SQL, NoSQL and Beyond

Lorna Jane Mitchell, IBM

Slides: https://lornajane.net/resources

Beyond MySQL

MySQL is great!

If you're ready for something different, how about:

- PostgreSQL
- Redis
- CouchDB

PostgreSQL

About PostgreSQL

Homepage: https://www.postgresql.org/

- Open source project
- Powerful, relational database

PostgreSQL Myths and Surprises

Myth 1: PostgreSQL is more complicated than MySQL

PostgreSQL Myths and Surprises

Myth 1: PostgreSQL is more complicated than MySQL

Not true. They are both approachable from both CLI and other web/GUI tools, PostgreSQL has the best CLI help I've ever seen.

PostgreSQL Myths and Surprises

Myth 1: PostgreSQL is more complicated than MySQL

Not true. They are both approachable from both CLI and other web/GUI tools, PostgreSQL has the best CLI help I've ever seen.

Myth 2: PostgreSQL is more strict than MySQL

PostgreSQL Myths and Surprises

Myth 1: PostgreSQL is more complicated than MySQL

Not true. They are both approachable from both CLI and other web/GUI tools, PostgreSQL has the best CLI help I've ever seen.

Myth 2: PostgreSQL is more strict than MySQL

True! But standards-compliant is a feature IMO

PostgreSQL Myths and Surprises

Myth 1: PostgreSQL is more complicated than MySQL

Not true. They are both approachable from both CLI and other web/GUI tools, PostgreSQL has the best CLI help I've ever seen.

Myth 2: PostgreSQL is more strict than MySQL

True! But standards-compliant is a feature IMO

Myth 3: PostgreSQL is slower than MySQL for simple things

PostgreSQL Myths and Surprises

Myth 1: PostgreSQL is more complicated than MySQL

Not true. They are both approachable from both CLI and other web/GUI tools, PostgreSQL has the best CLI help I've ever seen.

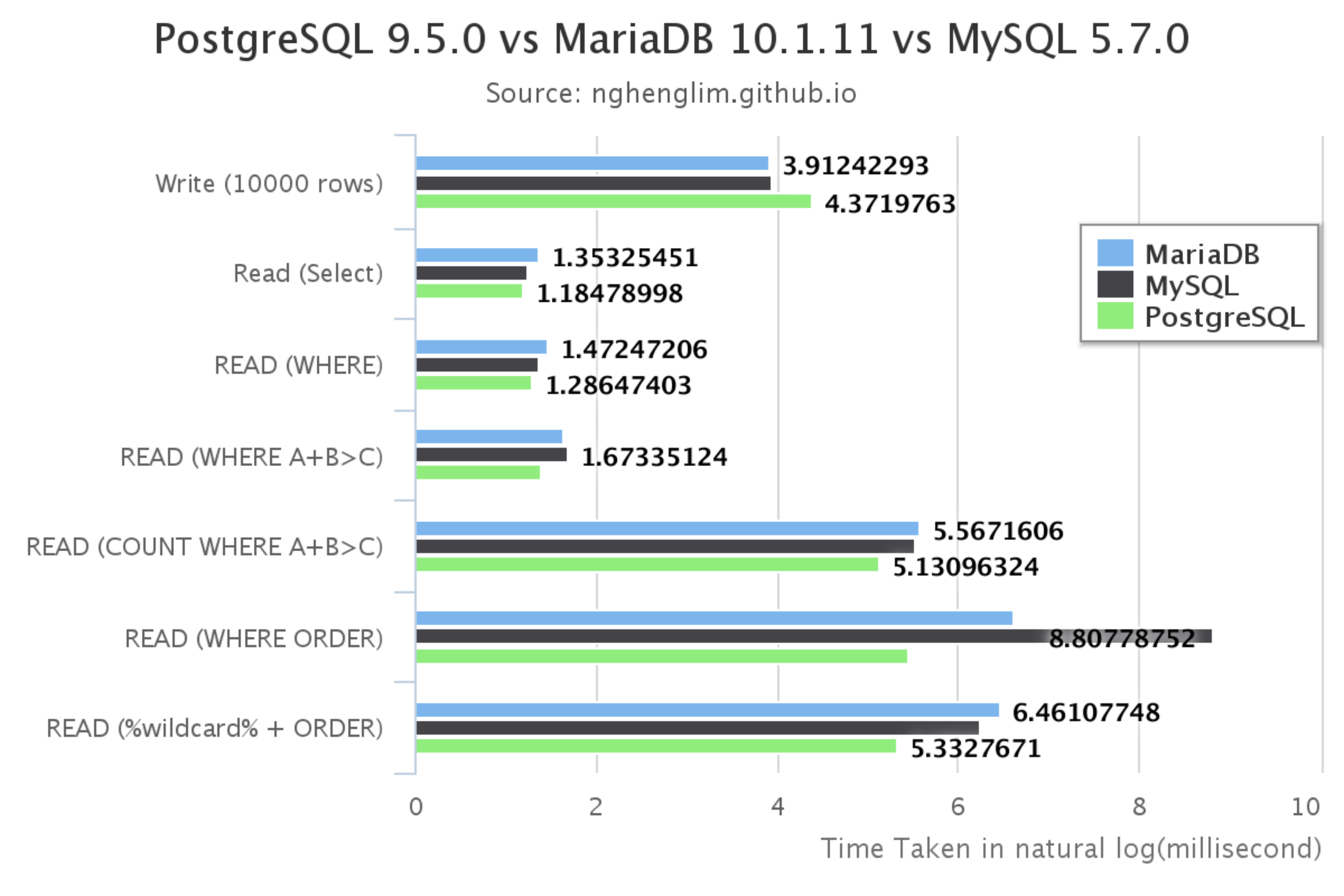
Myth 2: PostgreSQL is more strict than MySQL

True! But standards-compliant is a feature IMO

Myth 3: PostgreSQL is slower than MySQL for simple things

Not true. PostgreSQL has better query planning so is likely to be faster at everything, and also has more features.

PostgreSQL Performance



Data Types

PostgreSQL has data types to suit more data needs:

- UUID data type to create unique identifiers
- JSON and JSONB for working with JSON data

Data Types: UUID

We can use a UUID as a primary key:

```
CREATE TABLE products (
  product_id uuid primary key default uuid_generate_v4(),
  display_name varchar(255)
);

INSERT INTO products (display_name)
  VALUES ('Jumper') RETURNING product_id;
```

```
              product_id              | display_name
--------------------------------------+--------------
 73089ae3-c0a9-4c0a-8287-e0f6ec41a200 | Jumper
```

RETURNING Keyword

Look at that insert statement again

```
INSERT INTO products (display_name)
  VALUES ('Jumper') RETURNING product_id;
```

The `RETURNING` keyword allows us to retrieve a field in one step - removes the need for a `last_insert_id()` call.

Data Types: JSONB

Add a column to the table to hold attributes

```
ALTER TABLE products ADD COLUMN attrs jsonb;
```

Add some data

```
INSERT INTO products (display_name, attrs) VALUES
('Dress', '{"length": {"value": 61, "units":"inch"},
    "pockets":true, "colour":"teal"}');
```

Data Types: JSONB

We can use the JSON in our WHERE clause

```
SELECT display_name AS product, attrs->>'colour' AS colour
    FROM products
    WHERE attrs->>'pockets' = 'true';
```

```
 product | colour
---------+--------
 Cardi   | red
 Dress   | teal
 Jeans   | indigo
(3 rows)
```

Indexes

Examples might be:

- Primary key ensuring uniqueness
- Some other unique key
- Indexes facilitating fast lookup on one or more columns
- Indexes that use expressions

Indexes: Primary key

Primary keys are always unique

```
CREATE TABLE employees (
    id serial primary key,
    name text
    );
```

The `serial` data type is numeric and incrementing

Indexes: Expressions

Use an expression if you'll use one when fetching data

```
CREATE TABLE employees (
    id serial primary key,
    name text
    );

CREATE INDEX name_idx
    ON employees (lower(name));
```

Common Table Expressions (CTE)

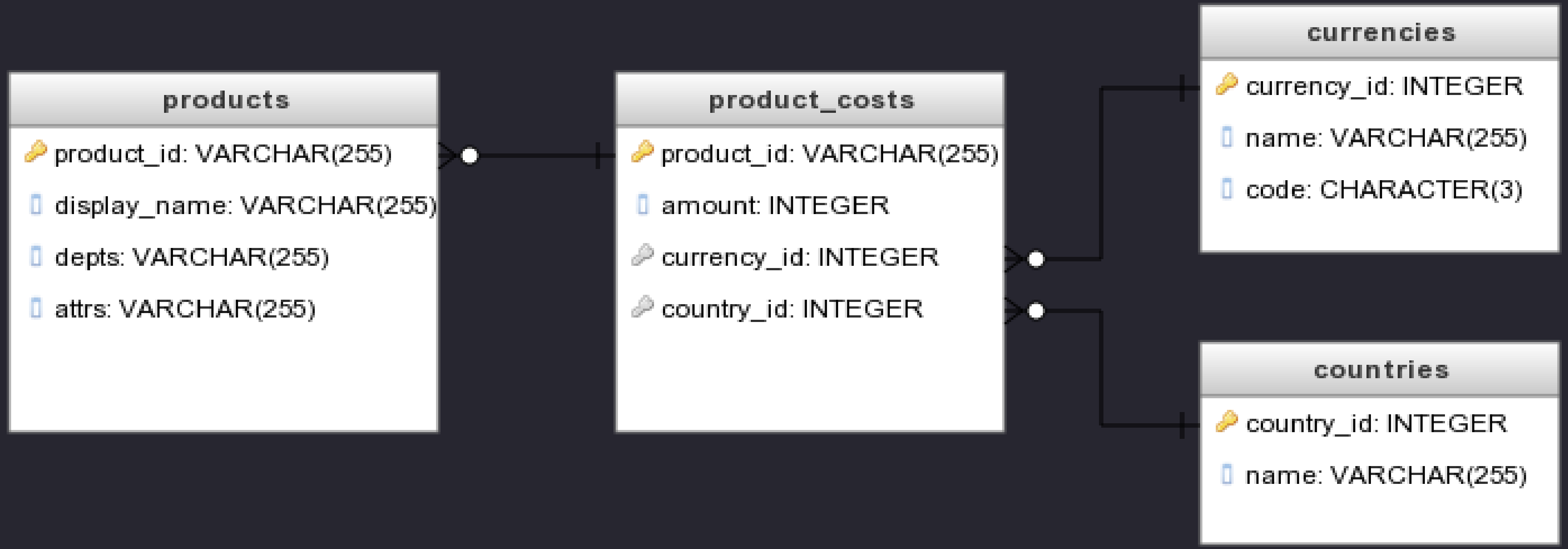
Feature enables declaring extra statements to use later

Moves complexity out of subqueries, making more readable and reusable elements to the query

Syntax:

```
WITH meaningfulname AS
  (subquery goes here joining whatever)
SELECT .... FROM meaningfulname ...
```

Common Table Expressions (CTE)



products
product_id: VARCHAR(255)
display_name: VARCHAR(255)
depts: VARCHAR(255)
attrs: VARCHAR(255)

product_costs
product_id: VARCHAR(255)
amount: INTEGER
currency_id: INTEGER
country_id: INTEGER

currencies
currency_id: INTEGER
name: VARCHAR(255)
code: CHARACTER(3)

countries
country_id: INTEGER
name: VARCHAR(255)

Common Table Expressions (CTE)

```
WITH costs AS
  (SELECT pc.product_id, pc.amount, cu.code, co.name
  FROM product_costs pc JOIN currencies cu USING (currency_id)
  JOIN countries co USING (country_id))
SELECT display_name, amount, code currency, name country
  FROM products JOIN costs USING (product_id);
```

```
display_name | amount | currency | count
-------------+--------+----------+---------
T-Shirt      |     25 | GBP      | UK
T-Shirt      |     30 | EUR      | Italy
T-Shirt      |     29 | EUR      | France
```

Window Functions

Window functions allow us to calculate aggregate values while still returning the individual rows.

e.g. a list of orders, including how many of this product were ordered in total

Window Functions

```
SELECT o.order_id, p.display_name,
  count(*) OVER (PARTITION BY product_id) AS prod_orders
FROM orders o JOIN products p USING (product_id);
```

```
               order_id                | display_name | prod_orders
----------------------------------+--------------+-------------
 74806f66-a753-4e99-aeae-6f947f08 | T-Shirt      |           6
 9ae83b3f-931e-4e6a-a8e3-910dd9ab | Hat          |           3
 0030c58a-122c-4fa5-90f4-231d3848 | Hat          |           3
 3d5a0d76-4c7e-433d-b3cf-2473912d | Hat          |           3
```

PostgreSQL Tips and Resources

- PhpMyAdmin equivalent: https://www.pgadmin.org/
- Best in-shell help I've ever seen (type `\h [something]`)
- JSON features
- Indexes on expression
- Choose where nulls go by adding `NULLS FIRST|LAST` to your `ORDER BY`
- Fabulous support for geographic data http://postgis.net/
- Get a hosted version from https://www.ibm.com/cloud/

Redis

About Redis

Homepage: http://redis.io/

Stands for: REmote DIctionary Service

An open source, in-memory datastore for key/value storage, and much more

Uses of Redis

Usually used in addition to a primary data store for:

- caching
- session data
- simple queues

Anywhere you would use Memcache, use Redis

Redis Feature Overview

- stores strings, numbers, hashes, sets ...
- supports key expiry/lifetime
- very simple protocols, use `redis-cli`
- great monitoring tools

Storing Key/Value Pairs

Store, expire and fetch values.

```
> set risky_feature on
OK
> expire risky_feature 3
(integer) 1
> get risky_feature
"on"
> get risky_feature
(nil)
```

Shorthand for set and expire: `setex risky_feature 3 on`

Storing Hashes

Use a hash for related data (h is for hash, m is for multi)

```
> hmset featured:hat name Sunhat colour white
OK
> hkeys featured:hat
1) "name"
2) "colour"
> hvals featured:hat
1) "Sunhat"
2) "white"
```

Finding Keys in Redis

The SCAN keyword can help us find things

```
127.0.0.1:6379> hset person:lorna twitter lornajane
(integer) 1
127.0.0.1:6379> scan 0 match person:*
1) "0"
2) 1) "person:Lorna"
2) "person:lorna"
127.0.0.1:6379> hscan person:lorna 0
1) "0"
2) 1) "twitter"
2) "lornajane"
```

Queues using Redis Lists

```
> LPUSH todo breakfast
(integer) 1
> LPUSH todo newspaper
(integer) 2

> BRPOP todo 1
1) "todo"
2) "breakfast"
> BRPOP todo 1
1) "todo"
2) "newspaper"
```

Configurable Durability

This is a tradeoff between risk of data loss, and speed.

- by default, redis snapshots (writes to disk) periodically
- the snapshot frequency is configurable by time and by number of writes
- use the `appendonly` log to make redis *eventually durable*

Redis: Tips and Resources

- Replication and clustering are simple!
- Sorted sets
- Supports pub/sub:
  - `SUBSCRIBE comments` then `PUBLISH comments message`
- Excellent documentation http://redis.io/documentation
- Reference card https://dzone.com/refcardz
- For PHP, `predis/predis` from composer or `phpiredis`
- Get a hosted version from https://www.ibm.com/cloud/

CouchDB

About CouchDB

Homepage: http://couchdb.apache.org/

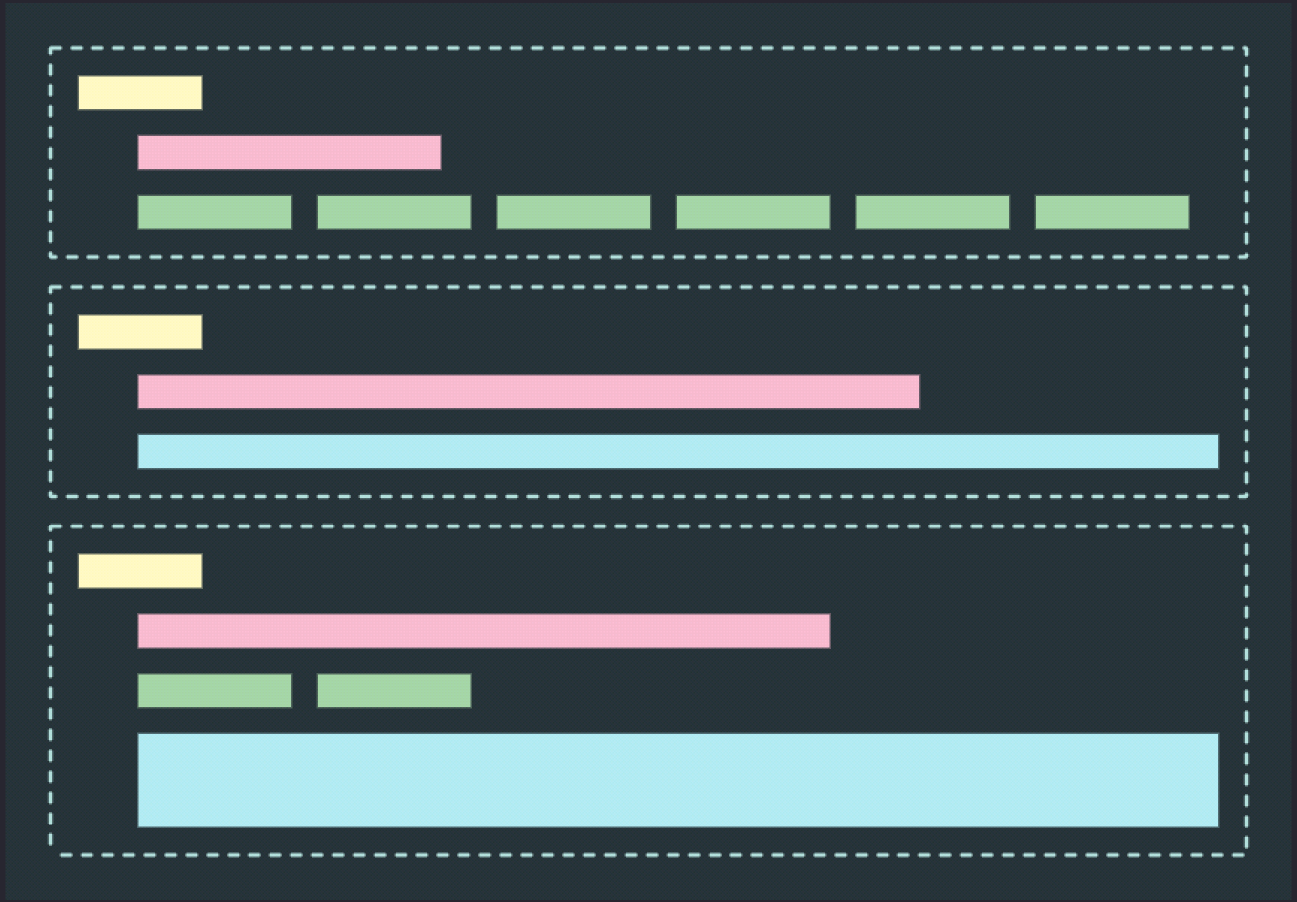
A database built from familiar components

- HTTP interface
- Web interface *Fauxton*
- JS map/reduce views

CouchDB is a NoSQL Document Database

Schemaless Database Design

We can store data of any shape and size



Documents and Versions

When I create a record, I supply an id and it gets a rev:

```
$ curl -X PUT http://localhost:5984/products/1234
  -d '{"type": "t-shirt", "dept": "womens", "size": "L"}'

{"ok":true,"id":"1234","rev":"1-bce9d948a37e72729e689145286fd3ee"}
```

(alternatively, POST and CouchDB will generate the id)

Update Document

CouchDB has awesome consistency management

To update a document, supply the rev:

```
$ curl -X PUT http://localhost:5984/products/1234
  -d '{"_rev": "1-bce9d948a37e72729e689145286fd3ee",
  "type": "t-shirt", "dept": "womens", "size": "XL"}'

{"ok":true,"id":"1234","rev":"2-4b8a7e1bde15d4003aca1517e96d6cfa"}
```

Changes API

Get a full list of newest changes since you last asked

http://localhost:5984/products/_changes?since=7

```
~ $ curl http://localhost:5984/products/_changes?since=7
{"results":[
{"seq":9,"id":"123",
    "changes":[{"rev":"2-7d1f78e72d38d6698a917f8834bfb5f8"}]}
],
```

Polling/Long polling or continuous change updates are available, and they can be filtered.

Replication

CouchDB has the best database replication options imaginable:

- ad-hoc or continuous
- one directional or bi directional
- conflicts handled safely (best fault tolerance ever)

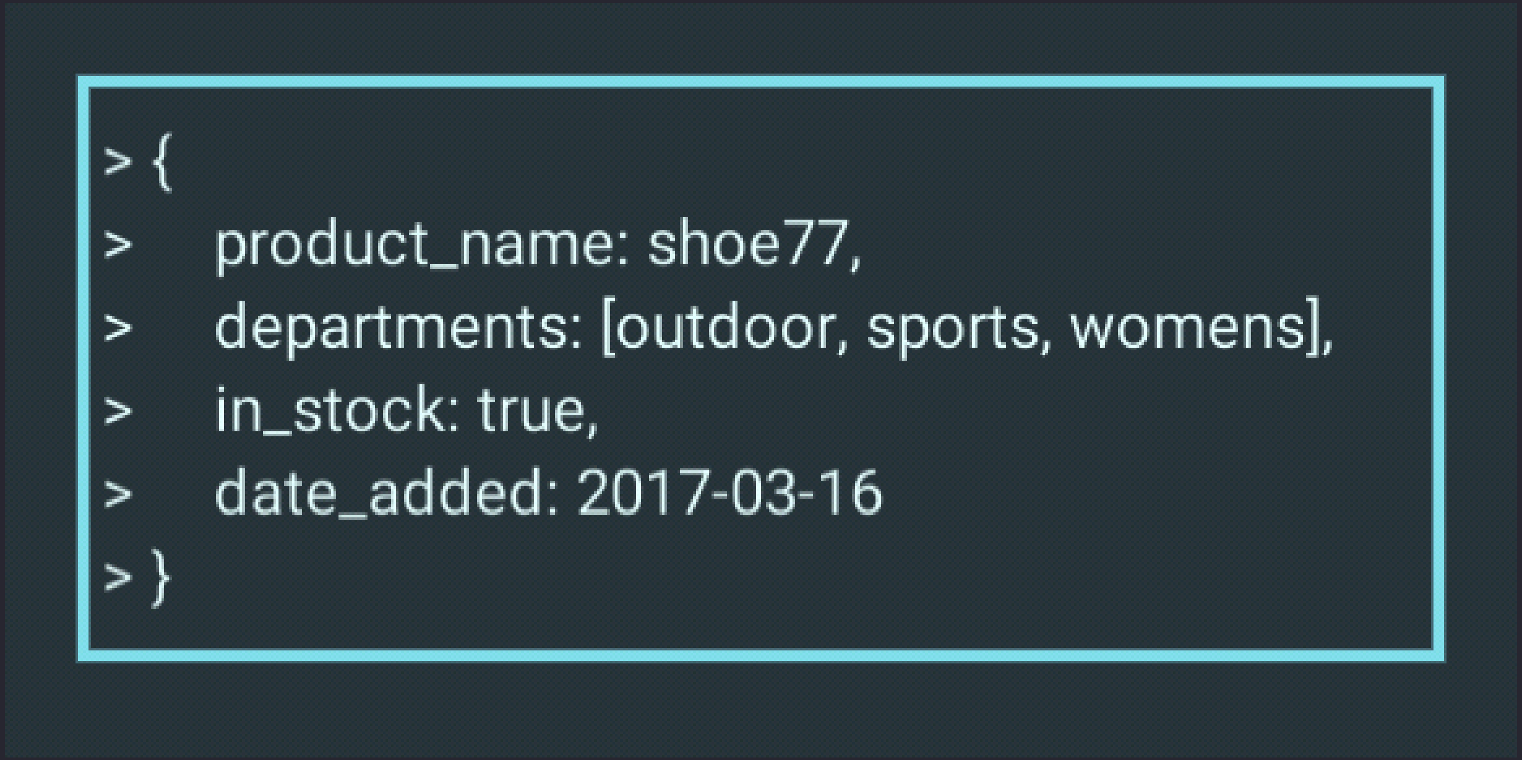
CouchDB Views

- Written in Javascript
- Use MapReduce
- The map results are stored
- Can be used either for filtering, or for aggregation

MapReduce Primer: Map

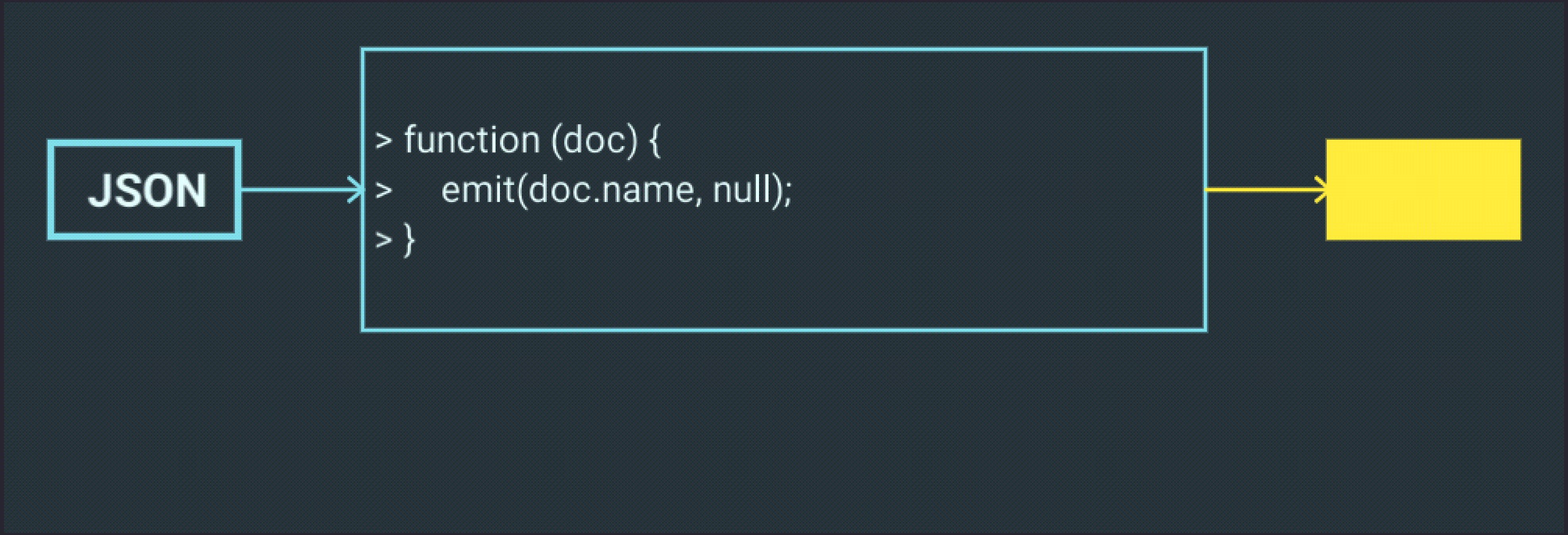
- Examine each document, "emit" 0+ keys/value pairs
- Scales well because each document is independent
- To filter a collection of documents, use map step only

MapReduce Primer: Map



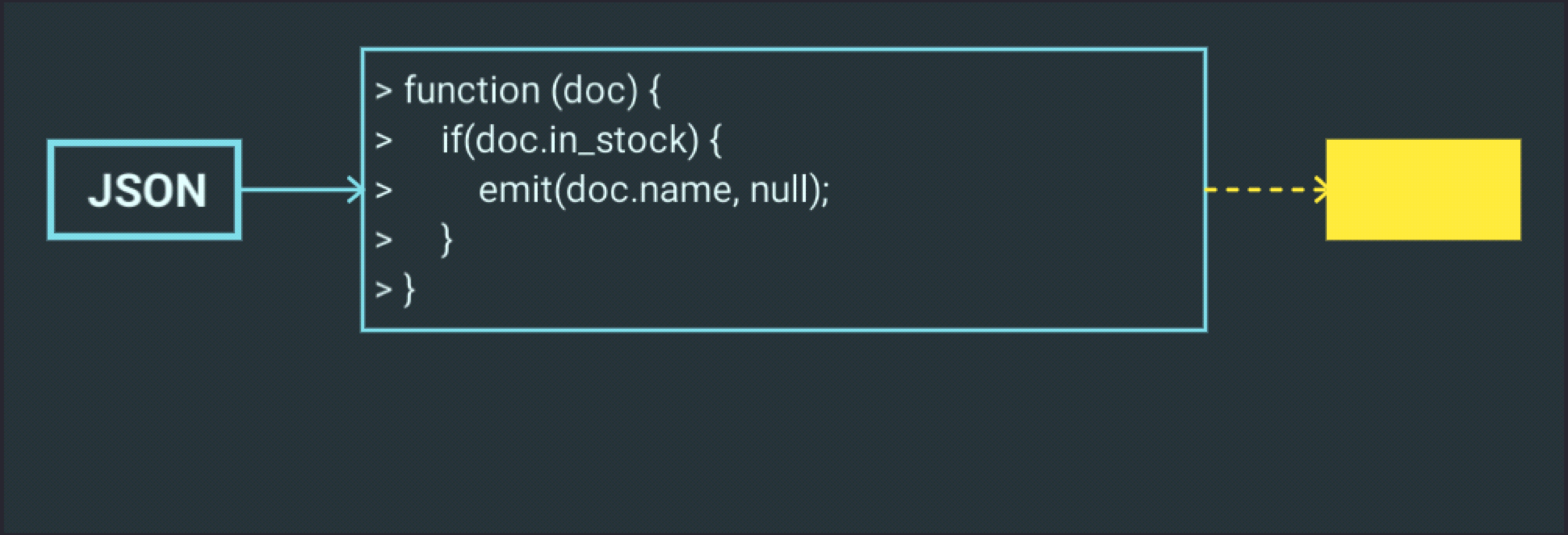
```
> {
>     product_name: shoe77,
>     departments: [outdoor, sports, womens],
>     in_stock: true,
>     date_added: 2017-03-16
> }
```

MapReduce Primer: Map



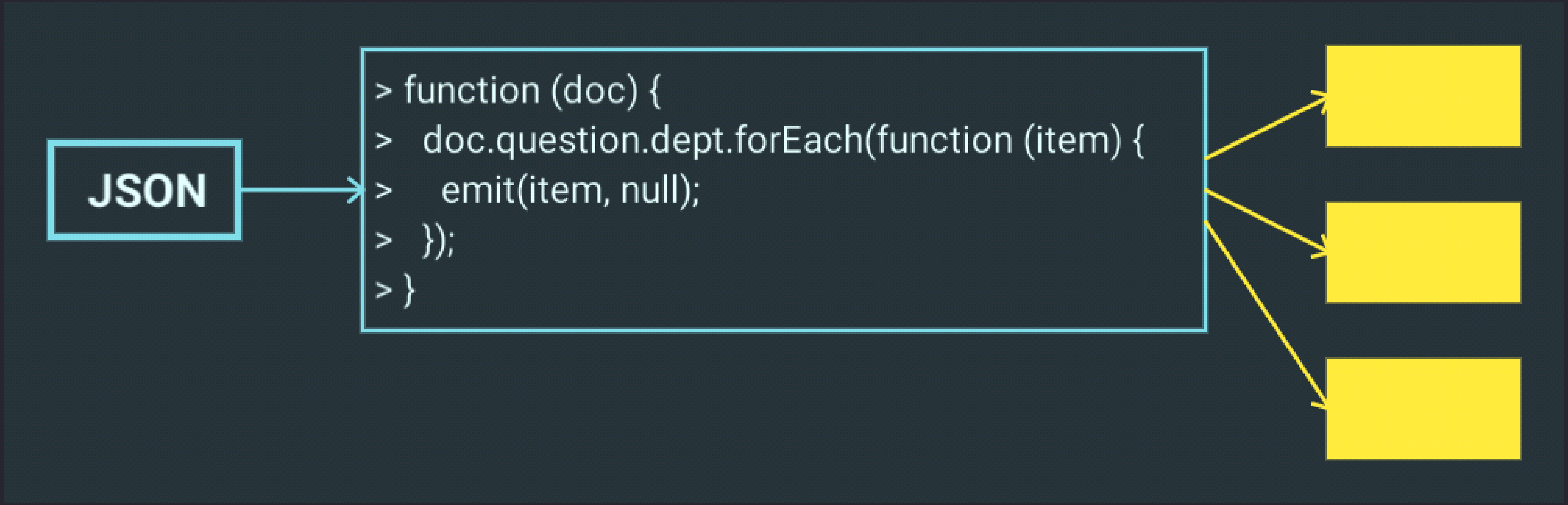
```
> function (doc) {
>     emit(doc.name, null);
> }
```

MapReduce Primer: Map



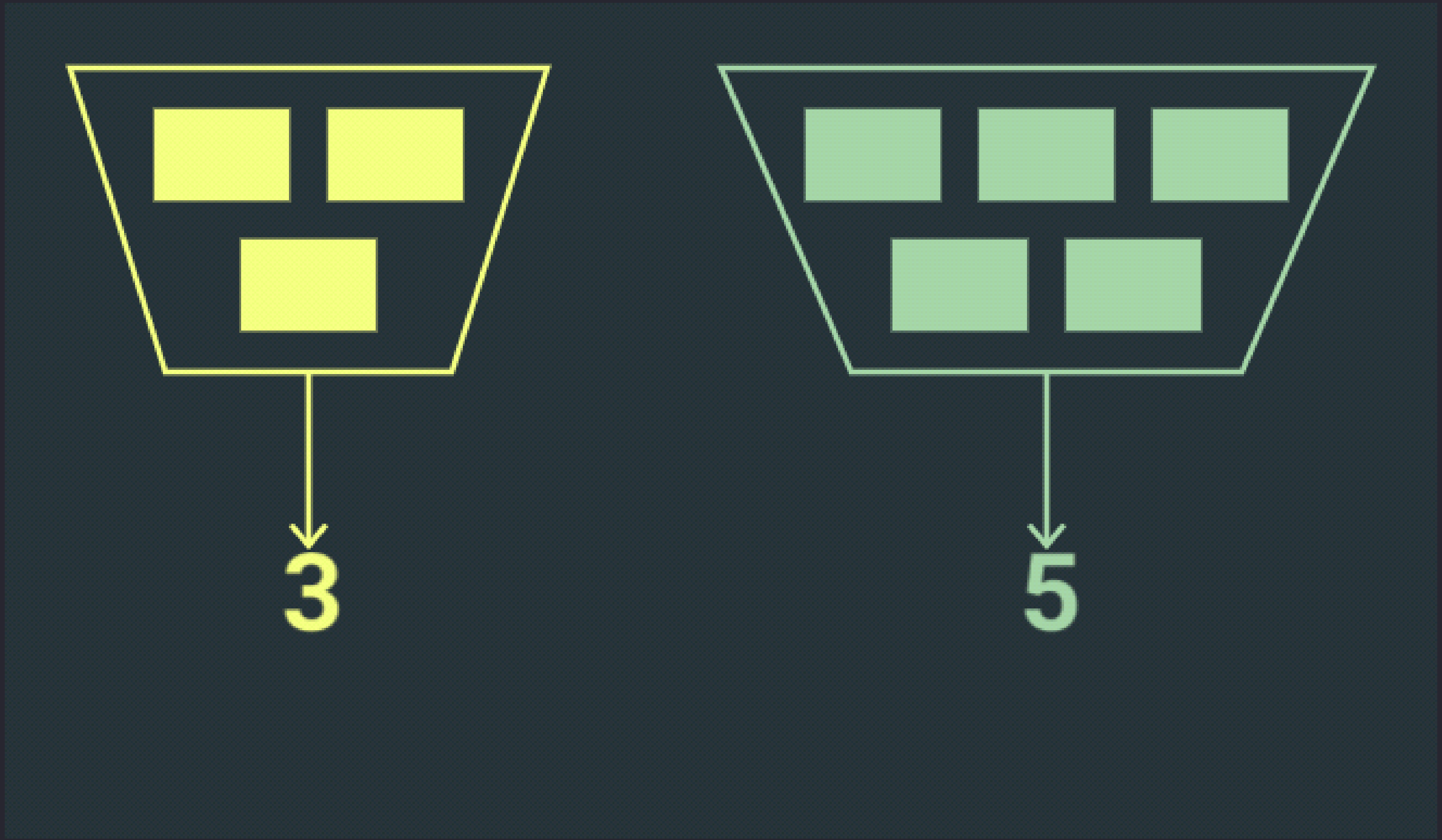
```
> function (doc) {
>     if(doc.in_stock) {
>         emit(doc.name, null);
>     }
> }
```

MapReduce Primer: Map



```
> function (doc) {
>   doc.question.dept.forEach(function (item) {
>     emit(item, null);
>   });
> }
```

MapReduce Primer: Reduce



MapReduce Primer: Reduce

- "Reduce" values in batches with the same key
- CouchDB has useful built in functions for most things
- Use reduce step when you want aggregate data
    - (SQL equivalent: a query with `GROUP BY`)

CouchDB Views: Example

http://localhost:5984/products/_design/products/_view/count?group=true

```
{"rows":[
  {"key":["mens","t-shirt"],"value":1},
  {"key":["womens","bag"],"value":3},
  {"key":["womens","shoes"],"value":1},
  {"key":["womens","t-shirt"],"value":2}
]}
```

CouchDB Views: Example

http://localhost:5984/products/_design/products/_view/count?group_level=1

```
{"rows":[
  {"key":["mens"],"value":1},
  {"key":["womens"],"value":6}
]}
```

CouchDB Tips and Resources

- CouchDB Definitive Guide http://guide.couchdb.org
- Javascript implementation https://pouchdb.com/
- PHP CouchDB library: https://github.com/ibm-watson-data-lab/php-couchdb
- Get a hosted version from https://www.ibm.com/cloud/

SQL, NoSQL and Beyond

Thanks

Feedback: https://joind.in/talk/f4061

Slides: http://lornajane.net/resources

Further reading: Seven Databases in Seven Weeks

Contact:

- `lorna.mitchell@uk.ibm.com`
- @lornajane