Offline Processing on App Engine: A Look Ahead

Brett Slatkin
May 28th, 2009
Agenda

- The Task Queue API
  - Tasks, Webhooks
  - Push versus Pull, Performance
  - Idempotence, Queues, Throttling
  - Names, ETA

- Example applications (interspersed)
  - Sending email
  - Schema migration
  - Write-behind cache

- The future
Moderator and Feedback

● Moderator questions
  ○ http://tinyurl.com/offlinetalk

● Immediate feedback about this presentation
  ○ http://haveasec.com/io/
Motivation
Motivation

● Google App Engine is great for web apps
  ○ Request-based, database backed apps

● Background and batch processing are highly requested features
  ○ Cron good for periodic jobs, but not enough
  ○ Would enable a range of new applications to be built entirely on App Engine
Motivation (2)

- Why do background processing?
  - Do work continuously without user requests
  - Incrementally process data, compute results
  - Smooth out load patterns, lower user latency

- A new style of computation on App Engine
Introduction
Overview

- New API for App Engine: Task Queue
- Part of App Engine Labs
  - API may change until it's graduated from Labs
  - Not yet specified how we will enable billing
- Not released; should launch in a couple weeks
- Live for demoing today with working code
What is a task queue?

- Simple idea in general:
  1. Describe the work you want to do now
  2. Save the description somewhere
  3. Have something else execute the work later

- Work executed in the order received (best-effort FIFO)
- If execution fails, work will be retried until successful

- Smallest example:

```python
taskqueue.add(description_of_work)
```
What is a task queue? (2)

● Benefits
  o Asynchronous
    ■ Why do work now when we can do it later?
  o Low-latency (for users)
    ■ Tasks are light-weight; ~3x faster than Datastore
  o Reliable
    ■ Once written, a task will eventually complete
  o Scalable
    ■ Storage of new tasks has no contention
    ■ Parallelizable with multiple workers

● Many features can extend this basic concept
What is a task queue, historically?

- UNIX had `at` and `batch` commands
- People use cron jobs and flat files
  - Append to a DB or file with work to do
  - Cron job periodically consumes the whole queue
- Lots of reliability and scalability issues here
What other task queue systems exist?

- There are many task-queue-like systems out there
  - *MQ, Amazon SQS, Azure queues, TheSchwartz, Twisted, Starling, beanstalkd, etc
  - Often conflated with publish-subscribe messaging
- Queueing systems maximize data throughput
  - Routers, data pipelines
  - Fully saturate network, CPU, disk
- Pub-sub systems maximize transactions, decoupling
  - Large numbers of small transactions per second
  - One-to-many fan-out with changing receivers
  - Guaranteed ordering, filtering, two-phase commit
- Our new API implements queueing, not pub-sub
How do traditional task queues work?
How do traditional task queues work? (2)

- Polling has problems:
  - Worker sits in a loop polling the front of the queue
    - Not event driven; wasted work
  - Workers stay resident when there's no work to do
    - Wastes machine resources
  - Fixed number of workers
    - Admins must manually add more workers to keep up or queue will grow without bounds

- Limited optimization possible
  - Many systems fake a polling interface with something event-driven under the hood
  - Long-lived, hanging connections
How does our Task Queue API work?

- We **push** tasks to your app; no polling necessary

- HTTP Web hooks!
  - RESTful, push-based interface for doing work
  - Concept used outside Google and App Engine
  - Many of our upcoming APIs use this style

- Tasks as web hooks
  - Task is just an HTTP request (URL, body, etc)
  - Enqueue and we send your app the request later
  - If the web hook returns HTTP 200 OK, it's done
  - Any other response causes back-off and retries
Concrete example: Mail sending queue

class MailWorker(webapp.RequestHandler):
    def post(self):
        mail.send_mail(
            'me@example.com',
            self.request.get('to'),
            self.request.get('subject'),
            self.request.get('body'))

    # To enqueue a task:
    taskqueue.add(url='/work/mail', params=dict(
        to='foo@example.com',
        subject='Hello',
        body='this is a message!'))
Concrete example: Mail sending queue demo
How does our Task Queue API work? (2)
How does our Task Queue API work? (3)
How does our Task Queue API work? (4)

- Worker threads added depending on work-load
  - Max number of threads depends on throughput
  - High maximum rate limits for safety

- Integrated into admin console as normal requests
  - Application and request logs searchable
  - Dashboard statistics and error-rate monitoring
  - Graphs include offline work
Working with Tasks: Idempotence

- Important for tasks to be idempotent
- Run the same task repeatedly without harmful effects
  - Or acceptable effects (e.g., duplicate emails)
- Necessary because failure may happen at any time
- Tasks will be retried until success
- Possible for a task to spuriously run twice even without server failures!

- It is your responsibility as the application developer to ensure idempotence of tasks
Working with Queues

- Each task added to a single Queue for execution
  - Multiple queues allowed per application
- Queues provide isolation and separation of tasks
- Configure how each queue is throttled
- Example queue.yaml

```yaml
queue:
- name: mail_queue
  rate: 2000/d
- name: speedy_queue
  rate: 5/s
```
Working with Queues (2)

- Why do you want to throttle?
  - Combine work periodically; execute in batches
  - Ensure stability of workload (CPU, bandwidth, $)
  - Not exceed maximum writes per second for a single entity group in Datastore
  - Not overload a partner site with web service calls
  - Not send too many emails at a time (to a single host, recipient, etc)

- Also enables prioritization of work
  - Tasks are only defined by the web hook request, not the queue they are on
  - Could have a queue for each level of service
Working with Queues (3)

- Many-to-many queue throttling
Concrete Example: Schema migration

- Without Task Queue API
  - Cron job slowly iterates through entities; migrates them; stores current entity location in memcache
  - Use remote_api or bulkloader to dump the whole dataset and reupload it
Concrete Example: Schema migration (2)

- With Task Queue API
  - Define handler to: query for next N entities; modify them; do a batch update; then enqueue a task to resume starting after the current position
  - Failures at any point will cause the task to be retried later, picking up exactly where it left off
Concrete Example: Schema migration (3)

class FirstUserKind(db.Model):
    name = db.StringProperty()

class SecondUserKind(db.Model):
    first = db.StringProperty()
    last = db.StringProperty()

def second_from_first(u):
    first, last = u.name.split(' ')
    return SecondUserKind(
        first=first, last=last)

def first_from_second(u):
    return FirstUserKind(
        name='%s %s' % (u.first, u.last))
Concrete Example: Schema migration (4)

```python
def migrate(kind, start):
    query = from_kind.all()
    if start:
        query.filter('__key__ >', db.Key(start))
    old = query.fetch(10)
    if not old:
        logging.info('All done!')
        return

    next_start = old[-1].key()
    new = [migrate(x) for x in old]
    db.put(new)
    db.delete(old)
    taskqueue.add(url='/worker/migration',
                   params=dict(start=next_start, kind=kind))
```
Concrete Example: Schema migration demo
Working with Tasks: ETA

- "Estimated time of arrival"
  - How long until a task should be executed
  - Different than "visibility timeouts" in other systems

- Useful for doing work in the relatively near future
  - More fine-grained, programmatic control than cron

- Example uses:
  - Periodically clear caches, flush buffers, report incremental results (via email, web service call), prioritize tasks
Working with Tasks: Names

- Each task may be given a unique name by the app.
  - When not supplied, an ID is auto-generated.
- After a named task completes, its "tombstone" will remain for a few days.
- Adding tasks with tombstoned names raises an error.
- Enforces "only-once" semantics.
  - Example: Migrate the schema for these entities once and **only** once.
Concrete Example: Write-behind cache
Concrete Example: Write-behind cache

- Minimizes writes with repeated cache flushing
  1. Write new data to the cache
  2. Periodically read cache and persist to disk

- Benefits
  - Database writes no longer increase as a function of overall user traffic!
  - 100 cache writes/sec becomes 1 DB write/sec

- Problem
  - Time window (small!) for loss of cached data
Concrete Example: Write-behind cache (2)
Concrete Example: Write-behind cache (3)

- Write-behind page-hit counter

- Page-hit counter without Task Queue API:
  - Sharded counters; relatively expensive
  - No control over write throughput
Concrete Example: Write-behind cache (3)

- Page-hit counter with Task Queue API:
  - All hits increment a counter in memcache
  - Copy memcache values to Datastore with a Task
  - Queue throttle used to limit max writes per second to the counter's entity group
  - Read from memcache or Datastore
Concrete Example: Write-behind cache (4)

class Counter(db.Model):
    count = db.IntegerProperty(indexed=False)

class CounterHandler(webapp.RequestHandler):
    def post(self):
        key = self.request.get('key')
        if (memcache.incr(key) is None and
            not memcache.add(key, 1)):
            memcache.incr(key)
        if memcache.add(key + '_dirty', 1):
            taskqueue.add(url='/worker',
            params={'key': key})
Concrete Example: Write-behind cache (5)

class PageHitWorker(webapp.RequestHandler):
    def post(self):
        key = self.request.get('key')
        memcache.delete(key + '_dirty'):
        value = memcache.get(key)
        if value is None:
            logging.error('Failure for %s', key)
            return
        Counter(key_name=key, count=value).put()
Concrete Example: Write-behind cache demo
The Future
The Future

- **Coming soon**
  - Release of Task Queue API in App Engine Labs
  - Python-only at first, Java soon after

- **Java support in the works**
  - Web hooks interface
  - JMS integration

- **More API features**
  - Queue management functions (e.g., flush)
  - Queue contents viewing in admin console
  - Notification of queue events (e.g., empty)
The Future

- Batch processing
  - Task API good for small datasets (< 100k rows)
  - More tools required for parallelization, high throughput processing of Datastore entities
  - Need rich features for aggregations, statistics

- Map Reduce
  - Plan to eventually support MapReduce abstraction
  - Need more tools: intermediary storage, sorting, etc
  - Want it to work with small (50k entities) and very large (> 1TB) datasets
Wrap-up
Wrap-up

- Use the Task Queue API! *(once it's launched =)*
- Make your existing app faster, lower latency
- Scale your app further with reduced costs
- Add new functionality you couldn't implement before
- Take advantage of web hooks for easy debugging
Questions?

- Add to Moderator
  - http://tinyurl.com/offlinetalk

- Give me feedback about this presentation!
  - http://haveasec.com/io/