Queues in PostgreSQL

Thomas Munro

PGCon 2016

About Me

- Joined EnterpriseDB's database server team ~1 year ago
- Working on EDB Postgres Advanced Server and PostgreSQL
- Minor contributor to PostgreSQL: **`SKIP LOCKED`**, `cluster_name`, `remote_apply`, various bug fixes (multixacts, SSI, portability, testing), review

What's a Queue?

Why Put One in an RDBMS?

Example Use Cases

Implementation

Problems

What Could We Do Better?

queue /kjuː/

noun

1. *Chiefly British* A line or sequence of people or vehicles awaiting their turn to be attended to or to proceed.

Definition: Oxford Dictionary
Image: paphotos.co.uk

queue /kjuː/

noun

1. *Chiefly British* A line or sequence of people or vehicles awaiting their turn to be attended to or to proceed.

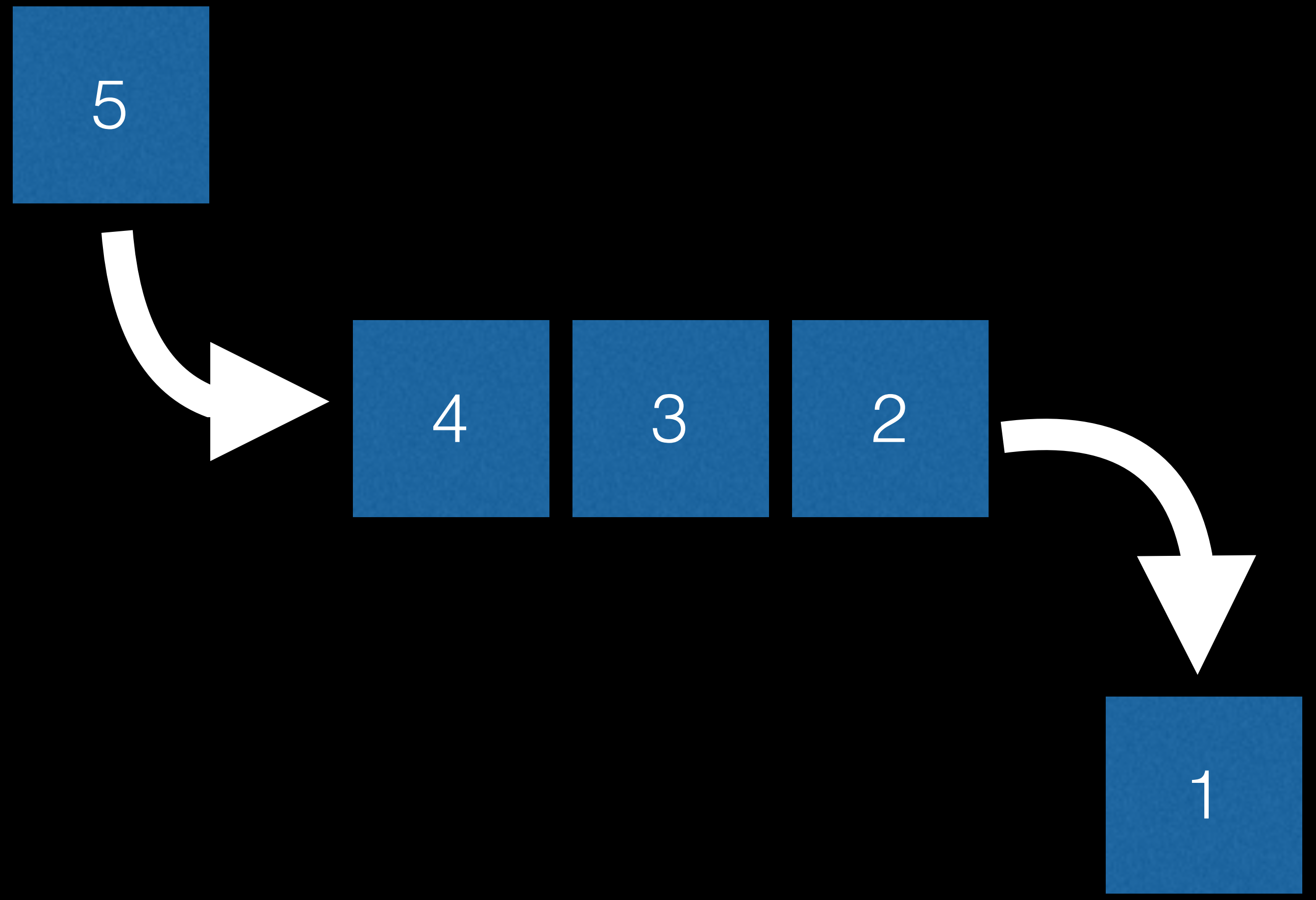
[“Americans have started saying ‘Queue’. Blame Netflix”
- *New Republic*]

Definition: Oxford Dictionary
Image: paphotos.co.uk

queue /kjuː/

noun

2. *Computing* A list of data items, commands, etc., stored so as to be retrievable in a **definite order**, usually the order of insertion.



Definition: Oxford Dictionary

Informal Taxonomy

- Queues

1. FIFO: First-in-first-out queues
2. Priority queues

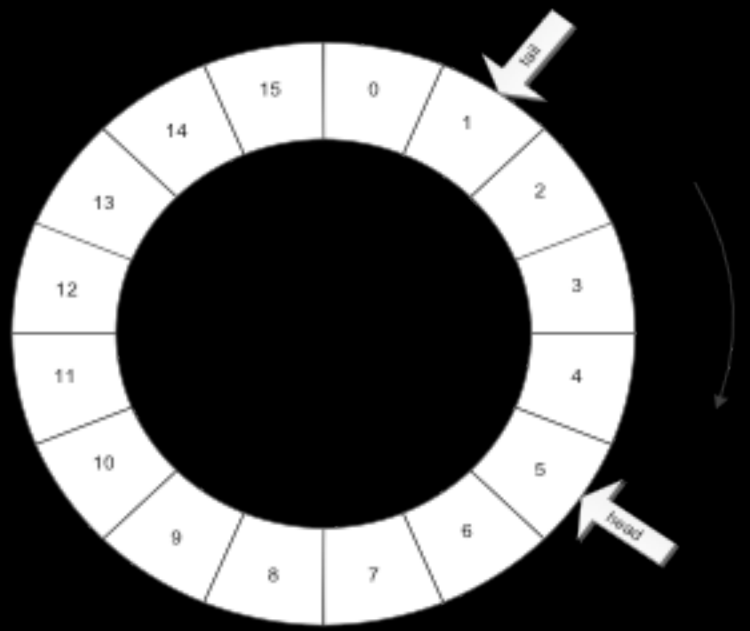
- “Queues”

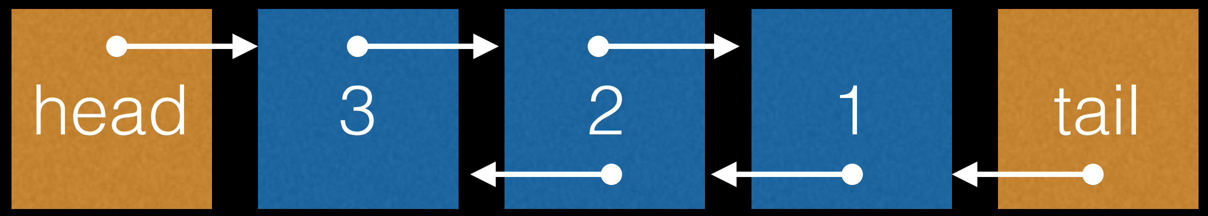
3. Specialised queues (merging, reordering)
4. Unordered/approximately ordered queues

1. FIFO Queues

- The order most people think of when they hear the word “queue”

- Often used in low level code because the implementation is simple and fast: physical layout reflects logical ordering





2. Priority Queues

- Sometimes a different explicit logical order is needed
- Implementation techniques include sets of FIFO queues, **trees** and other data structures associated with sorting

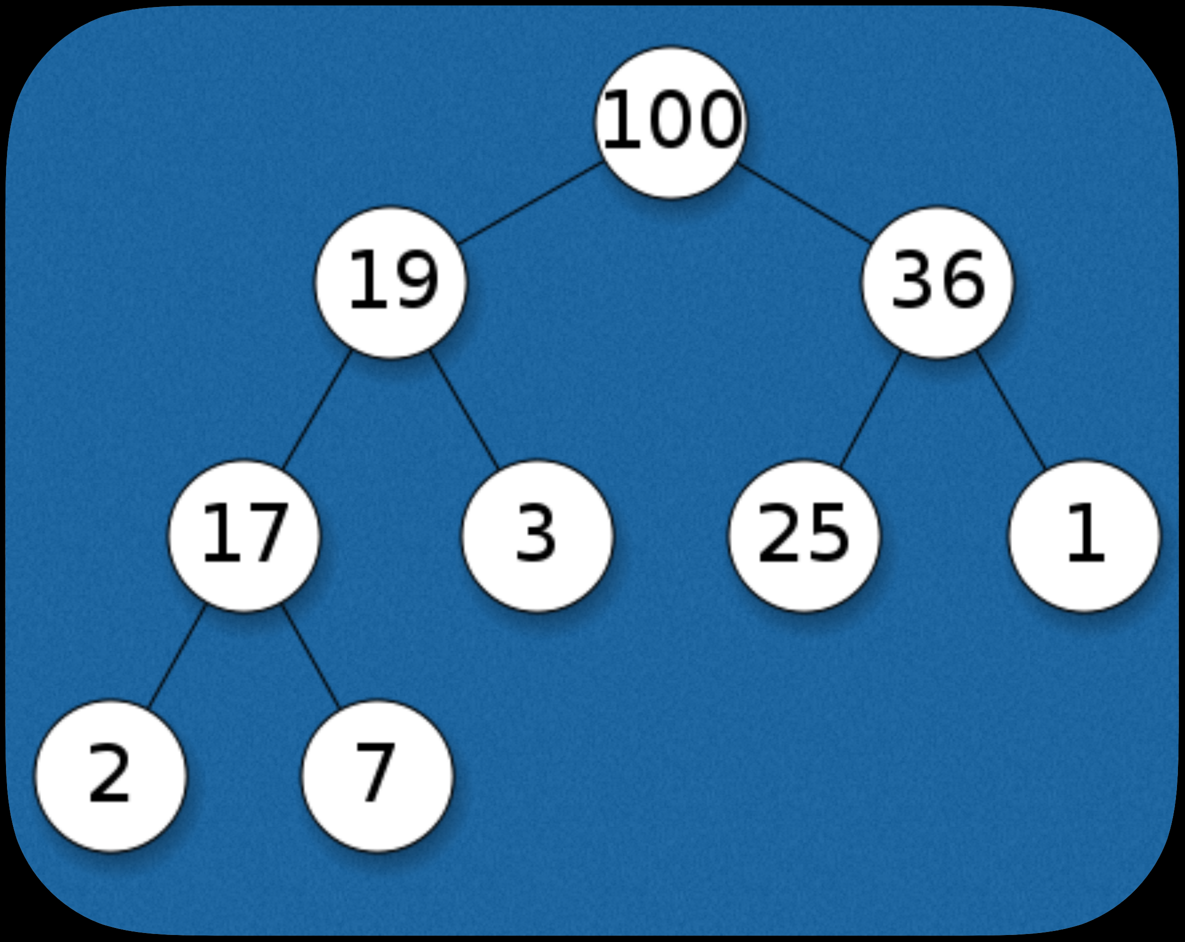


Image: Wikipedia

3. Specialised "Queues"

- Sometimes we use the word queue more loosely to describe something that gives up strict logical ordering to meet some other goal

- Operating system IO schedulers and elevators/lifts allegedly improve global efficiency by merging and reordering queued requests

Image: epicioci pixabay.com

4. Unordered & Approximately Ordered "Queues"

- Sometimes we don't care about the order that items are retrieved in at all, we just want to process them as quickly as possible

- … but usually we want at least approximate time ordering for fairness (no arbitrarily stuck messages), but don't need strict global ordering for correctness

- Transactional and concurrent systems blur the order of both insertion and retrieval

What’s a Queue?

Why Put One in an RDBMS?

Example Use Cases

Implementation

Problems

What Could We Do Better?

“Meh, why not use RabbitMQ/Redis/PGQ/ <*thing*>?”

Free clipart: pngimg.com

You might consider using a plain old database if…

- … you want reliable persistent message processing that is atomic with respect to other database work (without the complications of distributed transactions)

- … you don’t want the maintenance, backups, failover and risks of new moving parts (message broker daemons)

- … your message rates and number of consumers are in the range that PostgreSQL and your hardware can handle

- … you like PostgreSQL enough to attend a conference

What's a Queue?

Why Put One in an RDBMS?

Example Use Cases

Implementation

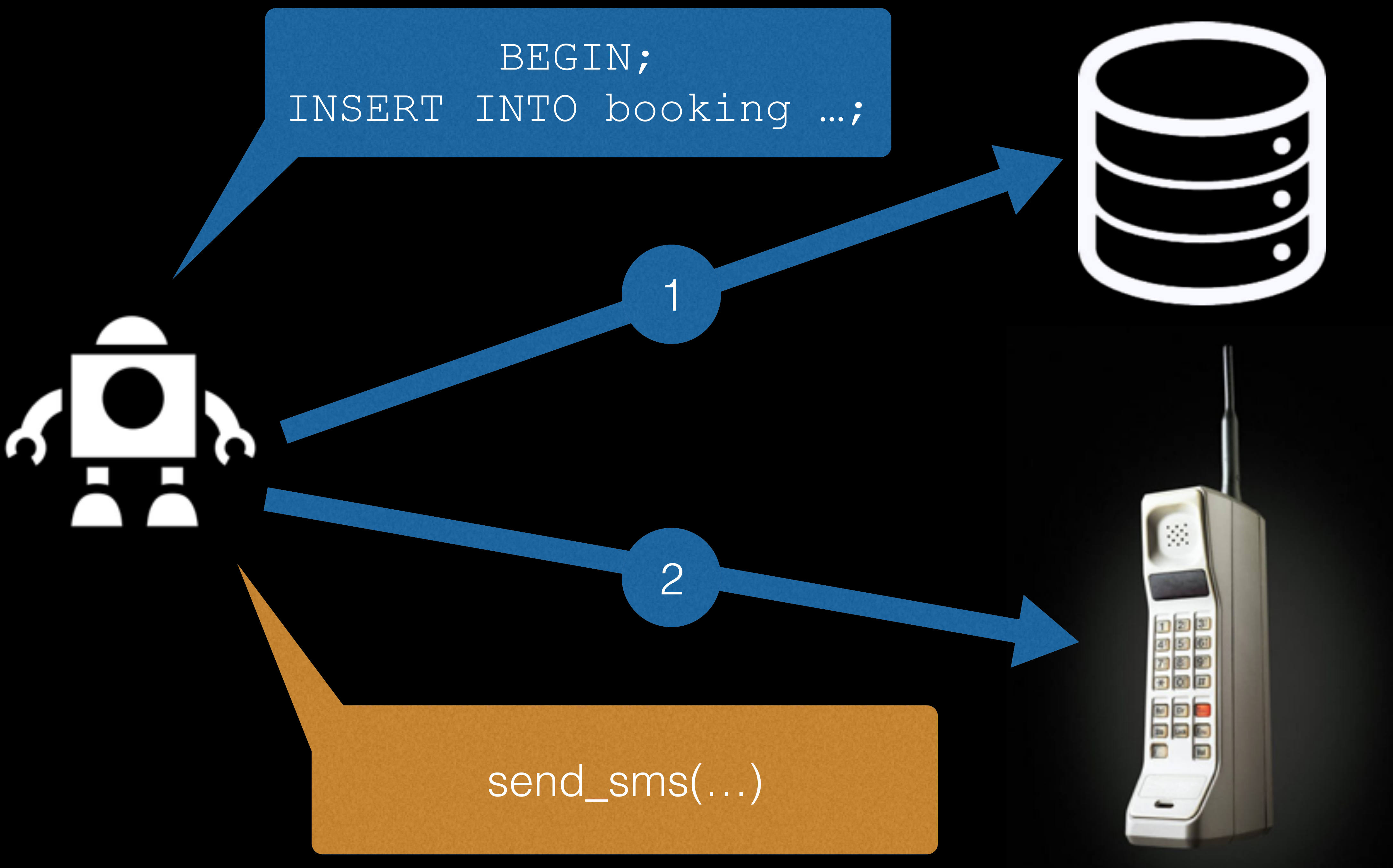
Problems

What Could We Do Better?

Mixing Transactions with External Effects

- We want to book a seat on a plane
- We also want to send an SMS message with confirmation of the booking and seat number

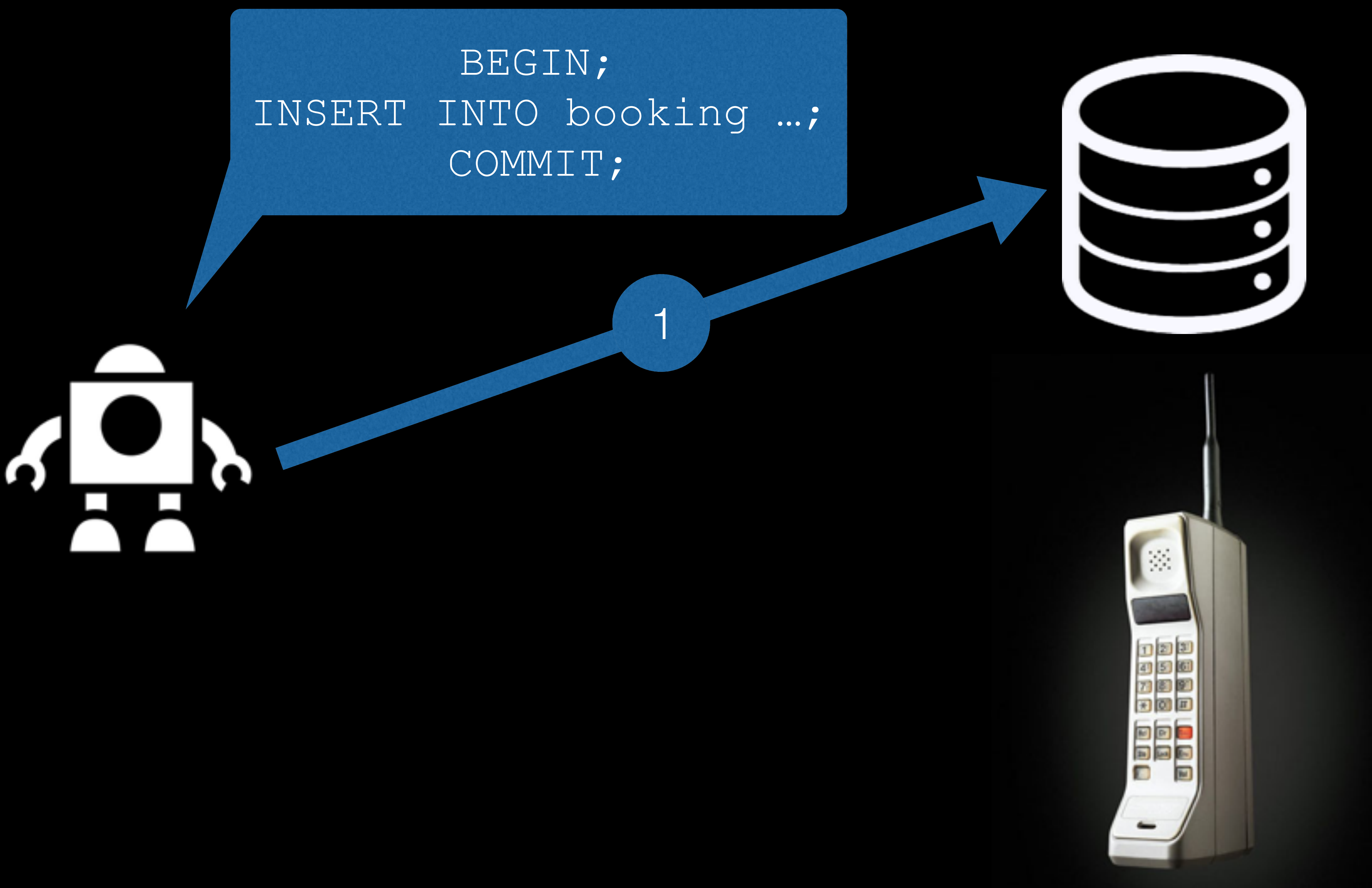
Mixing Transactions with External Effects: Take 1



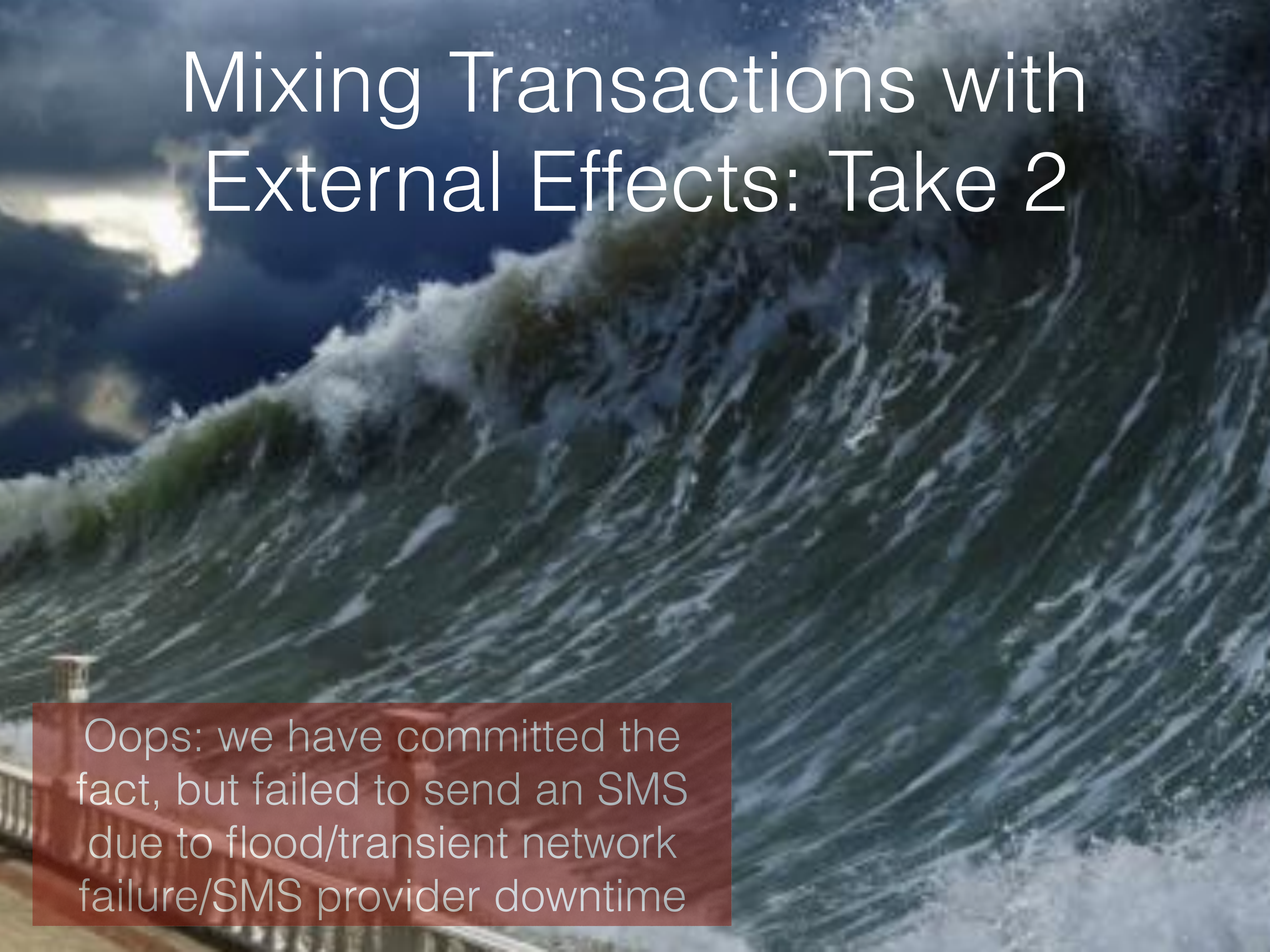
Mixing Transactions with External Effects: Take 1

Oops: we have sent an SMS but forgot the fact it represents due to an asteroid/bug/hardware failure before COMMIT

Mixing Transactions with External Effects: Take 2

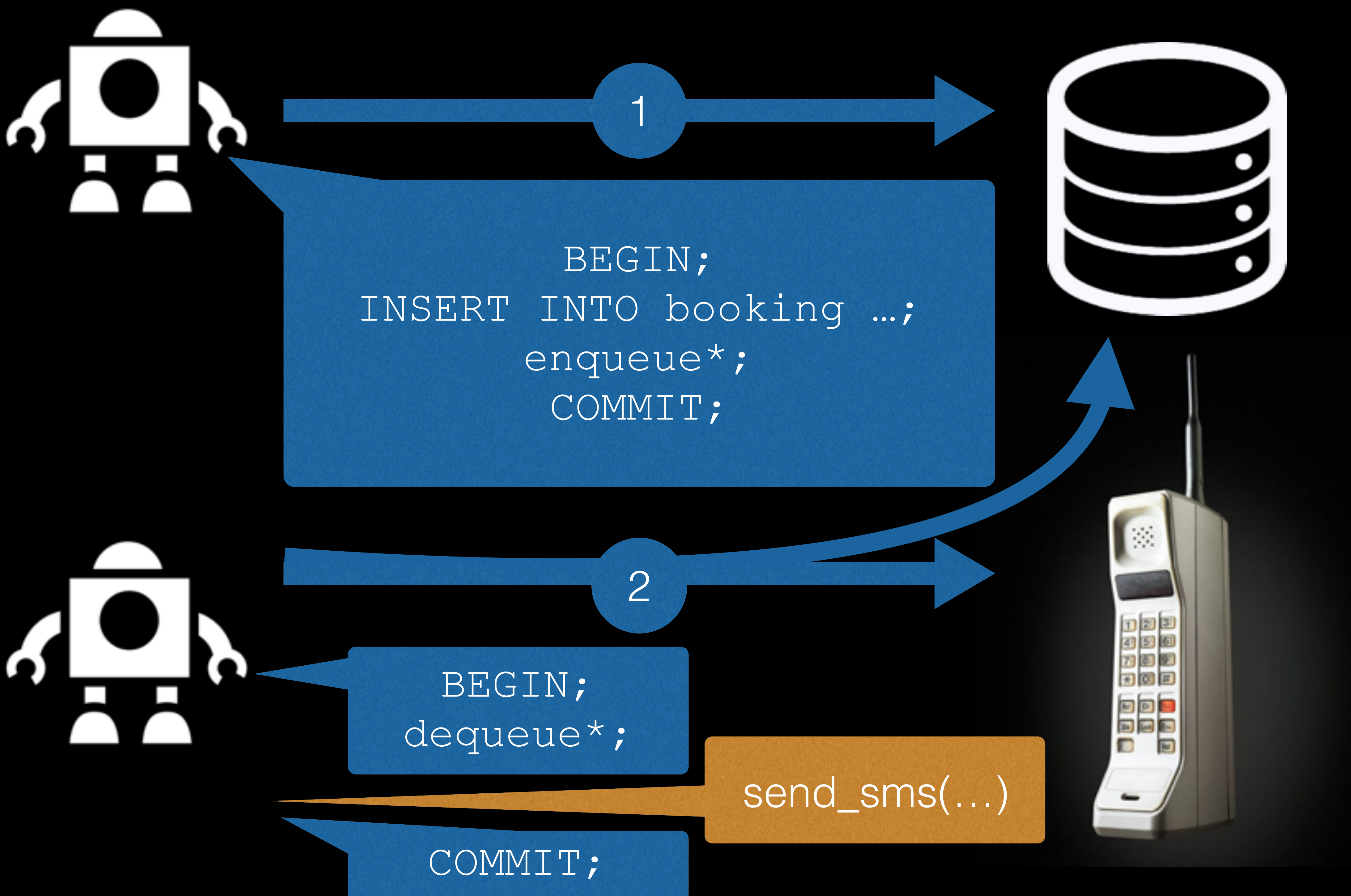


Mixing Transactions with External Effects: Take 2



Oops: we have committed the fact, but failed to send an SMS due to flood/transient network failure/SMS provider downtime

Mixing Transactions with External Effects: Take 3



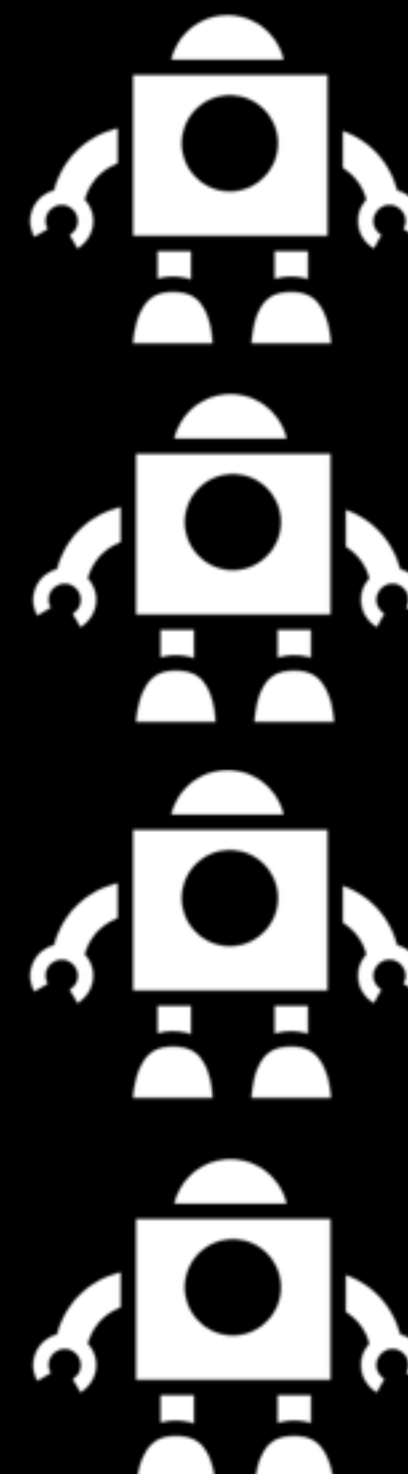
Mixing Transactions with External Effects

- We establish a new fact (the booking) and record our intention to notify the customer (the entry in the SMS queue) **atomically**
- We remove the queued item after sending successfully (and probably have a retry system if the SMS service is temporarily failing)
- The SMS sending operation should ideally be **idempotent** so that if we fail after sending but before committing the dequeue operation, sending the same message again won't be problematic

Distributed Computing

- Job control for farming out expensive external computation to worker processes

- Job control for database aggregation work moved out of interactive transactions



What’s a Queue?

Why Put One in an RDBMS?

Example Use Cases

Implementation

Problems

What Could We Do Better?

Ingredients

- Messages: Rows in plain old tables
- Priority ordering: `ORDER BY`
- Signalling: `NOTIFY` & `LISTEN`
- Concurrency:
  - None, course grained locking or `SERIALIZABLE`
  - … or explicit fine grained locking

No Physical FIFO

- The relational model (and therefore its approximate earthly embodiment SQL) doesn’t expose details of physical ordering or insertion order to the user
- Ordering will therefore need to be a function of values in records supplied at `INSERT` time, and explicitly requested with `ORDER BY` when they are retrieved (it’s always a “priority queue”), or unordered

Enqueue Protocol

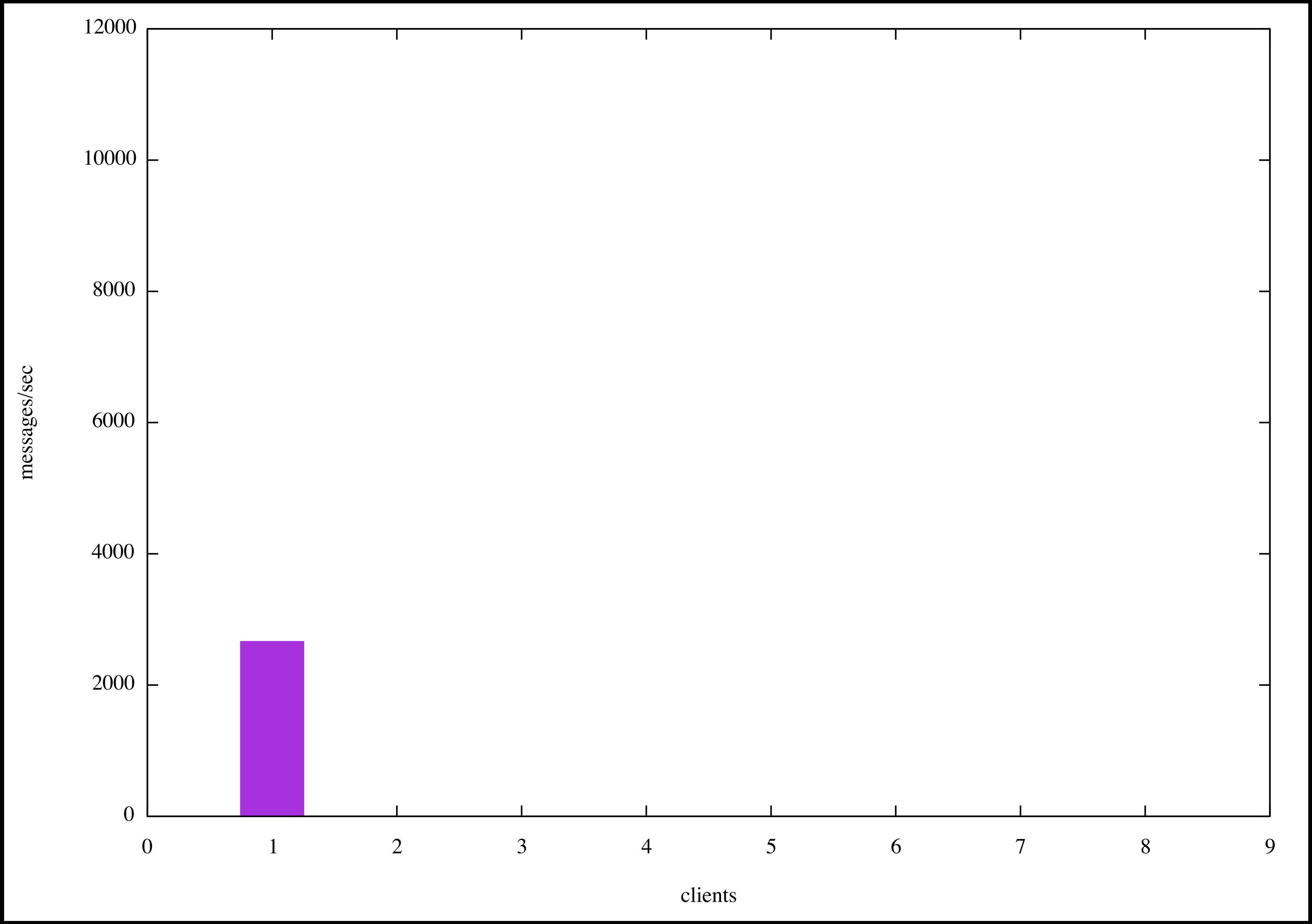
- ```
  BEGIN;
  — any other work
  INSERT INTO sms_queue (…)
  VALUES (…);
  NOTIFY sms_queue_broadcast;
  COMMIT;
  ```

- Note: if inserting transactions overlap, then it is difficult to generate a key that increases monotonically with respect to commit/transaction visibility order!
  ```

Dequeue Protocol: Take 1

- `LISTEN sms_queue_broadcast;`

- ```
  BEGIN;
  SELECT message_uuid, destination, body
    FROM sms_queue
   ORDER BY insert_time
   LIMIT 1;
  — if found, do something (internal or
  — external + idempotent) and then:
  DELETE FROM sms_queue
   WHERE message_uuid = $1;
  COMMIT;
  ```

- `— repeat previous step until nothing found`

- `— wait for notifications before repeating`
  ```

Dequeue Protocol: Take 1

- At isolation levels below SERIALIZABLE, this protocol won't work correctly if there are concurrent sessions dequeuing
- At SERIALIZABLE level, at most one such overlapping session can succeed (worst case workload for SERIALIZABLE)

Dequeue Protocol: Take 1

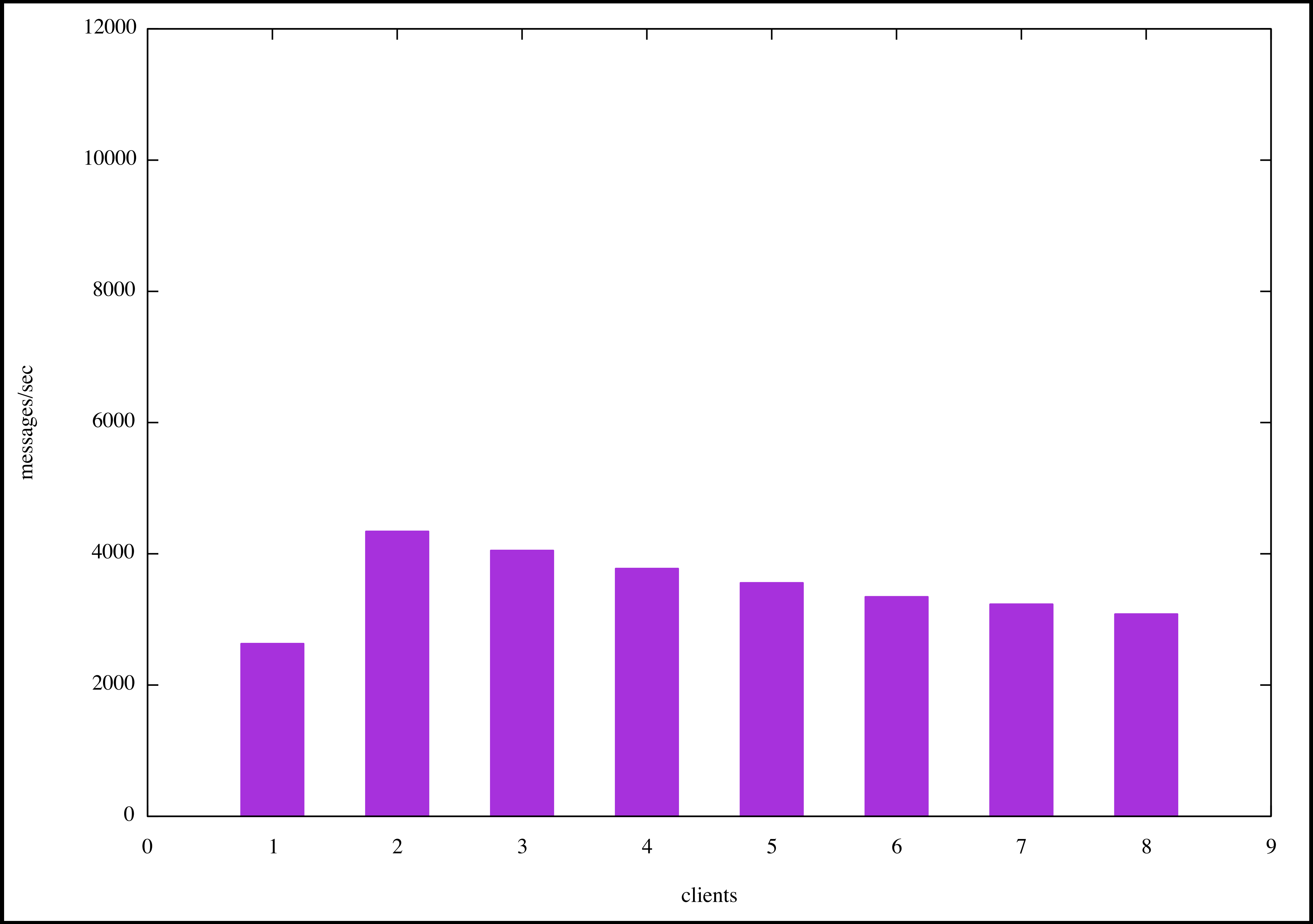
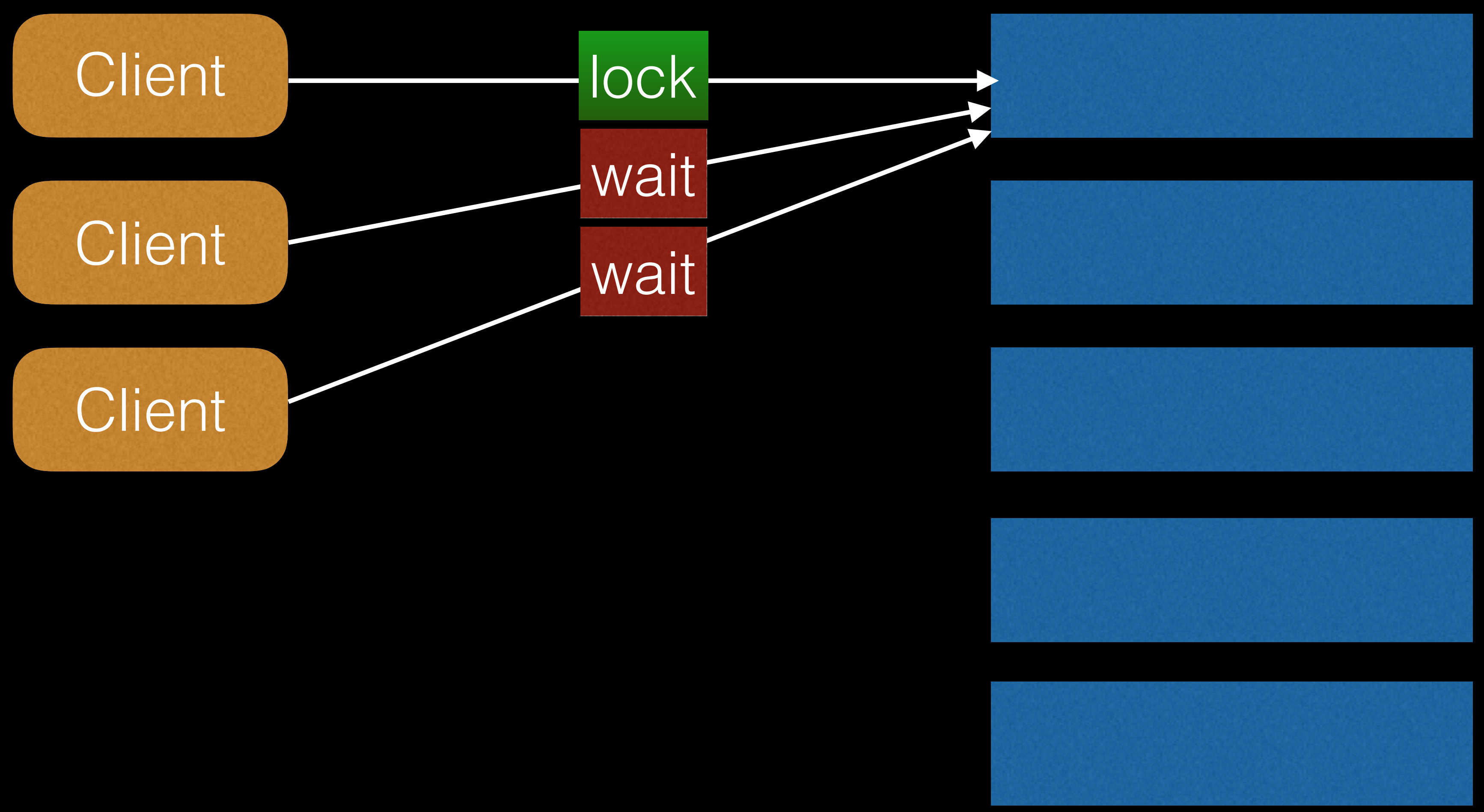


Dequeue Protocol: Take 2

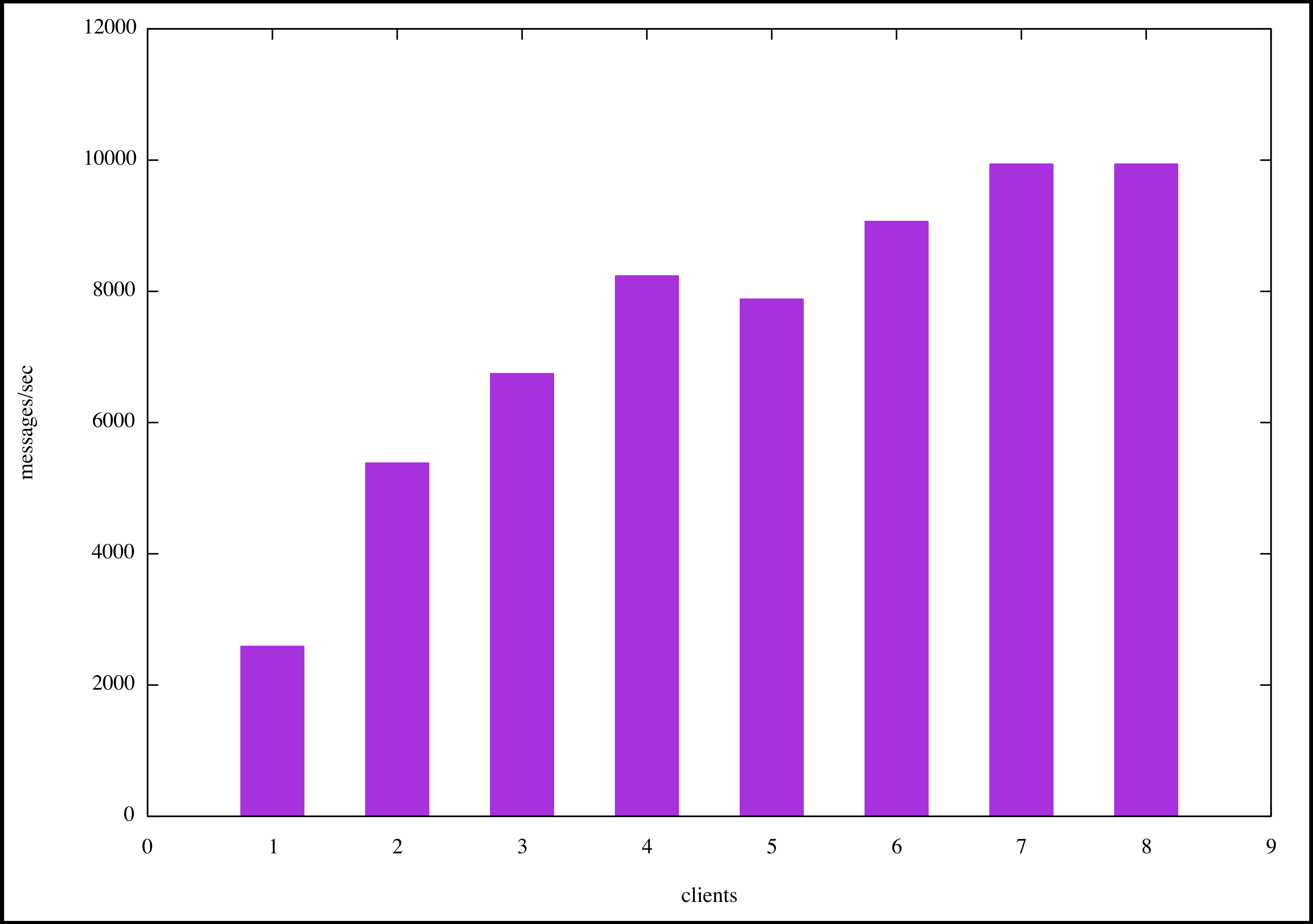
- `LISTEN sms_queue_broadcast;`

- ```
  BEGIN;
  SELECT message_uuid, destination, body
    FROM sms_queue
     FOR UPDATE
   ORDER BY insert_time
   LIMIT 1;
  — if found, do something (internal or
  — external + idempotent) and then:
  DELETE FROM sms_queue
   WHERE message_uuid = $1;
  COMMIT;
  ```

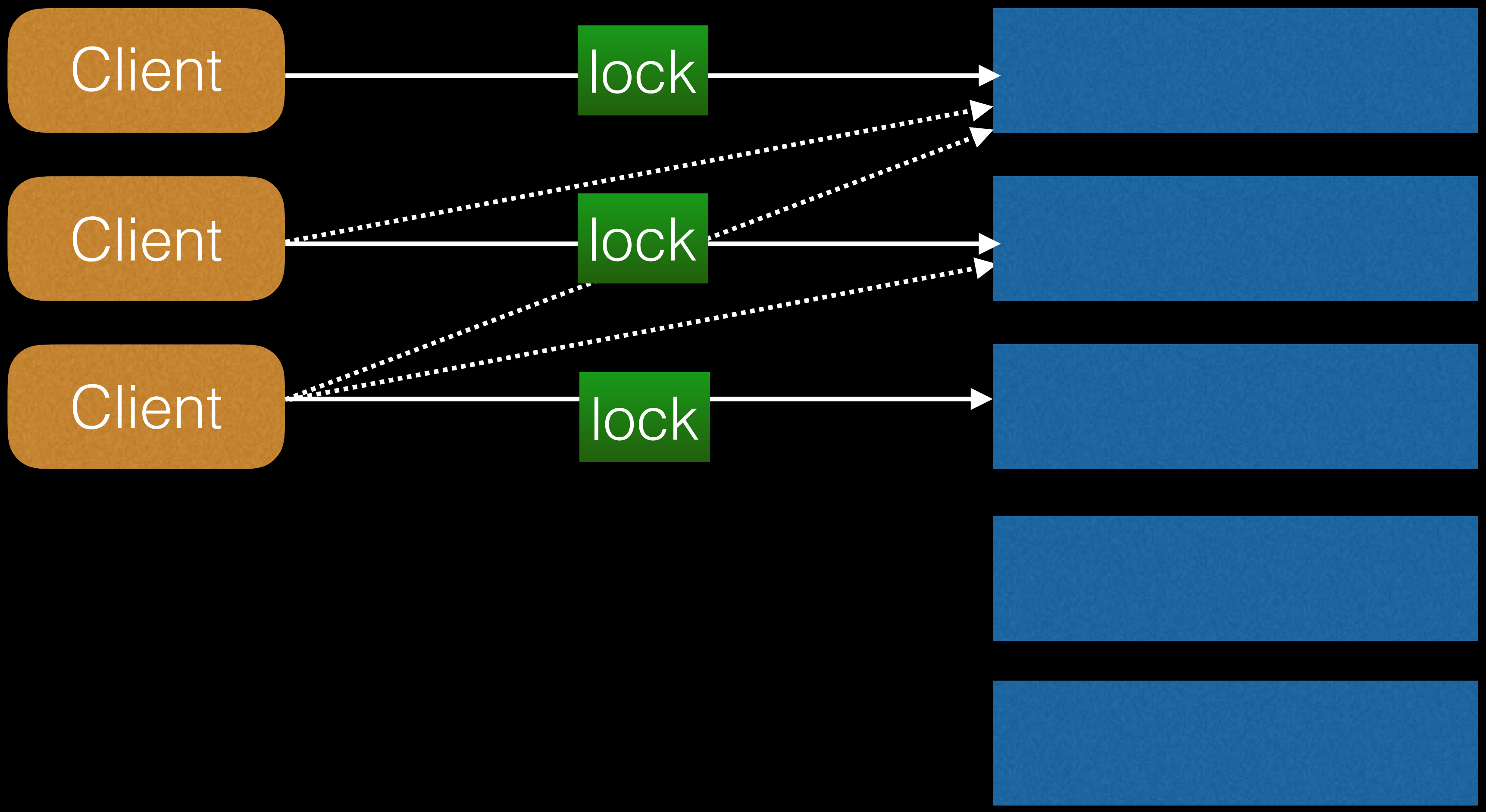
- `— repeat previous step until nothing found`

- `— wait for notifications before repeating`
  ```

Dequeue Protocol: Take 2



Dequeue Protocol: Take 2



Dequeue Protocol: Take 3

- `LISTEN sms_queue_broadcast;`
- ```
  BEGIN;
  SELECT message_uuid, destination, body
    FROM sms_queue
     FOR UPDATE SKIP LOCKED
   ORDER BY insert_time
   LIMIT 1;
  — if found, do something (internal or
  — external + idempotent) and then:
  DELETE FROM sms_queue
   WHERE message_uuid = $1;
  COMMIT;
  ```
- `— repeat previous step until nothing found`
- `— wait for notifications before repeating`

In PostgreSQL 9.4 and earlier which don't have SKIP LOCKED, use pg_try_advisory_lock(x) in the WHERE clause, where x is somehow derived from the message ID
  ```

Dequeue Protocol: Take 3



Dequeue Protocol: Take 3



Dequeue Protocol: Take 3

- The `ORDER BY` clause is still controlling the time we **start** processing each item, but no longer controlling the order we commit
- Dequeuing transactions that roll back cause further perturbation of the processing order
- Looser ordering is good for concurrency while still approximately fair to all messages
- Stricter ordering is needed for some replication-like workloads with a semantic dependency between messages

What’s a Queue?

SQL

Example Use Cases

Implementation

Problems

What Could We Do Better?

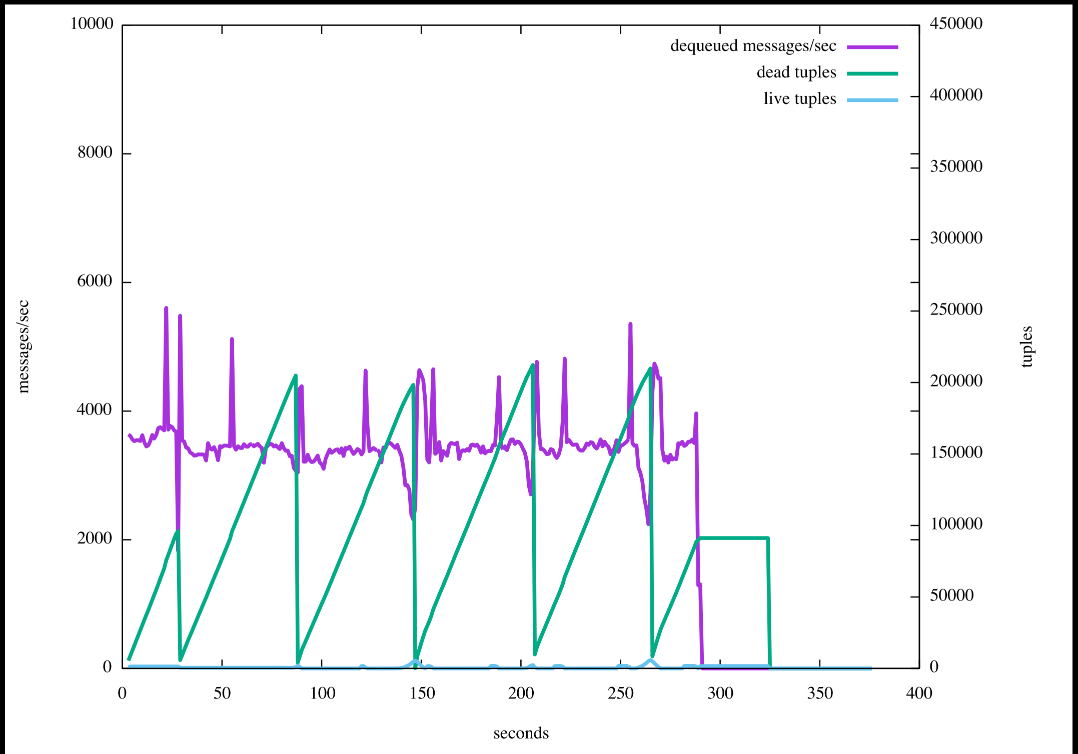
Resilience

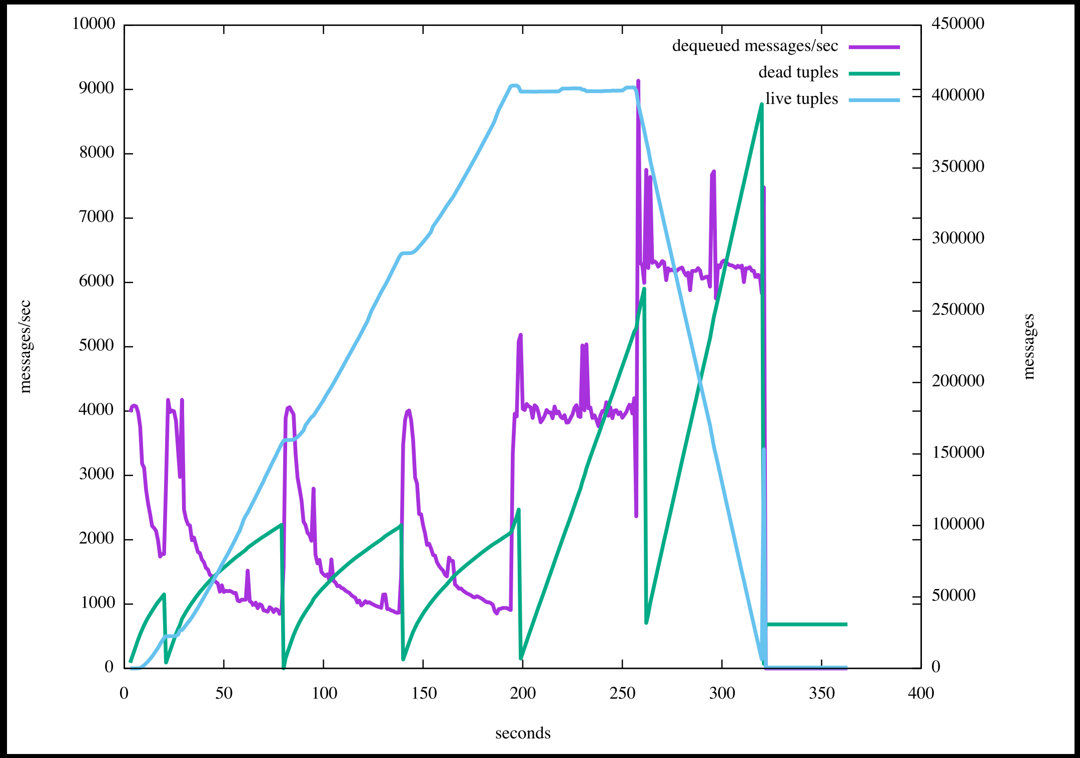
- The protocol discussed so far has messages which are locked, worked on and then deleted in the same transaction is simple, but doesn't help us manage failures very conveniently
- Some ideas for improvement, depending on requirements:
  - Handle failure of external systems by incrementing a retry counter on a message and giving up on messages after some maximum number of retries
  - Prevent such retries from happening too fast by setting a time column to a future time when incrementing message, which the dequeue operation should respect
  - Resilience against crashing or hanging workers is trickier because we can't increment a retry count in an transaction that never commits; one approach is to have one transaction update a message state, and then do the real work in a separate transaction — this requires a protocol for cleaning up/stealing work items if they aren't completed within a time frame

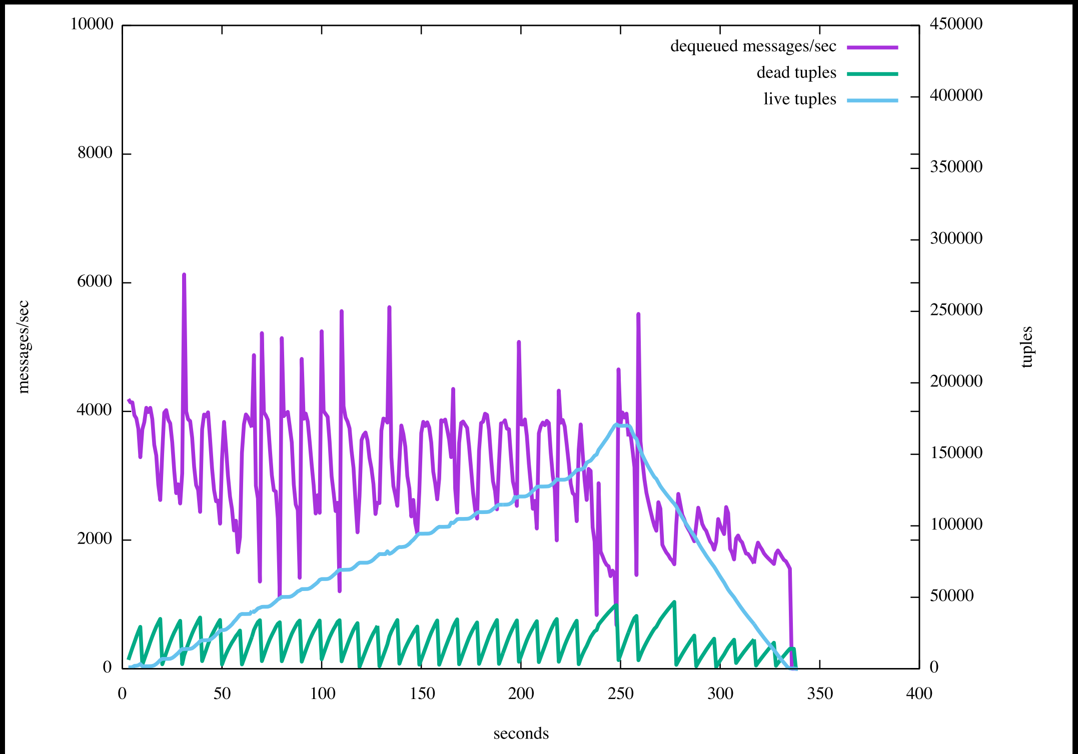
Some Other Considerations

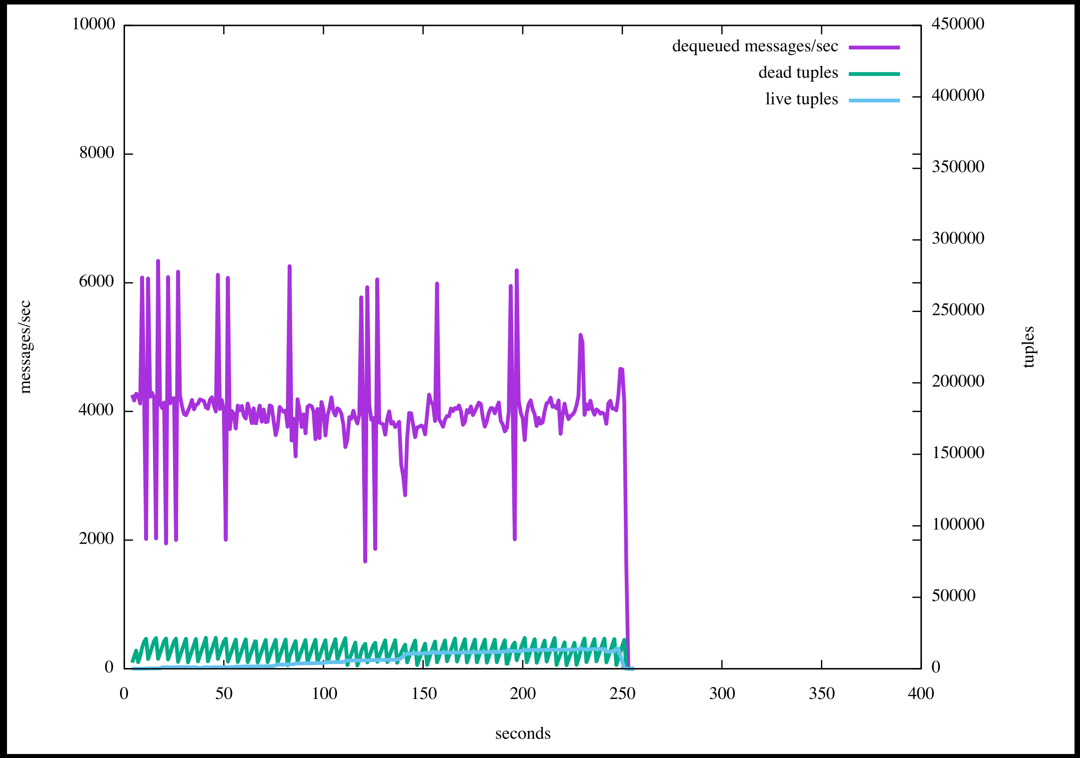
- Watch out for ID space running out (32 bit integers)
- If using a SEQUENCE to generate a strict order, be careful of cycling and be aware of behaviour when transactions overlap
- Btrees not correlated with insert/delete order can develop a lot of bloat in high churn tables
- Statistics for volatile tables might cause trouble (CF DB2 VOLATILE)
- If there is no ordering requirement at all, in theory you might not even need an index on a queue table (you could use ctid to refer to arbitrarily selected locked rows)
- Default vacuum settings may be insufficient, depending on your workload, leading to bloat and unstable performance

Vacuuming


dequeued messages/sec
dead tuples
live tuples
messages/sec
tuples
seconds


dequeued messages/sec
dead tuples
live tuples
messages/sec
messages
seconds


dequeued messages/sec
dead tuples
live tuples
messages/sec
tuples
seconds


dequeued messages/sec
dead tuples
live tuples
messages/sec
tuples
seconds

What’s a Queue?
SQL
Example Use Cases
Performance
Problems
What Could We Do Better?

Notifications

- It would be nice to have a new wait/notify feature that could handle ‘broadcast’ like NOTIFY, but also ‘notify one’: to avoid stampedes of otherwise idle workers when only one item has been enqueued

- It might be better to do that with a blocking ‘wait’ function rather than the NOTIFY asynchronous message approach (?)

UNDO

- UNDO-log based MVCC should provide continuous recycling of space, avoiding bloat and giving smoother performance

- … but no doubt bring new problems, and be extremely difficult to build

Serializable

- Queue-like workloads are the worst case for SERIALIZABLE
- The executor could in theory consider returning tuples in a different order when there is a LIMIT, no [complete] ORDER BY, and another transaction has SIREAD locks on a tuple being returned
- Perhaps this could reduce conflicts in such workloads, allowing higher throughput without giving up the benefits of SERIALIZABLE

<EOF>