Problems that Occur when Multiple Things Use a Database at the Same Time

… and some suggested solutions and workarounds.

A Quick Caveat.

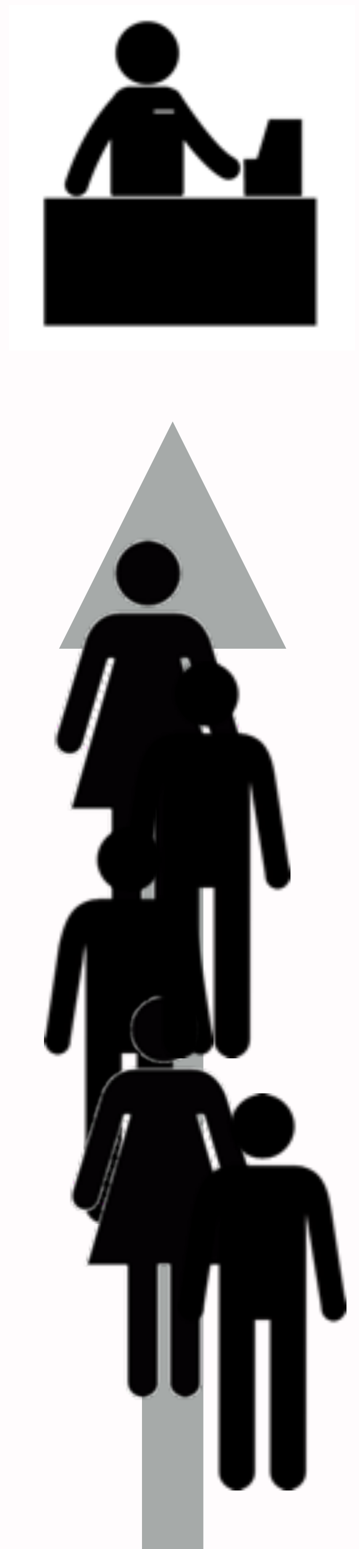
A Quick Caveat.

The Problem

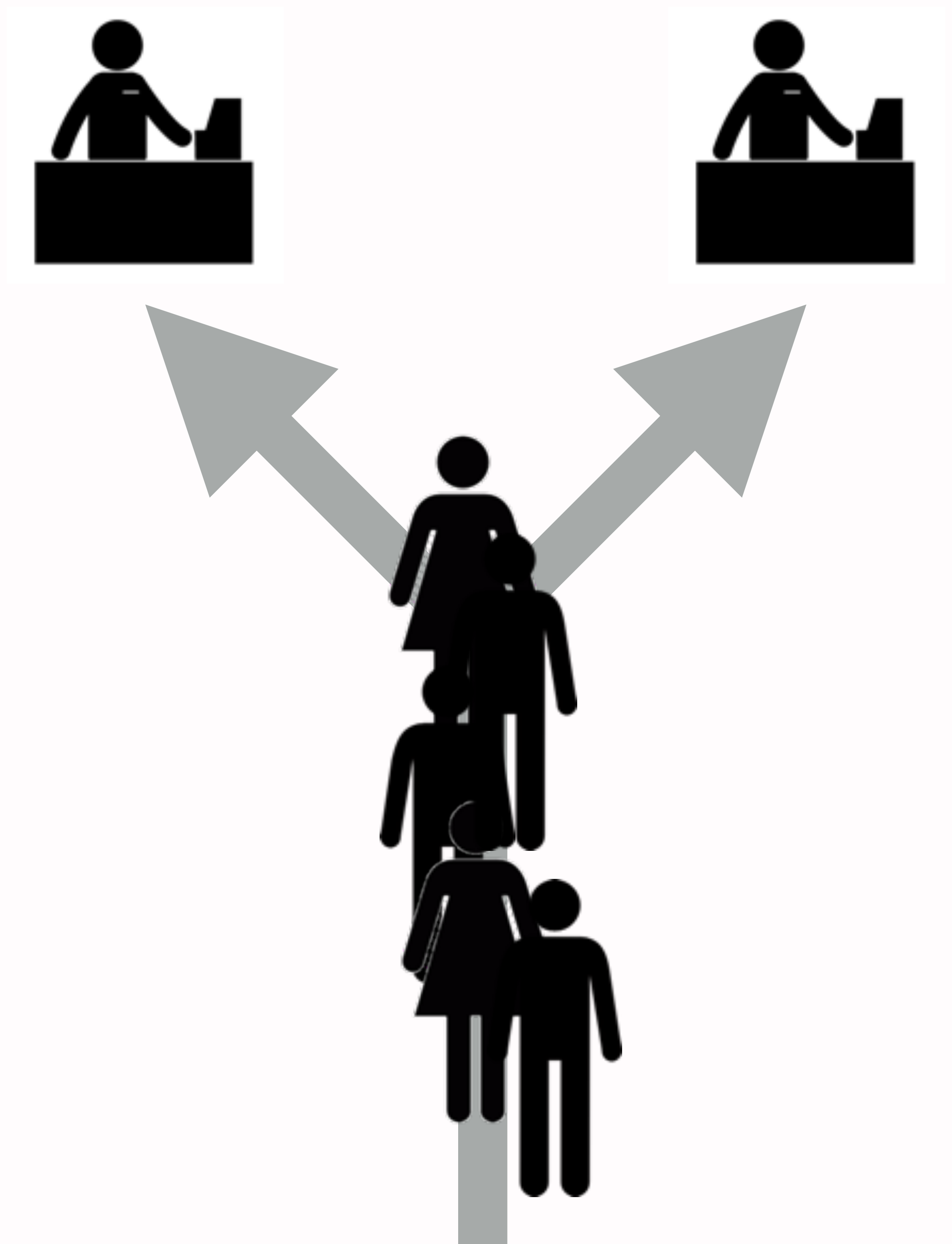
An example.

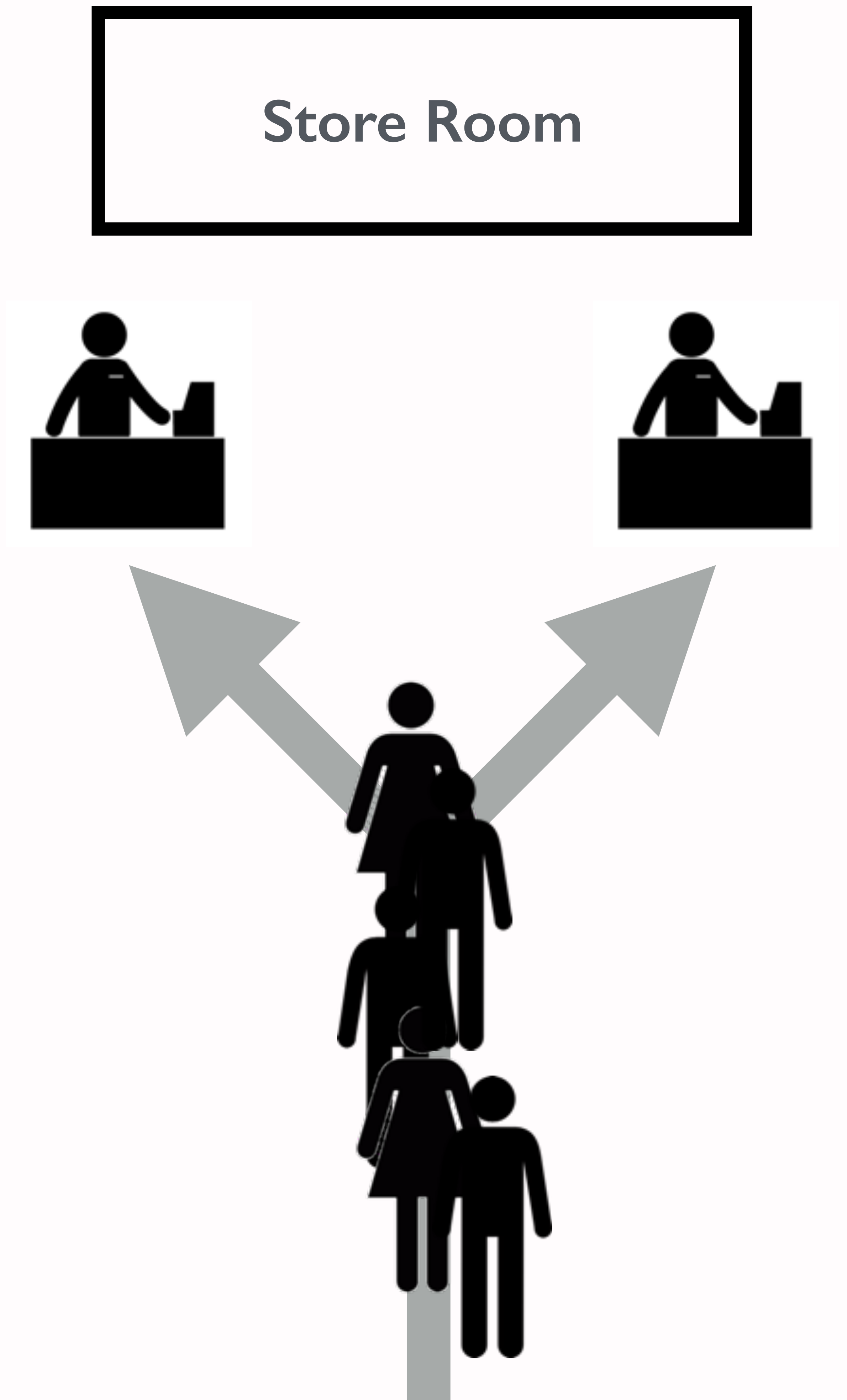
An example.

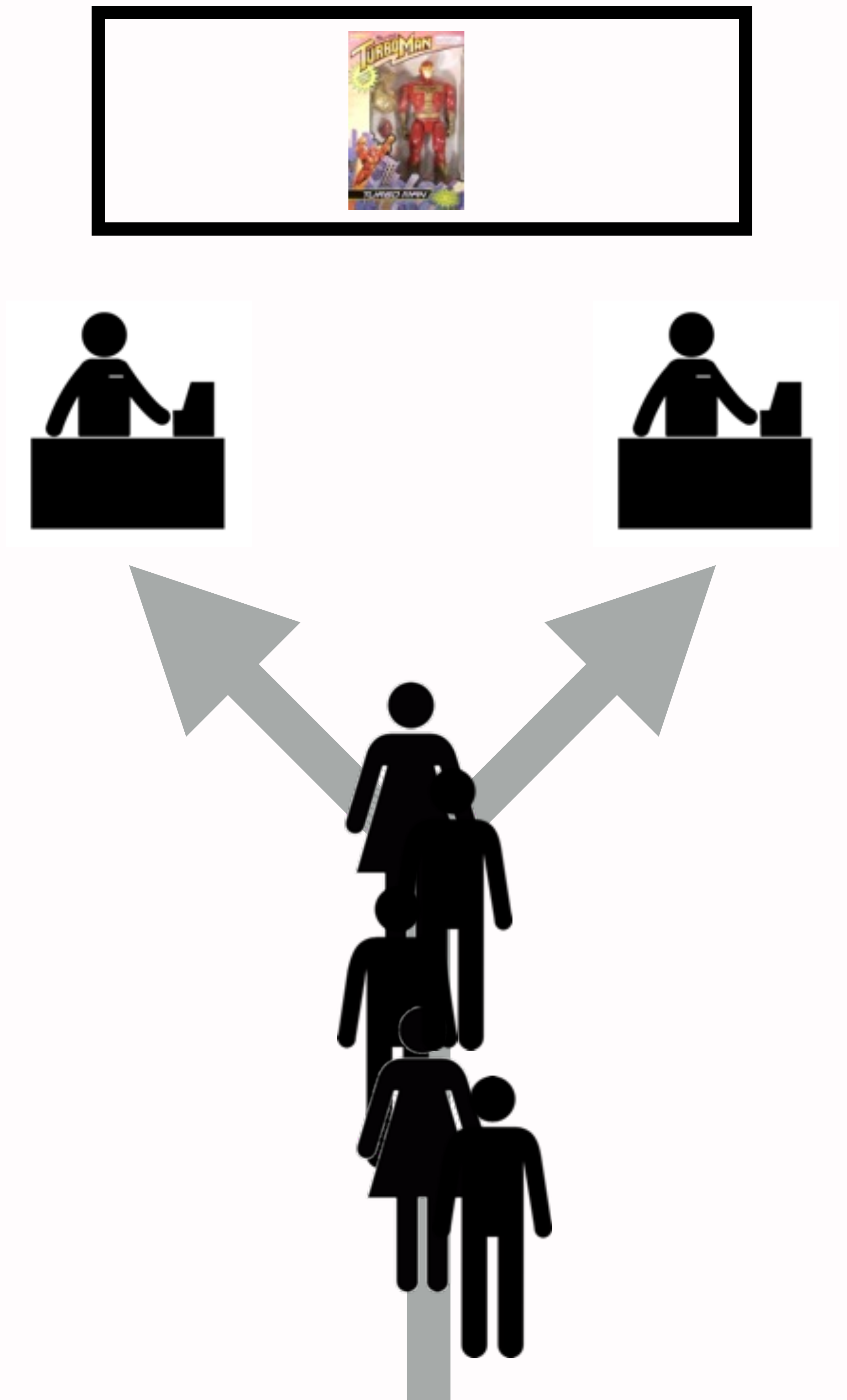
An example.

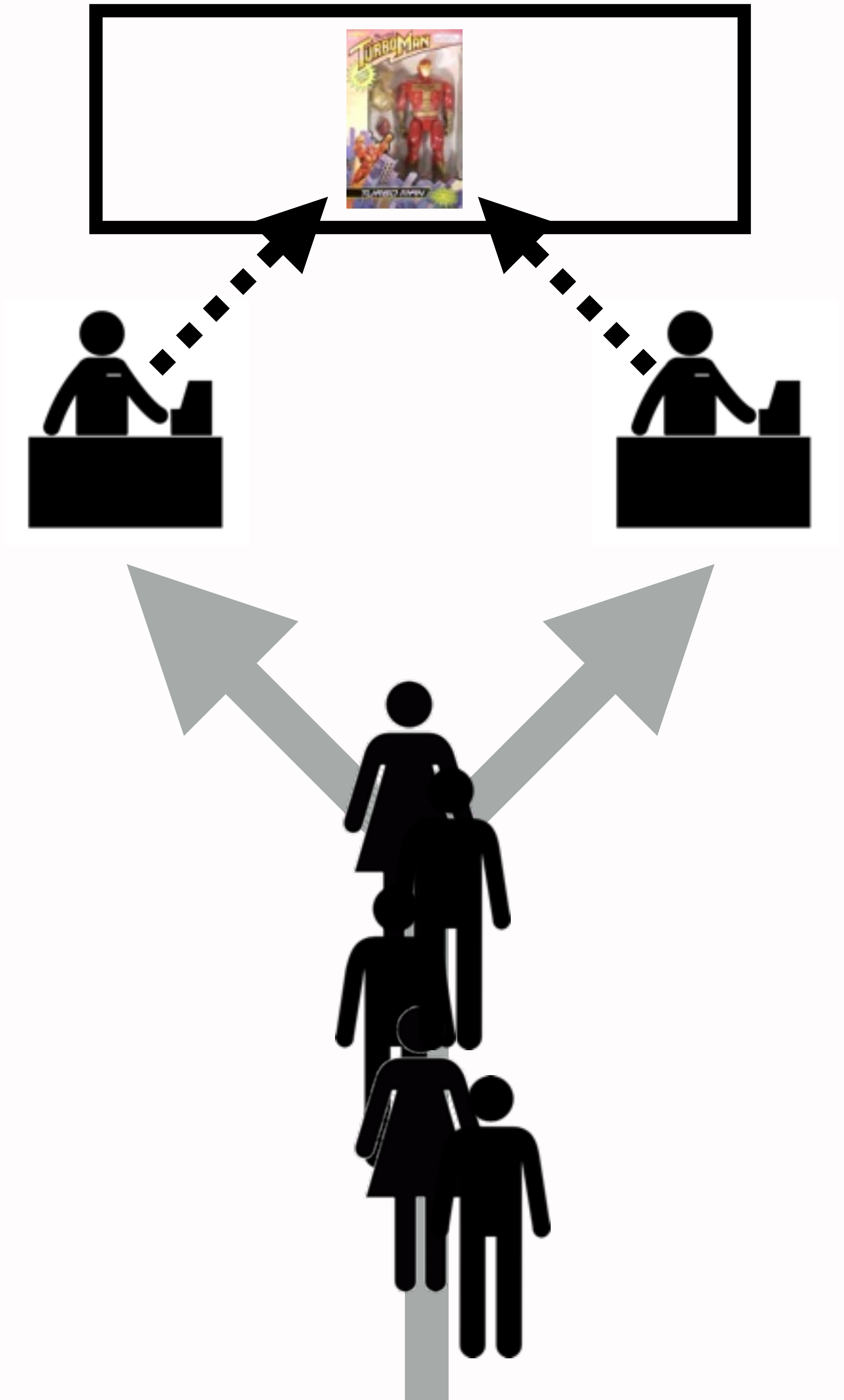


An example.




Store Room


TURBO MAN


TURBO MAN

An example.

An example.

```
p = Post.find(123)
```

```
SELECT *
FROM posts
WHERE id = 123;
```

An example.

```
p = Post.find(123)   SELECT *
                     FROM posts
                     WHERE id = 123;


p.name = "Wow"
p.save               UPDATE posts
                     SET name = "Wow"
                     WHERE id = 123;
```

An example.

Something can happen in here →

```
SELECT *
FROM posts
WHERE id = 123;


UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

Two examples at once.

A

B

```
SELECT *
FROM posts
WHERE id = 123;
```

A

```
SELECT *
FROM posts
WHERE id = 123;
```

B

```
SELECT *
FROM posts
WHERE id = 123;
```

A

```
SELECT *
FROM posts
WHERE id = 123;


UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

B

```
SELECT *
FROM posts
WHERE id = 123;
```

A

```
SELECT *
FROM posts
WHERE id = 123;

UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

B

```
SELECT *
FROM posts
WHERE id = 123;

UPDATE posts
SET name = "Nope"
WHERE id = 123;
```

A

```
SELECT *
FROM posts
WHERE id = 123;


UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

B

```
SELECT *
FROM posts
WHERE id = 123;


UPDATE posts
SET name = "Nope"
WHERE id = 123;
```

Winner: B

```
{
  id: 123,
  name: "nope",
  body: "........"
}
```

ATOMICITY

Atoms

```
p = Post.find(123)    SELECT *
                      FROM posts
                      WHERE id = 123;

p.name = "Wow"
p.save                UPDATE posts
                      SET name = "Wow"
                      WHERE id = 123;
```

Atoms

```
p = Post.find(123)
```

```
SELECT *
FROM posts
WHERE id = 123;
```

```
p.name = "Wow"
p.save
```

```
UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

Atoms

```
p = Post.find(123)
```

```
SELECT *
FROM posts
WHERE id = 123;
```

```
p.name = "Wow"
p.save
```

```
UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

Atoms

```
p = Post.find(123)
```

```
SELECT *
FROM posts
WHERE id = 123;
```

```
p.name = "Wow"
p.save
```

```
UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

Making Database Interactions Atomic

Three Ways

Three Ways

1. UPDATE a column's value based on its *current* value.
    - Get the database to figure out the new value. Don't assume we know what the value is in advance.

1) Push it to the DB.

1) Push it to the DB.

```
p = Post.find(123)
```

```
SELECT *
FROM posts
WHERE id = 123;
```

1) Push it to the DB.

```
p = Post.find(123)
```

```
SELECT *
FROM posts
WHERE id = 123;
```

```
p.increment(
  :views, 1
)
```

```
UPDATE posts
SET views =
 COALESCE(views, 0)
 + 1
WHERE id = 123;
```

1) Push it to the DB.

```
Post.increment_counter(
  :views, 1
)
```

```
UPDATE posts
SET views =
 COALESCE(views, 0)
 + 1
WHERE id = 123;
```

1) Push it to the DB.

```
p = Post.find(123)

p.tags << "a new tag"

# Rails 4 w/ Postgres
# and a DB migration,
# eg.
# t.string(
#   :tags,
#   array: true
# )
```

```
-- SELECT ...

UPDATE posts
SET tags =
 array_append(
   tags,
   "a new tag"
  )

WHERE id = 123;
```

Three Ways

1. UPDATE a column's value based on its *current* value.
   - Get the database to figure out the new value. Don't assume we know what the value is in advance.

2. Add conditions to the UPDATE.
   - Only update if our assumptions are true.

2) Add Conditions to UPDATE.

2) Add Conditions to UPDATE.

```ruby
p = Post.find(123)

Post.where(
 id: 123,
 name: p.name,
).update_all(
 name: "Wow",
)

# => 1 means it worked
# => 0 means it didn't
```

```sql
-- SELECT ...

UPDATE posts
SET
 name = "Wow"
WHERE
 id = 123 AND
 name = "Old Name";
```

2) Add Conditions to UPDATE.

```
# Database Migration
add_column :posts,
  :lock_version,
  :integer
```

2) Add Conditions to UPDATE.

```
# Database Migration
add_column :posts,
  :lock_version,
  :integer

# Form View
<%= form.hidden_field
    :lock_version %>
```

2) Add Conditions to UPDATE.

```
add_column :posts,
  :lock_version,
  :integer
```

```
<%= form.hidden_field :lock_version %>
```

2) Add Conditions to UPDATE.

```
add_column :posts,
  :lock_version,
  :integer
```

```
<%= form.hidden_field :lock_version %>
```

```
# Controller
pp = post_params
p = Post.find(pp[:id])
p.lock_version =
   pp[:lock_version]
p.name = pp[:name]
p.save
```

2) Add Conditions to UPDATE.

```
add_column :posts,
  :lock_version,
  :integer
```

```
<%= form.hidden_field :lock_version %>
```

```
# Controller
pp = post_params
p = Post.find(pp[:id])       -- SELECT ...
p.lock_version =
   pp[:lock_version]
p.name = pp[:name]
p.save
```

2) Add Conditions to UPDATE.

```
add_column :posts,
  :lock_version,
  :integer
```

```
<%= form.hidden_field :lock_version %>
```

```
# Controller
pp = post_params
p = Post.find(pp[:id])   -- SELECT ...
p.lock_version =
    pp[:lock_version]
p.name = pp[:name]
p.save
```

2) Add Conditions to UPDATE.

```
add_column :posts,
  :lock_version,
  :integer
```

```
<%= form.hidden_field :lock_version %>
```

```
# Controller
pp = post_params
p = Post.find(pp[:id])
p.lock_version =
   pp[:lock_version]
p.name = pp[:name]
p.save
```

```
-- SELECT ...
UPDATE posts
SET
 name = "Wow",
 lock_version = 2
WHERE
 id = 123 AND
 lock_version = 1;
```

2) Add Conditions to UPDATE.

- Problematic...?
  - The "did it update or not?" counter is too coarse.
  - You get another race condition: does the record still exist? Can't differentiate between Stale and Gone.
- But you can (in most cases) just ask the DB again to check.

Three Ways

1. UPDATE a column's value based on its *current* value.
   - Get the database to figure out the new value. Don't assume we know what the value is in advance.
2. Add conditions to the UPDATE.
   - Only update if our assumptions are true.
3. Put everything inside a container.
   - Suddenly the *container* is the atom.

3) Put actions inside a container.

Atoms

```
p = Post.find(123)
```

```
SELECT *
FROM posts
WHERE id = 123;
```

```
p.name = "Wow"
p.save
```

```
UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

Atoms

```
p = Post.find(123)
```

```
SELECT *
FROM posts
WHERE id = 123;
```

```
p.name = "Wow"
p.save
```

```
UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

"Just Add a Transaction"

```
ActiveRecord::Base.transaction do |t|

  p = Post.find(123) # SELECT ...

  p.name = "Wow"
  p.save             # UPDATE ...

end
```

220g NET
Cottee's
SETS HARD IN SECONDS
BEGIN TRANSACTION
chocolate
FLAVOURED ICE CREAM COATING
PX17341F/A
Serving Suggestion

A

```
SELECT *
FROM posts
WHERE id = 123;

UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

B

```
SELECT *
FROM posts
WHERE id = 123;

UPDATE posts
SET name = "Nope"
WHERE id = 123;
```

A

```
SELECT *
FROM posts
WHERE id = 123;


UPDATE posts
SET name = "Wow"
WHERE id = 123;
```

B

```
SELECT *
FROM posts
WHERE id = 123;


UPDATE posts
SET name = "Nope"
    E id = 123;
```

3) Put actions inside a container.

3) Put actions inside a container.

- Locking (rows, tables, advisory)

3) Put actions inside a container.

- Locking (rows, tables, advisory)
- Transaction Isolation Levels

Locks

- Locking a particular row.
- Locking an entire table.
- Arbitrary application-level locks.

Locks: SELECT FOR UPDATE

```
p = Post.lock(true).find(123)
# SELECT ... FOR UPDATE

p.name = "Foo"
p.save
# UPDATEs, COMMITs
```

Locks: Table Lock

- The nuclear options; blocks access entirely.

```
Post.connection.execute(
  'LOCK TABLE posts IN ... MODE'
);
```

Locks: Application Lock

https://github.com/heroku/pg_lock

```
post = Post.new(...)
key = "#{Post}-#{post.author}"
PgLock.new(name: key).lock do
  # Stuff, eg. call post.save
  # to insert the new Post.
end
```

Isolation Levels

- Read Uncommitted
- Read Committed **(the default)**
- Repeatable Read
- Serializable

(Good reference: http://www.postgresql.org/docs/9.1/static/transaction-iso.html)

Isolation Levels

- Read Uncommitted
- Read Committed **(the default)**
- Repeatable Read
- Serializable

(Good reference: http://www.postgresql.org/docs/9.1/static/transaction-iso.html)

Isolation Level: Default

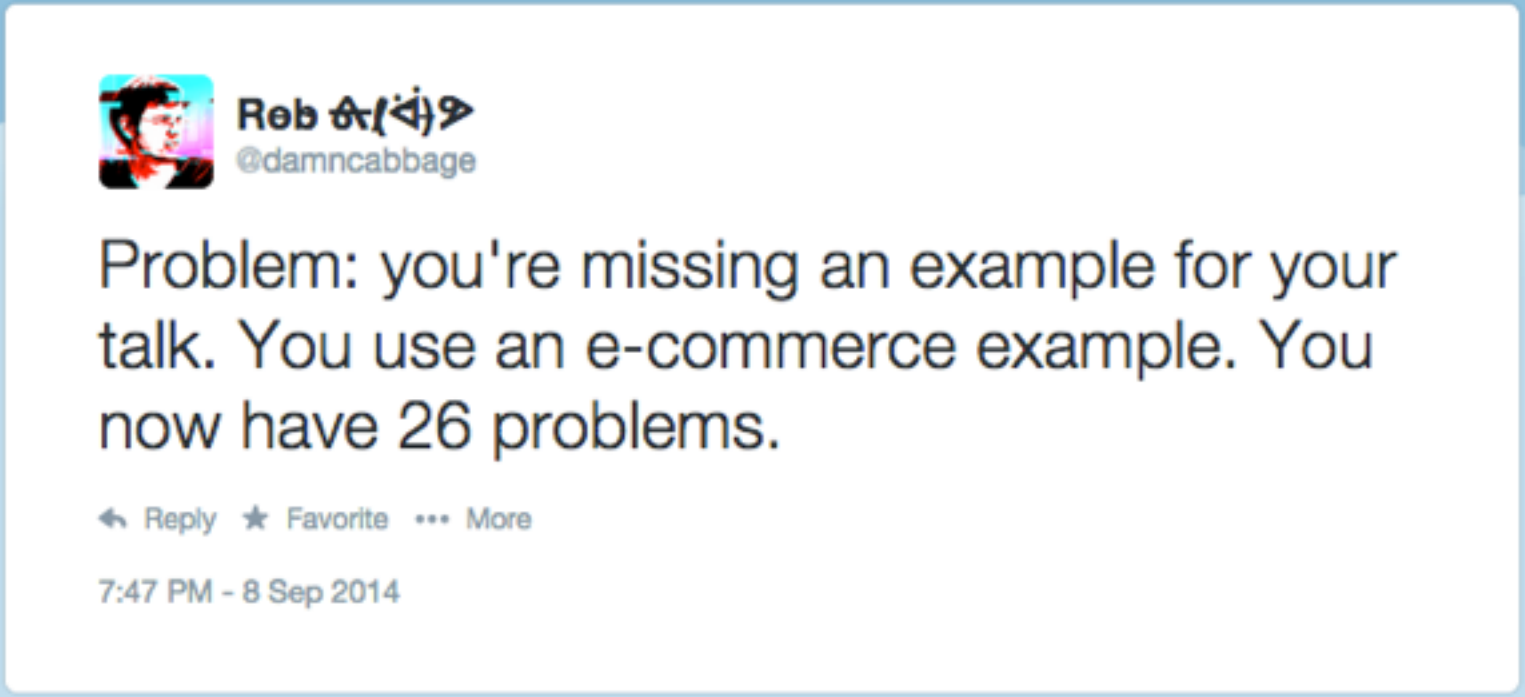
```
ActiveRecord::Base.transaction do |t|

  p = Post.find(123) # SELECT ...

  p.name = "Wow"
  p.save             # UPDATE ...

end
```

Isolation Level: Maximum

```
Post.transaction(
  isolation: :serializable
) do |t|

  p = Post.find(123) # SELECT ...

  p.name = "Wow"
  p.save             # UPDATE ...

end
```

Summin' up.

- In-database modifications where possible.
- Isolation or Locks where not.
- The more restrictions, the slower it's gonna go.
- More work, but makes the problem **visible**.



Rɵb ᕦ(ᐛ)ᕤ
@damncabbage

Problem: you're missing an example for your talk. You use an e-commerce example. You now have 26 problems.

7:47 PM - 8 Sep 2014

Fin.

Rob Howard
@damncabbage