DEVELOPER-FRIENDLY TASKQUEUES

WHAT WE LEARNED BUILDING MRQ

& WHAT YOU SHOULD ASK YOURSELF BEFORE CHOOSING ONE

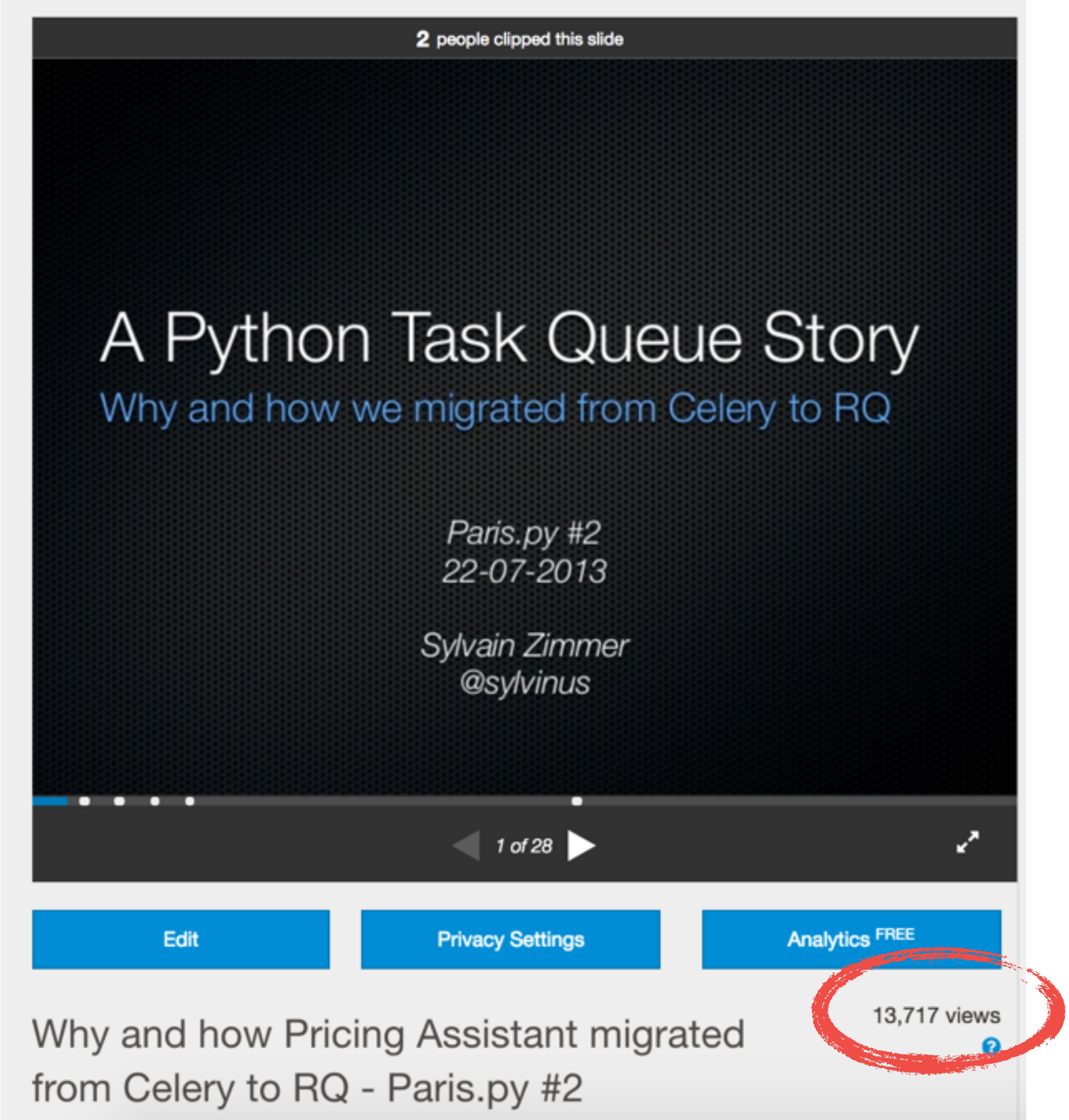
Sylvain Zimmer / @sylvinus

PyParis 2017

/usr/bin/whoami

- (SpaceX nerd)
- Founder, dotConferences
- CTO Pricing Assistant
- Co-organizer Paris.py meetup
- User of Python taskqueues for 10+ years
- Main contributor of MRQ

4 years ago...



2 people clipped this slide

A Python Task Queue Story

Why and how we migrated from Celery to RQ

Paris.py #2
22-07-2013

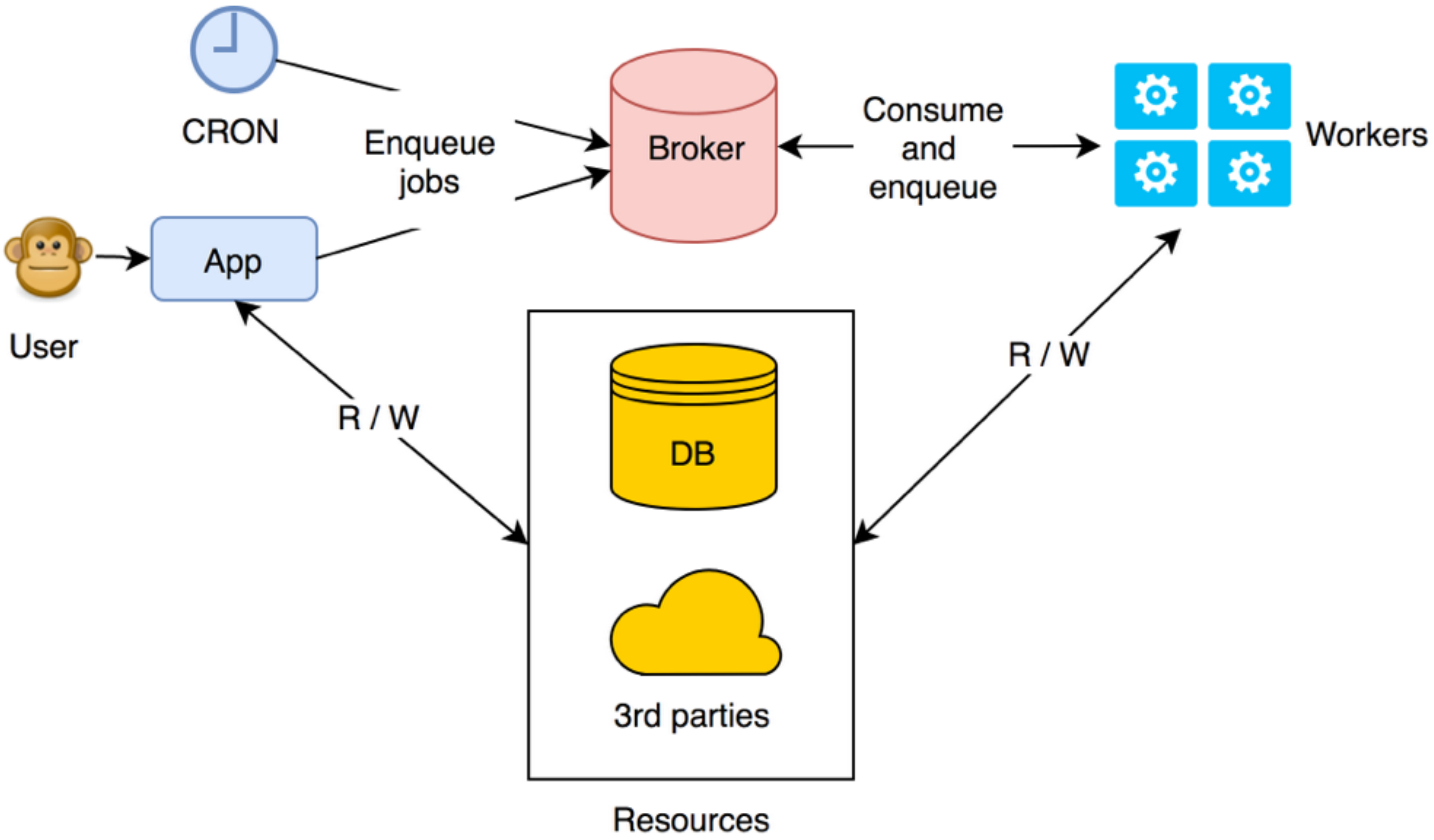
Sylvain Zimmer
@sylvinus

13,717 views

Why and how Pricing Assistant migrated from Celery to RQ - Paris.py #2

OMG !1!!!

TASKQUEUE FUNDAMENTALS



Credit: Adrien Di Pasquale

A typical job/task

```
def send_an_email(email_type, user):
  html = template(email_type, user)
  status = email.send(html, user["email"])
```

KERNEL PANIC

```
  metrics.send("email_%s" % status, 1)
  return status
```

Task properties

Re-entrant < Idempotent < Nullipotent

- Safe to interrupt and then retry

```
def reentrant(a):
    value = a + random()
    db.insert(value)
```

- Safe to call multiple times
- Result will be the same

```
def idempotent(key, value):
    db.update(key, value)
```

- Free of side-effects

```
def nullipotent(a):
    return a ** 2
```

Other task properties & best practices

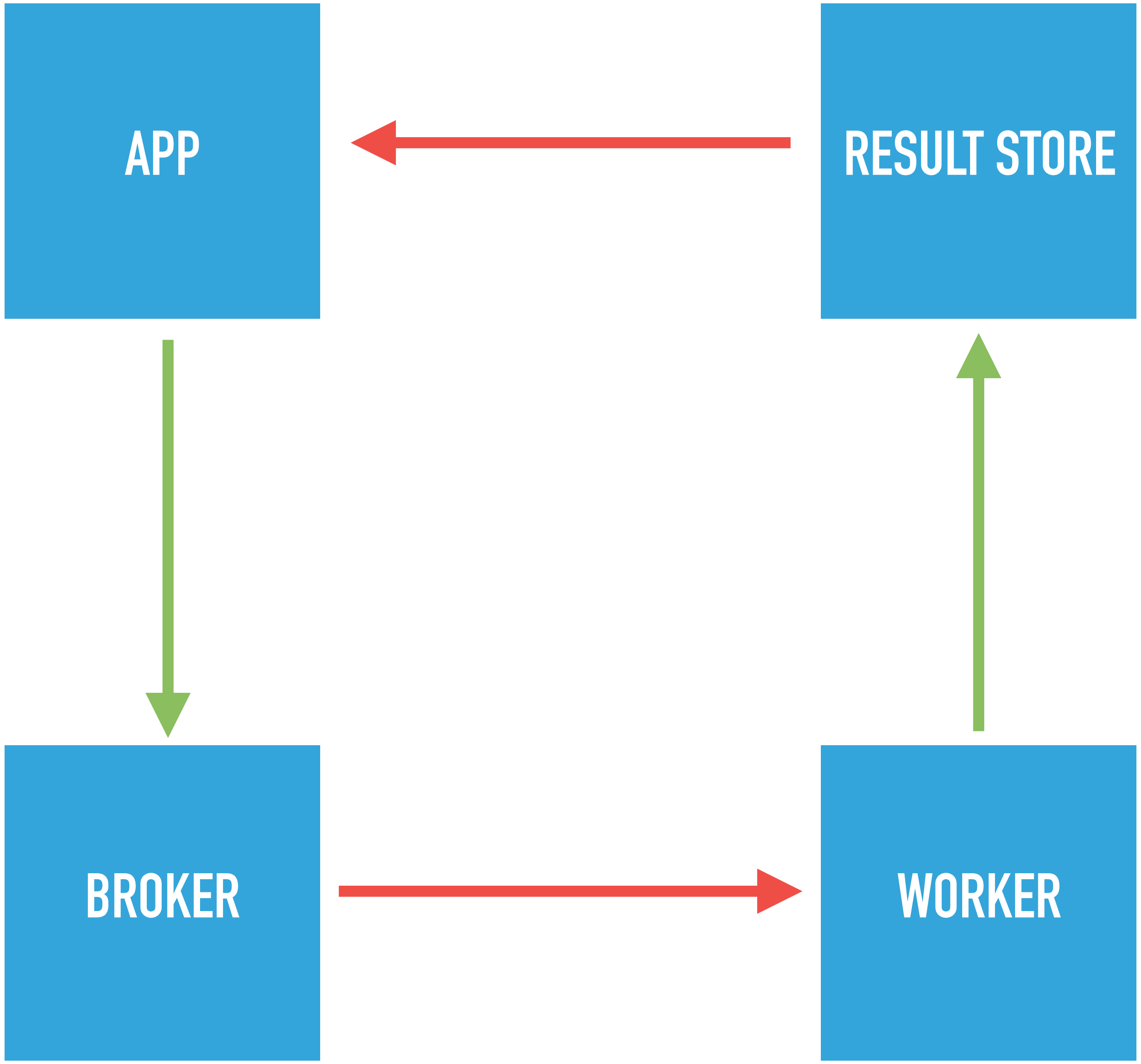
- Serializable args, serializable result
- Args validation / documentation
- Least args possible
- Canonical path vs. registration
- Concurrent safety
- Statuses

Coroutines vs. Threads vs. Processes

- IO-bound tasks vs. CPU-bound tasks
- Threads offer few benefits for a Python worker (GIL)
- Coroutines/Greenlets are ideal for IO-bound tasks
- Processes are required for CPU-bound tasks
- If you have heterogenous tasks, your TQ should support both!

```
$ mrq-worker --greenlets 25 --processes 4
```

Performance: latency & throughput



OPS & TOOLING

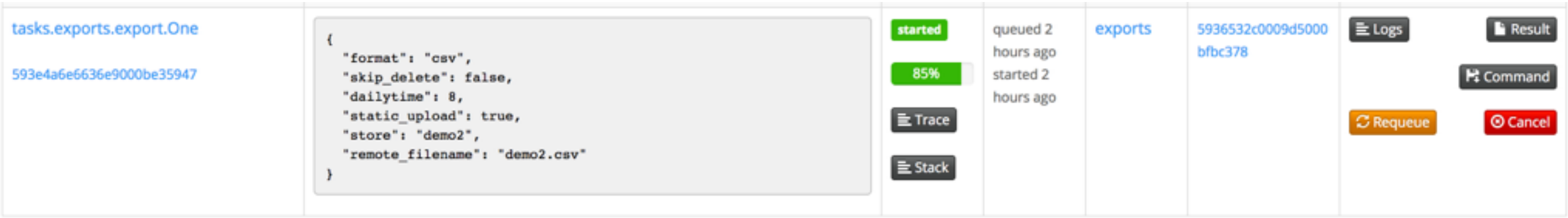
MURPHY'S LAW DOESN'T MEAN
THAT SOMETHING BAD WILL HAPPEN.
IT MEANS THAT WHATEVER CAN HAPPEN, **WILL HAPPEN.**

Errors

- Exception handlers
- Timeouts
- Retry rules
- Sentry & friends
- gevent: test your tracebacks!
- Priorities
- Human process to manage failed tasks!

Task visibility

- Tasks by status, path, worker, ...
- Tracebacks & current stack
- Logs
- Timing info
- Cancel / Kill / Move tasks
- Progress



tasks.exports.export.One

593e4a6e6636e9000be35947

```
{
  "format": "csv",
  "skip_delete": false,
  "dailytime": 8,
  "static_upload": true,
  "store": "demo2",
  "remote_filename": "demo2.csv"
}
```

started

85%

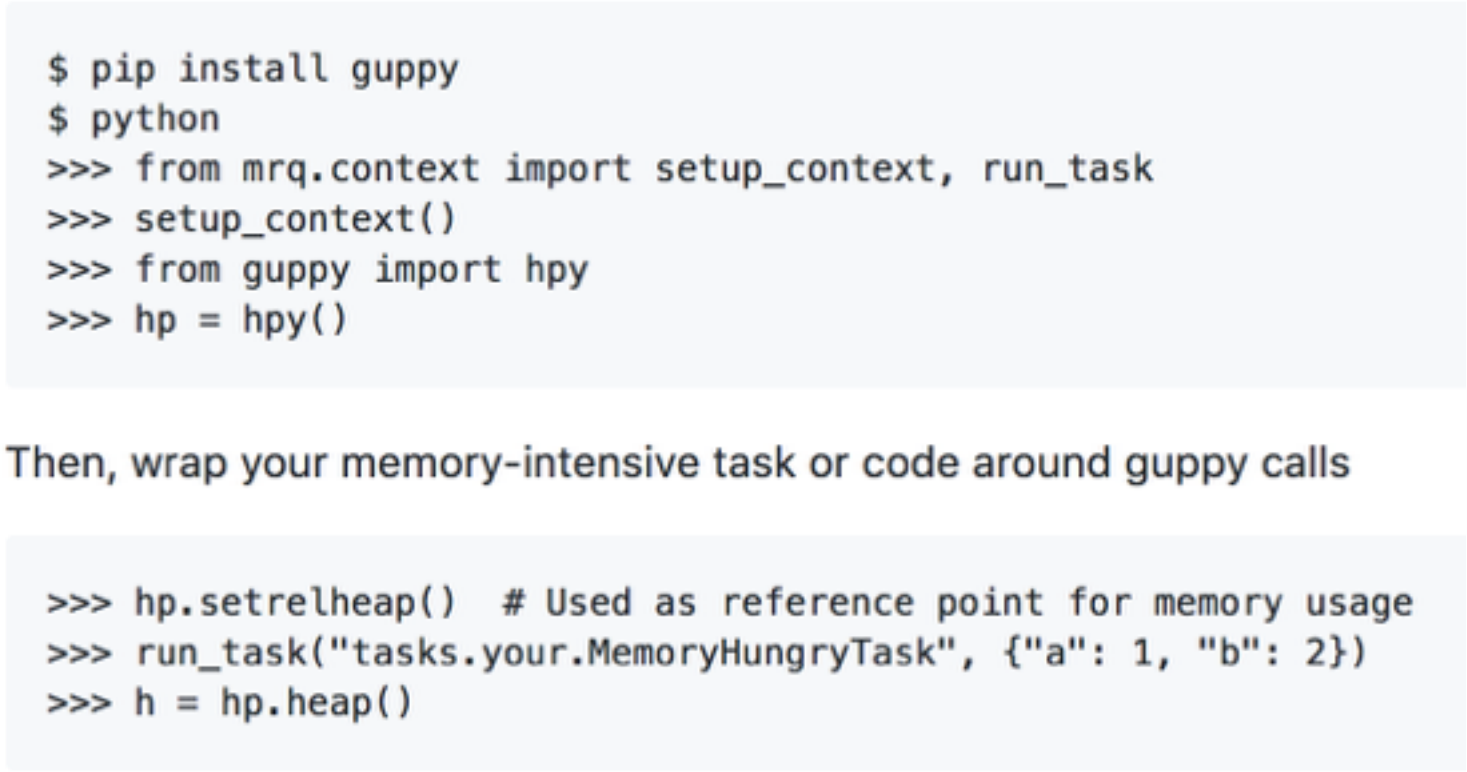
queued 2 hours ago started 2 hours ago

exports

5936532c0009d5000bfbc378

Memory leaks

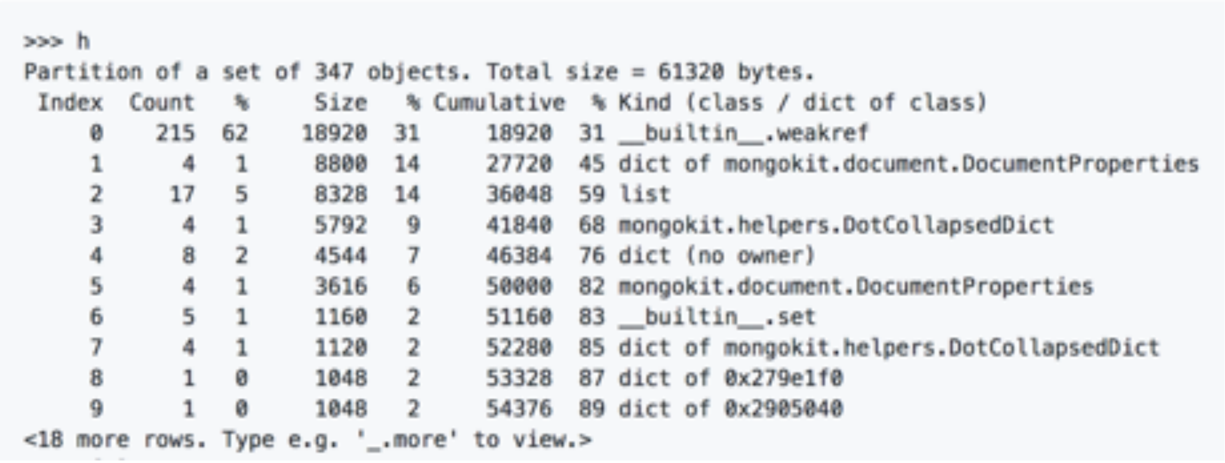
- Workers = long-running processes
- gevent makes debugging harder
- Watch out for global variables or mutable class attributes!
- Python's ecosystem is surprisingly poor in this area
- guppy, objgraph can usually help



```
$ pip install guppy
$ python
>>> from mrq.context import setup_context, run_task
>>> setup_context()
>>> from guppy import hpy
>>> hp = hpy()
```

Then, wrap your memory-intensive task or code around guppy calls

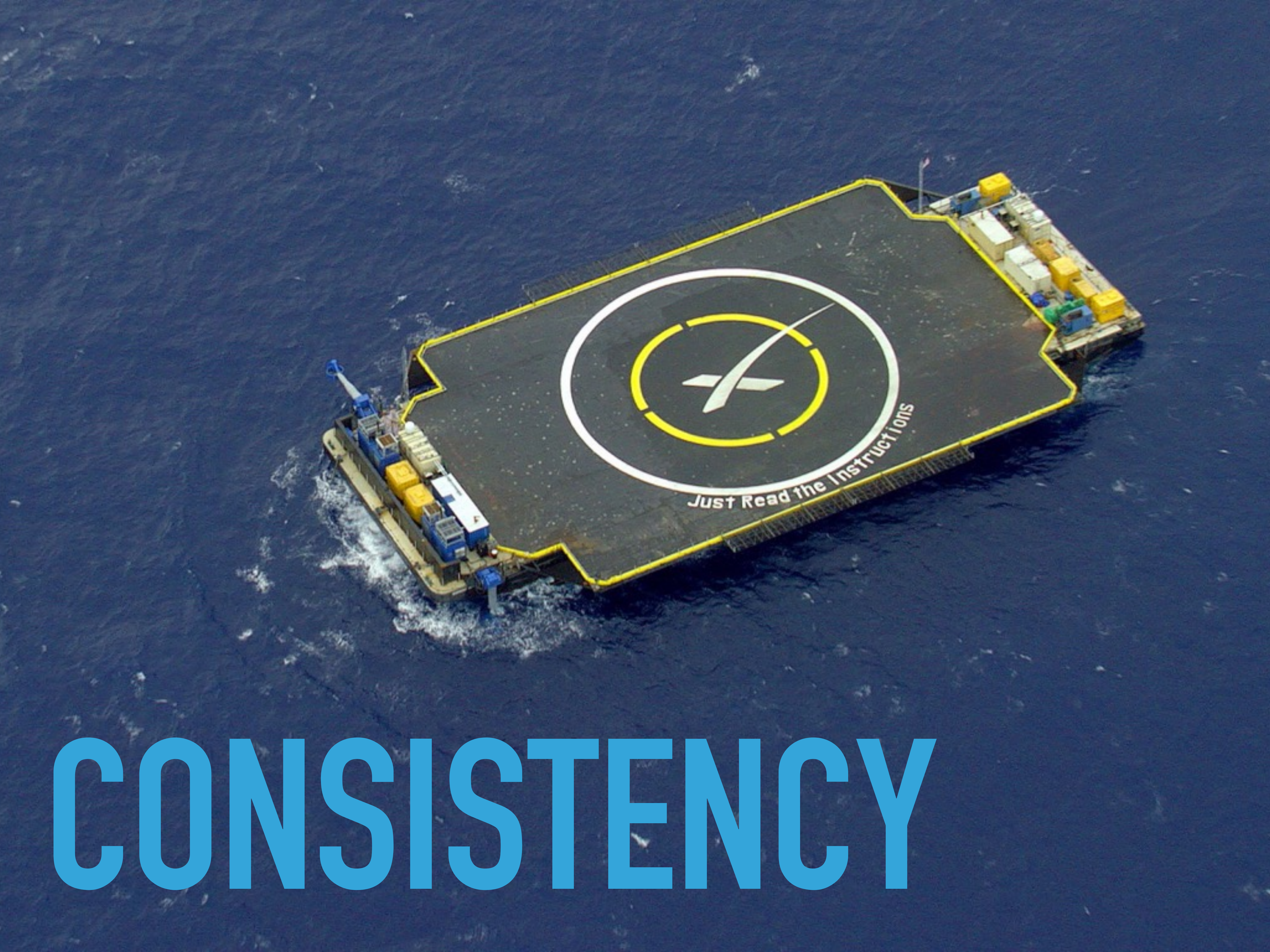
```
>>> hp.setrelheap()  # Used as reference point for memory usage
>>> run_task("tasks.your.MemoryHungryTask", {"a": 1, "b": 2})
>>> h = hp.heap()
```



```
>>> h
Partition of a set of 347 objects. Total size = 61320 bytes.
 Index  Count   %     Size   % Cumulative  % Kind (class / dict of class)
     0    215  62    18920  31      18920  31 __builtin__.weakref
     1      4   1     8800  14      27720  45 dict of mongokit.document.DocumentProperties
     2     17   5     8328  14      36048  59 list
     3      4   1     5792   9      41840  68 mongokit.helpers.DotCollapsedDict
     4      8   2     4544   7      46384  76 dict (no owner)
     5      4   1     3616   6      50000  82 mongokit.document.DocumentProperties
     6      5   1     1160   2      51160  83 __builtin__.set
     7      4   1     1120   2      52280  85 dict of mongokit.helpers.DotCollapsedDict
     8      1   0     1048   2      53328  87 dict of 0x279e1f0
     9      1   0     1048   2      54376  89 dict of 0x2905040
<18 more rows. Type e.g. '_.more' to view.>
```

Misc tools

- Scheduler
- Command-line runner, e.g. `mrq-run tasks.myTask {"a": 1}`
- Autoscaling
- Profiler



CONSISTENCY

Consistency guarantees

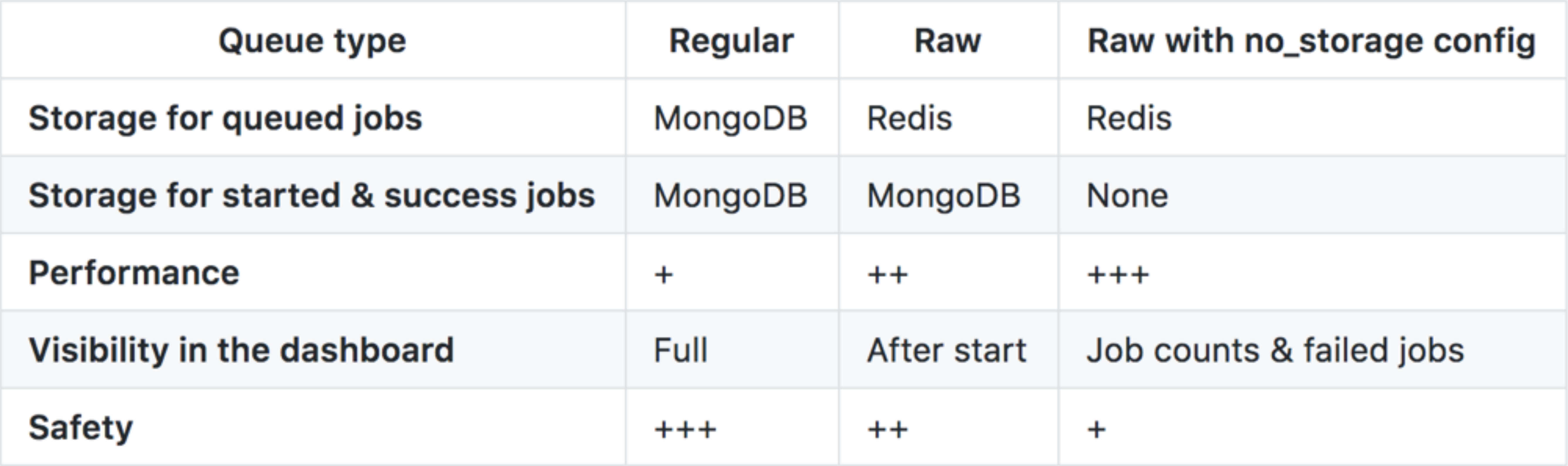
- At least once vs. At most once vs. Exactly once
- Ordering
- Critical operations:
  - Queueing
  - Marking tasks as started
  - Timeouts & retries

Types of brokers

- **Specialized message queues** (RabbitMQ, SNS, Kafka, …)
  - Performance, complexity, poor visibility
- **In-memory data stores** (Redis, …)
  - Performance, simplicity, harder to scale
- **Regular databases** (MongoDB, PostgreSQL, …)
  - Often enough for the job!

At the heart of the broker

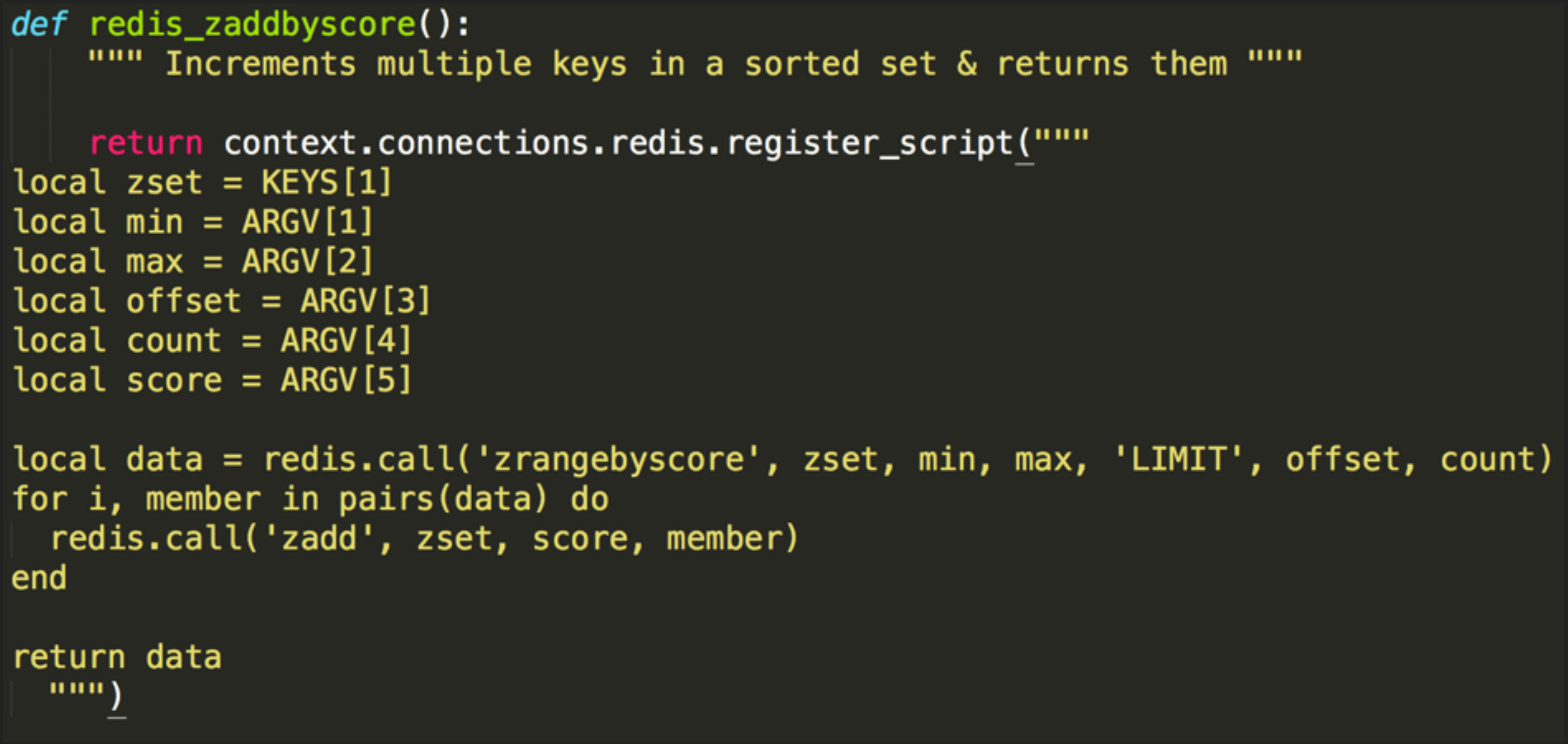
- Atomic update from "queued" to "started"
- MRQ with MongoDB broker: find_one_and_update()
- MRQ with Redis broker: Pushback in a ZSET



Queue type	Regular	Raw	Raw with no_storage config
Storage for queued jobs	MongoDB	Redis	Redis
Storage for started & success jobs	MongoDB	MongoDB	None
Performance	+	++	+++
Visibility in the dashboard	Full	After start	Job counts & failed jobs
Safety	+++	++	+

ZSETs in Redis

- Sorted sets with O(log(N)) scalability
- set/get by key, order by key, lookups by key or value
- Very interesting properties for task queues: Unicity, Ordering, Atomicity of updates, Performance
- MRQ's "Pushback" model:
  - Queue with key=timestamp
  - Unqueue by fetching key range & setting new keys in the future
  - After completion the task adjusts or removes the key



```
def redis_zaddbyscore():
    """ Increments multiple keys in a sorted set & returns them """

    return context.connections.redis.register_script("""
local zset = KEYS[1]
local min = ARGV[1]
local max = ARGV[2]
local offset = ARGV[3]
local count = ARGV[4]
local score = ARGV[5]

local data = redis.call('zrangebyscore', zset, min, max, 'LIMIT', offset, count)
for i, member in pairs(data) do
  redis.call('zadd', zset, score, member)
end

return data
  """)
```

Consistency guarantees

- Must be thought of for the whole system, not just the broker!
- Brokers can be misused or misconfigured
- The workers can drop tasks if they want to ;-)
- Consistency starts at queueing time!

TIME TO CHOOSE!

Think hard about what you need

- Will your taskqueue be the foundation of your architecture, or is it just a side project?
- What performance do you need? (IO vs. CPU, latency, ...)
- What level of visibility and control do you need on queued & running tasks?
- Can workers terminate abruptly? Lots of design consequences!
- What language interop do you need?

And then all the usual questions...

- Is it supported by a lively community?
- License
- Documentation
- Future plans

Which one to pick?

- **Celery:** High performance, large community, very complex, major upgrades painful
- **RQ:** Extremely simple to understand, low performance
- **MRQ:** Adjust task visibility vs. performance, simple to understand, 1.0 soon
- Lots of other valid options! Just be sure to ask yourself the right questions ;-)

REMINDER

BE GRATEFUL FOR THE OSS YOU USE!

PricingAssistant

Hiring Pythonistas!

Photo credits: https://www.flickr.com/photos/spacex/

THANKS!

QUESTIONS?