Google™ 10 IO

Building high-throughput data pipelines on Google App Engine

Brett Slatkin
May 20th, 2010

View live notes and ask questions about this session on Google Wave

http://tinyurl.com/app-engine-pipelines

Me

http://onebigfluke.com

Agenda

- Intro
- Fan-out
- Transactional sequences
- Fan-in
- Bonus round
- Future directions

Intro

What are pipelines?

- Constant trickle/torrent of inputs and outputs
    - Assembly-line
- Optimize for end-to-end latency of input to output (~seconds)
- Minimize incremental cost
- Not lossy, eventually consistent, all inputs served

What are **NOT** pipelines?

- Offline systems like MapReduce
- Batch processing, report generation
- Outputs are from a snapshot of inputs
- Latency from input to output is ~hours

Example apps

- Pipelines
  - Email, Twitter, PubSubHubbub (routing)
  - Reddit, Digg (voting, agg)
  - CRM (~yeah, really)
- Not pipelines
  - Guestbook (flat)
  - Terasort (snapshot)
  - Chat (transient)
- Hybrid
  - YouTube, Vimeo (transcode)
  - Flickr, Picasa (face recog, tags)

Fan-out: Continuations

What is fan-out?

- One action leads to many others

- Datastore-based inbox systems (eg, microblogging)
- Send notification emails, XMPP, SMS, Channel API, APN
- Web service calls
- Enqueue more tasks

Example fan-out

- Update a party invitation, send an email to everyone

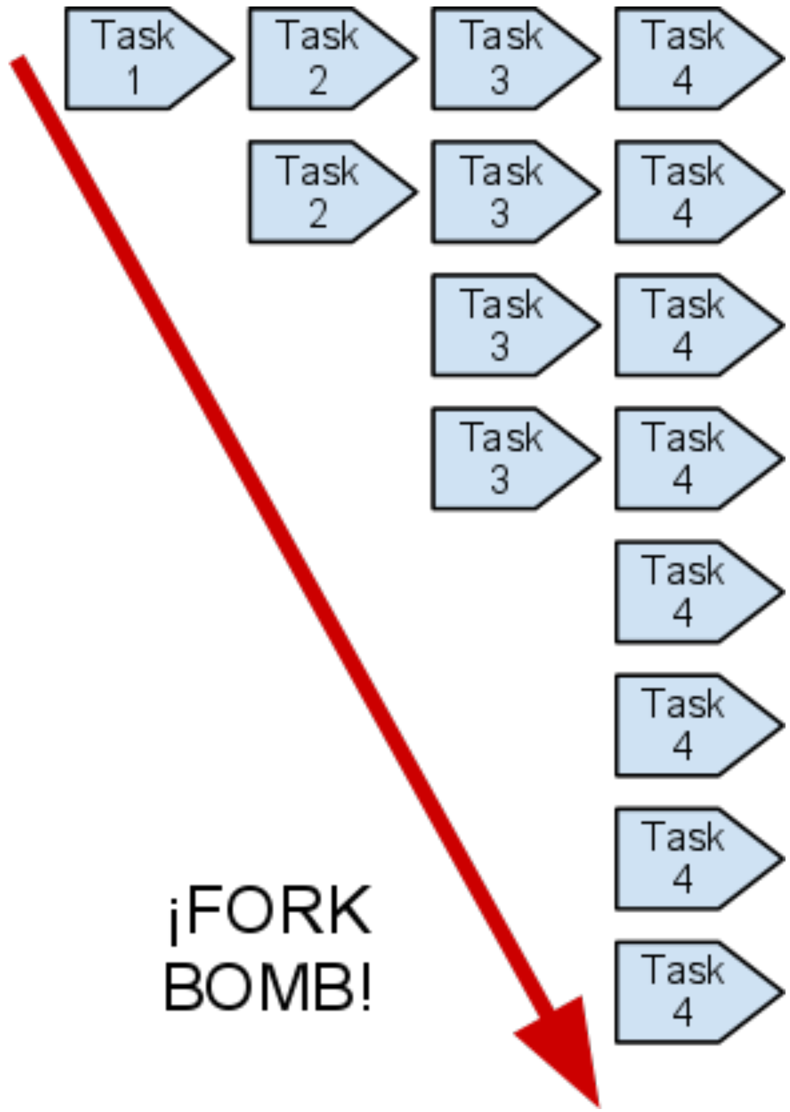
```
class Party(db.Model):
  when = db.DateTimeProperty()
  host = db.UserProperty()
class PartyGoer(db.Model):
  party = db.ReferenceProperty(Party)
  name = db.StringProperty()
  address = db.EmailProperty()
```

Continuation passing (naively)

```
class EmailHandler(webapp.RequestHandler):
  def post(self):
    my_party = self.request.get("party_key")
    cursor = self.request.get("cursor")
    query = PartyGoer.all().filter(
      "party =", db.Key(my_party))
    if cursor:
      query.with_cursor(cursor)
    goers = query.fetch(10)
    # Send some emails ...
    if len(goers) == 10:
      taskqueue.add(url='/work/email',
        params={'party_key': my_party,
                'cursor': query.cursor()})
```

Continuation passing (the wrong way)

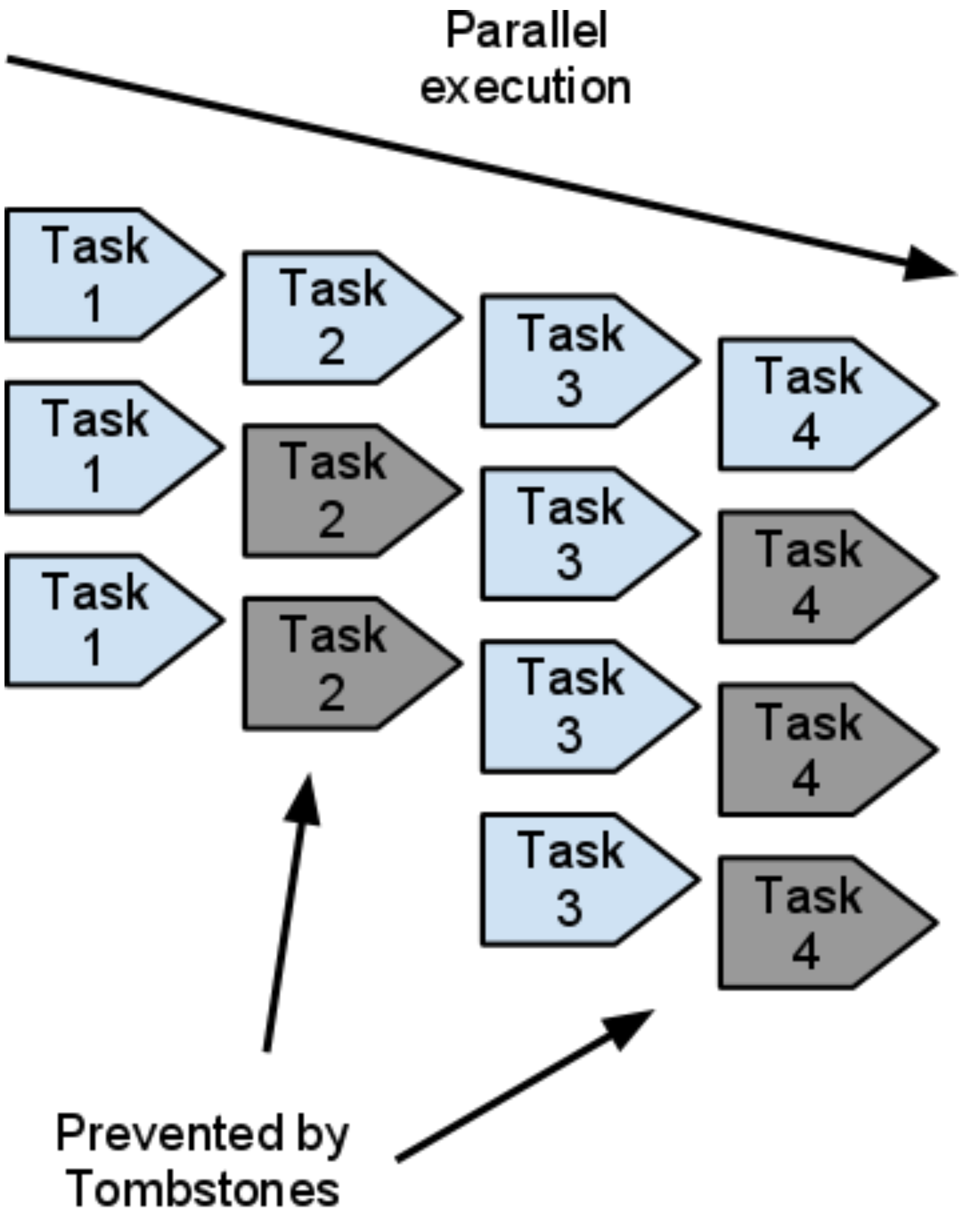
- Any failures and...



Continuation passing (the right way)

```
class EmailHandler(webapp.RequestHandler):
  def post(self):
    my_party = self.request.get("party_key")
    cursor = self.request.get("cursor")
    query = PartyGoer.all().filter(...)
    if cursor:
      query.with_cursor(cursor)
    goers = query.fetch(10)
    if len(goers) == 10:
      taskqueue.add(
        url='/work/email',
        params={'party_key': my_party,
                'cursor': query.cursor()},
        name=int(self.request.get('gen')) + 1)
    # Send some emails ...
```

Continuation passing (the right way)



Continuation passing benefits

- Failures and spurious retries are isolated
- Execute continued work in parallel

Continuation passing benefits 2

- Pairs well with asynchronous APIs
  - Async URLFetch in Python
    - Java support since 1.3.1 (February)
  - Async Datastore
    - Python: http://asynctools.googlecode.com
    - Java: http://twig-persist.googlecode.com

- Used in PubSubHubbub reference hub
  - **100-300** worker requests/sec constantly

Transactional sequences

What are transactional sequences?

- Datastore transactions and transactional tasks
- Guarantee that tasks run after data is written
  - Strong consistency when task is run

- Enables roll-forward semantics to fanned-out data
  - Build materialized views

What are materialized views good for?

- A query that's saved back into the database
  - Read-heavy, cached, secondary indexes
  - Eventually consistent views

- Incremental aggregations (commutative)
- Natural and left-joins
- Filter/query/sorting materialized results

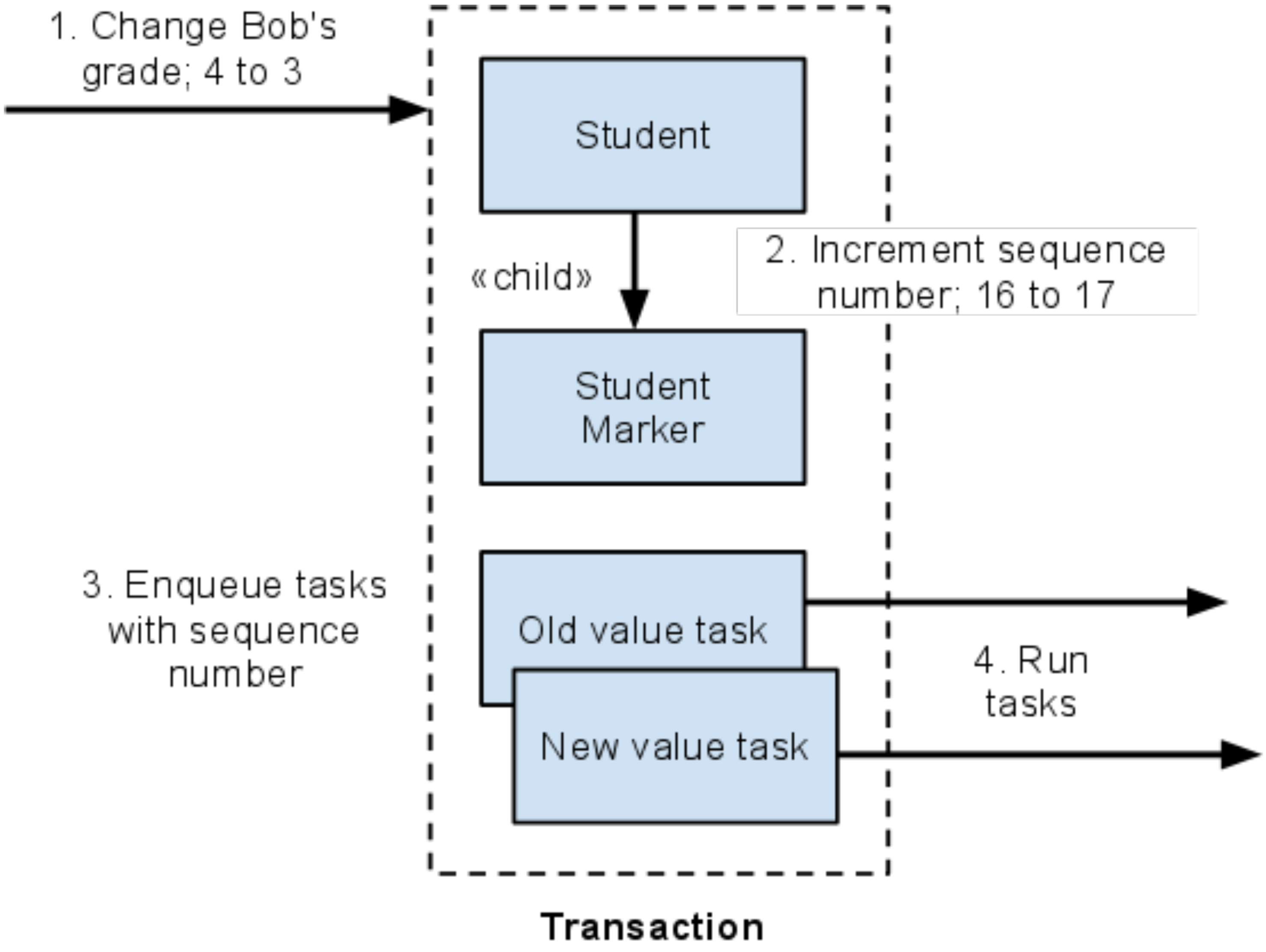
SQL Example: Students in school

```
Student | Grade
Bob     |     4
Daisy   |     3
...

SELECT grade, count(*) as count
    FROM Student
    GROUP BY grade;

grade   | count
3       |     5
4       |     7
...
```

App Engine Example: Students in school

```
class Student(db.Model):
  name = db.StringProperty()
  grade = db.IntegerProperty()


class Marker(db.Model):
  sequence = db.IntegerProperty(default=0)
  present = db.BooleanProperty()


class GroupCount(db.Model):
  grade = db.IntegerProperty()
  count = db.IntegerProperty(default=0)
```

Roll-forward semantics: Update source data



Update source data

```
def update(name, new, id):
  def txn():
    if id:
      student = Student.get_by_id(id)
      old, student.grade = student.grade, new
    else:
      student = Student(name=name, grade=new)
      student.put()  # Assign ID
      old, id = None, student.key().id()

    marker_key = db.Key.from_path(
        'Marker', id, parent=student.key())
    marker = db.get(marker_key)
    if not marker: marker = Marker(key=marker_key)
    marker.sequence += 1
    # continues on next slide
```

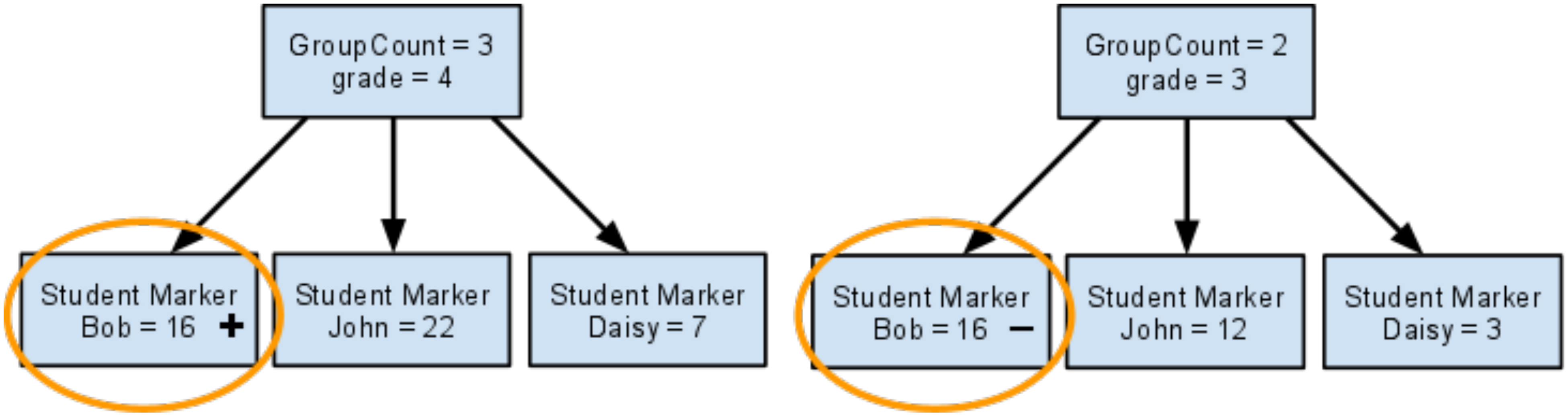
Update source data continued

```
    db.put([student, marker])
    taskqueue.Task(
      url='/work',
      params={'student_id': id, 'grade': new,
              'sequence': marker.sequence,
              'present': True}
    ).add(transactional=True)
    if old is not None:
      taskqueue.Task(
      url='/work',
      params={'student_id': id, 'grade': old,
              'sequence': marker.sequence,
              'present': False}
    ).add(transactional=True)
  db.run_in_transaction(txn)
```

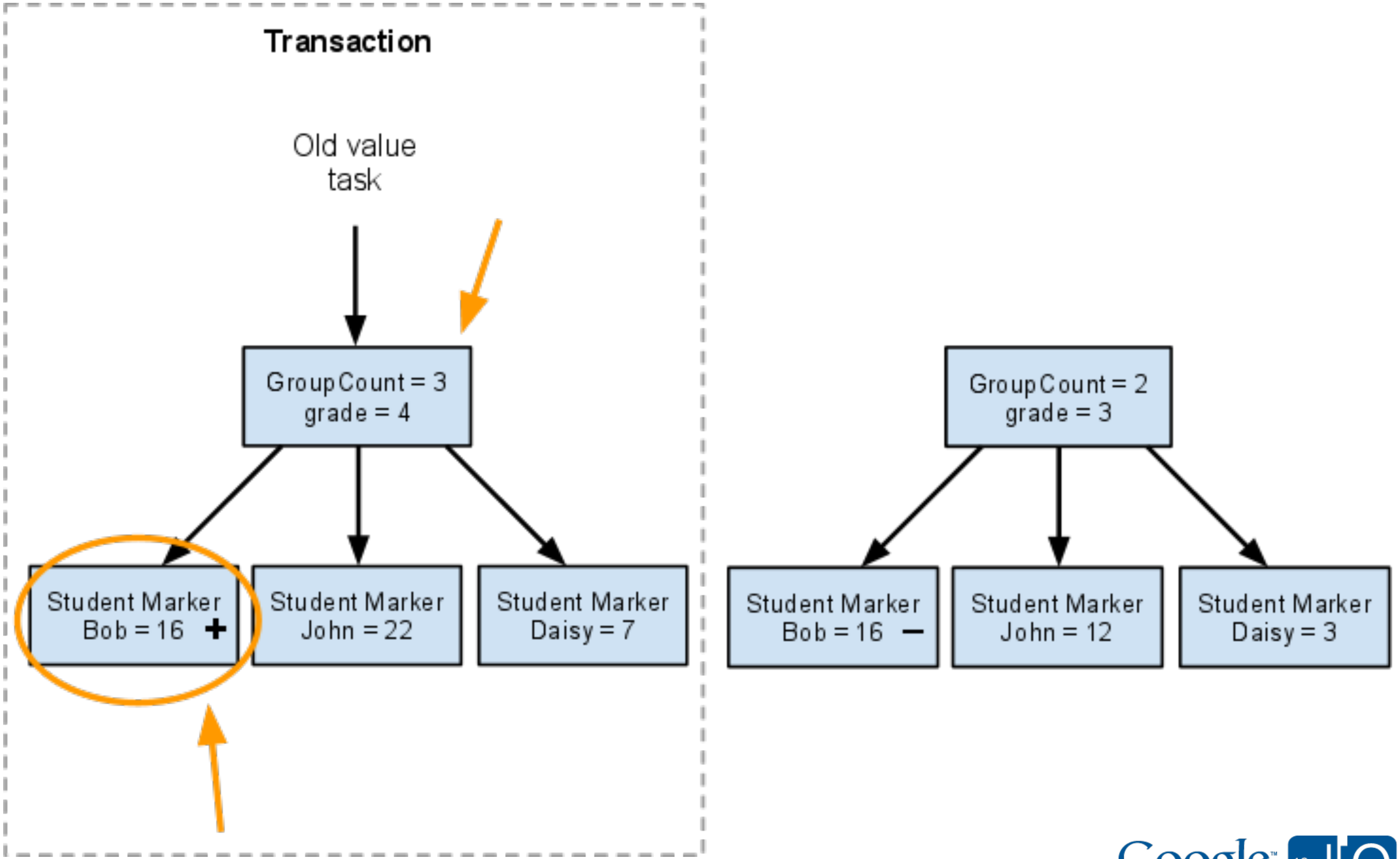
App Engine Example: Students in school

```
class Student(db.Model):
  name = db.StringProperty()
  grade = db.IntegerProperty()


class Marker(db.Model):
  sequence = db.IntegerProperty(default=0)
  present = db.BooleanProperty()


class GroupCount(db.Model):
  grade = db.IntegerProperty()
  count = db.IntegerProperty(default=0)
```

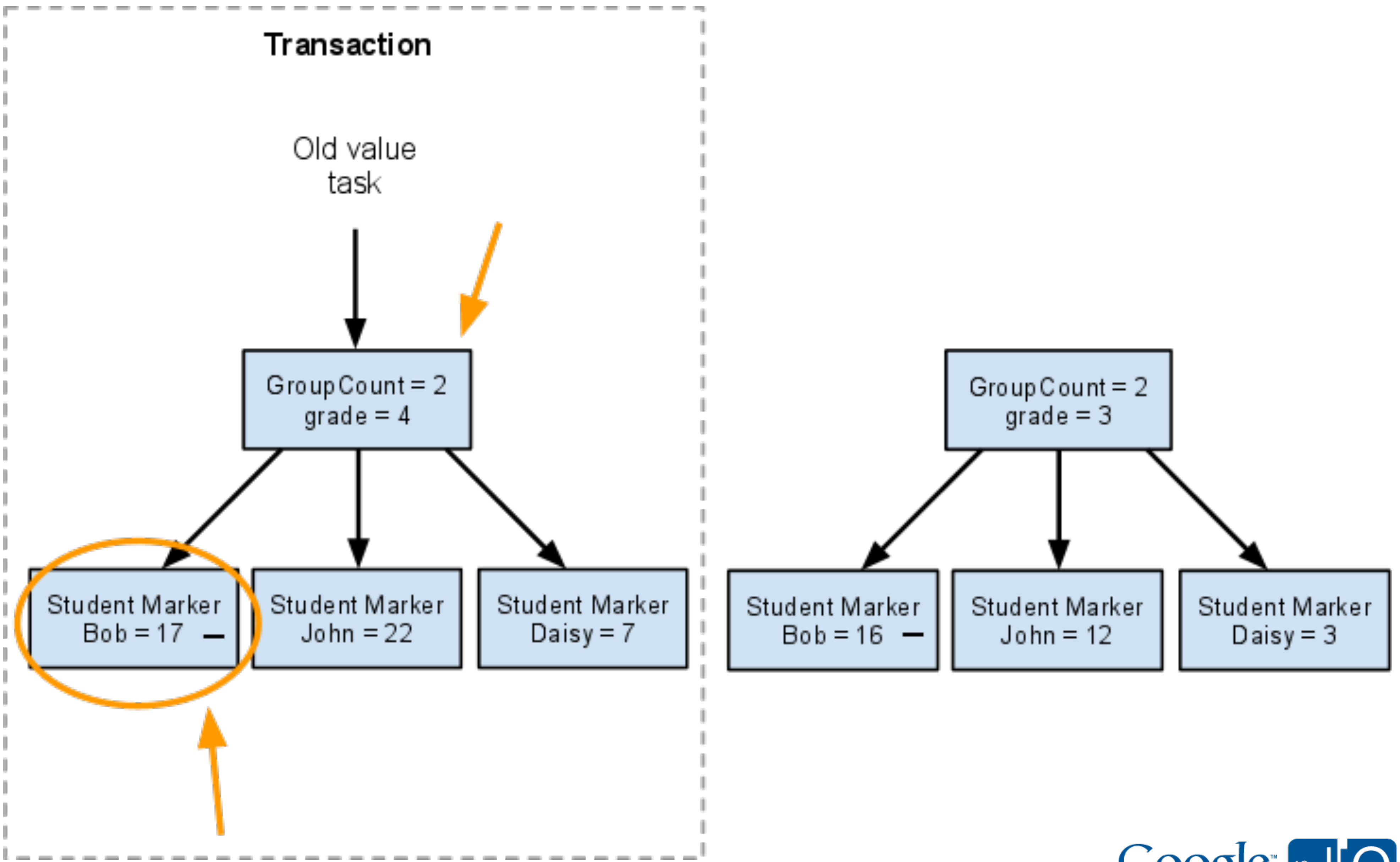
Roll-forward semantics: View initial state



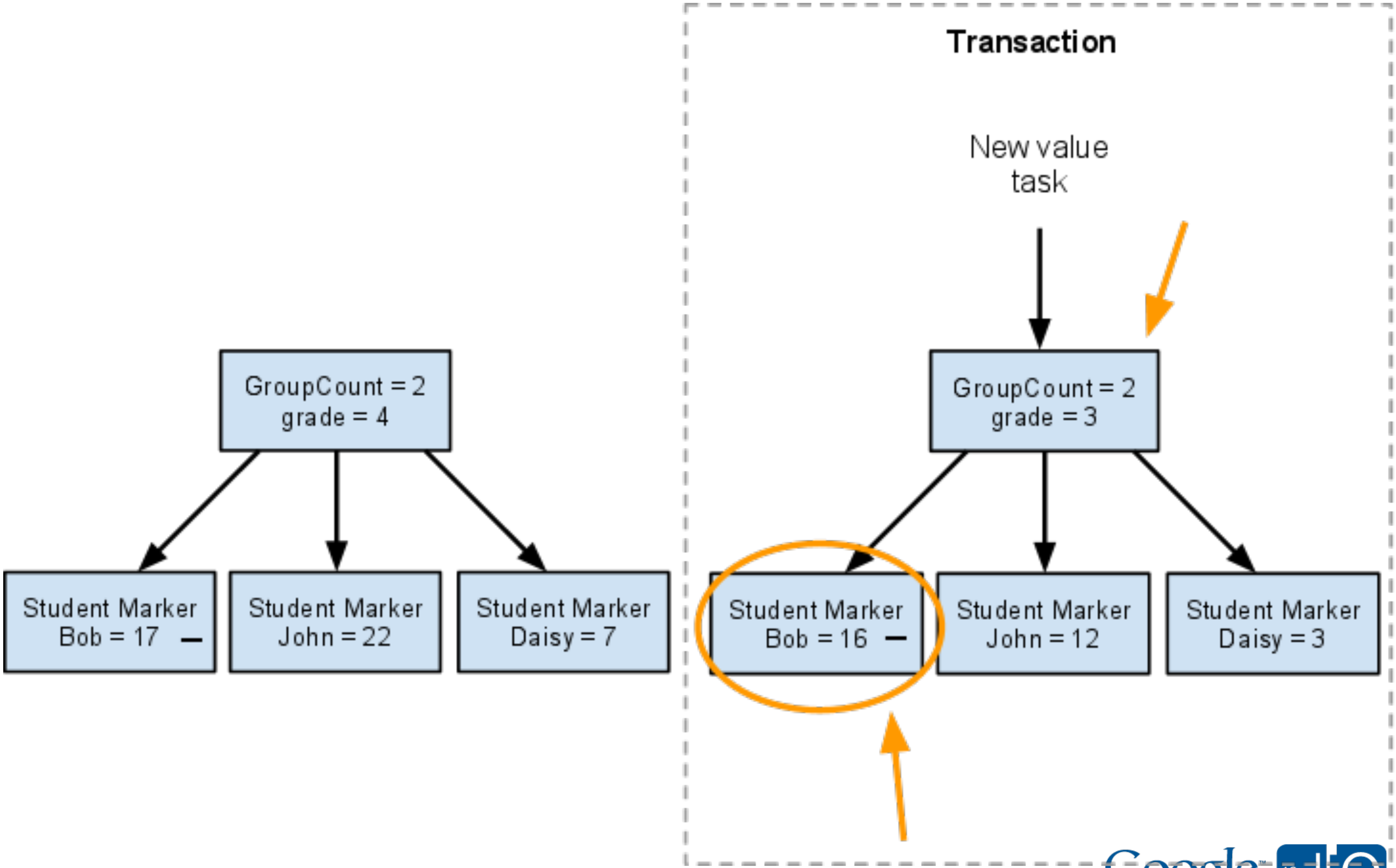
Roll-forward semantics: Update old value



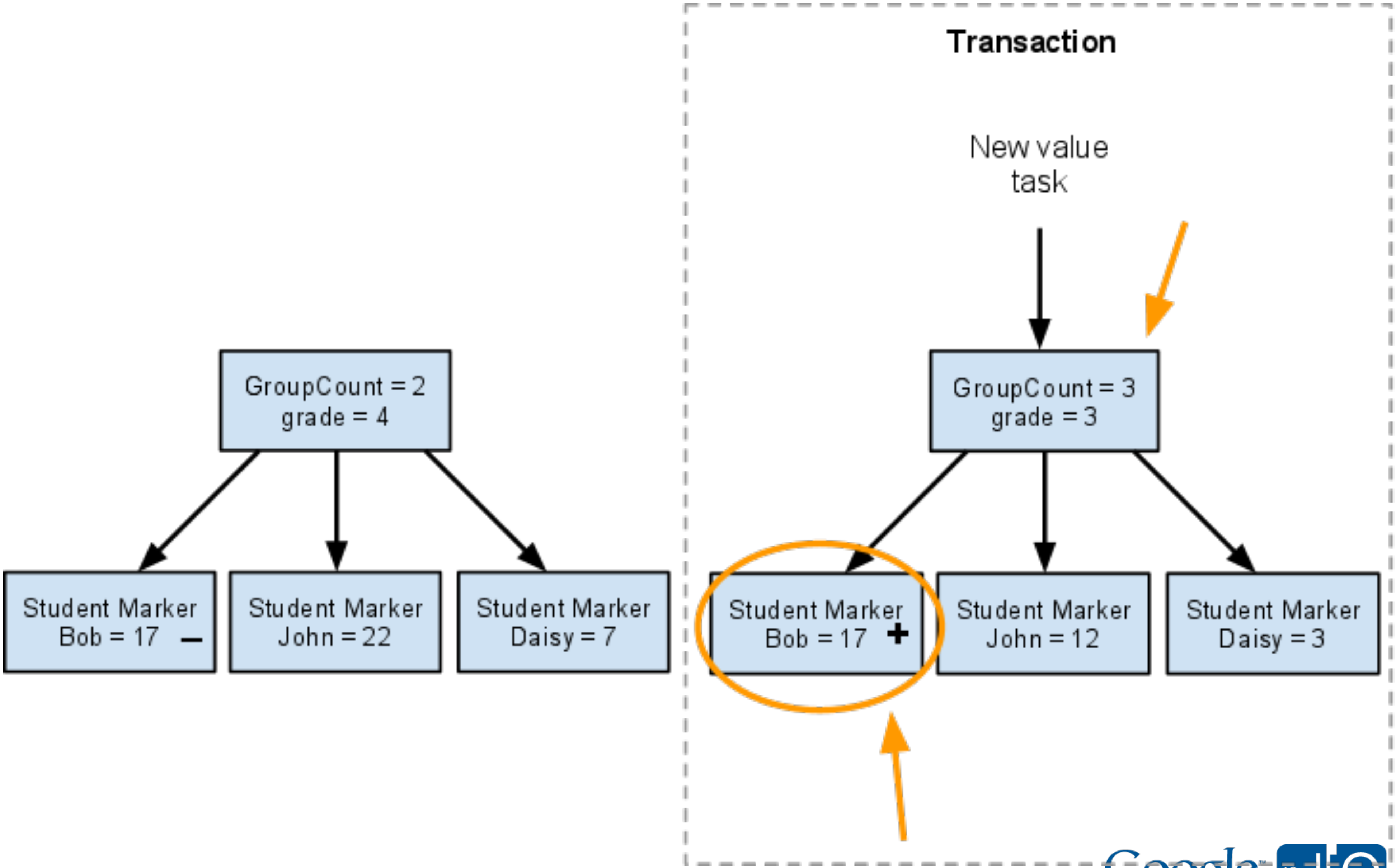
Roll-forward semantics: Update old value



Roll-forward semantics: Update new value



Roll-forward semantics: Update new value



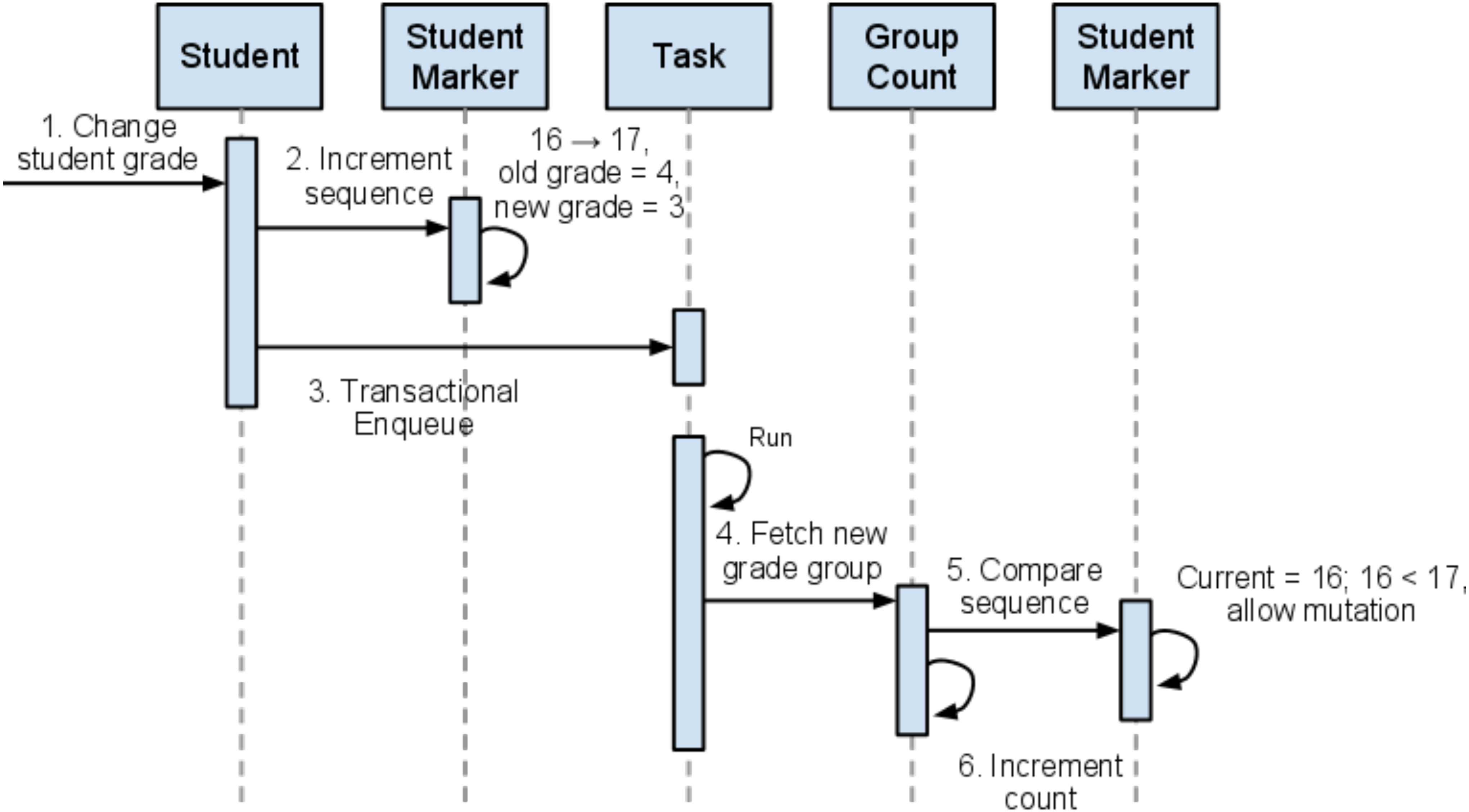
Update group counts

```
def apply(sequence, present, grade, id)
  group_key = db.Key.from_path('GroupCount', grade))
  marker_key = db.Key.from_path(
    'Marker', id, parent=group_key)

  def txn():
    group, marker = db.get([group_key, marker_key])
    if not group:
      group = GroupCount(key=group_key)
    if not marker:
      marker = Marker(key=marker_key)
    if marker.sequence >= sequence:
      raise db.Rollback('Ignore out-of-order')
    # continues on next slide
```

Update group counts countinued

```
    old, marker.present = marker.present, present
    marker.sequence = sequence
    db.put(marker)

    if old:
      group.count -= 1
    if present:
      group.count += 1

    if group.count == 0:
      group.delete()
    else:
      db.put(group)

  db.run_in_transaction(txn)
```

Coordination sequence



Transactional sequences demo

Sequencing details

- Each aggregation row is in its own entity group
  - Update aggregation rows in separate transactions
  - Order of task application doesn't matter

- Marker entity is child of each Count (aggregation) row
  - Marker indicates presence of Student in aggregation
  - Sequence numbers let you ignore old/stale updates

- **Bridge transactions** across entity groups

Sequencing details 2

- Works well for commutative operations (count, sum)
  - Toggling presence is add or subtract
  - Ancestor queries for more complex functions
  - Use continuations and cursors to continue queries

- Enqueue multiple tasks at source data write time
  - Update many aggregations in parallel

Sequencing details 3

- Max throughput proportional to number of aggregation rows
  - Watch out: Data distribution across aggregation
  - Rate-limit materialized view tasks to 1/sec for safety

- Storage cost for "presence" entities

Fan-in: Fork-join queues

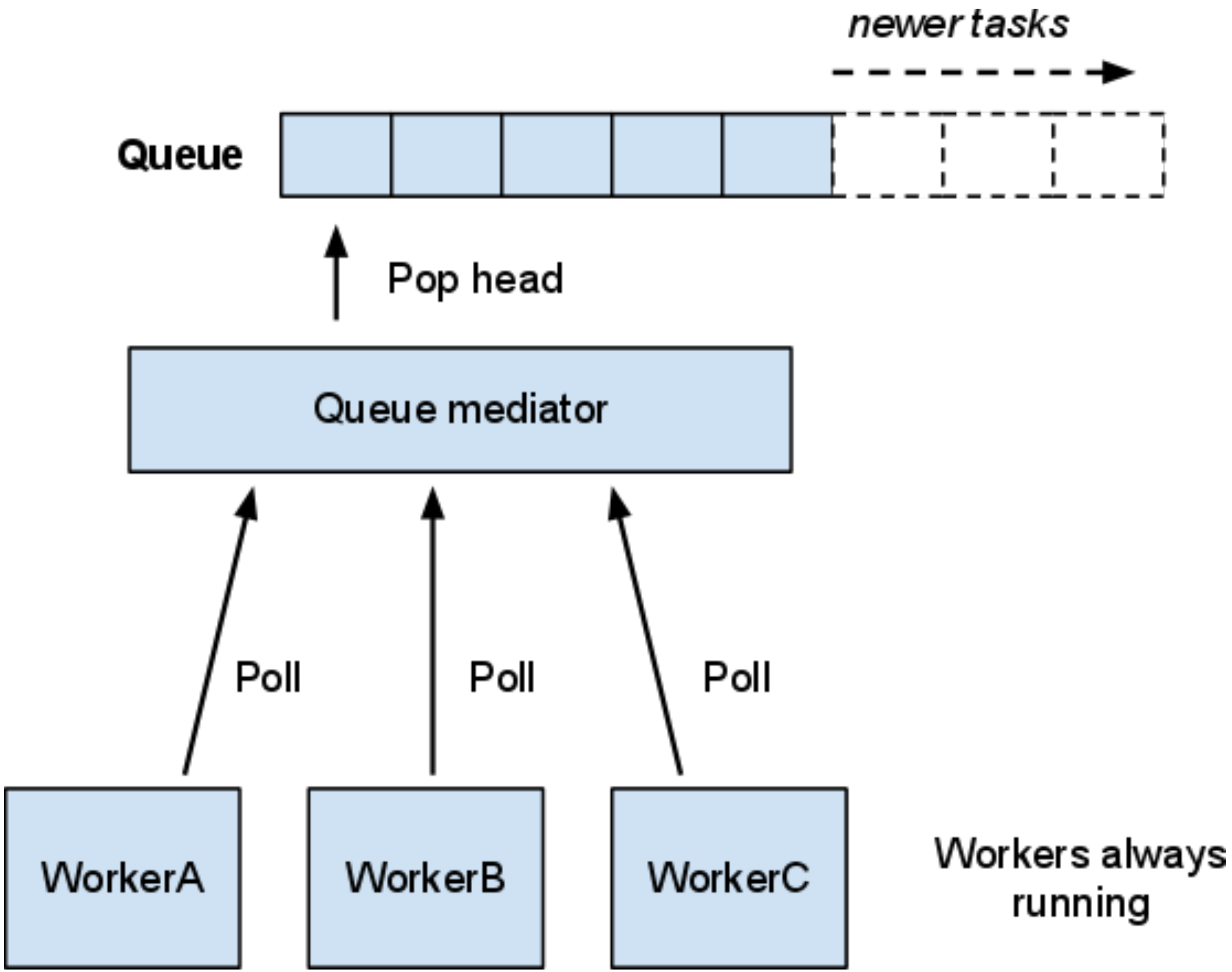
Why fan in?

- Apply multiple data transforms in batches
    - Counters, aggregations, roll-ups, reservations
    - Reddit/Digg-style: Save users' voting history
    - Beat the ~1 write/sec per entity group safety margin

- Wait for high-latency API calls simultaneously
    - RSS aggregators, microblog data sinks
    - Use fewer threads = more throughput
    - Ensure queues do not back up

- Amortize overhead costs with parallel work

Polling workers: Traditional approach



Polling workers: Problems

- How many polling workers do you need to ensure 50 new tasks per second are serviced within 500ms?
- What if tasks take at least 10 seconds?
- How do you guarantee exclusivity?

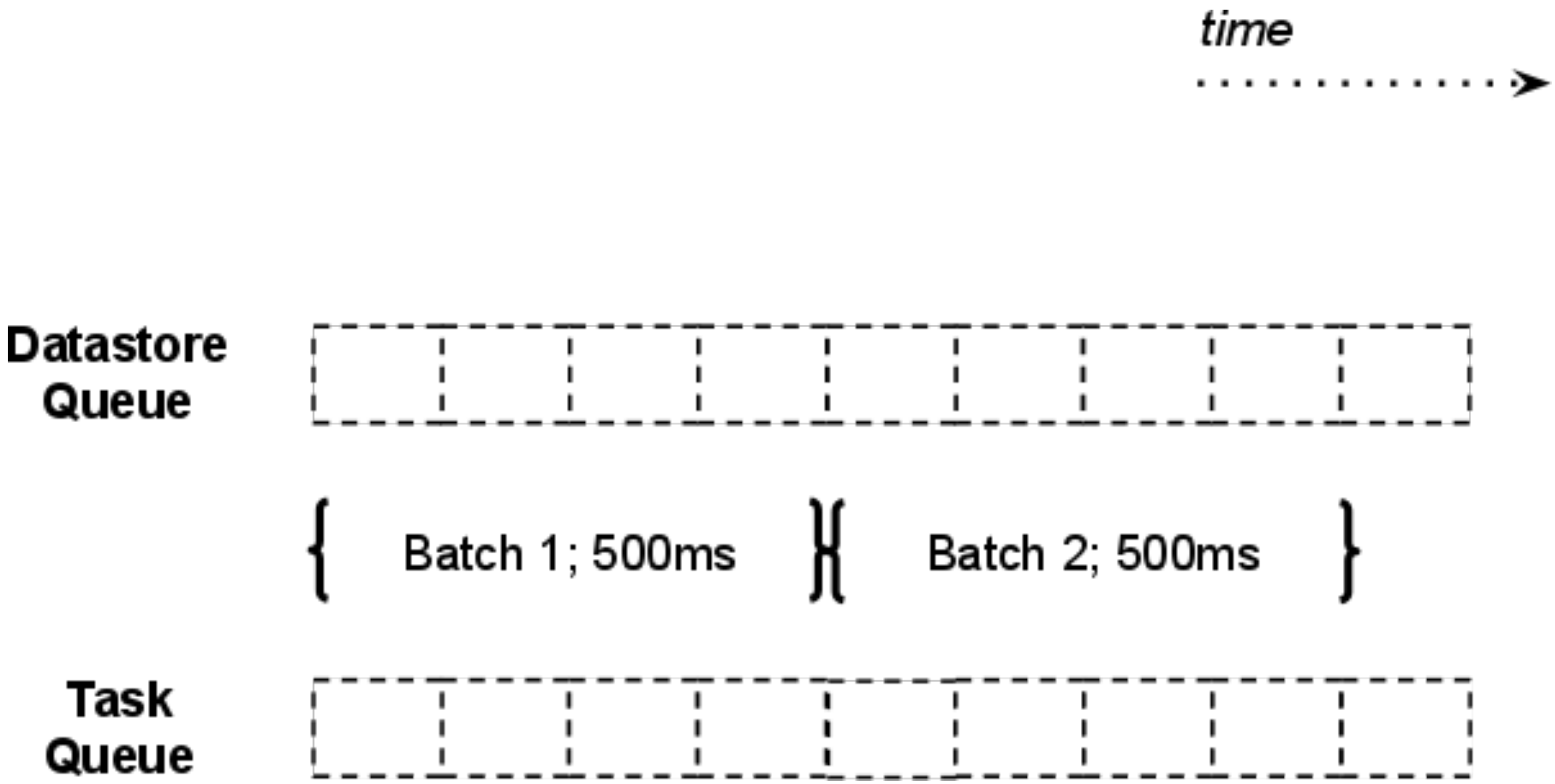
Polling workers are for offline processing

- Not dynamic, not low-latency
- Must have enough workers running to match peak load
- Eliminates the benefits of App Engine's *push* task queue

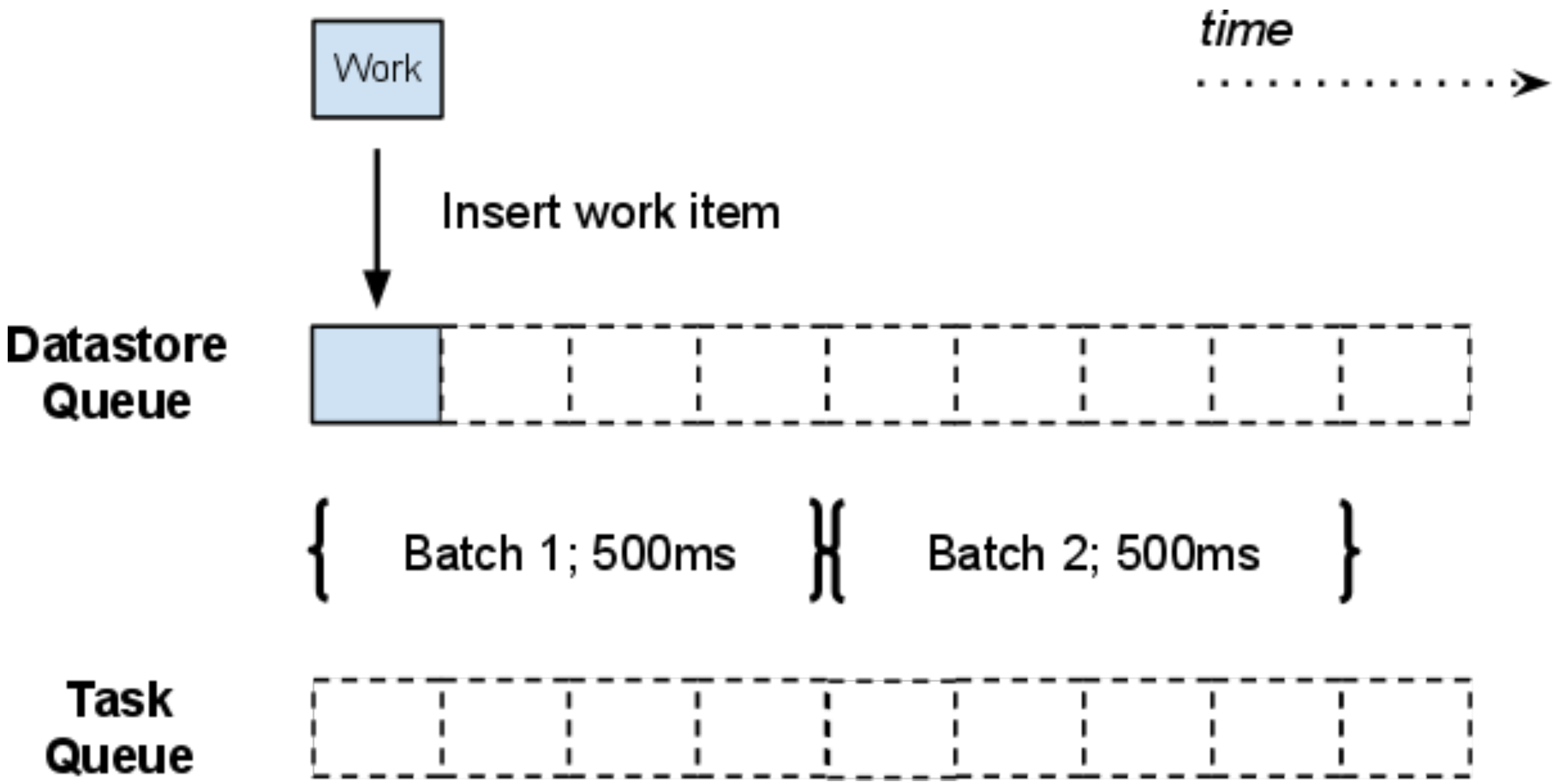
What is a fork-join queue?

- Fork incoming work items apart as they enter
- Work starts within maximum fixed period after arrival
- Execute work in batch for efficiency
- Join completed work items together into a result (optional)

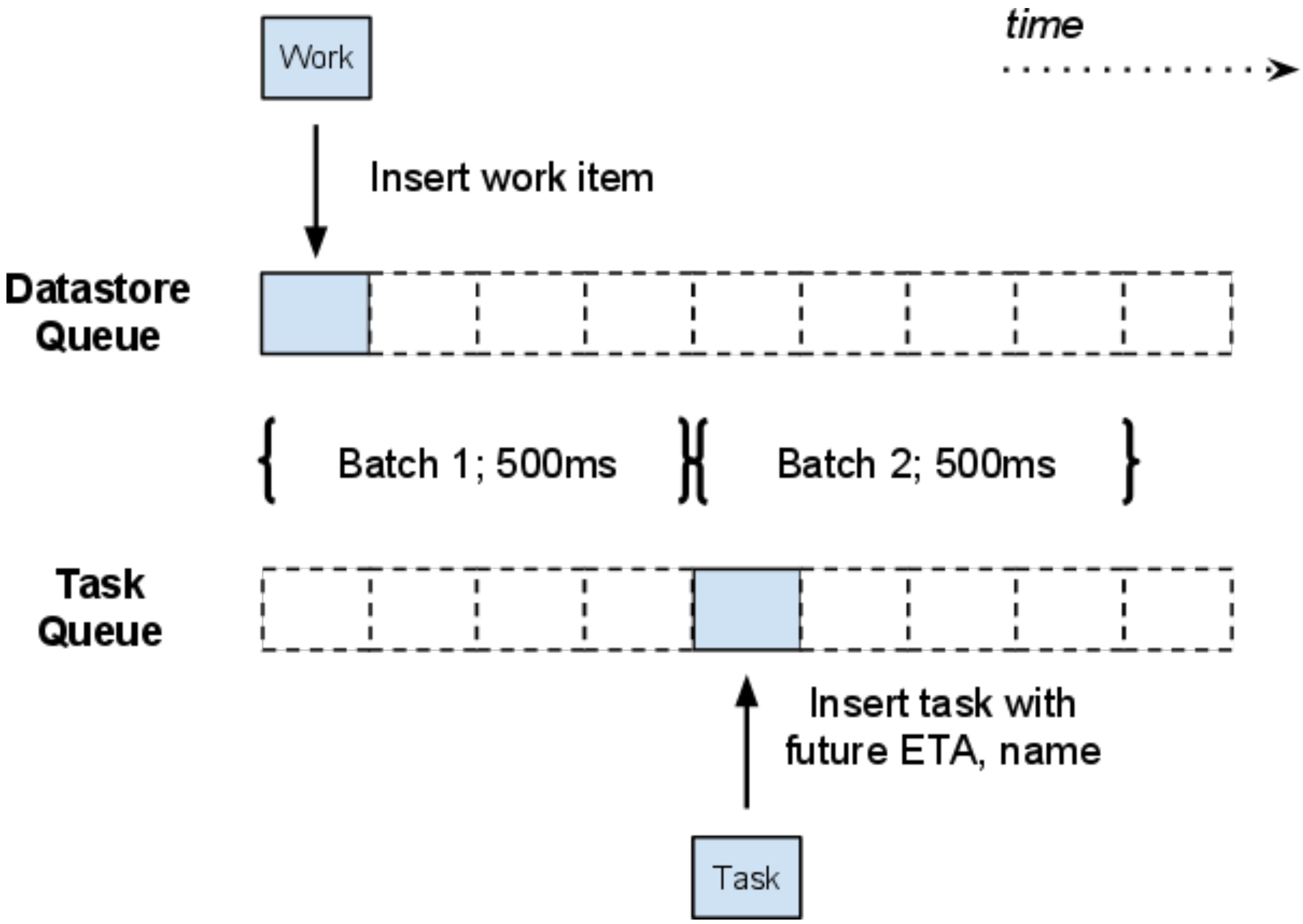
Fork-join queue with Datastore



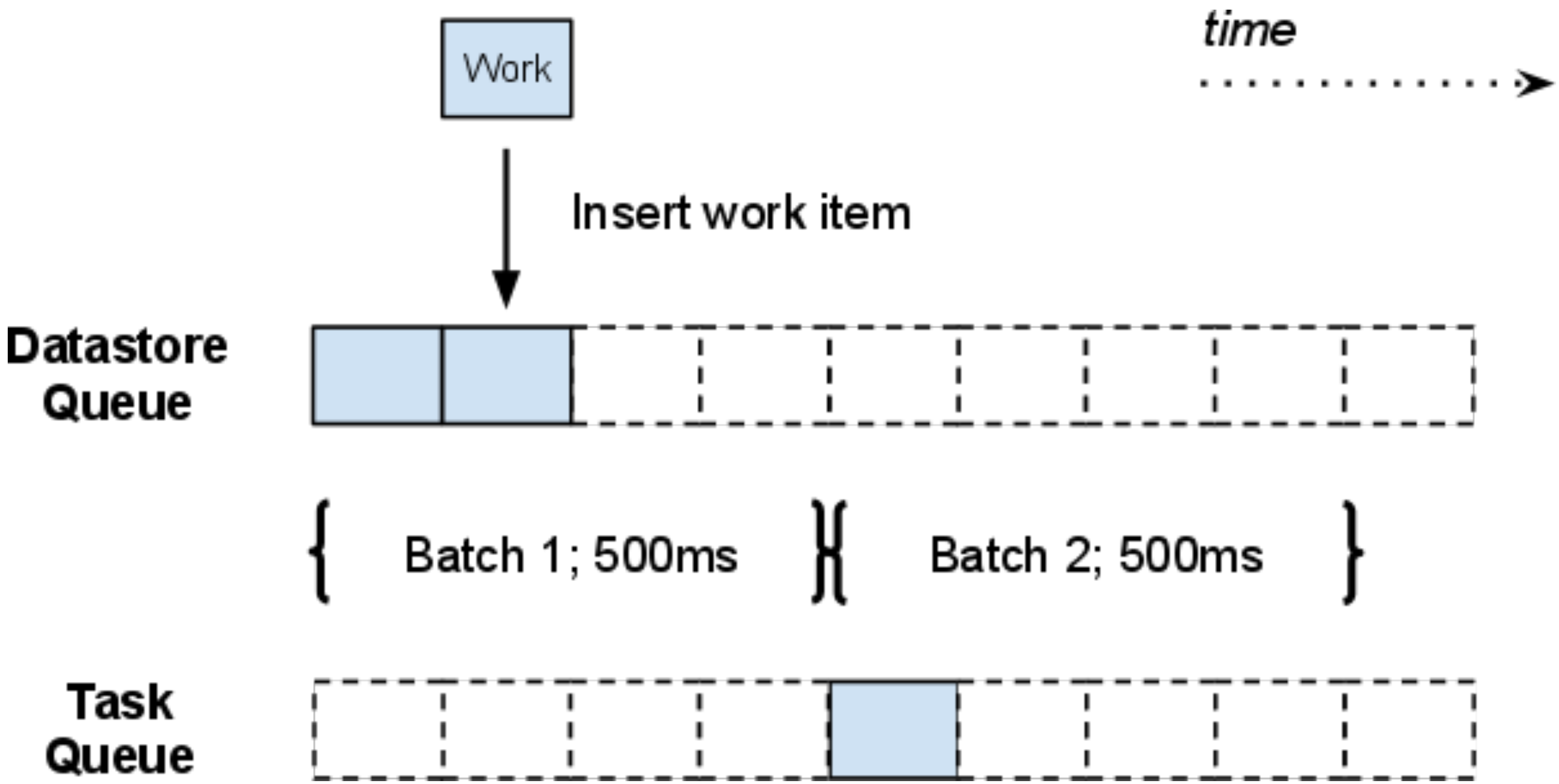
Fork-join queue with Datastore



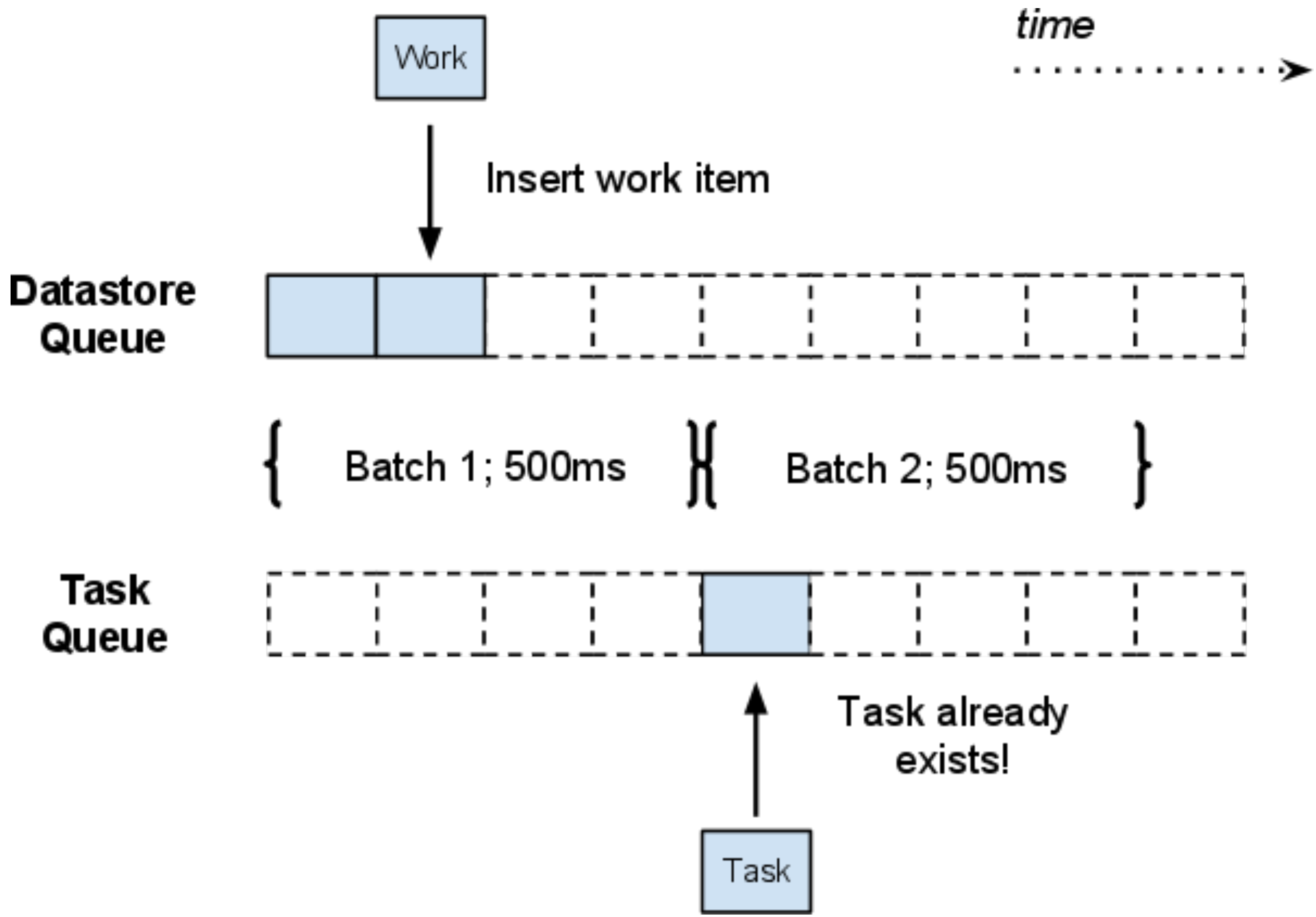
Fork-join queue with Datastore



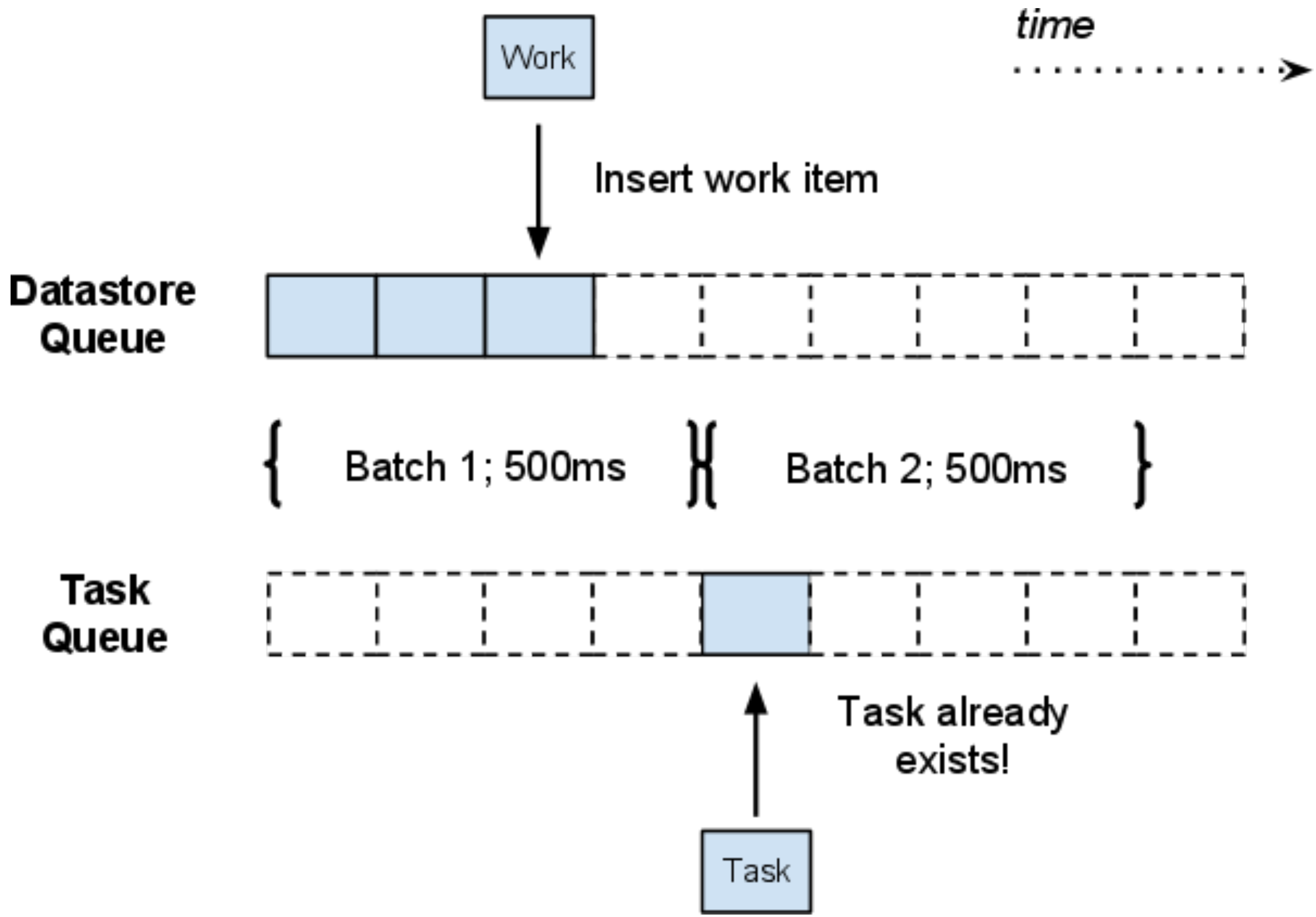
Fork-join queue with Datastore



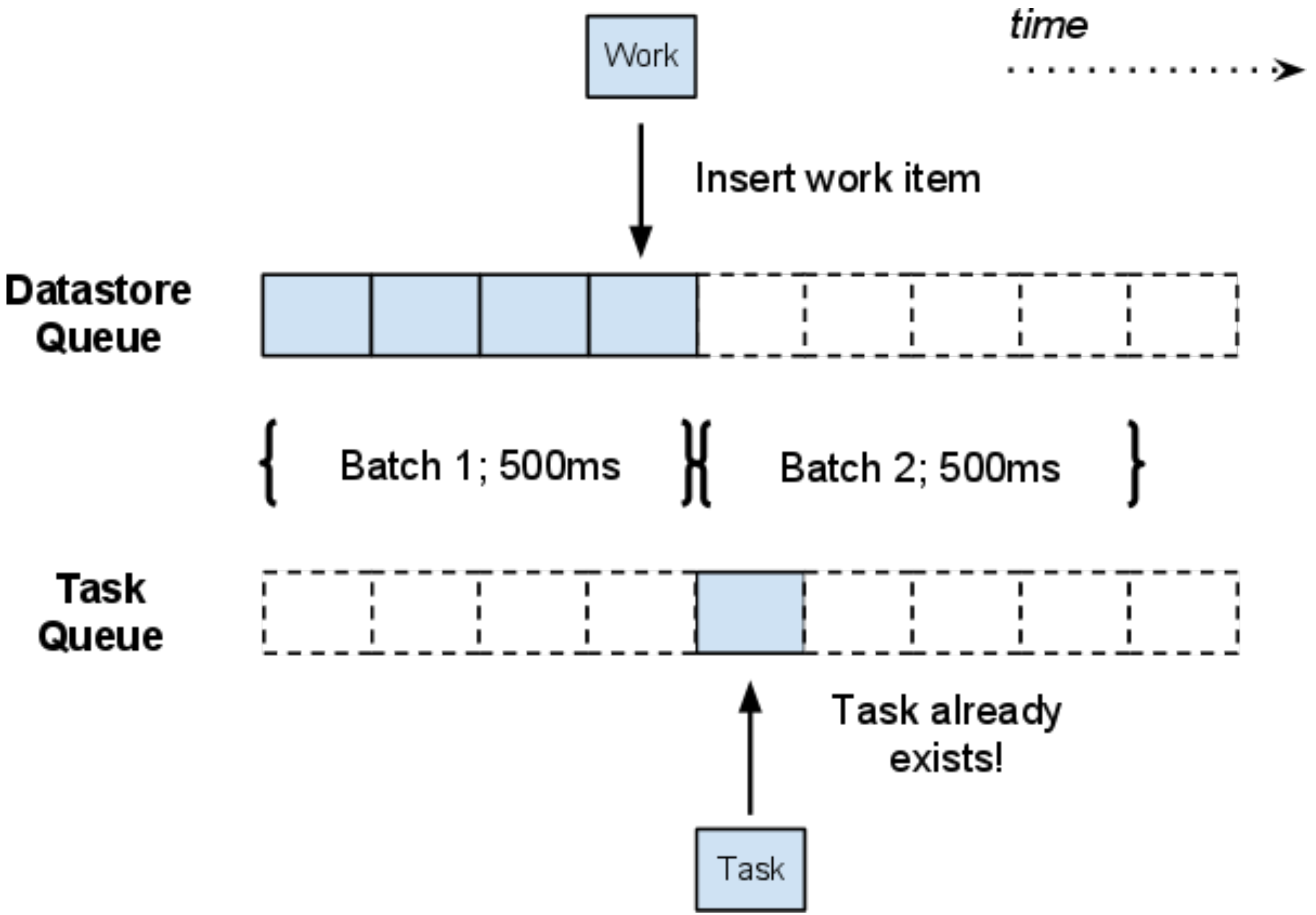
Fork-join queue with Datastore



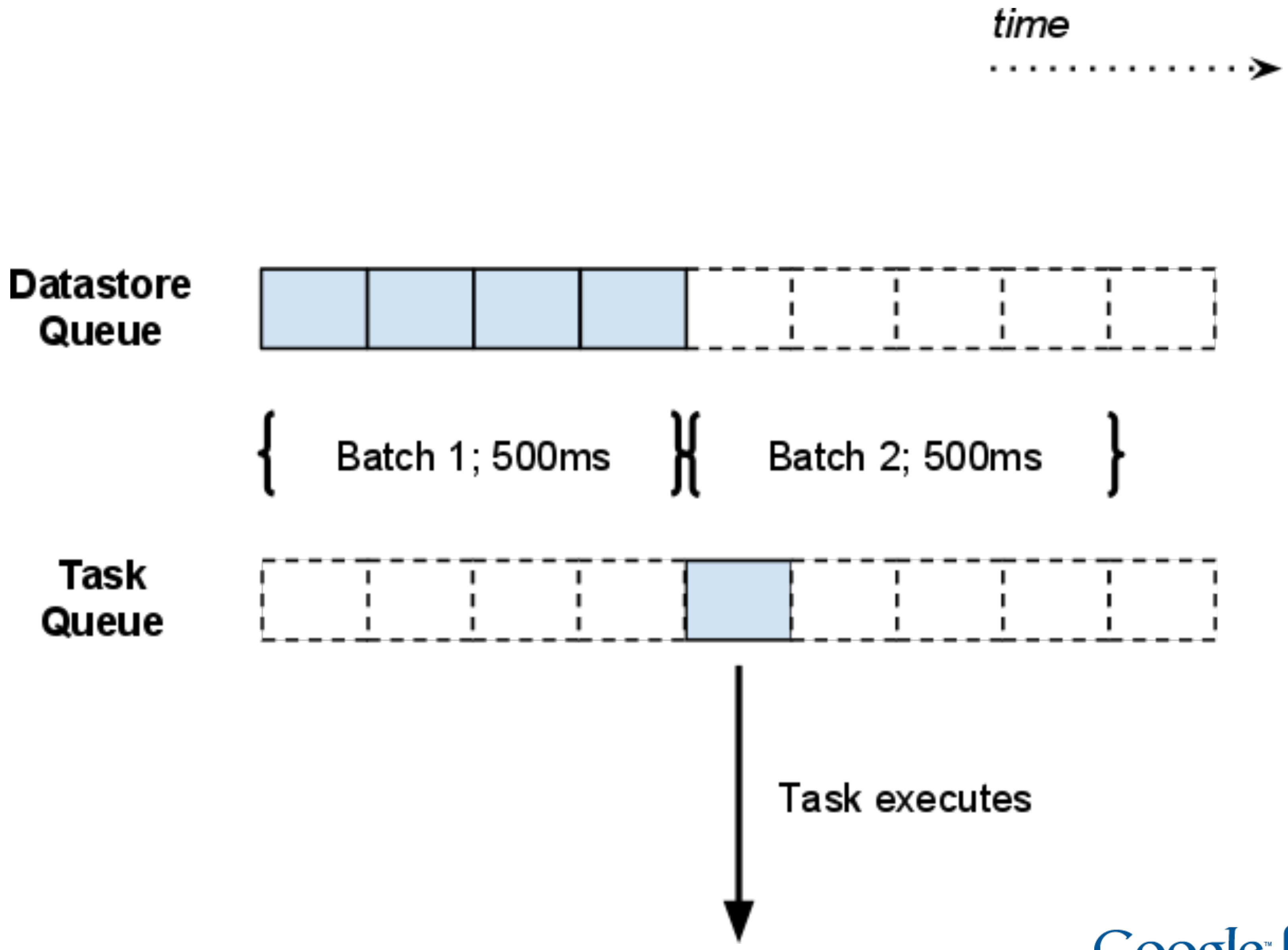
Fork-join queue with Datastore



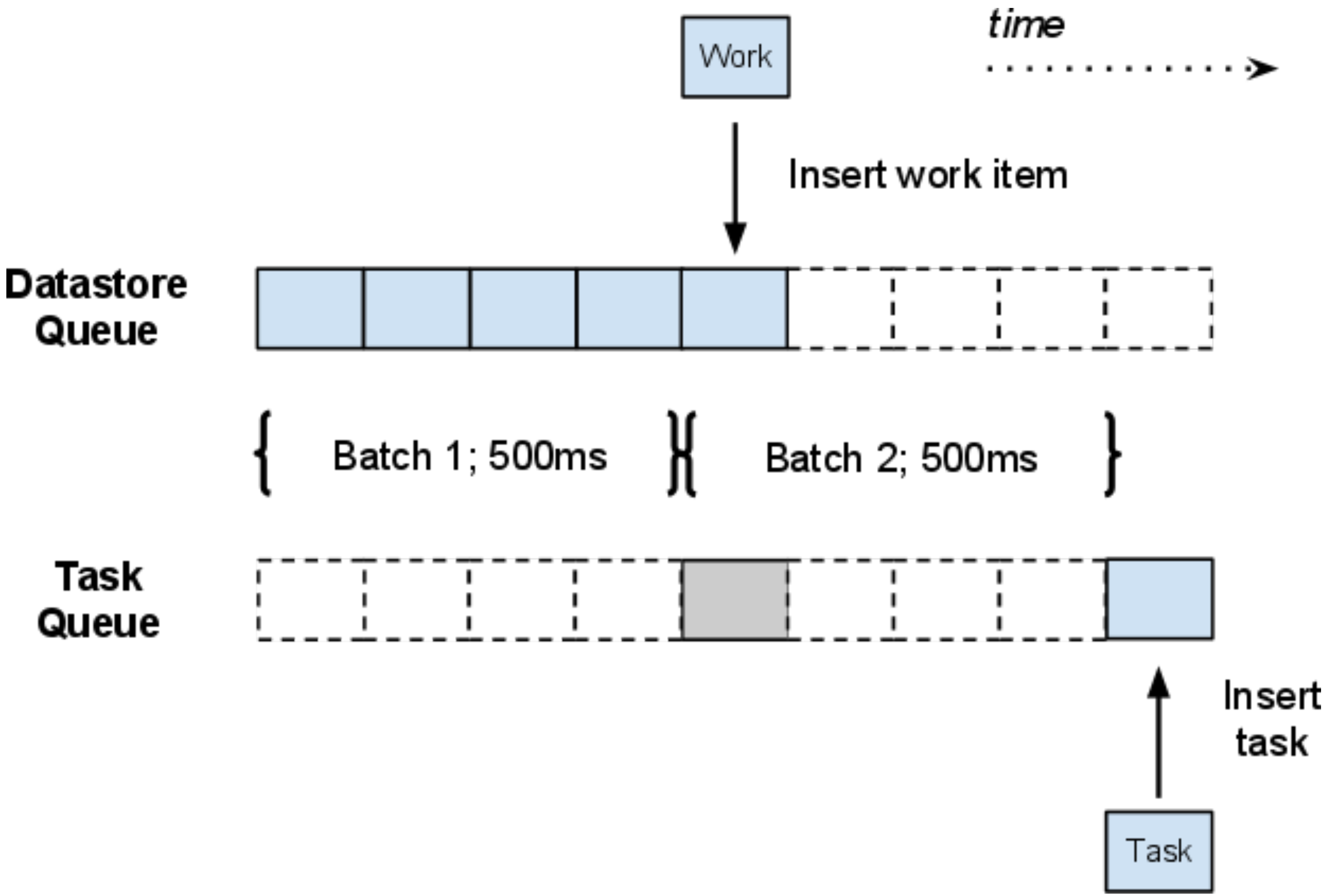
Fork-join queue with Datastore



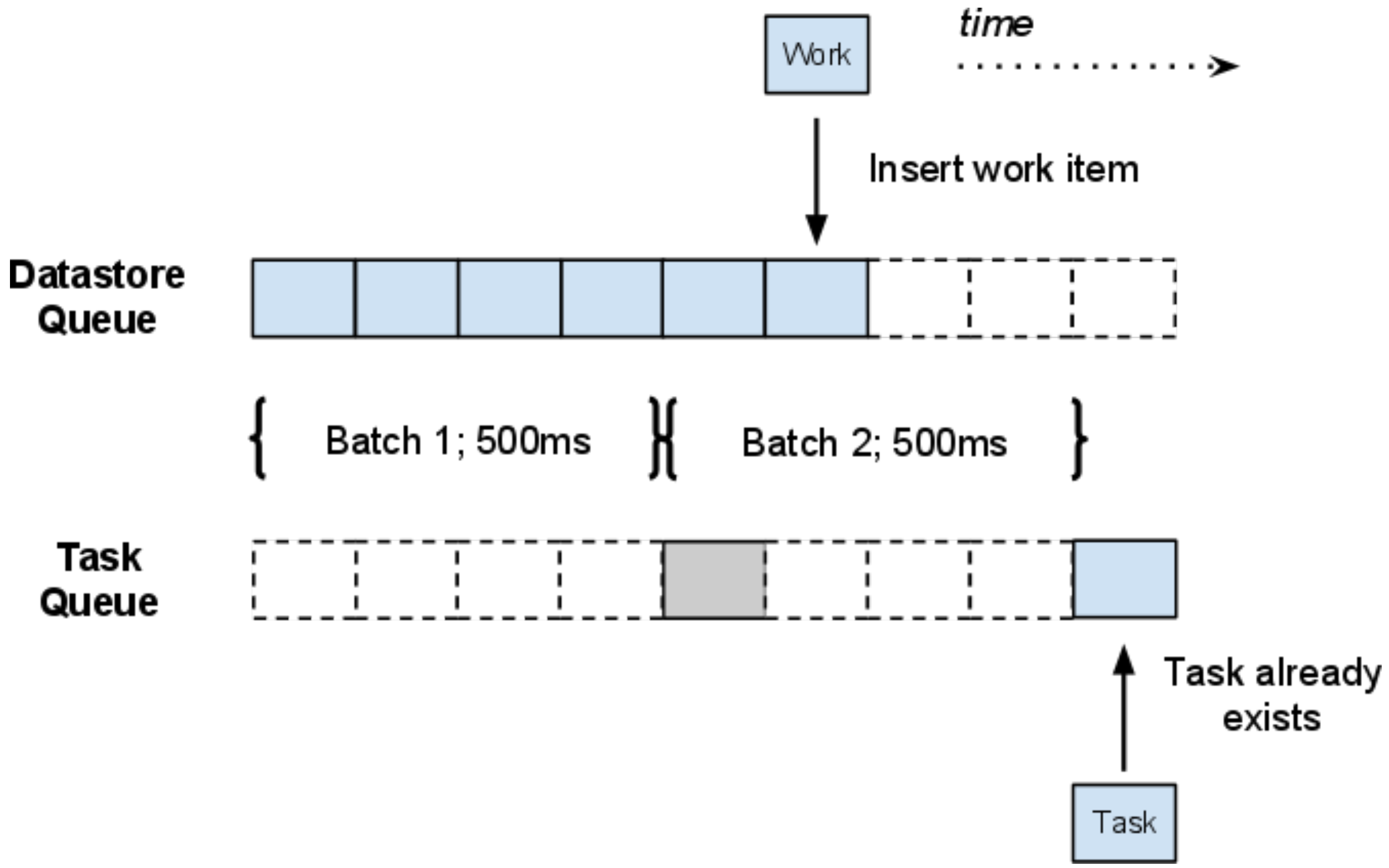
Fork-join queue with Datastore



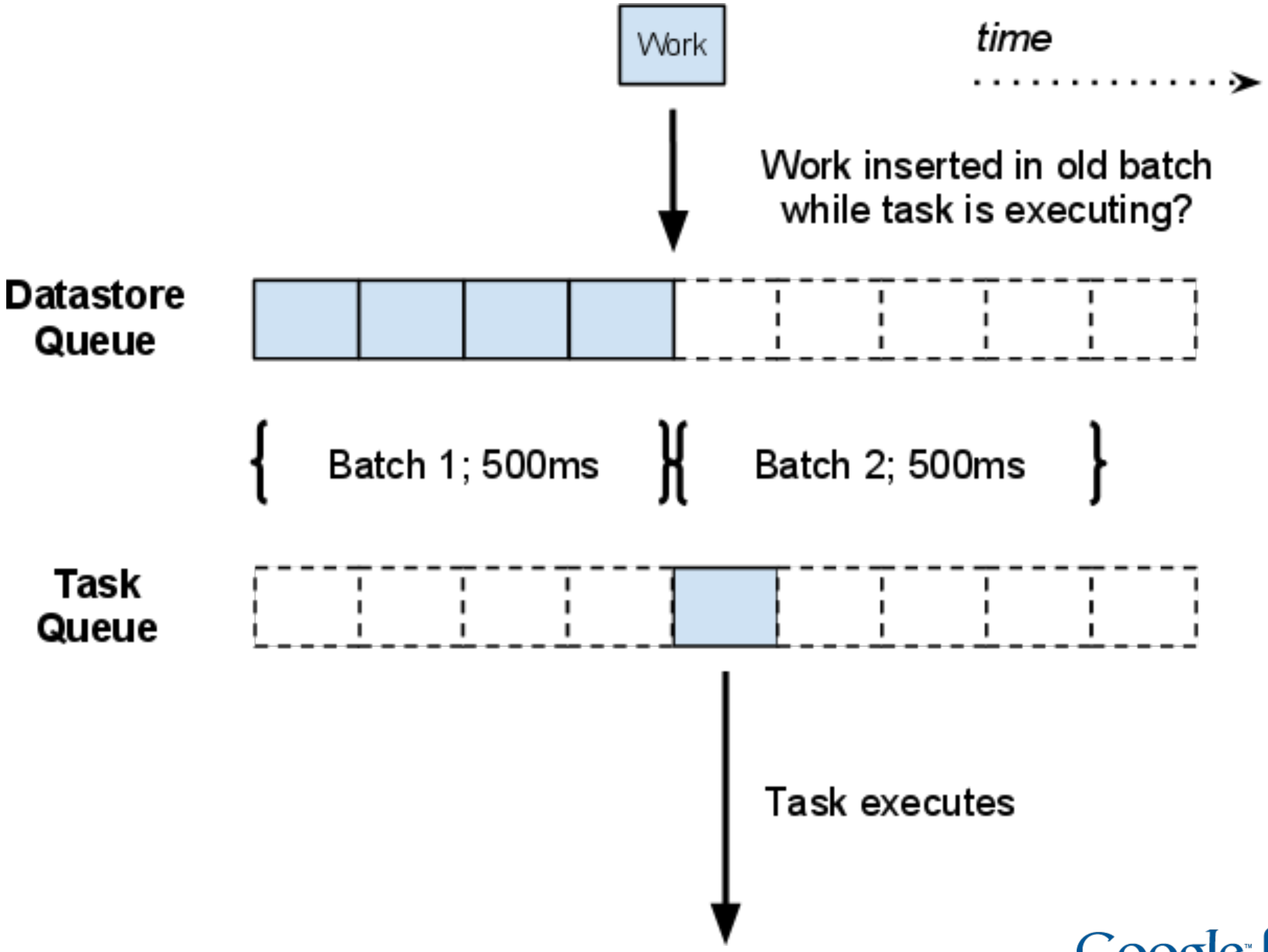
Fork-join queue with Datastore



Fork-join queue with Datastore



Fork-join queue with Datastore: Race conditions



Fork-join queue example: Models

```
class MySum(db.Model):
  name = db.StringProperty()
  total = db.IntegerProperty()

class MyWork(db.Model):
  work_index = db.StringProperty()
  delta = db.IntegerProperty(indexed=False)
```

Fork-join queue example: Insert

```
def insert(sum_name, delta):
  index = memcache.get('index-' + sum_name)
  if index is None:
    memcache.add('index-' + sum_name, 1)
    index = memcache.get('index-' + sum_name)

  lock = '%s-lock-%d' % (sum_name, index)
  writers = memcache.incr(lock, initial_value=2**16)
  if writers < 2**16:
    memcache.decr(lock)
    return False  # Insert fails, try again
  work = MyWork(delta=delta, work_index='%s-%d' %
                (sum_name, knuth_hash(index)))
  work.put()
  # ... continues on next slide
```

Fork-join queue example: Insert continued

```
now = time.time()
try:
  taskqueue.add(
    name='%s-%d-%d' % (
      sum_name, int(now / 30), index),
    url='/work',
    eta=datetime.datetime.utcfromtimestamp(now) +
        datetime.timedelta(seconds=1))
except taskqueue.TaskAlreadyExistsError:
  pass # Fan-in magic
finally:
  memcache.decr(lock)

return True
```

Google 10 I/O

Fork-join queue example: Join

```
def join(sum_name, index):
  # force new writers to use the next index
  memcache.incr('index-' + sum_name)

  lock = '%s-lock-%d' % (sum_name, index)
  memcache.decr(lock, 2**15) # You missed the boat

  # busy wait for writers
  for i in xrange(20):  # timeout after 5s
    counter = memcache.get(lock)
    if counter is None or int(counter) <= 2**15:
      break
    time.sleep(0.250)

  # ... continues on next slide
```

Fork-join queue example: Join continued

```
results = list(MyWork.all()
    .filter('work_index =', '%s-%d' %
            (sum_name, knuth_hash(index)))
    .order('__key__'))
delta = sum(r.delta for r in results)
def txn():
  my_sum = MySum.get_by_key_name(sum_name)
  if my_sum is None:
    my_sum = MySum(key_name=sum_name,
                   name=sum_name, total=0)
  my_sum.total += delta
  my_sum.put()
db.run_in_transaction(txn)
db.delete(results)
```

Fork-join queue example: Demo

Fork-join queue details

- Task names are the fan-in mechanism
- Task ETA for periodic batching
- memcache reader/writer locks for batch coordination
  - Spin locks with timeout
- Datastore queries to find work

- Use offline job to pick up drops (memcache failures)

Fork-join queue performance

- Depends on your batch size (work items per task)
  - Can achieve 80 to 1 easily.

Items per second	Average insert latency
22.1	223ms
39.7	245ms
55.6	258ms
75.8	249ms

Fork-join queue performance 2

- Work index **must** be a hash
  - Distribute load across Bigtable tablets
  - Alternative is tablet splits, unavailability

- Eliminate all other indexes on work items
  - Prevent overloading contiguous Bigtable rows
  - Can keep indexes if you're "boxcar"-ing transactions

- The magic of batch period = 0

Bonus: Fan-in with materialized views

Fan-in with materialized views: A sketch

1. Configure fan-in queue to batch once per second
2. User starts transaction on input data, update its value
   - Get fork-join work indexes for target aggregations
   - Assign work indexes to your input sequence markers
   - Enqueue update tasks (unnamed), Commit
3. Optimistically insert *named* fan-in tasks
   - Guarantees completion; ignored in common case
4. Later: Fan-in worker queries for inputs by work index
5. Worker transacts on aggregation data rows
   - Batch get of aggregation markers for inputs
   - Compare old markers to input sequence numbers
   - Compute commutative diff of up-to-date inputs
   - Update aggregation rows, Commit

Fan-in with materialized views: A sketch

- Please build this!

Future directions

Future directions

- Background servers
  - No wall-clock limits (30 sec deadline removed)
  - Chunk through fan-in queues in bulk

- Addressable servers
  - Send RPCs from user-facing requests to backends
  - Fan-in queues can be in memory

- Order of magnitude faster, skip disk writes

View live notes and ask questions about this session on Google Wave

http://tinyurl.com/app-engine-pipelines

Me

http://onebigfluke.com

Google™ 10 I|O

Google™ 10 I O