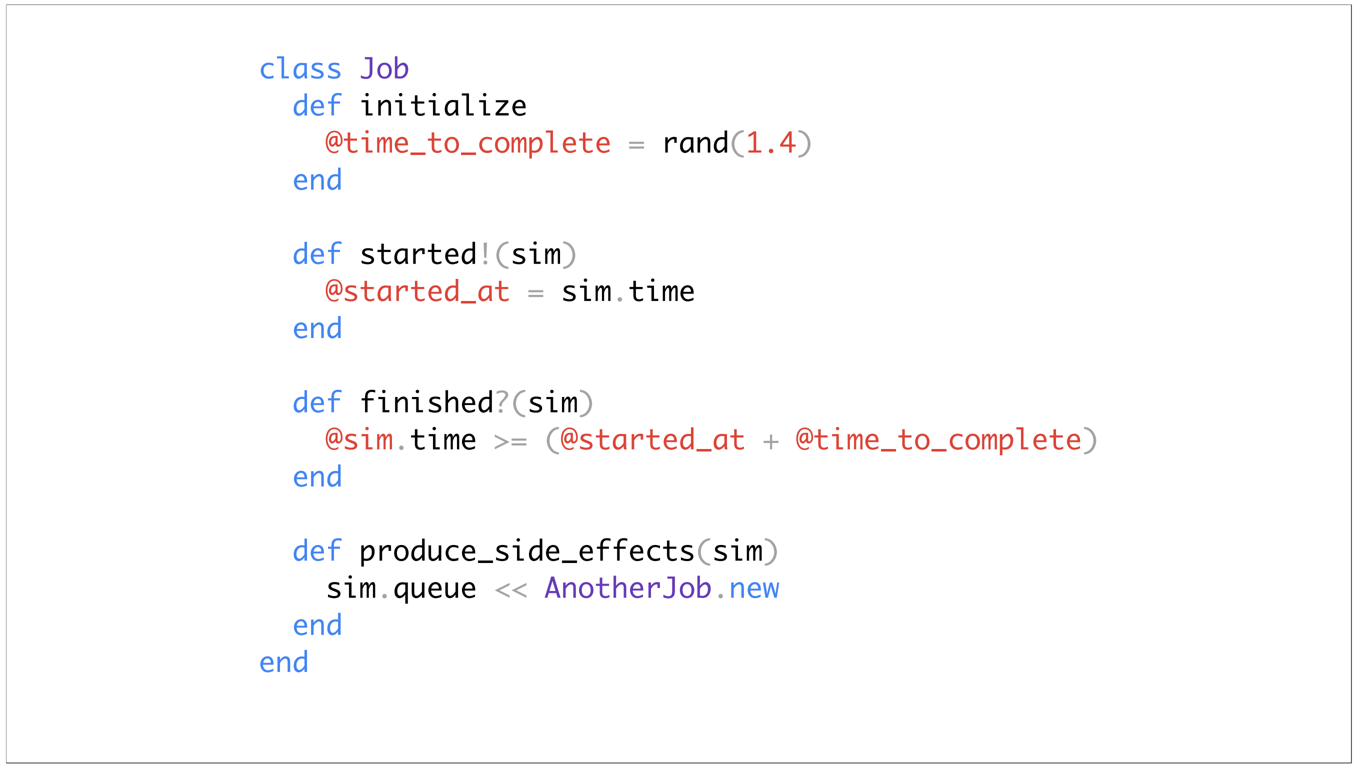
Anything involving probability and/or queues you will need to approach with humility and care, as they are often deceivingly difficult: How many people with their random, erratic behaviour can you let into the checkout at once to make sure it doesn't topple over? How many connections should you allow open to a database when it's overloaded? What is the best algorithm to prioritize asynchronous jobs to uphold our SLOs as much as possible?

https://sirupsen.com/napkin/problem-16-simulation

That is a hypothesis. We can argue hypotheses all we want, but there is a great saying in this blog article: https://sirupsen.com/napkin/problem-16-simulation

"Simulate anything that involves more than one probability, probabilities over time, or queues."

We had exactly this type of situation: we had a process which, in realtime, took 12 hours. We know that with improper inputs we can bring our system down if we don't configure it well. We must have a way to replicate this, right?

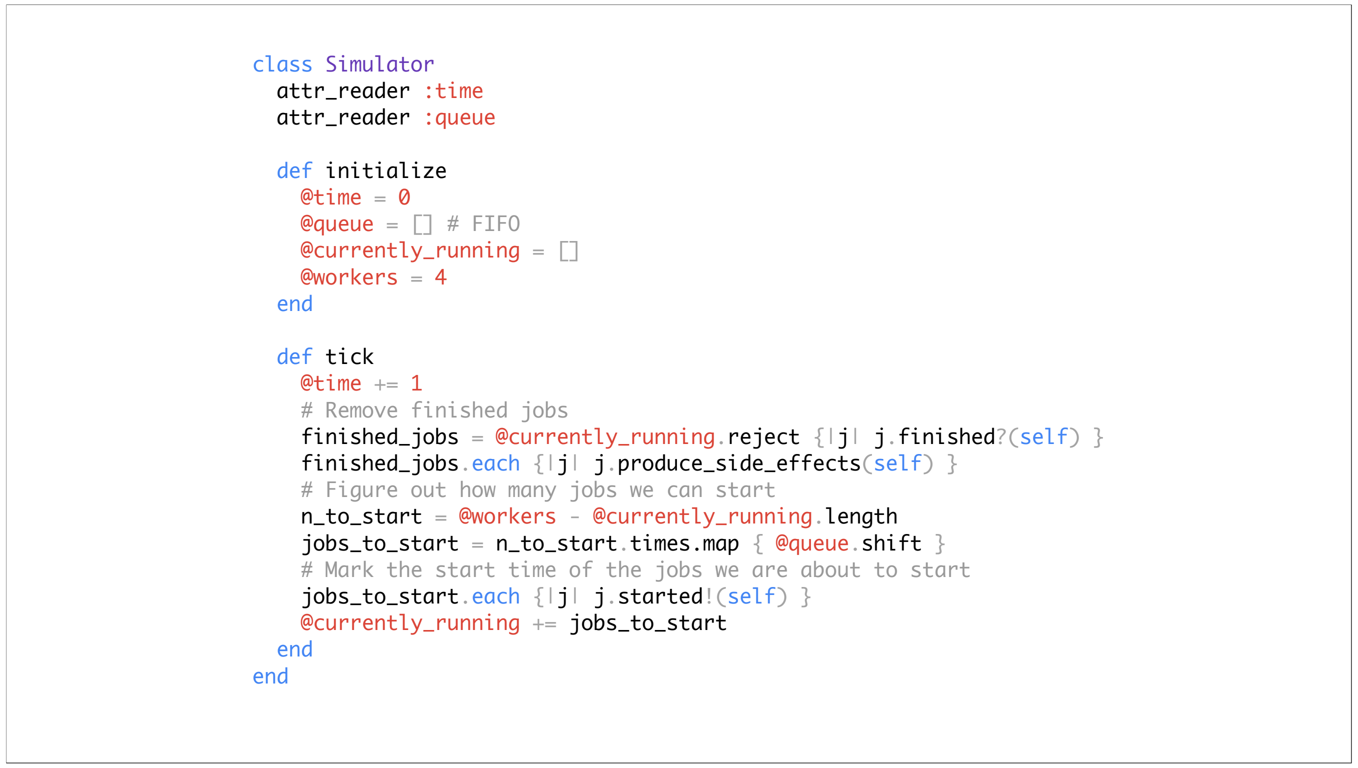


```ruby
class Job
  def initialize
    @time_to_complete = rand(1.4)
  end

  def started!(sim)
    @started_at = sim.time
  end

  def finished?(sim)
    @sim.time >= (@started_at + @time_to_complete)
  end

  def produce_side_effects(sim)
    sim.queue << AnotherJob.new
  end
end
```

Of course. We have a system with the following constraints - our model if you will.

* We have a number of worker threads (let's call them "workers"). A worker can process one job at a time.
* We have a number of jobs in the queue, which get evenly distributed among workers as those become available
* A job takes a certain amount of time to complete

This is how a job in such a simulator could look like:



```ruby
class Simulator
  attr_reader :time
  attr_reader :queue

  def initialize
    @time = 0
    @queue = [] # FIFO
    @currently_running = []
    @workers = 4
  end

  def tick
    @time += 1
    # Remove finished jobs
    finished_jobs = @currently_running.reject {|j| j.finished?(self) }
    finished_jobs.each {|j| j.produce_side_effects(self) }
    # Figure out how many jobs we can start
    n_to_start = @workers - @currently_running.length
    jobs_to_start = n_to_start.times.map { @queue.shift }
    # Mark the start time of the jobs we are about to start
    jobs_to_start.each {|j| j.started!(self) }
    @currently_running += jobs_to_start
  end
end
```

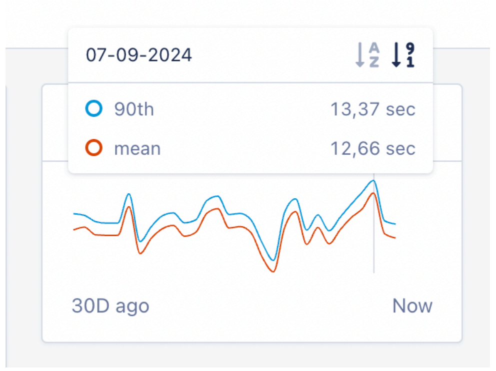
We start a simulator, which supports ticks. Every tick we increment the clock by 1, and we check whether jobs have finished. We allow jobs to produce side effects.

Now, there is a little detour to make here. You can write such a queue simulator using two approaches - "discrete event simulation" and "incremental time progression". I've settled for the latter, as it is somewhat easier to code. Note however, that the incremental time progression needs to implement "ticks" - the time increments of the system - in a way that they be small enough at the job durations you are using.

But:

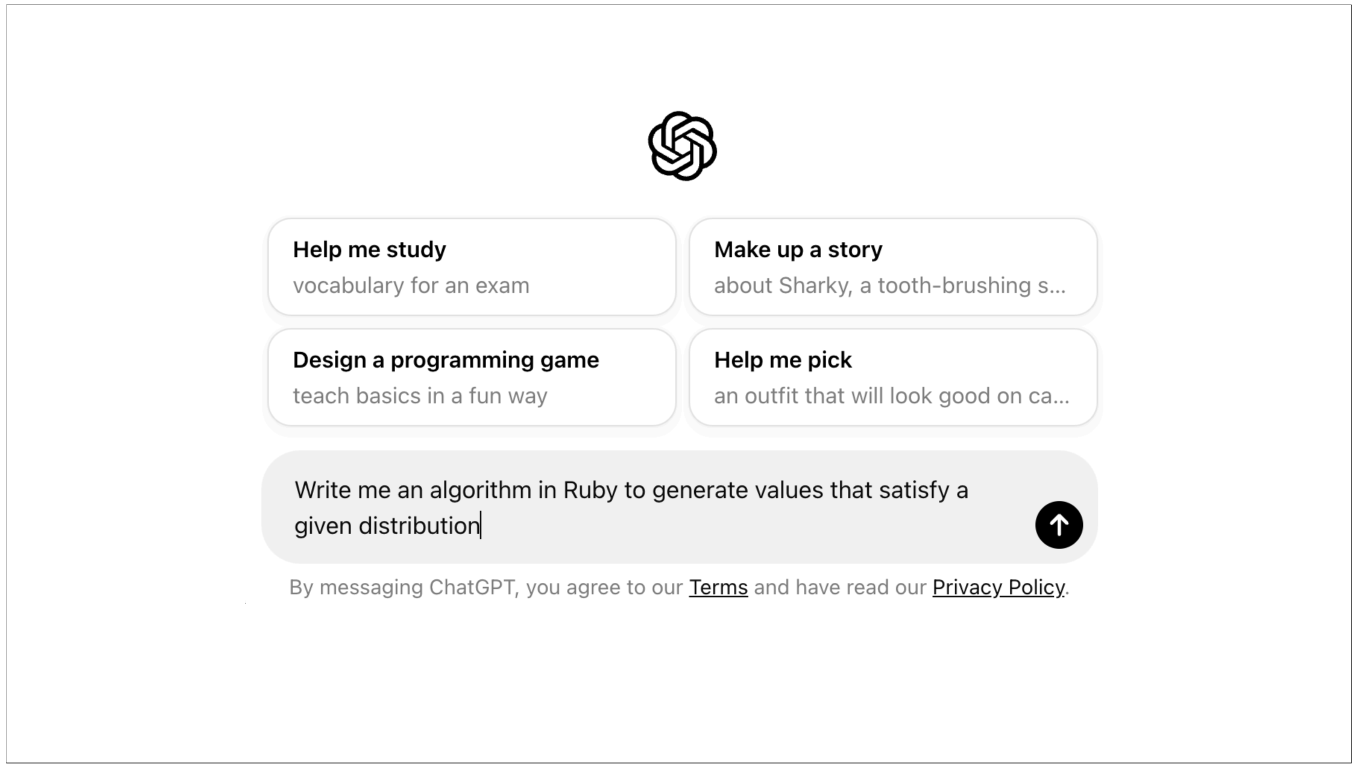
Remember how we initialized a job with a certain "runtime"? That bit is crucial.

While a job takes "a certain amount" of time to complete, that amount is not really the same for every job. It differs depending on how slow a particular bak is, how many transactions we need to download, whether we need to do an OAuth2 token refresh - and a number of other factors. But there is a way to model these durations, if you know where you start.



Enter statistics. Our APM package of choice - Appsignal - gives us a great visualisation for every job type that we run - it displays us percentiles. I haven't done maths properly and I did art school (no joke, I also worked for a CTO who did art school and was quite good at his job, AMA). So I knew that there should be a way, given percentiles - or a statistical distribution of some description - to produce "fake" durations that would satisfy that distribution.

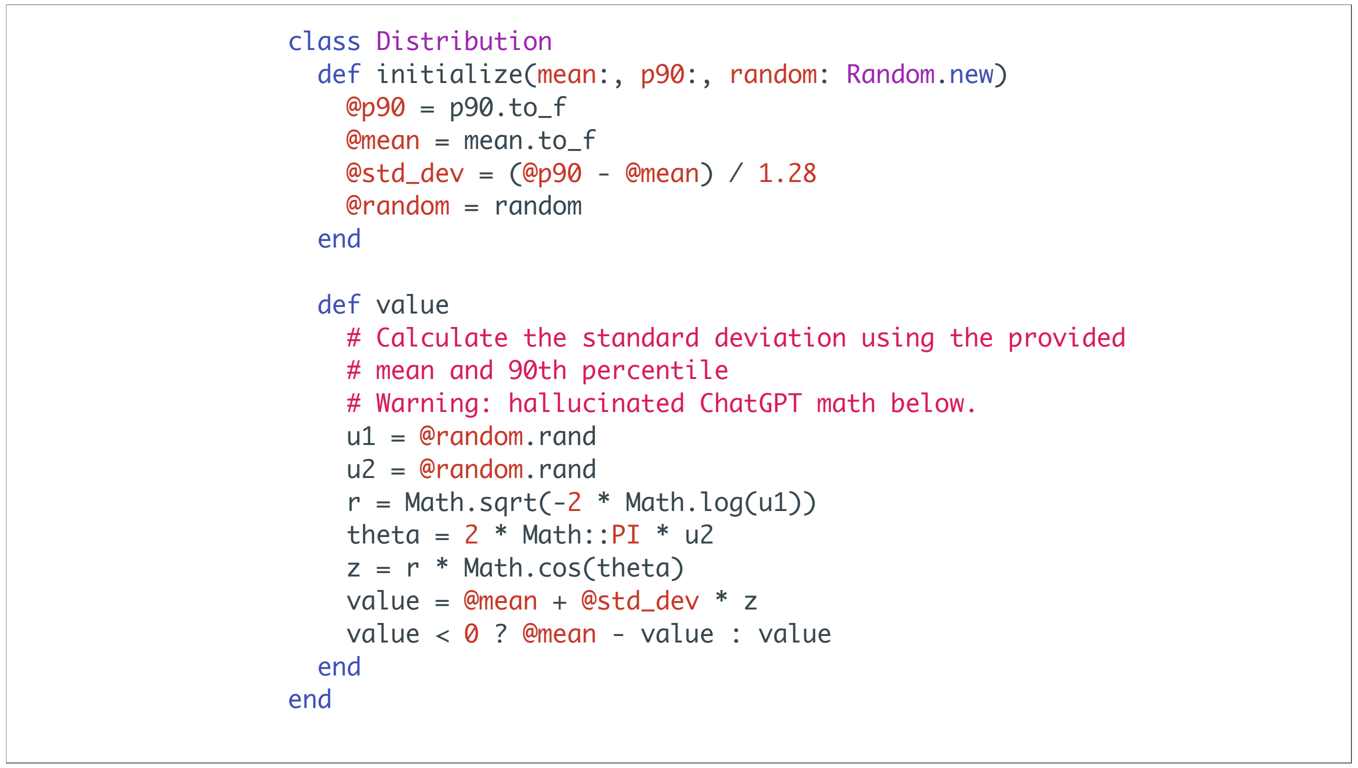
Now, when we have a hunch that "there must be an algorithm to..." but we don't know the algorithm, where do we go, with our art school degree? Exactly...


Help me study
vocabulary for an exam
Make up a story
about Sharky, a tooth-brushing s...
Design a programming game
teach basics in a fun way
Help me pick
an outfit that will look good on ca...

Write me an algorithm in Ruby to generate values that satisfy a given distribution

By messaging ChatGPT, you agree to our Terms and have read our Privacy Policy.

Now, I am not ashamed of this in the slightest. Why? Consider ChatGPT as a drunken pub companion. They may give you utter nonsense. They may trick you around your ears and feed you gibberish - urban legends, if you will. But what they also do is activate your mind, and they do have decent ideas from time to time. This time around, after some prodding, ChatGPT did deliver.

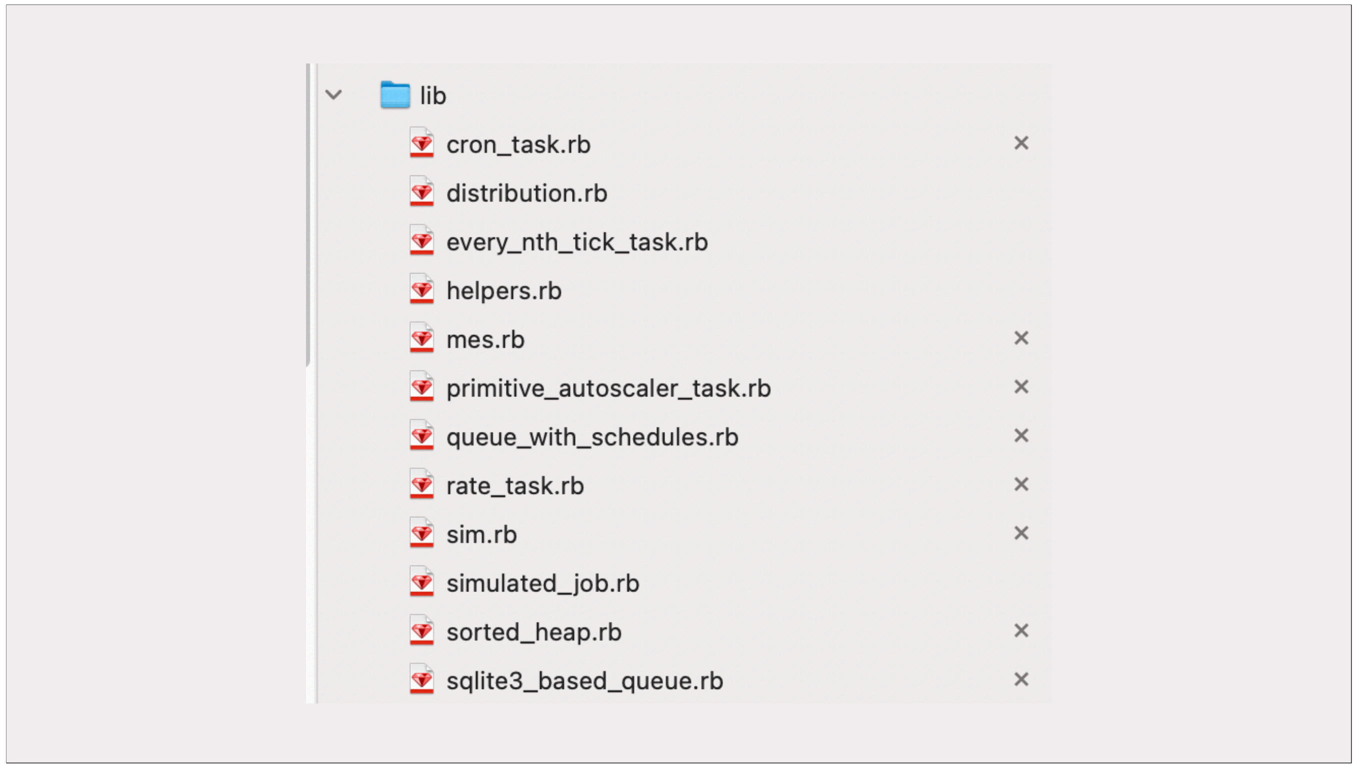


```ruby
class Distribution
  def initialize(mean:, p90:, random: Random.new)
    @p90 = p90.to_f
    @mean = mean.to_f
    @std_dev = (@p90 - @mean) / 1.28
    @random = random
  end

  def value
    # Calculate the standard deviation using the provided
    # mean and 90th percentile
    # Warning: hallucinated ChatGPT math below.
    u1 = @random.rand
    u2 = @random.rand
    r = Math.sqrt(-2 * Math.log(u1))
    theta = 2 * Math::PI * u2
    z = r * Math.cos(theta)
    value = @mean + @std_dev * z
    value < 0 ? @mean - value : value
  end
end
```

Of course there is such a process, called "inverse transform sampling". And after some prodding, ChatGPT gave me the desired code:

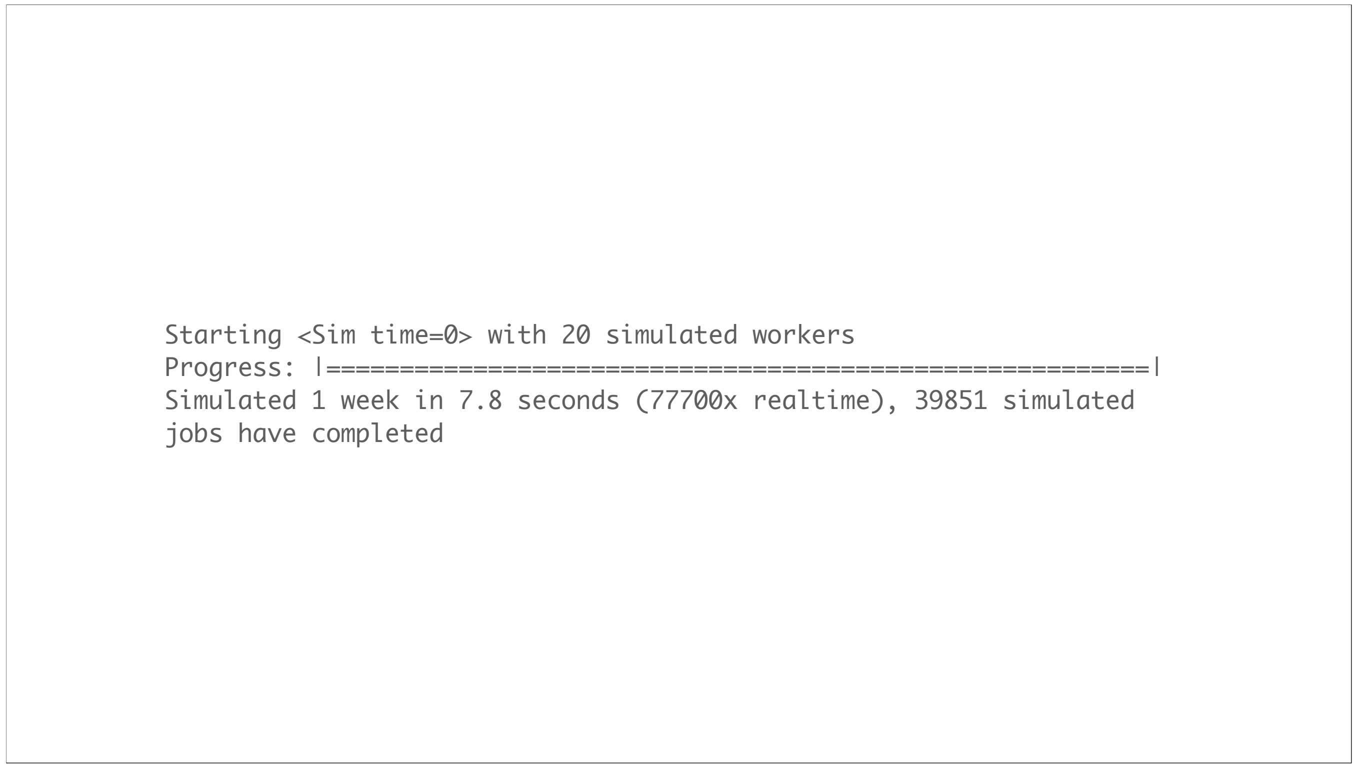
```ruby
require_relative "../lib/sim"

class EmailJob < SimulatedJob
end

sim = Sim.new
dist = Distribution.new(mean: 4, p90: 120) # Durations in ticks
every_tick = -> {
  # Lots of emails are enqueued every hour
  if (sim.time % (60 * 60)).zero?
    rand(200..500).times do
      sim.enqueue EmailJob.new(runtime: dist.value)
    end
  end
}
sim.run(n_ticks: 7 * 24 * 60 * 60, every_tick:)
```

This way we simulate a lot of emails getting placed on the queue every hour. Let's run it for a week of simulated time.



```
lib
    cron_task.rb                    ×
    distribution.rb
    every_nth_tick_task.rb
    helpers.rb
    mes.rb                          ×
    primitive_autoscaler_task.rb    ×
    queue_with_schedules.rb         ×
    rate_task.rb                    ×
    sim.rb                          ×
    simulated_job.rb
    sorted_heap.rb                  ×
    sqlite3_based_queue.rb          ×
```

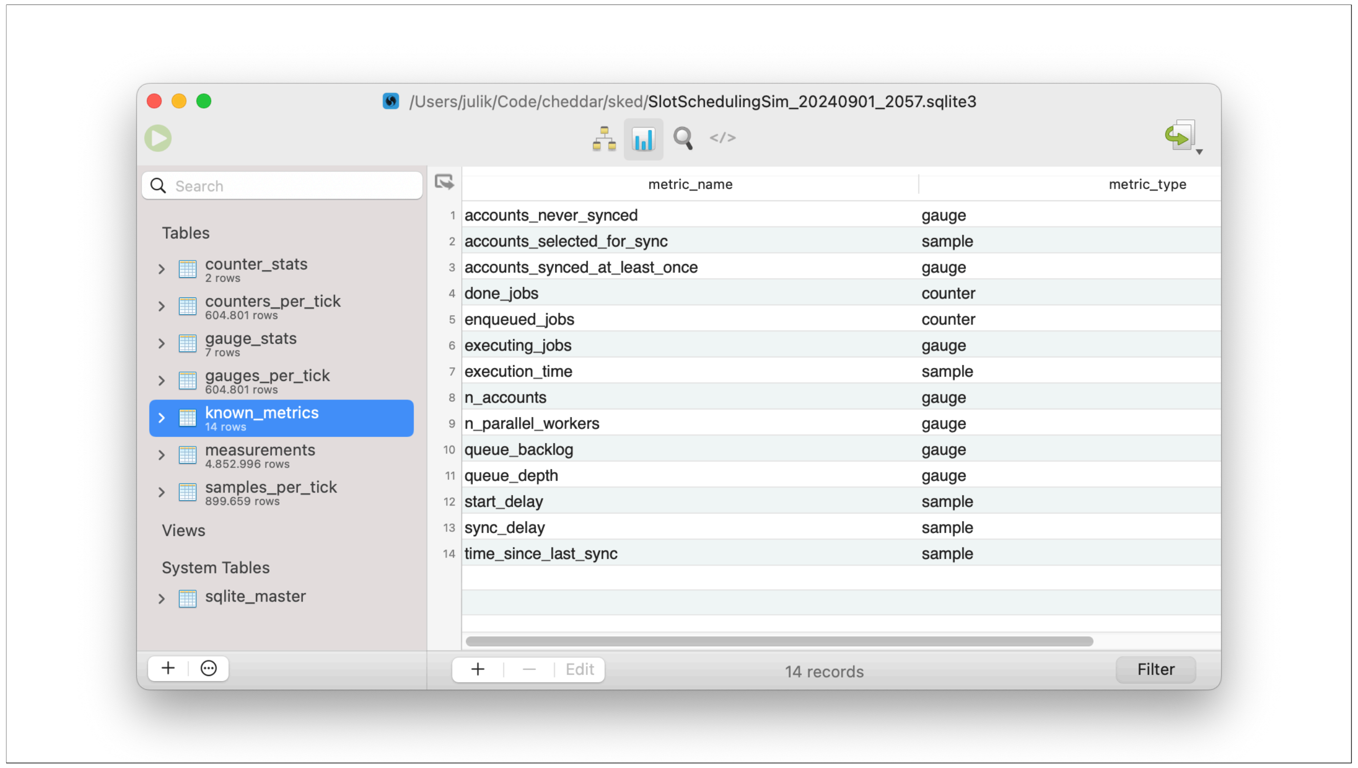
Of course there is a bit more to it but this is what it consists of



```
Starting <Sim time=0> with 20 simulated workers
Progress: |========================================================|
Simulated 1 week in 7.8 seconds (77700x realtime), 39851 simulated
jobs have completed
```

Now, it is great of course that we are, apparently, able to run our simulator at... whatever it is the realtime speed. But just knowing we are able to do that doesn't give us much useful info, right? Of course not. That simulator outputs a little something. And that little something is a SQLite database full of _metrics_:

```
Starting <SelfReschedulingSim time=0> with 15 simulated workers
Progress: |====================================================================================================|
Synced at least once: <redacted>, never synced: 0
Simulated 1 week in 37.2 seconds (16269x realtime), 1277360 simulated jobs have completed
Crunching the metrics for easier human consumption using 7388908 datapoints... this can take a little while

Starting <CentralSchedulingSim time=0> with 15 simulated workers
Progress: |====================================================================================================|
Synced at least once: <redacted>, never synced: 0
Simulated 1 week in 72.2 seconds (8377x realtime), 1316151 simulated jobs have completed
Crunching the metrics for easier human consumption using 6973022 datapoints... this can take a little while

Starting <SlotSchedulingSim time=0> with 15 simulated workers
Progress: |====================================================================================================|
Synced at least once: <redacted>, never synced: 1
Simulated 1 week in 20.2 seconds (29918x realtime), 608379 simulated jobs have completed
Crunching the metrics for easier human consumption using 4852996 datapoints... this can take a little while

Starting <SlotSchedulingSim2 time=0> with 15 simulated workers
Progress: |====================================================================================================|
Synced at least once: <redacted>, never synced: 36
Simulated 1 week in 20.8 seconds (29080x realtime), 608005 simulated jobs have completed
Crunching the metrics for easier human consumption using 4851875 datapoints... this can take a little while
```


/Users/julik/Code/cheddar/sked/SlotSchedulingSim_20240901_2057.sqlite3

Tables

counter_stats	2 rows
counters_per_tick	604.801 rows
gauge_stats	7 rows
gauges_per_tick	604.801 rows
known_metrics	14 rows
measurements	4.852.996 rows
samples_per_tick	899.659 rows

Views

System Tables

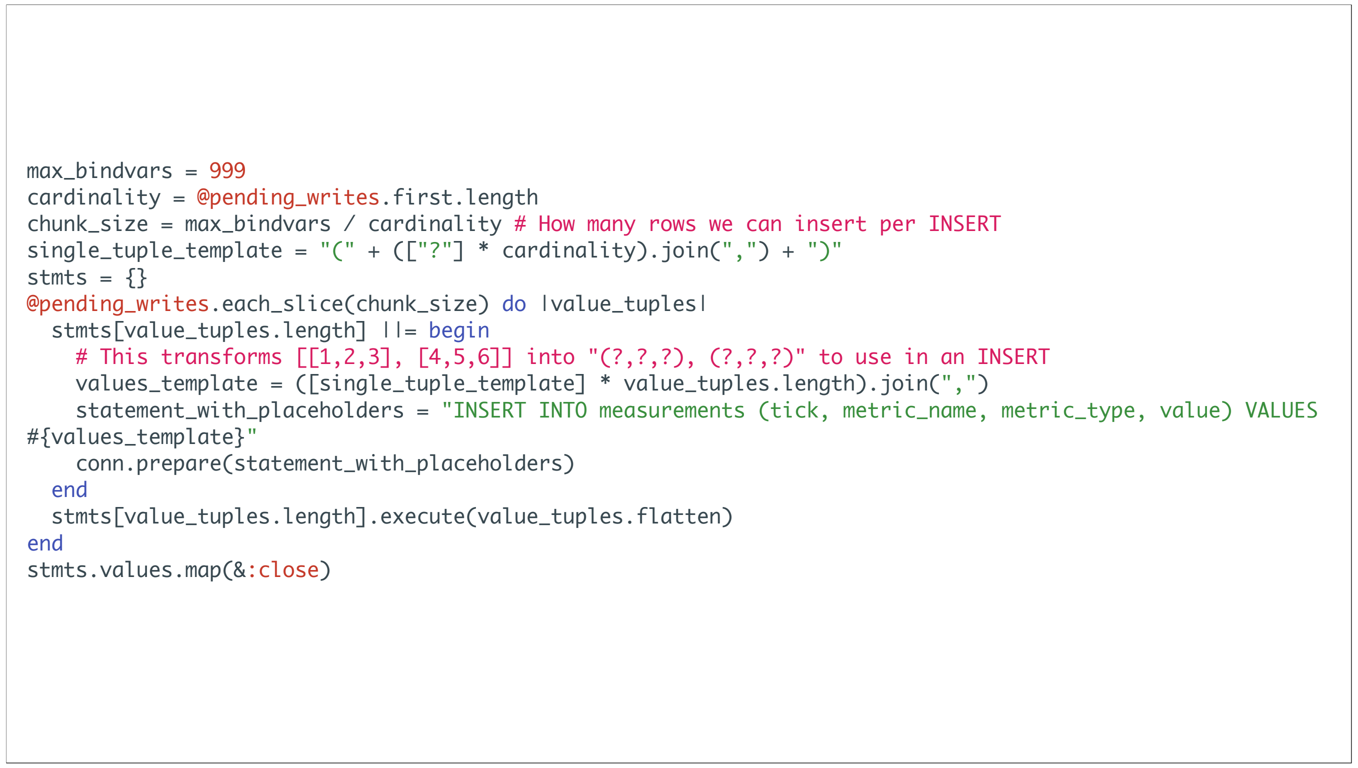
sqlite_master

	metric_name	metric_type
1	accounts_never_synced	gauge
2	accounts_selected_for_sync	sample
3	accounts_synced_at_least_once	gauge
4	done_jobs	counter
5	enqueued_jobs	counter
6	executing_jobs	gauge
7	execution_time	sample
8	n_accounts	gauge
9	n_parallel_workers	gauge
10	queue_backlog	gauge
11	queue_depth	gauge
12	start_delay	sample
13	sync_delay	sample
14	time_since_last_sync	sample

14 records

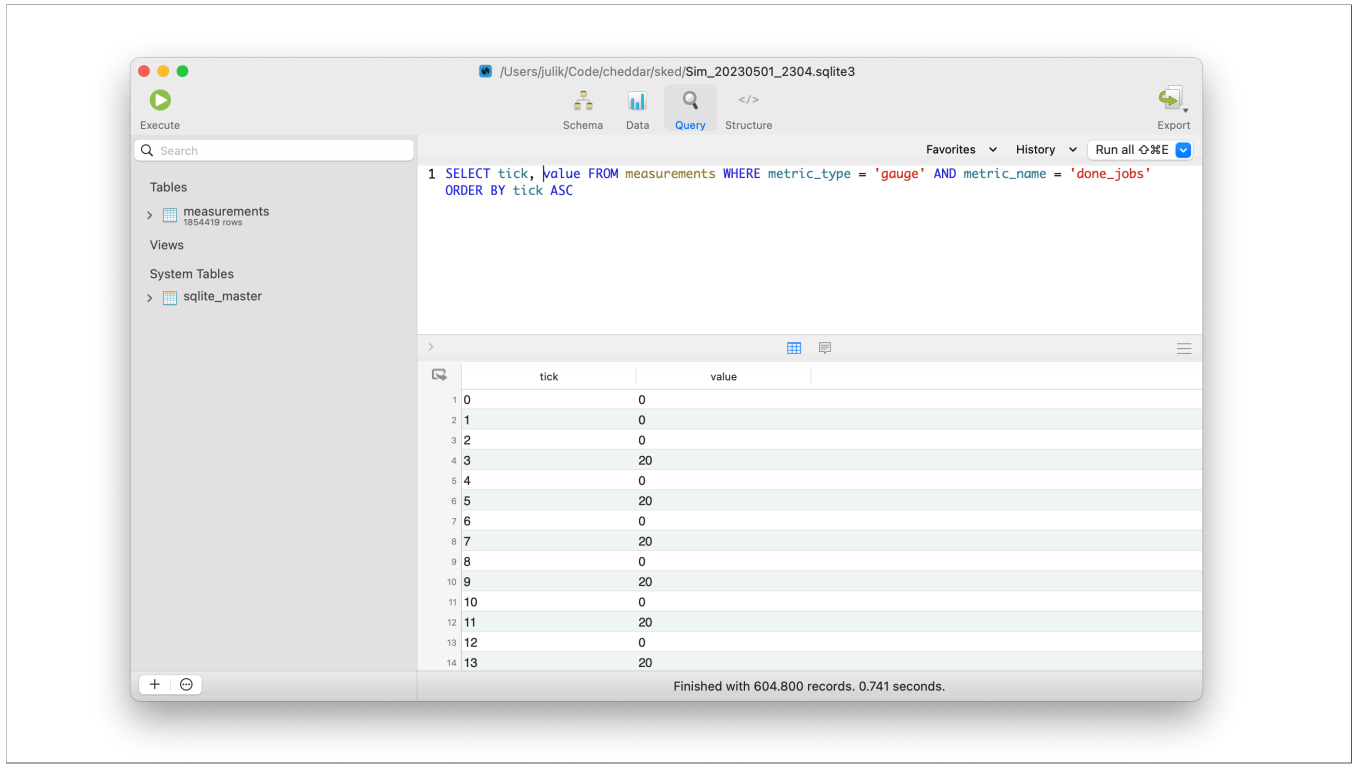
For example, it records how long the jobs were executing for - at every tick. Also, it records how many jobs were waiting on the queue at every tick. In essence, it "previews" your throughput graph for you in advance.

And we have written a few simulations using that simulation engine. These simulations would inject fake jobs into the queue, and those jobs would "mark" our fake accounts as synced at a certain time tick - as well as emit _more_ jobs into the queue.



```ruby
max_bindvars = 999
cardinality = @pending_writes.first.length
chunk_size = max_bindvars / cardinality # How many rows we can insert per INSERT
single_tuple_template = "(" + (["?"] * cardinality).join(",") + ")"
stmts = {}
@pending_writes.each_slice(chunk_size) do |value_tuples|
  stmts[value_tuples.length] ||= begin
    # This transforms [[1,2,3], [4,5,6]] into "(?,?,?), (?,?,?)" to use in an INSERT
    values_template = ([single_tuple_template] * value_tuples.length).join(",")
    statement_with_placeholders = "INSERT INTO measurements (tick, metric_name, metric_type, value) VALUES
#{values_template}"
    conn.prepare(statement_with_placeholders)
  end
  stmts[value_tuples.length].execute(value_tuples.flatten)
end
stmts.values.map(&:close)
```

This slide is here just for Stephen, find me after the talk if you are interested - this basically inserts rows into a SQLite database at the highest possible speed, so quick it's insane. Anyhow... having this data is valuable

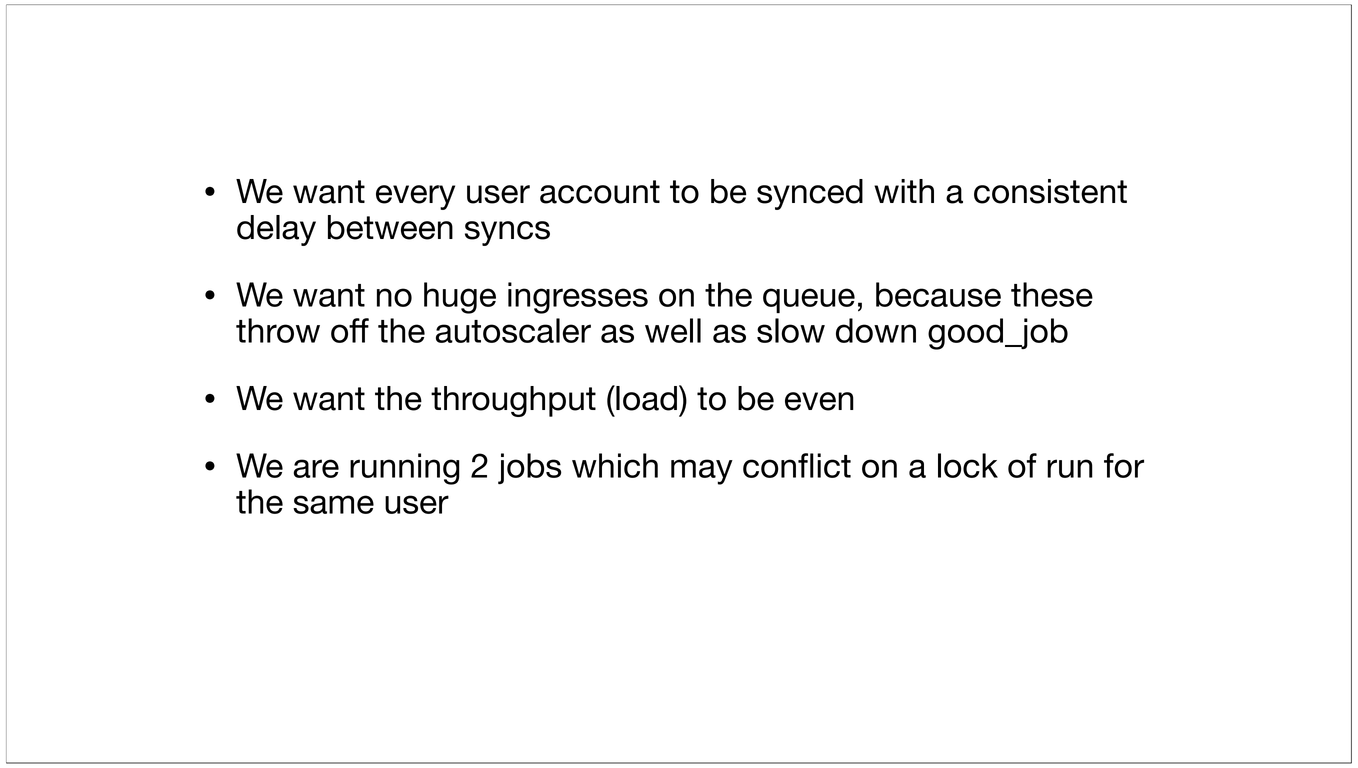


```
SELECT tick, value FROM measurements WHERE metric_type = 'gauge' AND metric_name = 'done_jobs'
ORDER BY tick ASC
```

	tick	value
1	0	0
2	1	0
3	2	0
4	3	20
5	4	0
6	5	20
7	6	0
8	7	20
9	8	0
10	9	20
11	10	0
12	11	20
13	12	0
14	13	20

Finished with 604.800 records. 0.741 seconds.

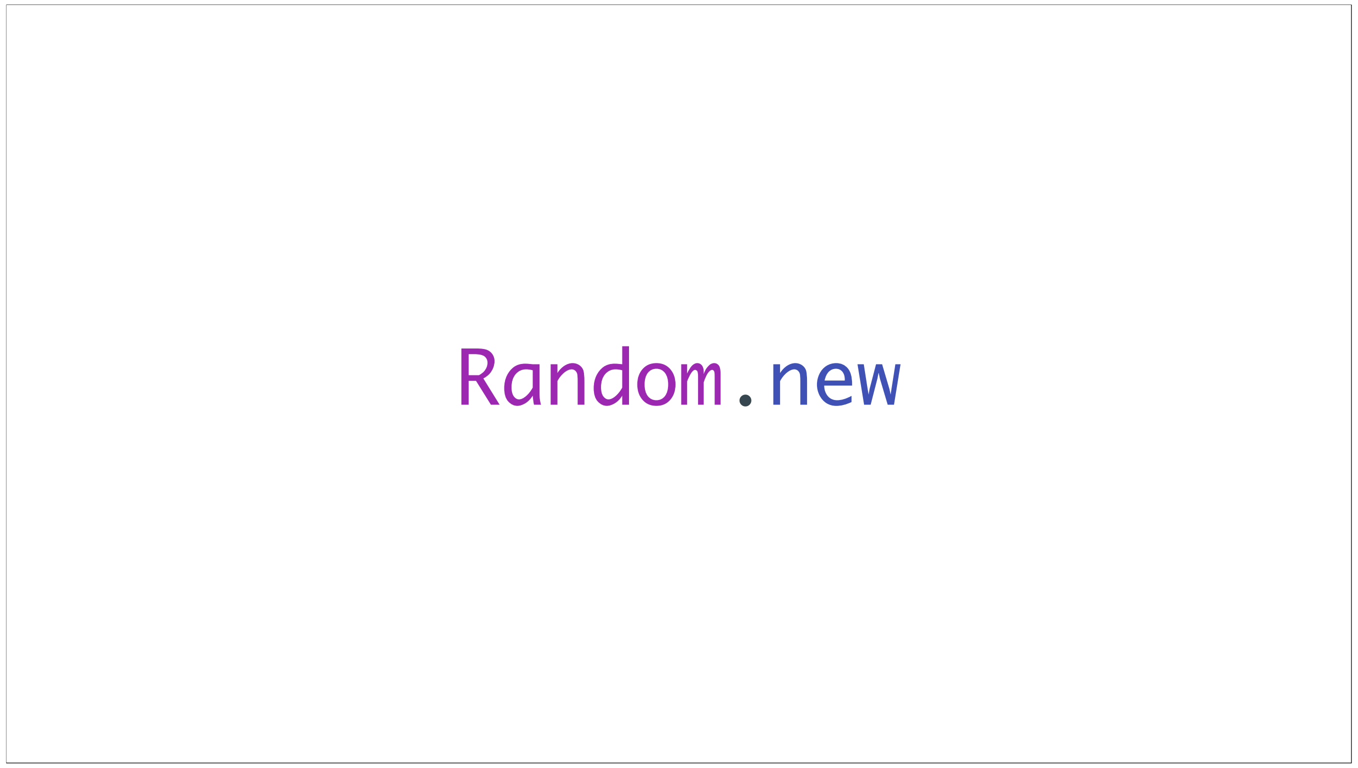
like here - we can see how many jobs would get finished at every tick of the simulation



- We want every user account to be synced with a consistent delay between syncs
- We want no huge ingresses on the queue, because these throw off the autoscaler as well as slow down good_job
- We want the throughput (load) to be even
- We are running 2 jobs which may conflict on a lock of run for the same user

And that `SlotScheduling` thing turned out to be quite neat in that regard. We thought it won't work, but it did.

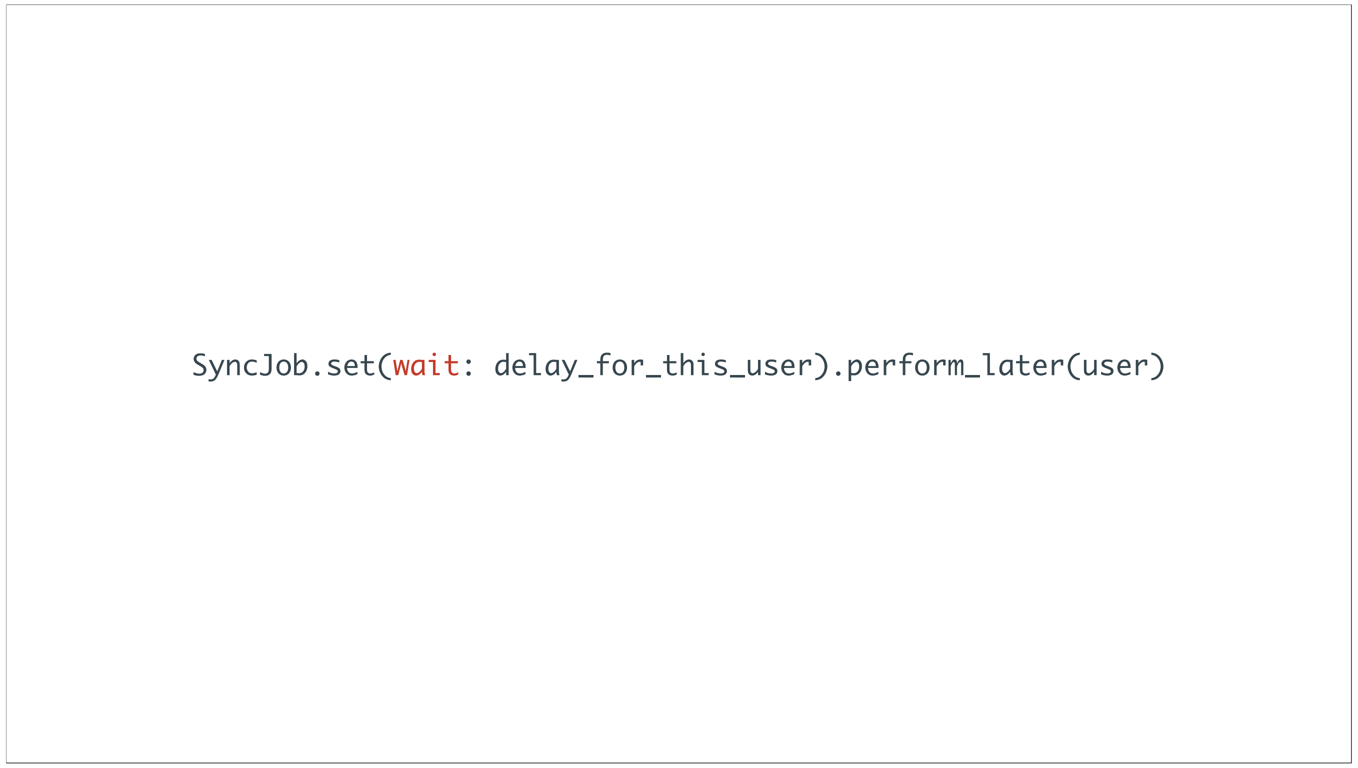
Here's the basic premise. We have N users (that N always changes as users join and leave the platform, as well as become inactive). We need to evenly spread them across an existing time interval of 12 hours - where we want all the syncs to have completed for them. How do we achieve that?



```
Random.new
```

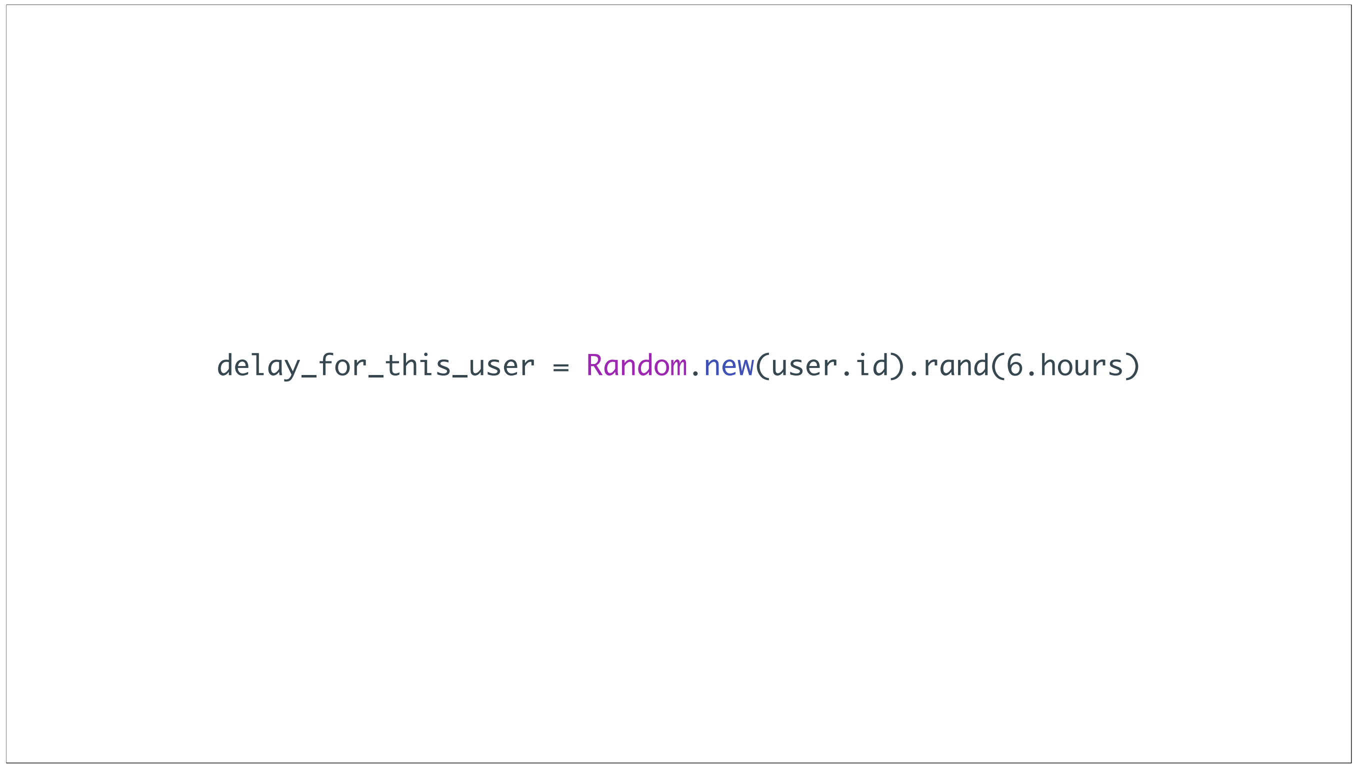
Seriously, that little class (Random) in Ruby is pretty underrated, and I'll show you some things that you can do. The reason why it is so useful is this: it is not, in fact, random at all.

It can give you _reproducible_ randomness. It is a deterministic memoizing algorithm called a Mersenne twister. Given a specific _seed_ it will, in fact, produce a repeatable, pre-determined sequence of random values. So: what if we did the most stupid thing possible, and just "smeared out" all of our user's sync tasks across our interval?



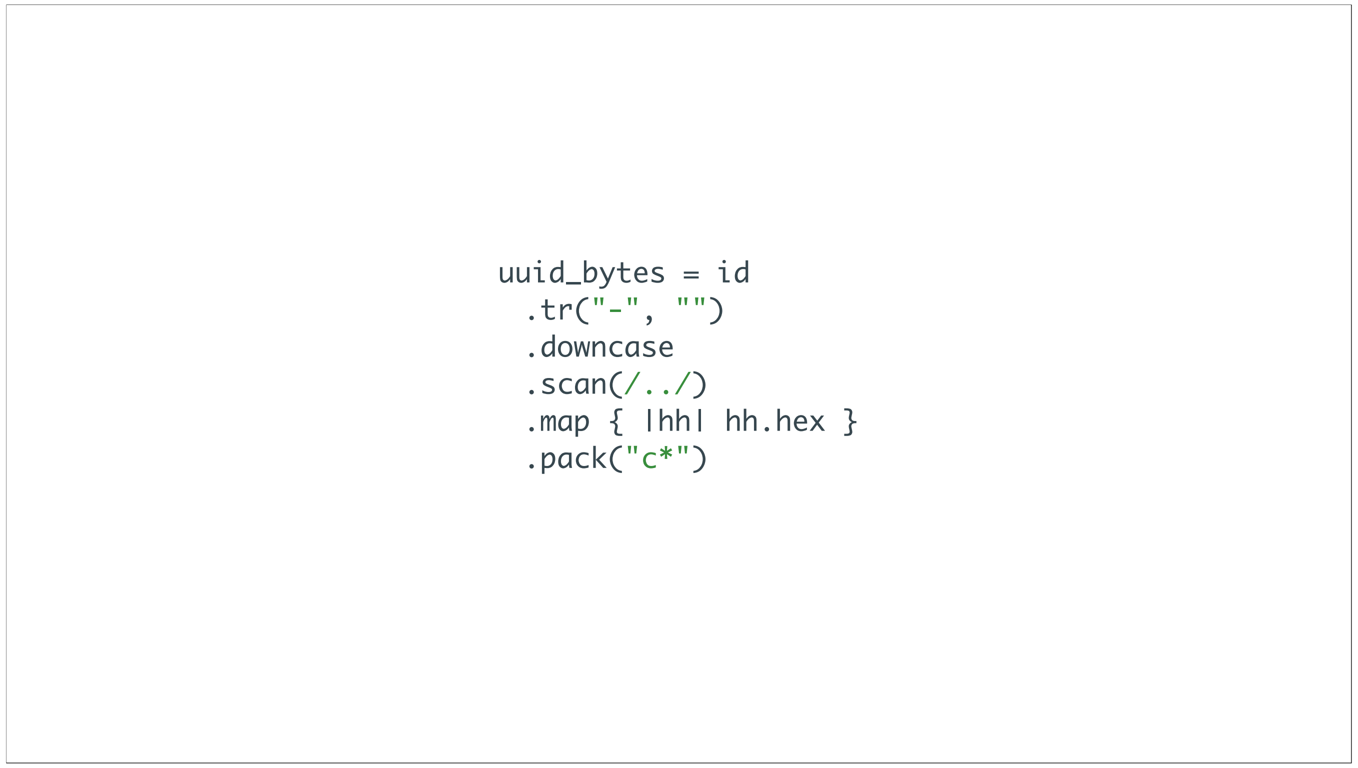
```
SyncJob.set(wait: delay_for_this_user).perform_later(user)
```

How do we do this? Well, if your users have integer primary keys (and I implore you: unless you have strong reasons not to do this - use bigint primary keys and be done with it), it is fairly simple. See, `Random` can be seeded with any integer value. So your user ID is already good enough for this:



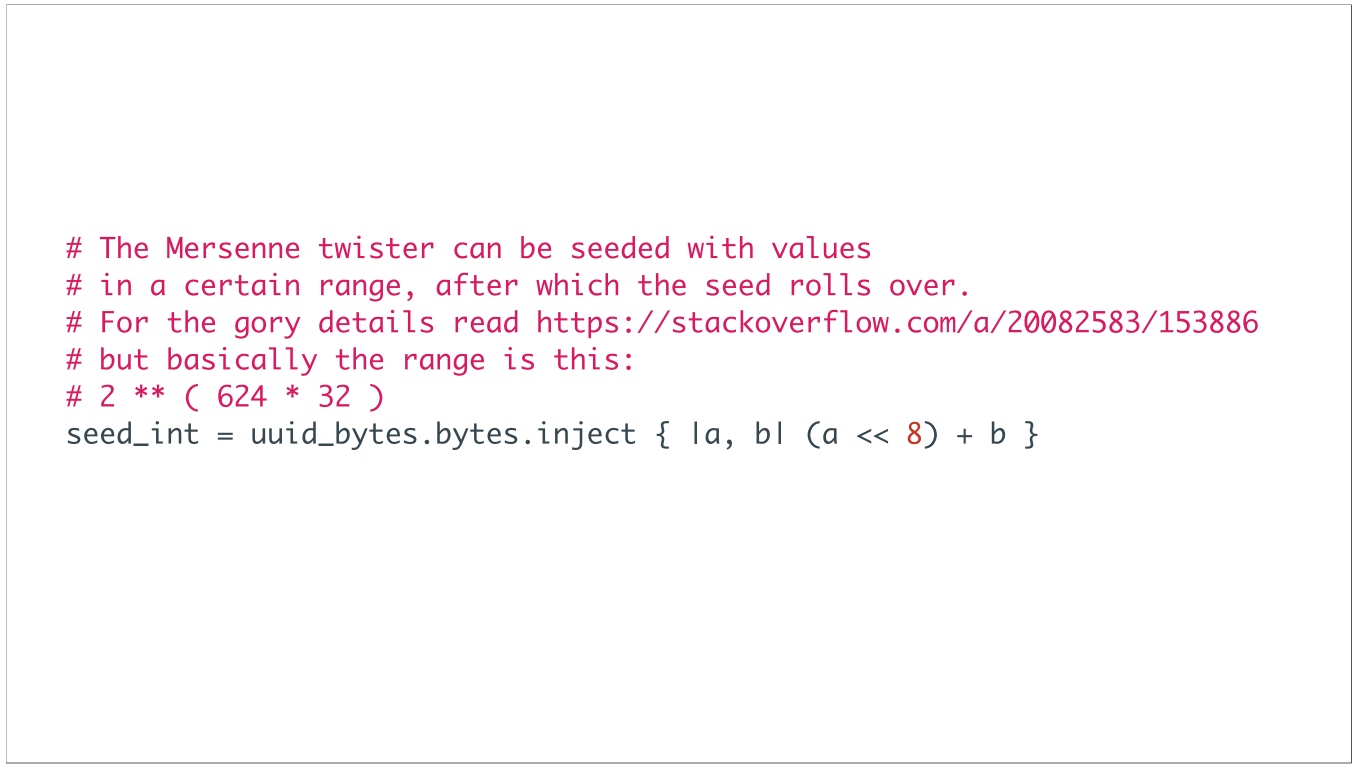
```ruby
delay_for_this_user = Random.new(user.id).rand(6.hours)
```

Boom, done. Here we take the user ID, we use it as the seed for the random number generator, and we make the generator give us a random number of seconds between 0 and 6 hours. And we would be done here, be it not for the UUIDs.



```ruby
uuid_bytes = id
  .tr("-", "")
  .downcase
  .scan(/../)
  .map { |hh| hh.hex }
  .pack("c*")
```

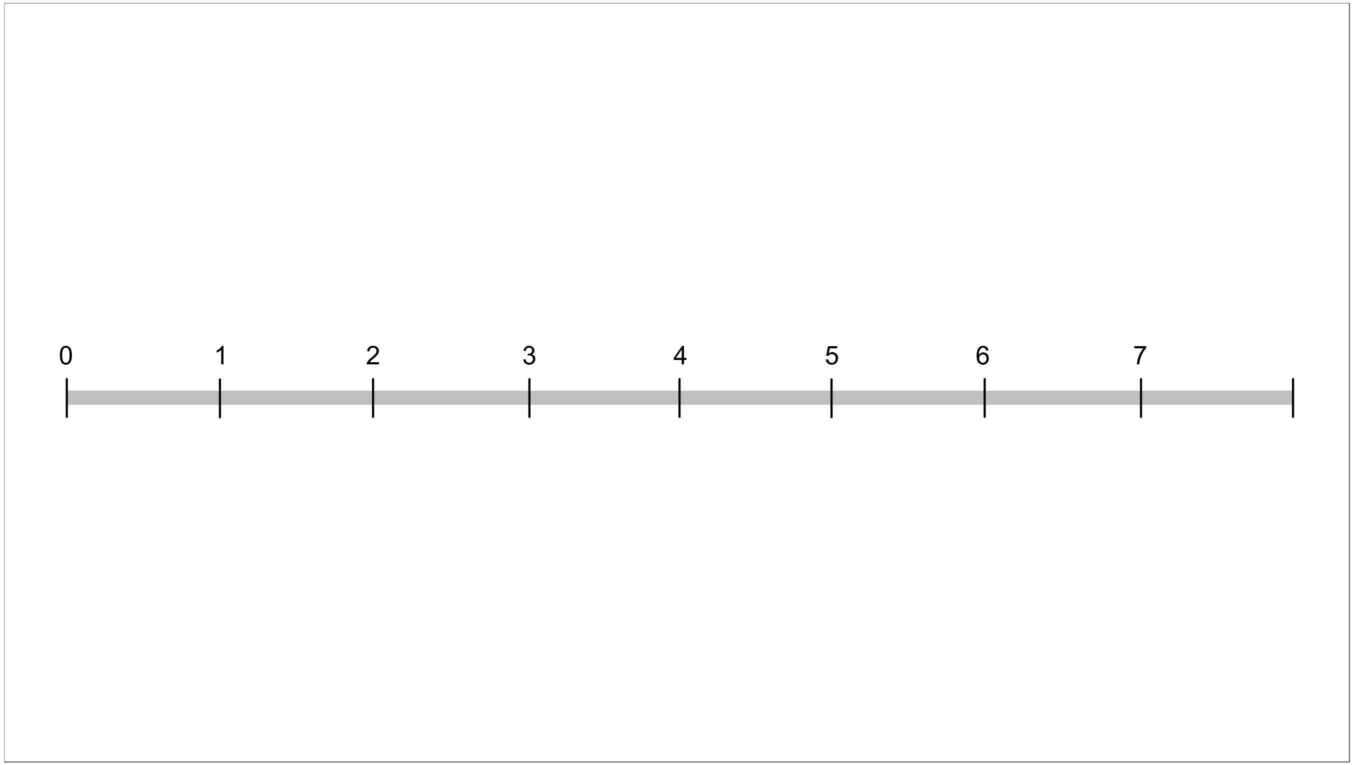
First, we need to convert the textual (string) representation of the UUID to a bag of bytes - what the UUID actually is. Afterwards, we accumulate bits from it to generate a seed:



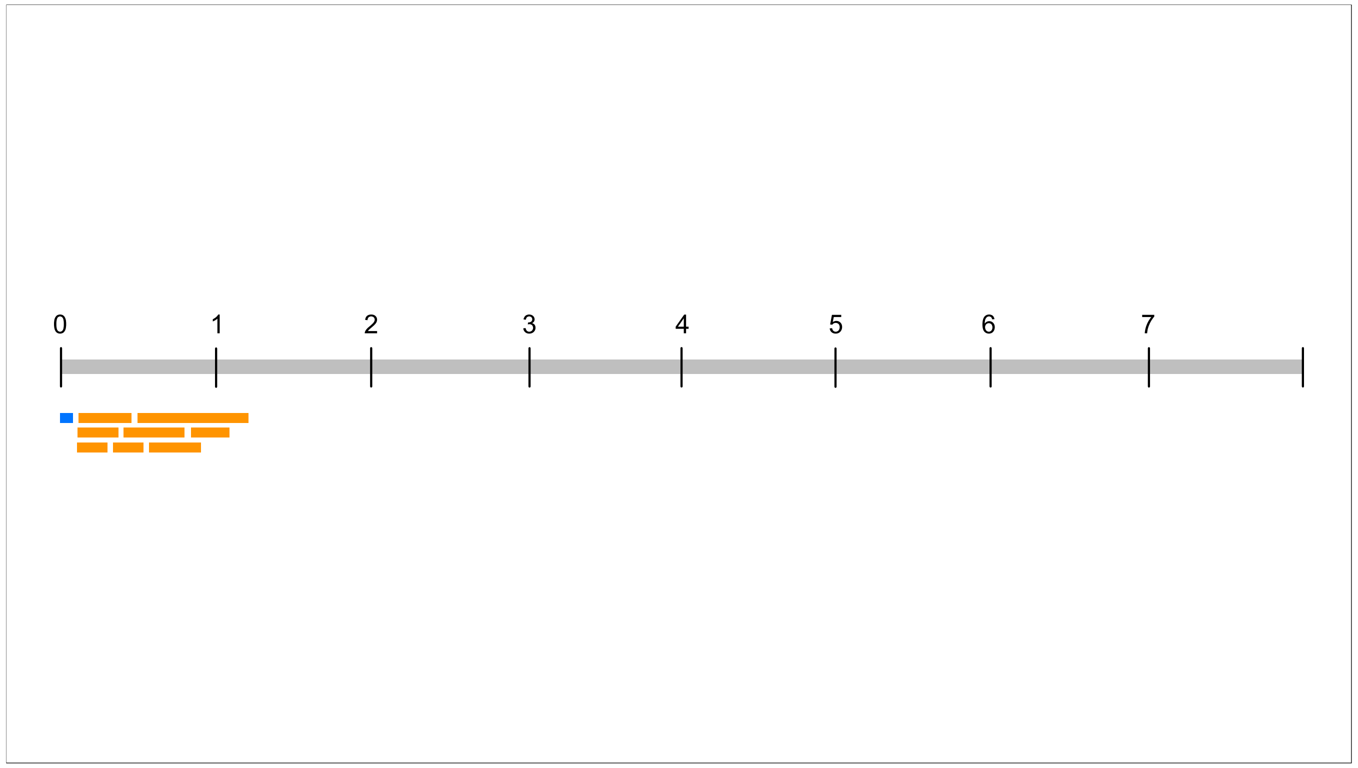
```
# The Mersenne twister can be seeded with values
# in a certain range, after which the seed rolls over.
# For the gory details read https://stackoverflow.com/a/20082583/153886
# but basically the range is this:
# 2 ** ( 624 * 32 )
seed_int = uuid_bytes.bytes.inject { |a, b| (a << 8) + b }
```

...and then we do Random dot new with that `seed_int` value.

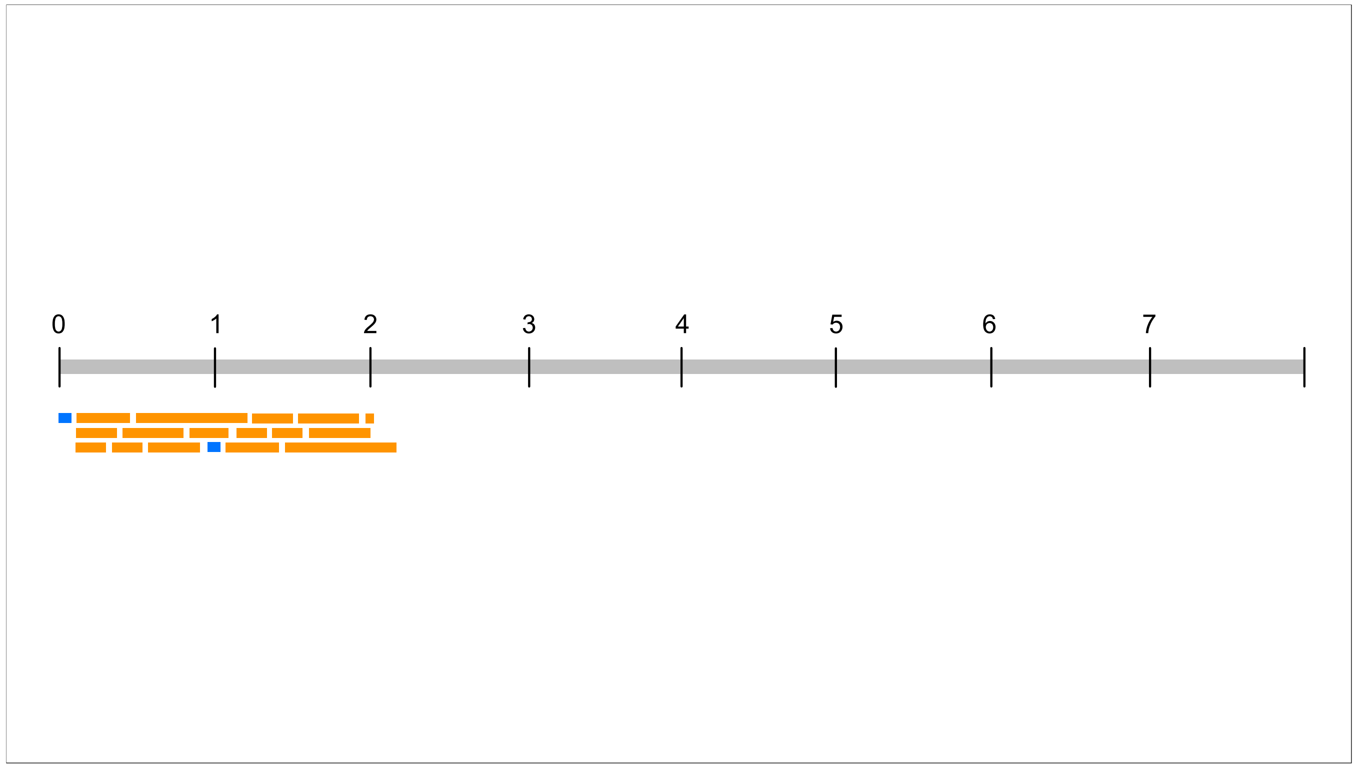
Now, this would have been great already - but we haven't solved our other problem. Remember how we were hurting a lot with this vicious circle of "more jobs on the queue - system heats up - load spikes"? So if we were to spool jobs for the entire userbase, all at once - the queues would have hundreds of thousands of jobs. This slows good_job down dramatically, and causes severe load on the DB. So "just" applying this uniformity delay was not enough, we needed to be a bit more sophisticated. So: why don't we portion out this workload instead?



It goes like this. Let's imagine our sync interval. If we divide it into a number of "slots" which would only perform tasks for a certain part of the userbase (and we would want to divide into a large number of those slots), we could spool one job per slot.

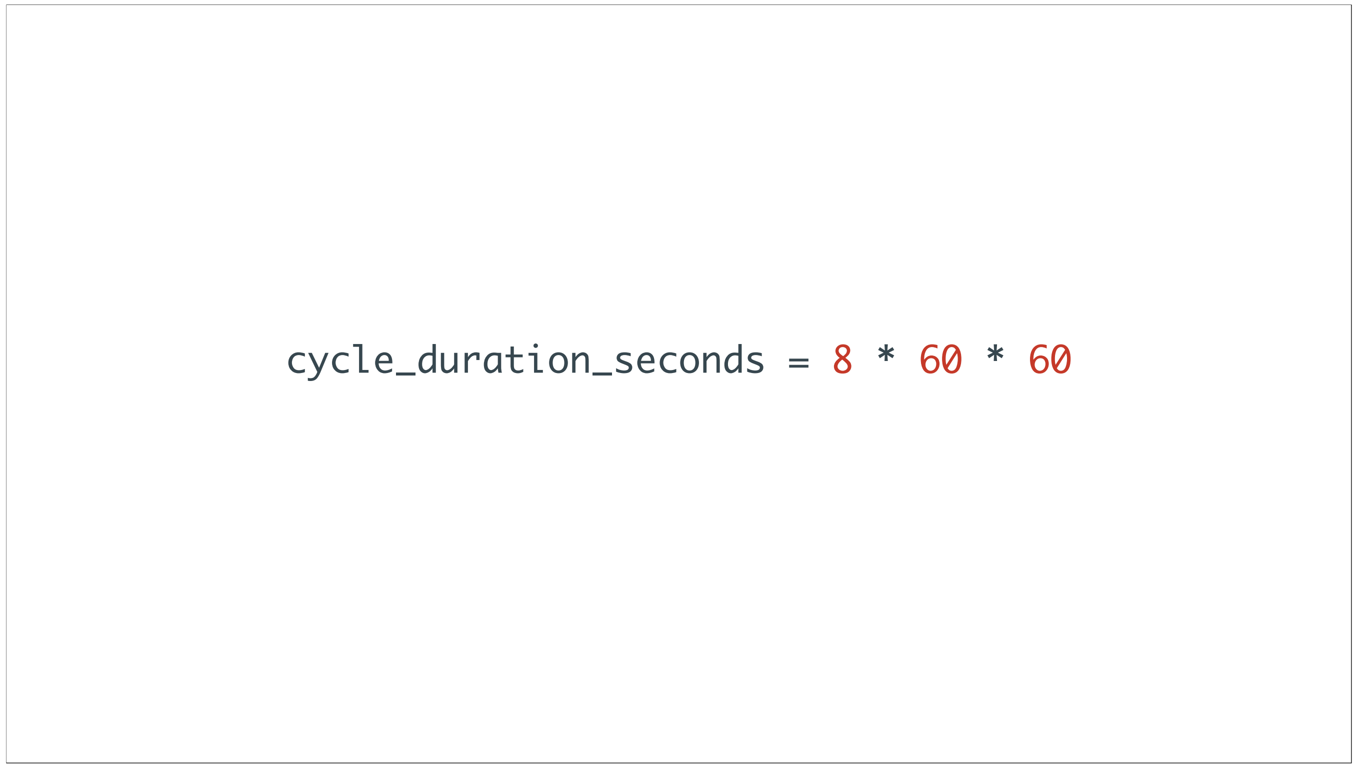


Here we will spool a job for slot 0 (it is blue). That job, in turn, spools up all the jobs for users in this slot 0, and spreads them over time until the start of the next slot - slot 1



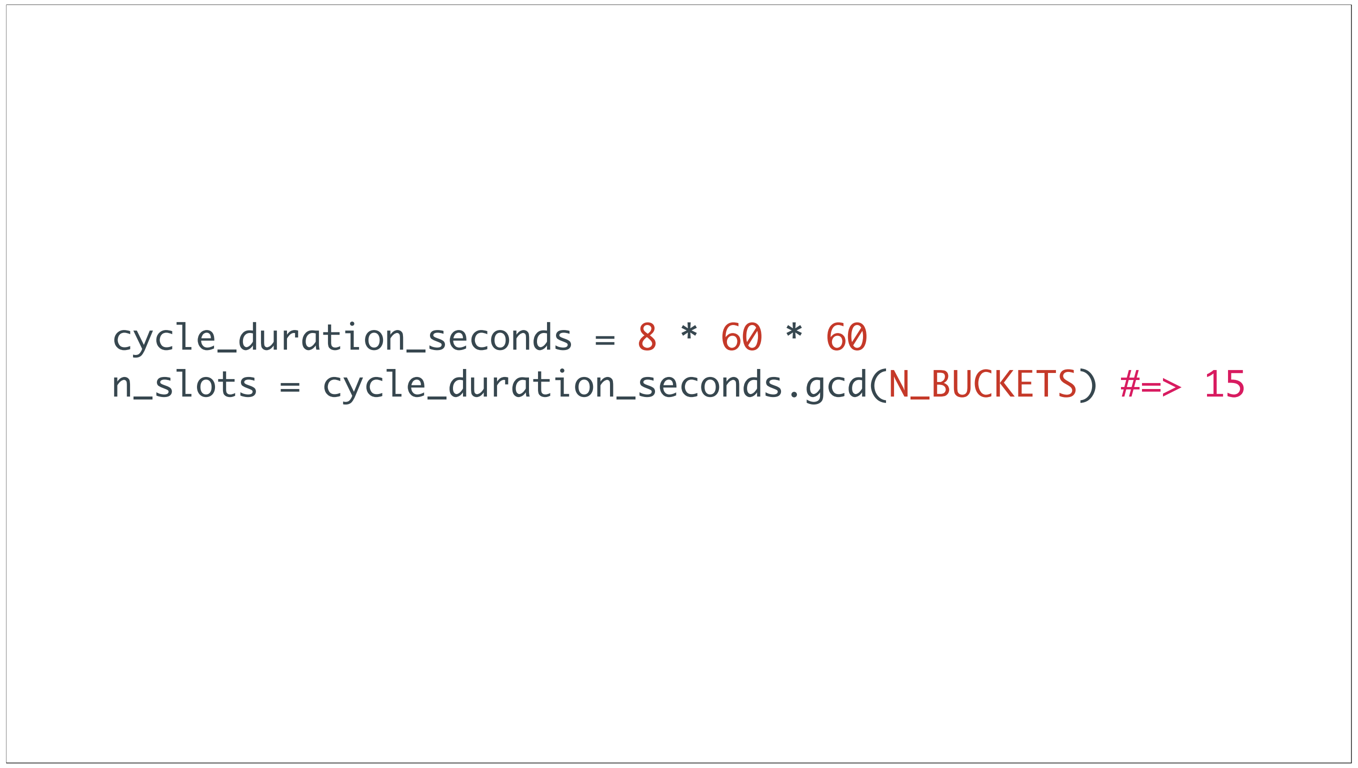
Then, slot 1 begins - and the same routine takes place. The job spools up tasks for users which fit between slot 0 and slot 2.

How many slots should we have? Well, there is a terrible interview question, which I hope you are not asking your candidates in interviews, because we care about people being able to do actual work and the ZIRP is over. There is actually a built in method for this. It is called "greatest common divisor". Imagine we have an interval to maintain of 8 hours:



```
cycle_duration_seconds = 8 * 60 * 60
```

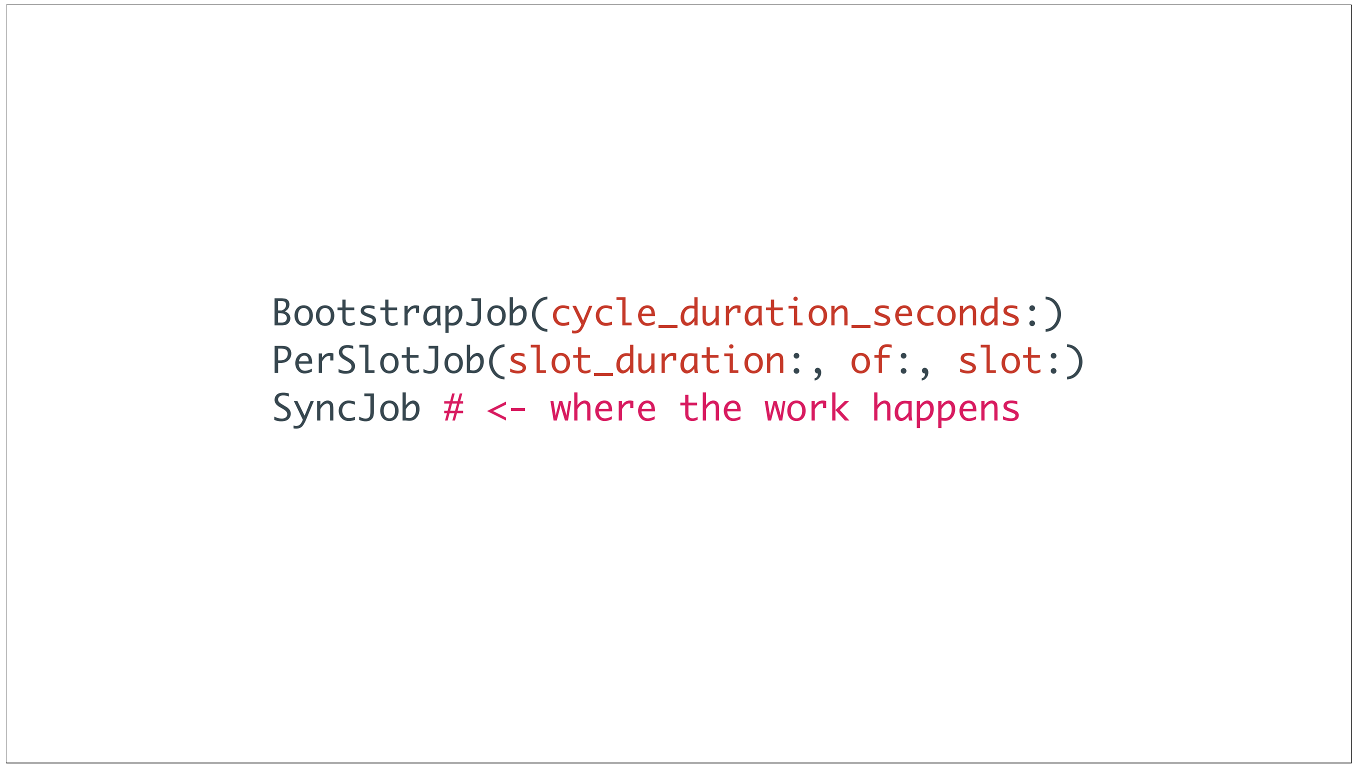
We know that we are dealing with IDs which can be transformed into "some kind of" byte representation, and therefore they are divisible by 256, and so on. We can also use two bytes, in which case we have even more "buckets" we can distribute users into - 65535. The balance here is to know the largest number of slots we can use as buckets for our users:



```ruby
cycle_duration_seconds = 8 * 60 * 60
n_slots = cycle_duration_seconds.gcd(N_BUCKETS) #=> 15
```

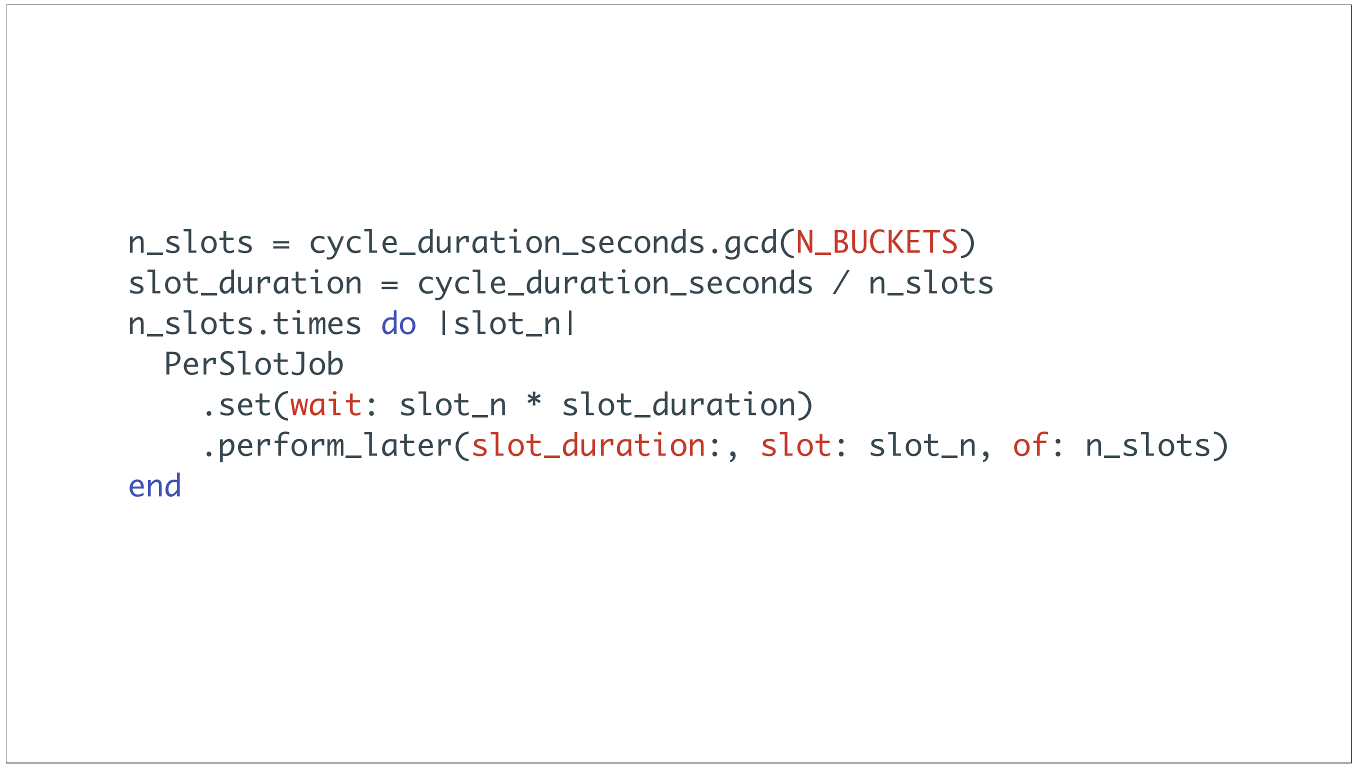
Yes, the Ruby stdlib has a `gcd` function, and it does exactly that - gives us the largest number of slots we can "slice up" our userbase into so that the slots contain the same number of possible buckets. And for this exercise, it gives us 15!

So, we are going to split our task into 3 parts. We will have:



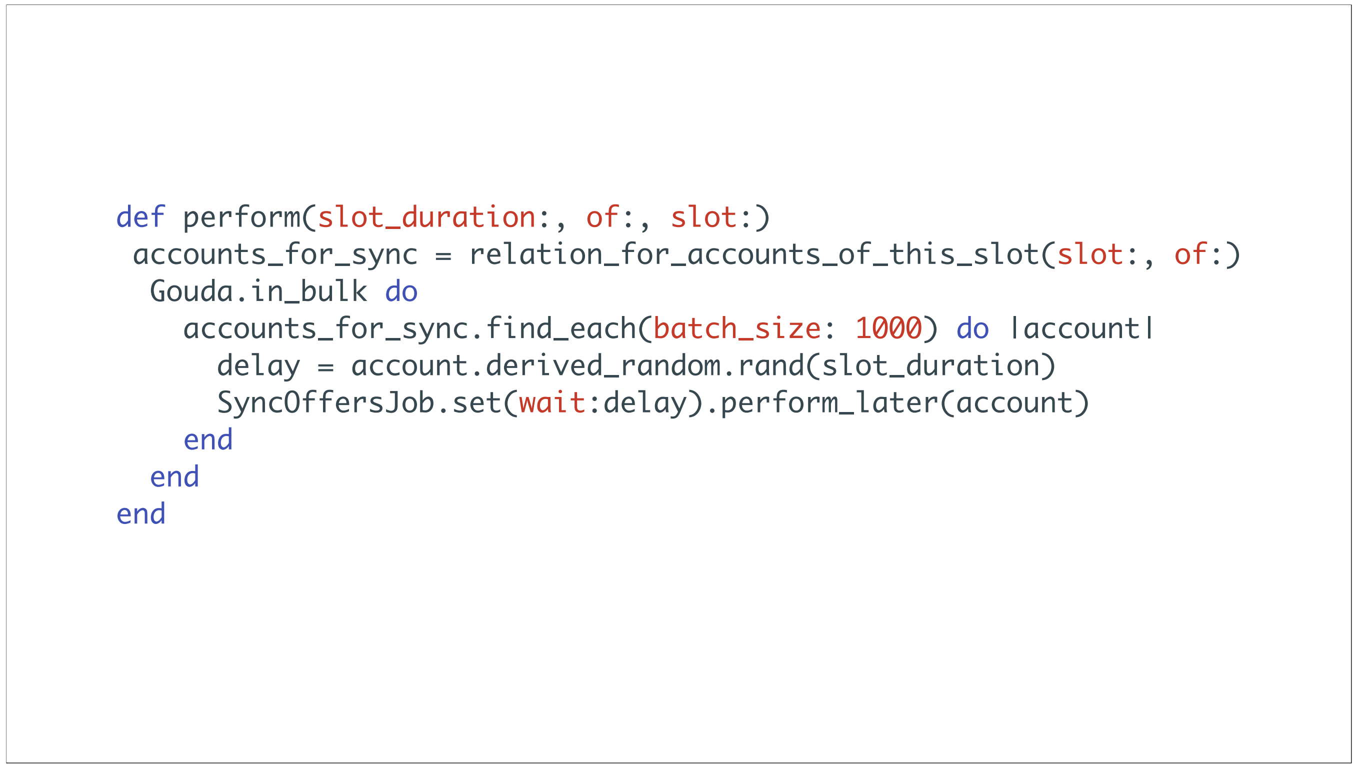
```
BootstrapJob(cycle_duration_seconds:)
PerSlotJob(slot_duration:, of:, slot:)
SyncJob # <- where the work happens
```

We have a setup of 3 jobs - the bootstrap (runs at the start of the cycle), the PerSlotJob (runs at the start of each slot) and SyncJob itself. When we start our sync, we run the bootstrap job. It computes how many slots we are going to have, and enqueues the jobs as follows:



```ruby
n_slots = cycle_duration_seconds.gcd(N_BUCKETS)
slot_duration = cycle_duration_seconds / n_slots
n_slots.times do |slot_n|
  PerSlotJob
    .set(wait: slot_n * slot_duration)
    .perform_later(slot_duration:, slot: slot_n, of: n_slots)
end
```

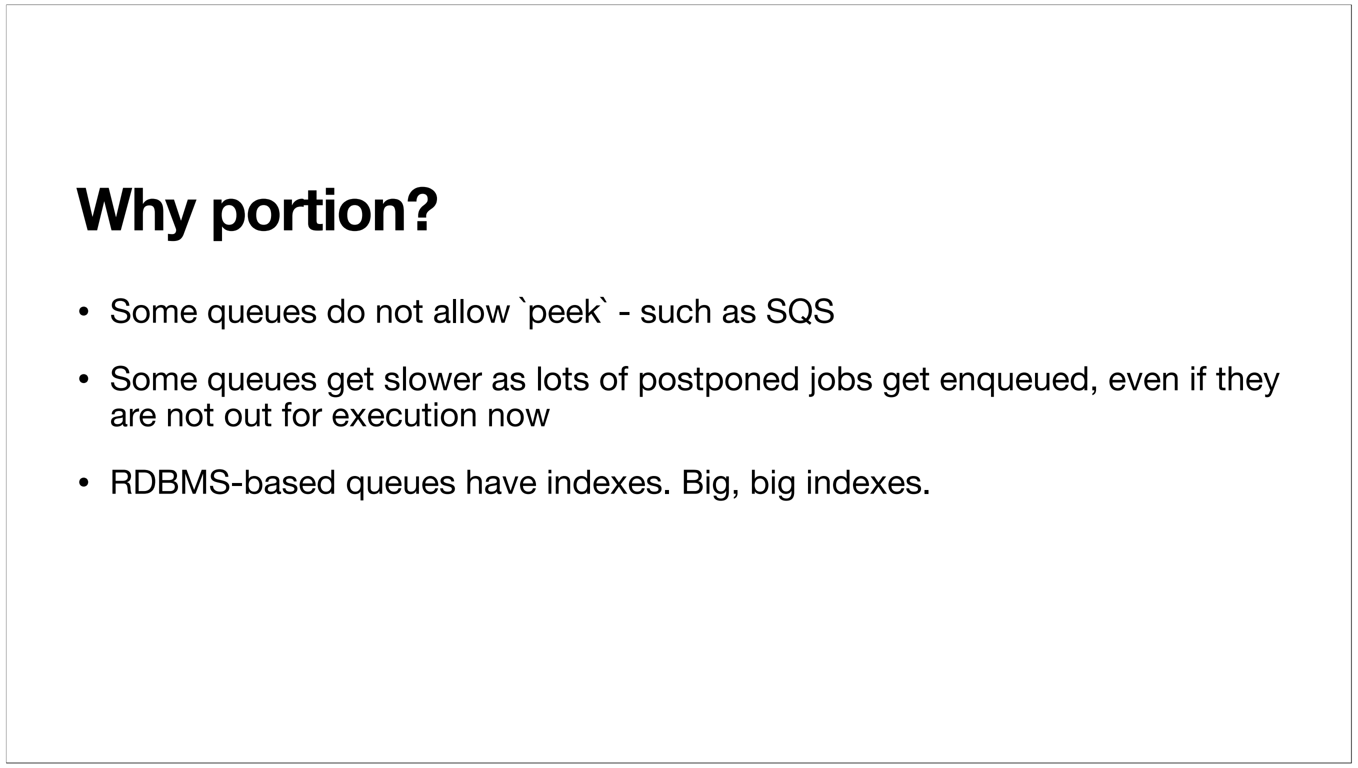
We enqueue as many PerSlotJobs as we have slots, and we delay the start of each PerSlot job by the amount of time that will pass until that slot will begin. We then tell that job that it is running for a slot number N, of M slots total, and that the slot has a duration. Then, when the PerSlot job runs, it will do this:



```ruby
def perform(slot_duration:, of:, slot:)
 accounts_for_sync = relation_for_accounts_of_this_slot(slot:, of:)
  Gouda.in_bulk do
    accounts_for_sync.find_each(batch_size: 1000) do |account|
      delay = account.derived_random.rand(slot_duration)
      SyncOffersJob.set(wait:delay).perform_later(account)
    end
  end
end
```

So within the `slot_duration` the `PerSlotJob` also distributes the jobs of specific users _within_ that slot, using the same account derived random. But it means that the ingress on the queue is going to be just 1/15th of the number of jobs we would do previously!

This has a lot of advantages. Firstly, you can operate your queues better. You get more time to kill those `PerSlotJobs` if something is malfunctioning, for example!

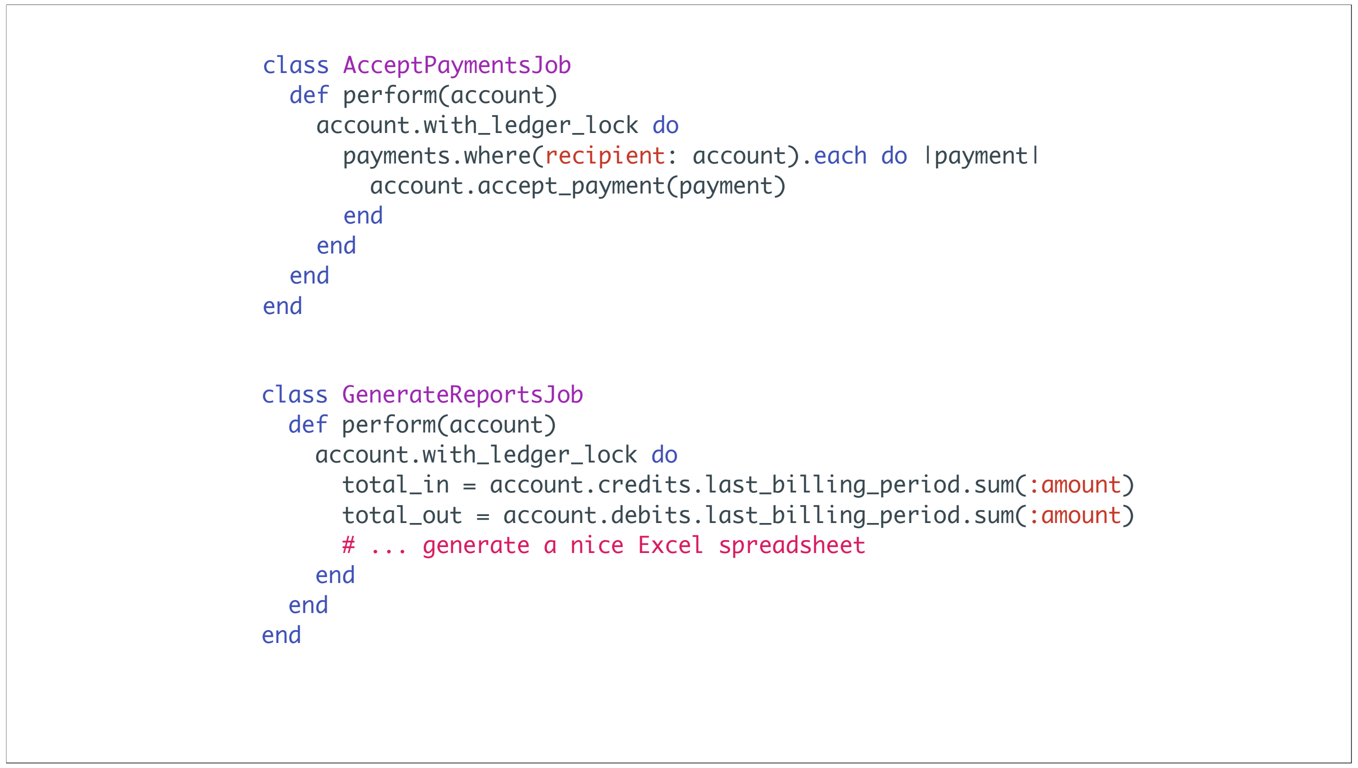


Why portion?

- Some queues do not allow `peek` - such as SQS
- Some queues get slower as lots of postponed jobs get enqueued, even if they are not out for execution now
- RDBMS-based queues have indexes. Big, big indexes.

For example, for us one of the bottlenecks was good_job. Good_job is great, but it takes advisory locks on jobs it wants to execute. The more jobs you have in the queue - the more locks it will try to acquire, and the more load there is going to be on the queue.

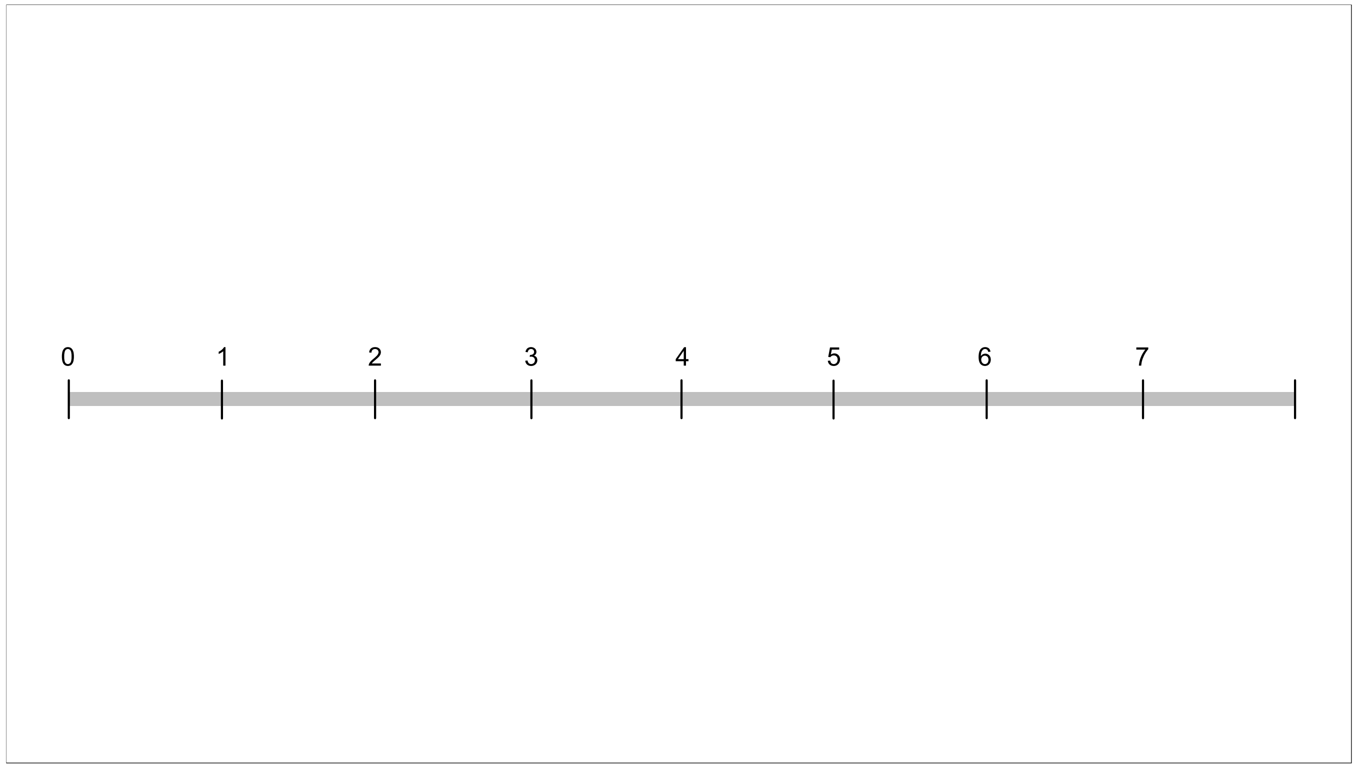
But it gets even better - with what we call "reciprocals". Imagine you have these two jobs:



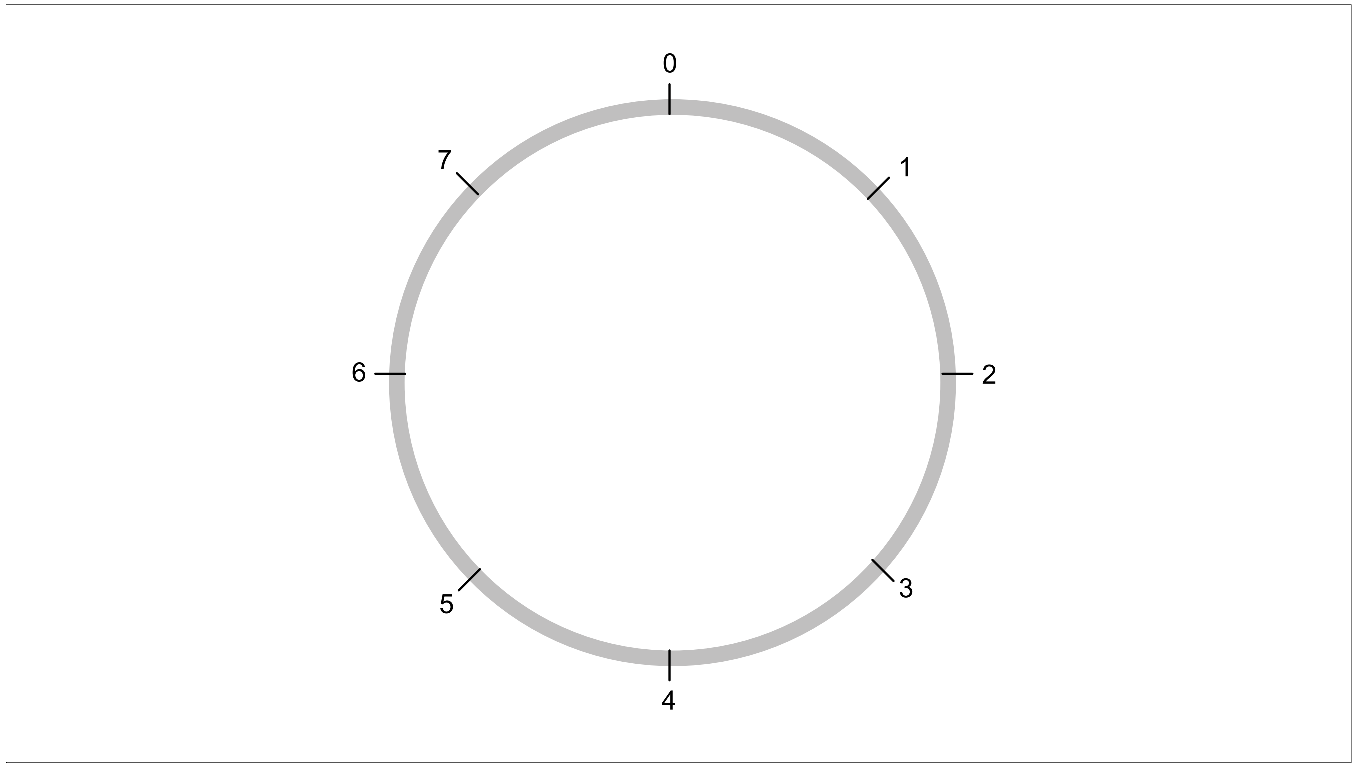
```ruby
class AcceptPaymentsJob
  def perform(account)
    account.with_ledger_lock do
      payments.where(recipient: account).each do |payment|
        account.accept_payment(payment)
      end
    end
  end
end


class GenerateReportsJob
  def perform(account)
    account.with_ledger_lock do
      total_in = account.credits.last_billing_period.sum(:amount)
      total_out = account.debits.last_billing_period.sum(:amount)
      # ... generate a nice Excel spreadsheet
    end
  end
end
```

When we are generating those reports we want to lock the ledger for that user, because we don't want payments to arrive just as we are crunching their numbers. Running those two jobs together would lead to lock contention, if we are not careful. They will also read and write on the same database table - so the DB will have to do locking for us too!

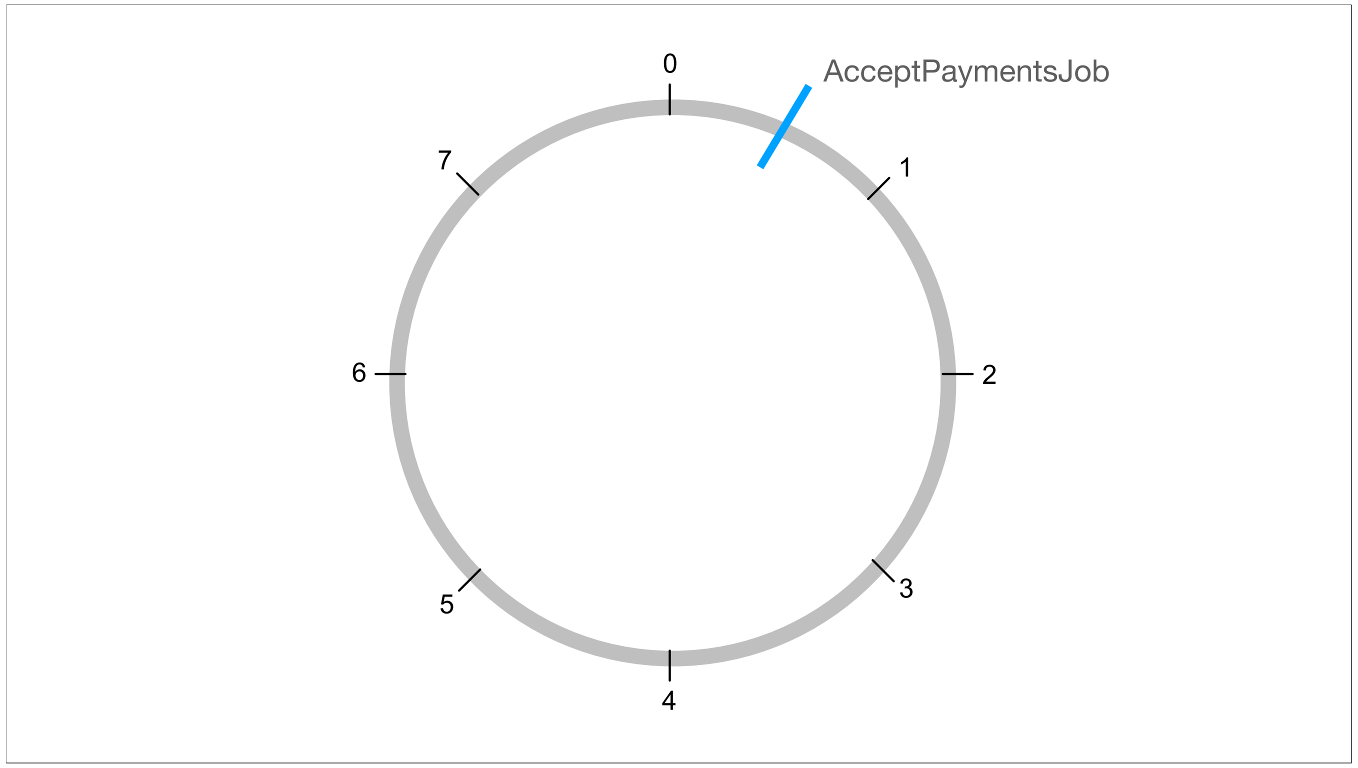
There is a really neat approach to this. Let's get back to our example with 8 slots:



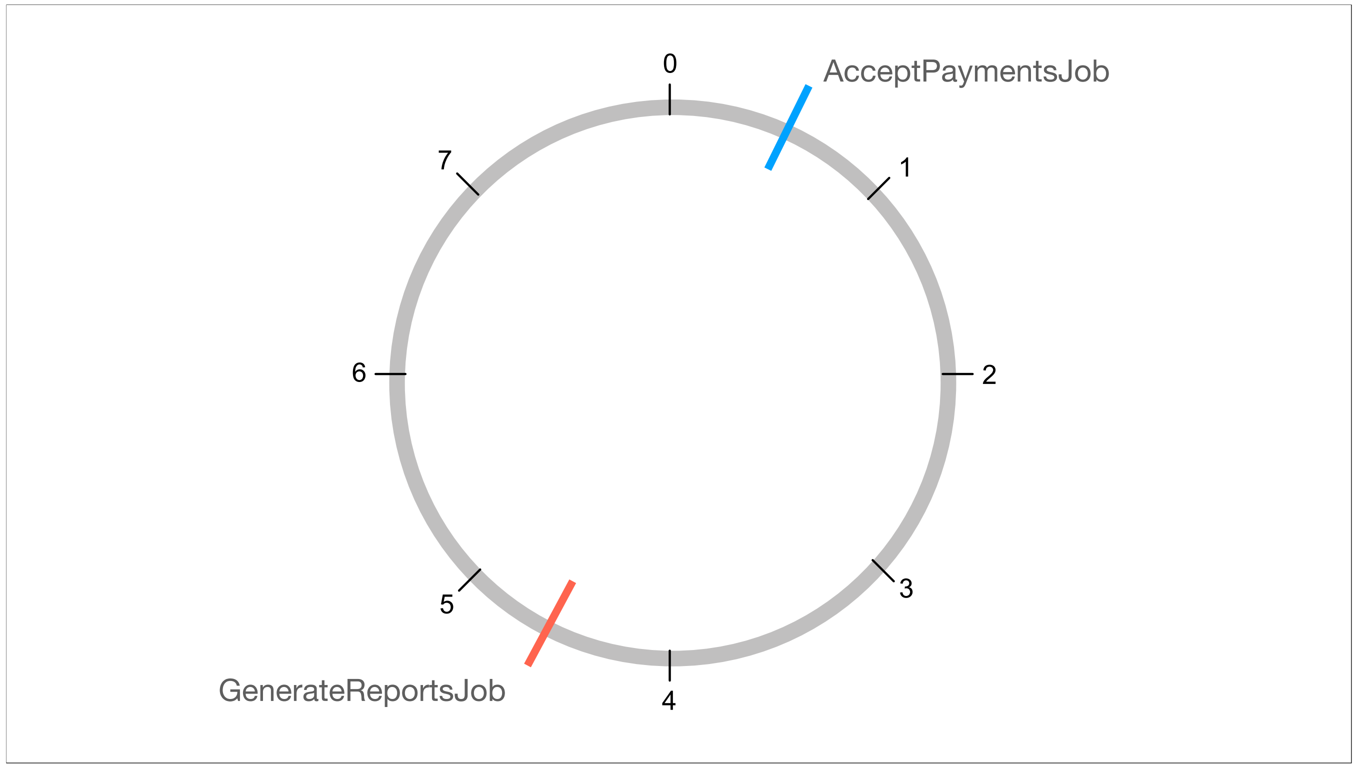
What if we take that interval and bend it into a donut, like so?



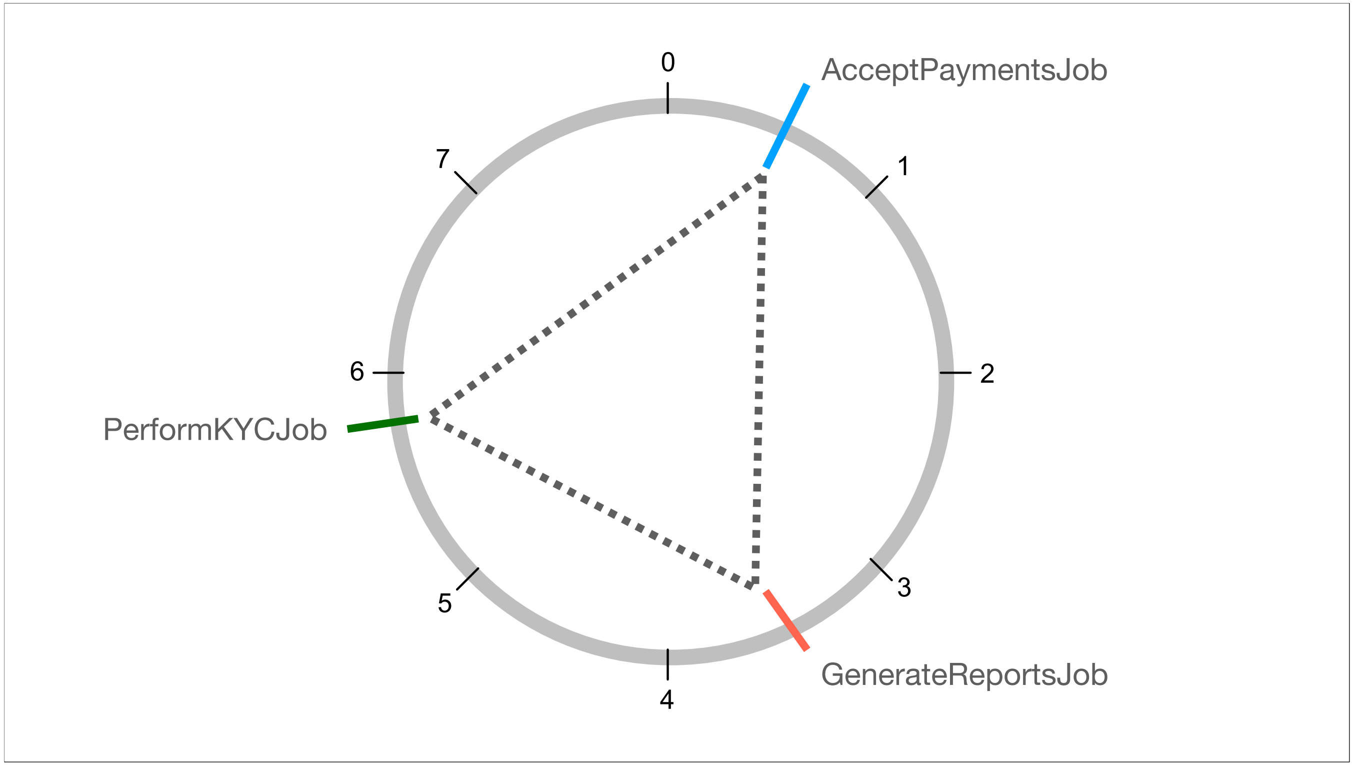
And then what we can do, is pinpoint for which slot we are now doing the `AcceptPaymentsJob`:



And then what we can do, is pinpoint for which slot we are now doing the `AcceptPaymentsJob`: Now, let's plot a diameter on this circle to find the furthest slot from this one. And when we run the AcceptPaymentsJobs for users in slot 0, we know that we can run the `GenerateReportsJob` for users in slot 4 at the same time - we have the smallest chance of these two jobs locking the same resources!

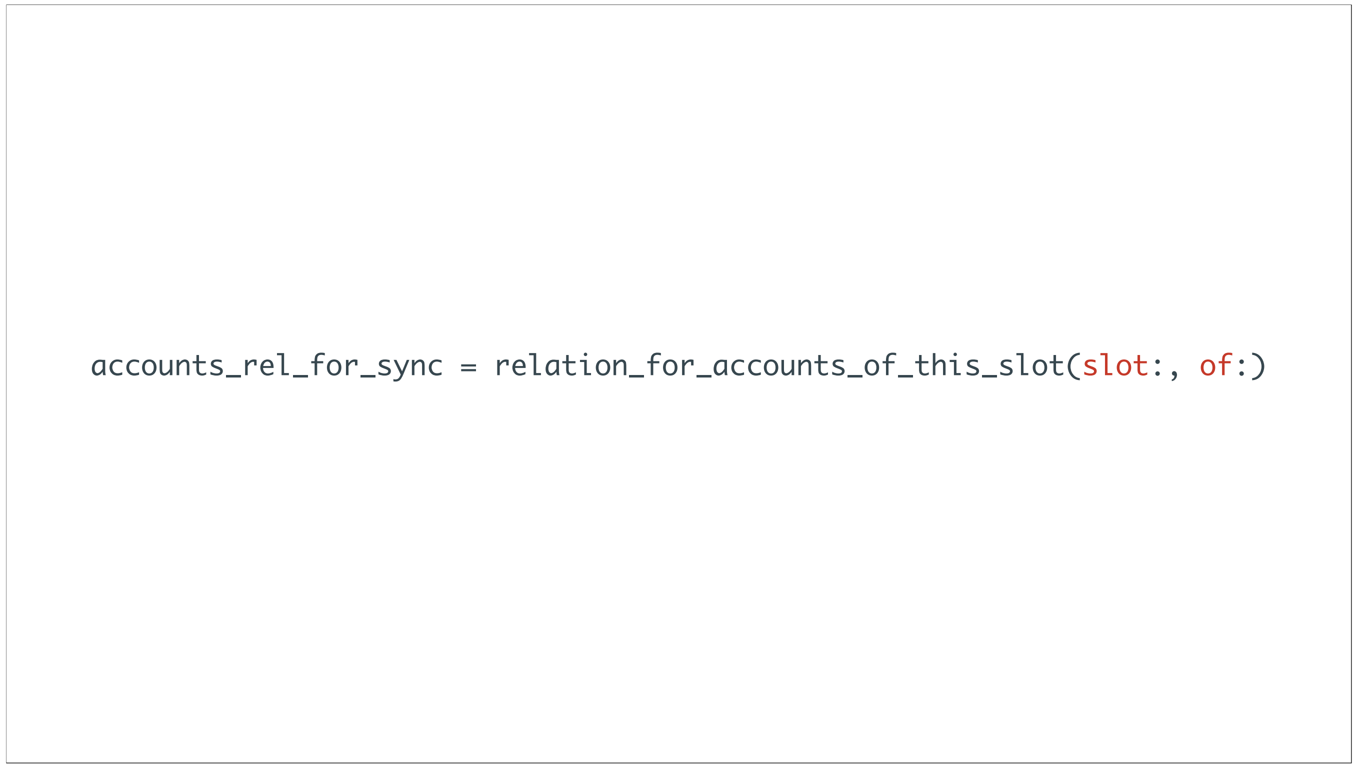


Like this. So: users currently in slot 0 will have their And this can be expanded. For example, imagine we add another job into the mix - say, `PerformKYCJob`. We want it not to interfere with either the first or the second job. Where do we place it?



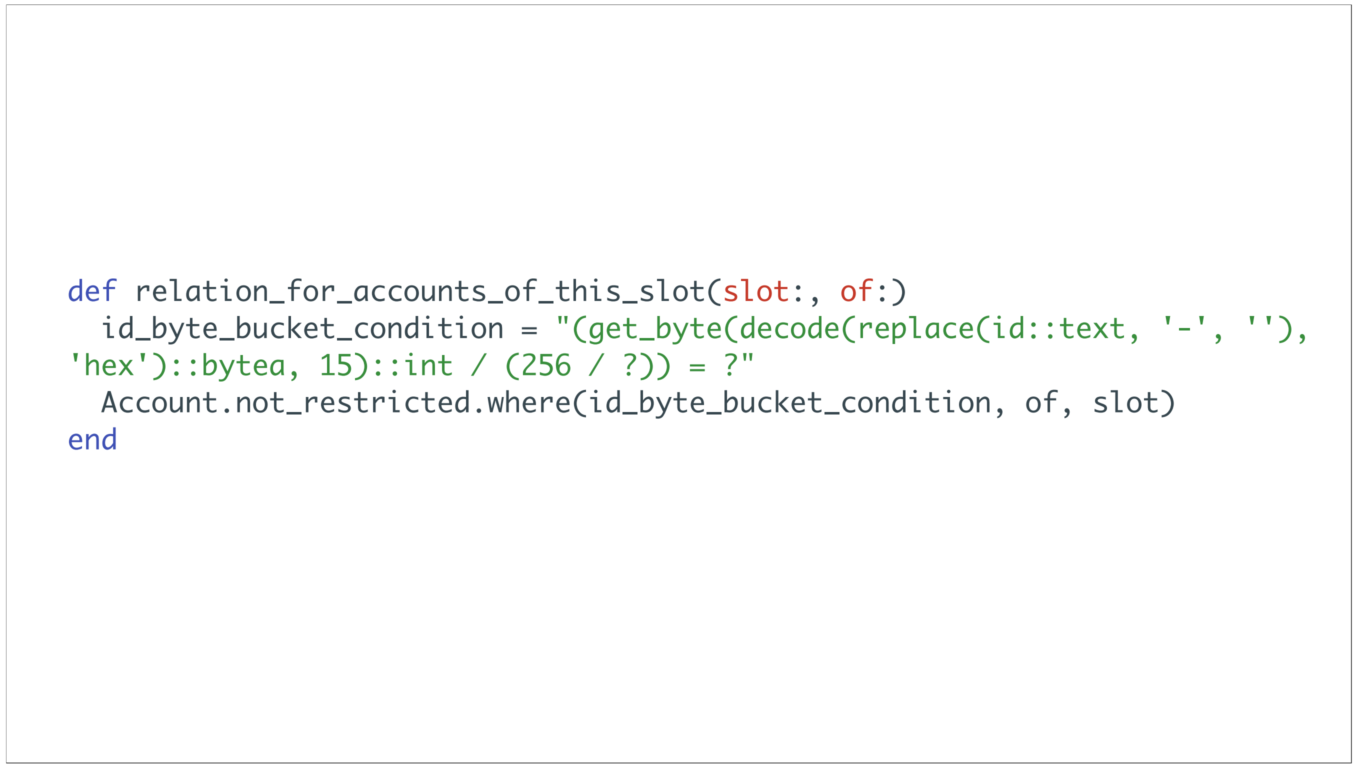
Here. We want to run jobs of these different kinds in such a way that they be "equidistant" from one another. This scales up - if we wanted to run 4 jobs that should not collide, we would place another one on the circle.

And now, for the last bit. Remember this scope we had in one of our jobs there?



```
accounts_rel_for_sync = relation_for_accounts_of_this_slot(slot:, of:)
```

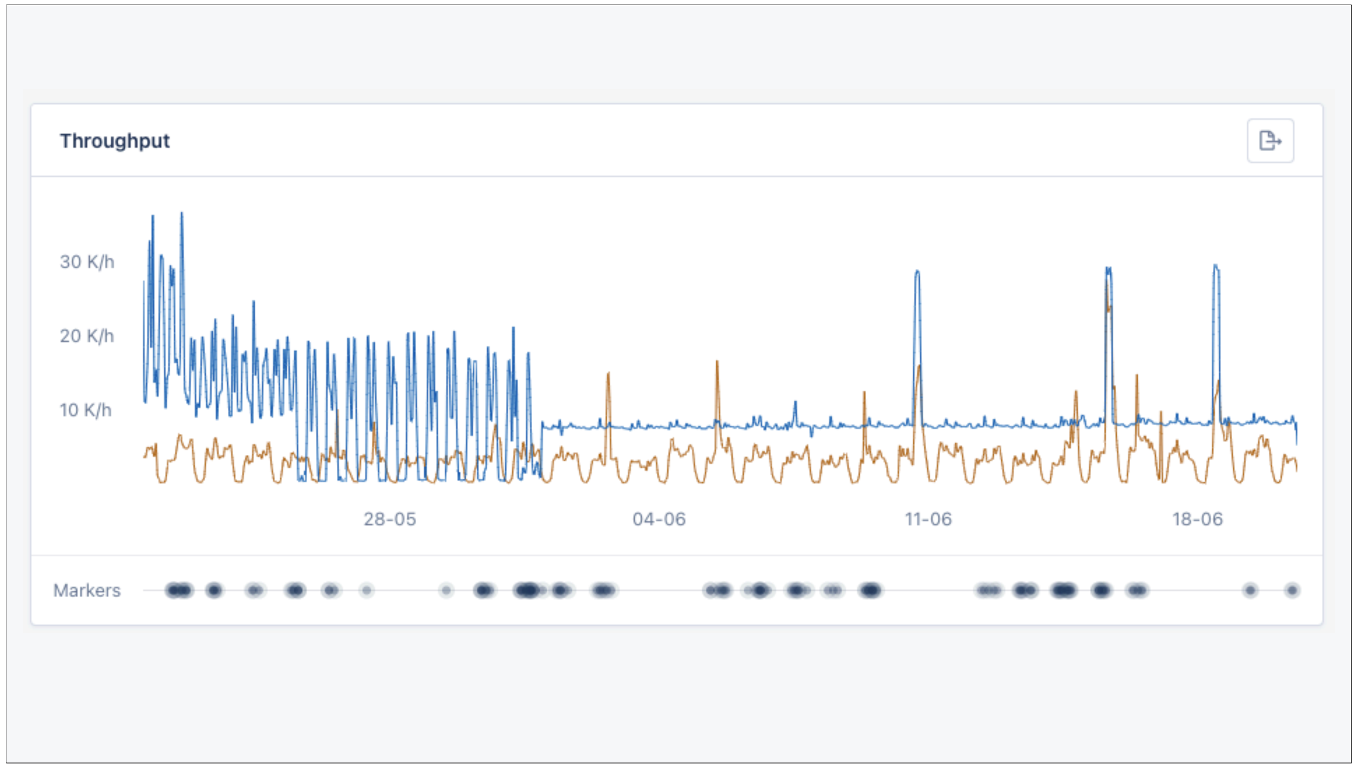
And now, for the last bit. Remember this scope we had in one of our jobs there? It is a neat method, but how does it actually work? Well, it is a bit of a Postgres hack:



```ruby
def relation_for_accounts_of_this_slot(slot:, of:)
  id_byte_bucket_condition = "(get_byte(decode(replace(id::text, '-', ''),
'hex')::bytea, 15)::int / (256 / ?)) = ?"
  Account.not_restricted.where(id_byte_bucket_condition, of, slot)
end
```

Let's unpack this for a second. First, we take the raw UUID typed Postgres column and interpret it as text. Then, we remove the dashes from it. Then we treat it as a hexadecimal value and decode it from hex, into raw bytes. Then we grab the last byte of that value, which we interpret as an integer. Note that we grab the last byte - with most UUID versions the random component is at the end, especially so with UUIDv7. The byte is then our"bucket identifier" amongst 256 possible buckets. We then check whether it matches the slot number when divided. This way we tell Postgres to give us just the accounts matching this particular slot out of that many other slots.

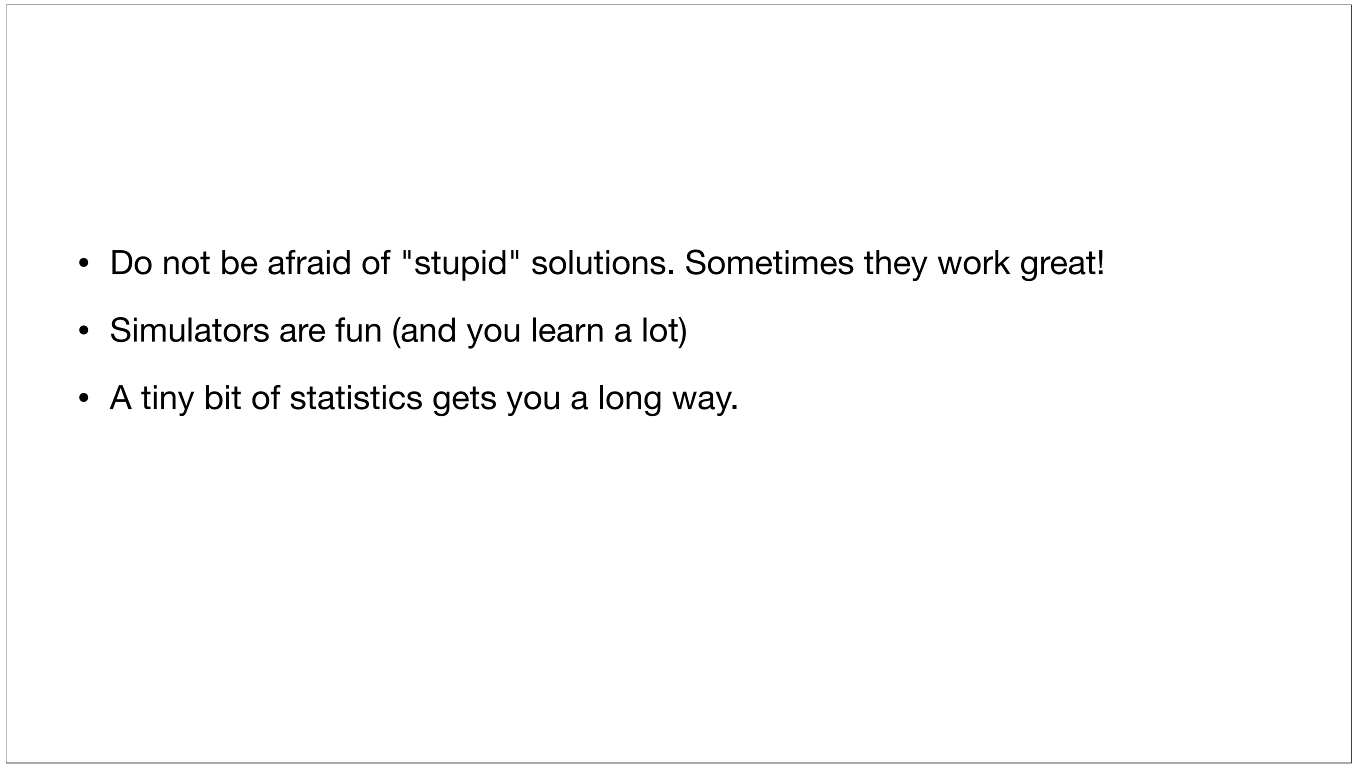
And so, in the end, we went to throughput like this:



In the middle is where the scheduler got deployed.

This has tremendous advantages. You can pre-size your capacity, There will be much less autoscaler flapping - you start a certain number of workers when your interval begins, and they keep running until the interval ends. You don't have CPU load spikes on your database, and you reduce contention on key resources.

Now, there is an important point here. A lot of articles on the topic say that "even spread" is the least favorable scheduling strategy, because you will have users who are "10x the size" of any other - your "jumbo users". They will have "10 times as much of everything" as others, and therefore executing their tasks will take way longer. And this is true. For us - the distribution was not as wild, first of all. Second, there is a concept you can introduce here and that would be of "weight", or "cost" of a specific user. You can then spread them over your schedule accounting not only for their slot, but also accounting for how long their operations are likely to take. If you have jumbo users, you usually will know who they are. By the same token, if you do have them - maybe you are already so far as to place them on a separate queue, separate infrastructure, separate shard... and sending them a bottle of champagne.



- Do not be afraid of "stupid" solutions. Sometimes they work great!
- Simulators are fun (and you learn a lot)
- A tiny bit of statistics gets you a long way.

* Do not be afraid of "stupid" solutions. Sometimes they work great!
* Simulators are fun (and you learn a lot)
* A tiny bit of statistics gets you a long way.

Thank you!

https://cheddar.me
https://github.com/cheddar-me

https://blog.julik.nl
https://x.com/julikt