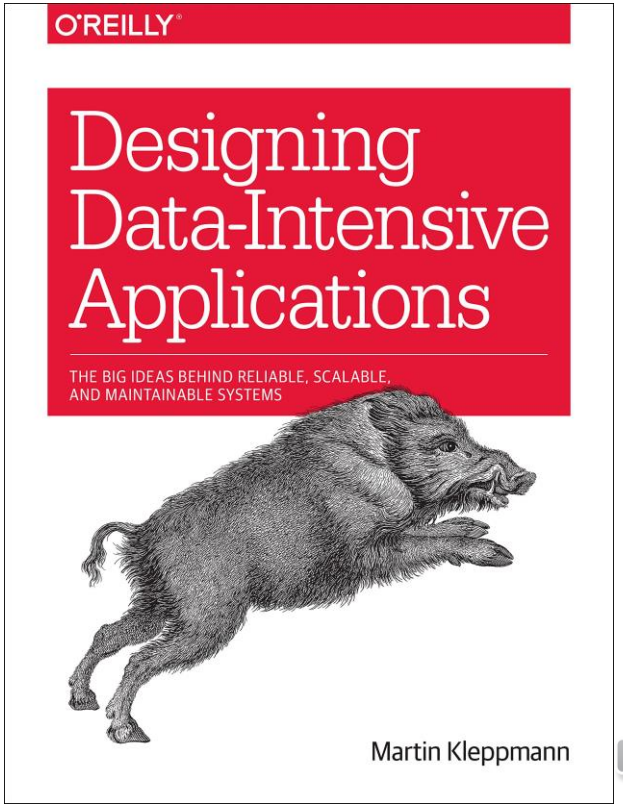
Data Intensive Applications

in PHP

Data Intensive Applications

Part book review

Part project implementation



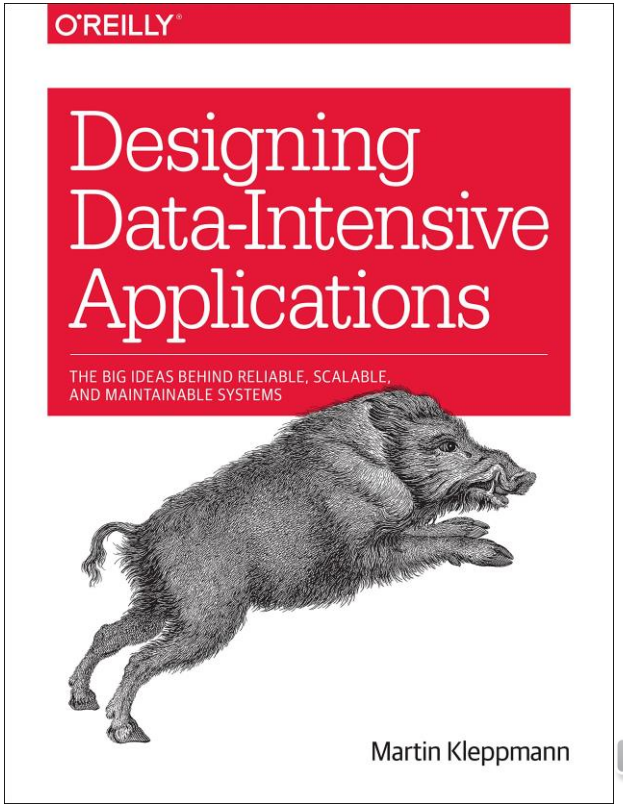
Data Intensive Applications

Martin Kleppman

Researcher at Cambridge on database systems

Voldemort / Kafka author

AllYourBaseConf 2015 “Bottled water” cdc system



Data Intensive Applications

Part 1 –

What is a database?

What kinds of database are there?

How does a database work?

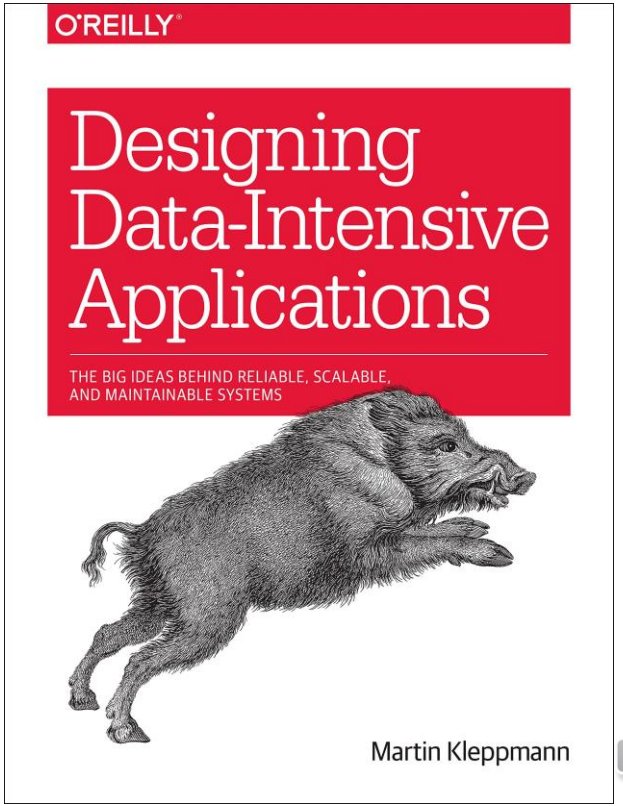
Part 2 –

Distributed data

Part 3 –

Derived data

batch vs streams vs future



Data Intensive?

- "data-intensive" means not "compute-intensive"
- Amount of data, complexity of data, rate of data change are bigger issues than CPU horsepower
- All web applications probably fall under this

Properties of good Data Systems

Reliable – continues to work correctly even when some things go wrong

- A fault is a single underlying issue, a failure is when the system as a whole stop working

Scalable – can the system cope with growth

- Simple statements like "X doesn't scale" or "Y is web-scale" are not useful

Maintainable

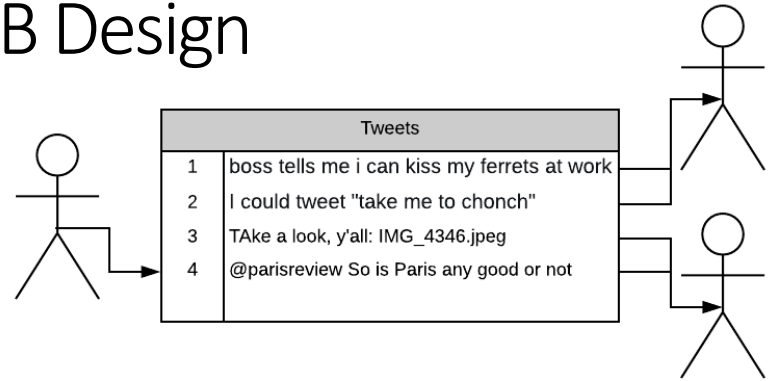
- Operatability, Simplicity, Evolvability

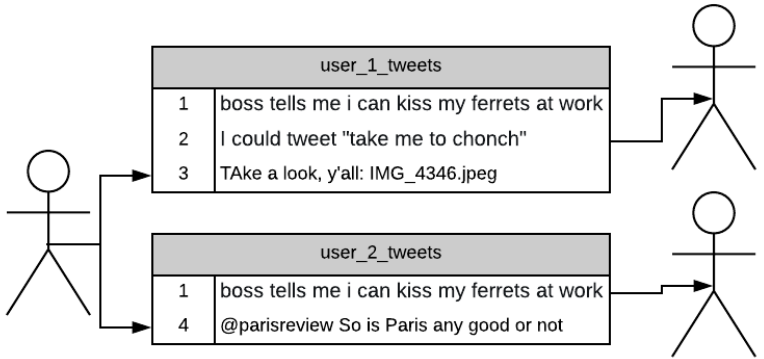
Twitter's DB Design

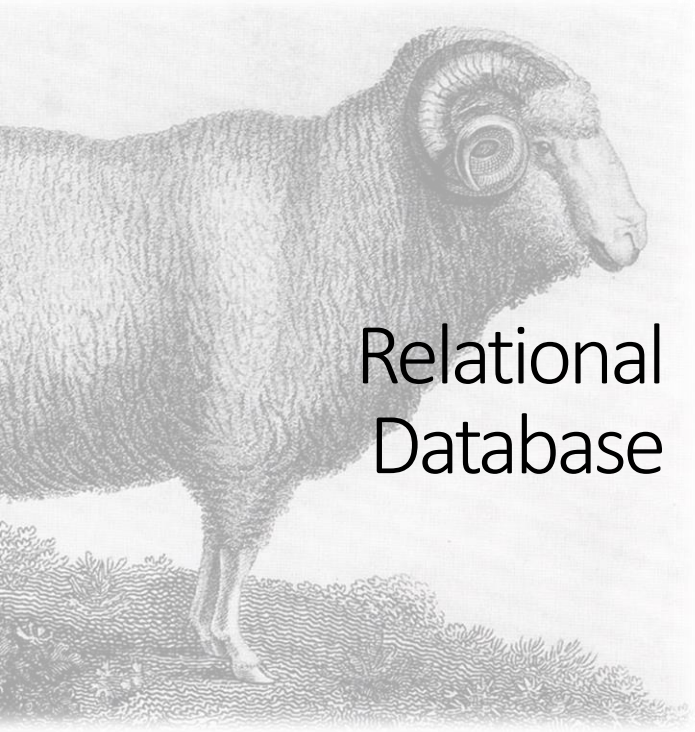
- All tweets written to "tweet" table, lots of random reads (expensive)

  or

- One feed per user, tweets written to all relevant feeds at write time
  - Cheap reads, just load precomputed table
  - Expensive writes for users with millions of follower

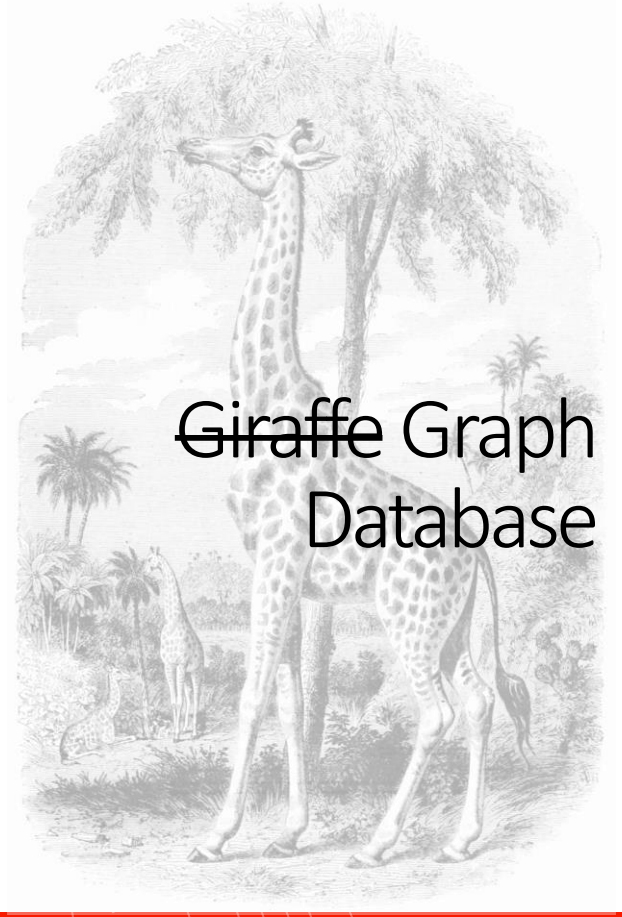






Relational Database

- 1970 – dominant paradigm since
- Columns in Rows in Tables in Schemas
- Data is stored in multiple tables with fixed columns per row
- Relationships between tables can be specified via keys
- Indexes provide fast lookup to individual rows
- SQL to query – declarative language
- MySQL / PostgresSQL / MS SQL Server / Amazon Aurora / Oracle

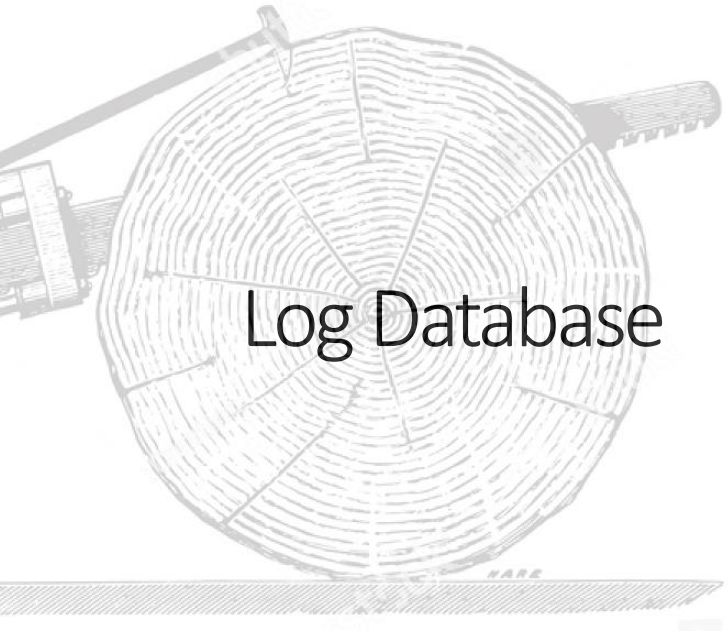


~~Giraffe~~ Graph Database

- Has two storage types – Vertex and Edge
- Edges link between two Vertexes
- Each can have a label and some key/value properties
- Good modelling many-to-many relationships between different entities
- Allows fast traversal
- Neo4J / Amazon Neptune

Document Database

- Aka NoSQL / JSON DB / Schemaless
- Allows dynamic fields and nested data per document
- Better impedance match – objects in memory map more easily to documents than rows
- Better locality – fewer queries when retrieveing complete object
- Bad at many-to-many / normalisation
- Harder / less efficient to query
- MongoDB / RavenDB / CouchDB / Amazon DynamoDB



- Write-optimised
- Same record can / should be stored multiple times
- Good plumbing between other parts
- Kafka / kinesis streams / event source db

Write your own DB

```php
function db_set($key, $value) {
    file_put_contents("database.db", "$key=$value\n", FILE_APPEND);
}

function db_get($key) {
    $all = file("database.db"),
    $entries = array_filter($all, function($entry) use($key) {
        return strpos($entry, "$key=") === 0;
    });
    return substr(array_pop($entries), strlen("$key="));
}
```

Write your own DB

- Log-structured
- Key-value
- Full history tracking
- Excellent write performance
- Easy to backup and restore
- Great compatibility with external tools

- Scales poorly
- Slow reads (no indexes)

- Probably better than many of the text-based databases that came with PHP scripts circa 2005

```
database.db:
dog=woof
cat=meow
dog=bark
json={“test”:“value”}
custom=[4,8,15,16,23,42]
serialised=O:8:"stdClass":0:{}
```

Polyglot Persistence

- Using multiple databases as part of a "data system"
- "Right tool for the job"
- Most applications over a certain size will use this
- Primary & Secondary storage
- Plumbing to connect them

Homebrew Polyglot Persistence

Key/value via table / Memcached plugin

Graph via `vertex` & `edge` table & awful query

Document via BLOB or JSON

Search via FULLTEXT index

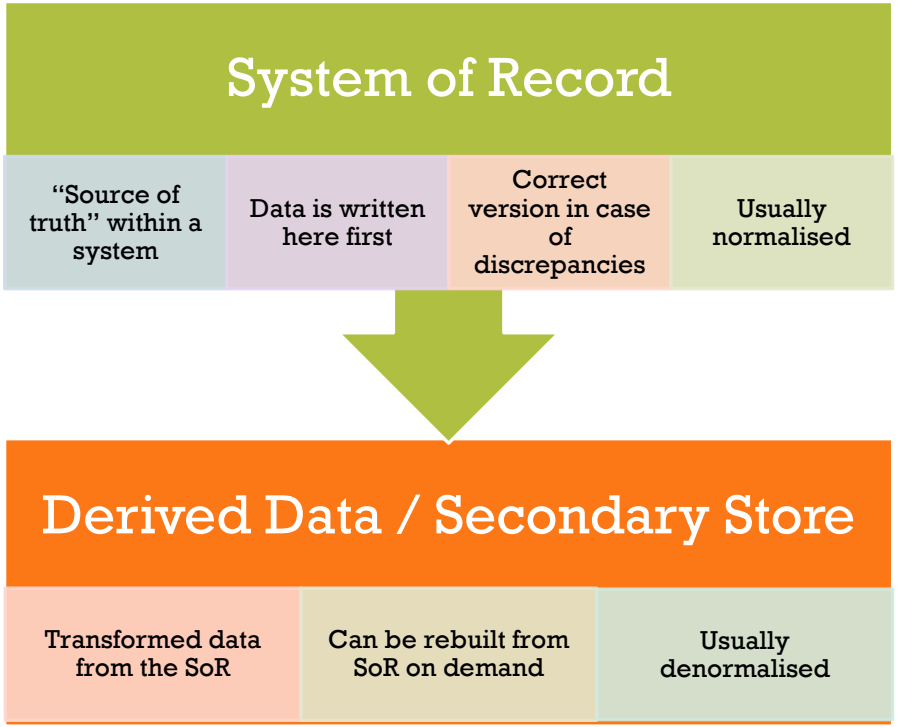
Log / queue via table

Batch vs Online

- Batch systems have throughput (number of requests per second)
- Online has response time (time between request and response)
- Lambda architecture - fast streaming (online) updates, accurate batch updates

“Data System”

Derived Data



Change Data Capture (CDC)

- Automatically builds "change" records by monitoring your primary database
- Save changes into a Log database like Kafka
- Downstream systems read from Kafka to rebuild
- confluent.io/blog/publishing-apache-kafka-new-york-times/

Implementing in PHP

System Brief

Ingest ticket data from N suppliers, all with different:

- Endpoint / API
- Data Model
- Update Schedule

Data Access

- API / XML file / CSV on FTP server
- Updated constantly / randomly / once a day
- Some tell you when something is updated, but not what
- Class should access data and emit all tickets – nothing else

Generators

- `yield` keyword
- Trivially create iterators
- Function calls `yield $x`, appears to return a `Generator`
- `foreach($generator as $value)` give you `$x`, next time loop is called original function picks up again to create another `$x` (if it wants)
- github.com/nikic/iter – php array functions reimplemented with generators

Big(ish) Data

- Some suppliers gave us too much data in one go
- Manually iterating over XML and nested structures is boring
- github.com/prewk/xml-string-streamer/
- github.com/pcrov/JsonReader

Source

```
function getFromPagedApi() {
  $page = 1;
  while(true) {
    $json = request("https://api.com?page=$page");
    $response = json_decode($json);

    yield from $response->Results;

    $page++;
    if($response->CurrentPage !== $response->TotalPages) {
      break;
    }
  }
}
```

Consumer

```
$results = getFromPagedApi();
foreach($results as $result) {
  //do something with $result
}
```

Normalisation

- Schema is important for forwards & backwards compatibility
- Built PHP DTO to contain data from third party
- Doesn't have to be same as your entities, but might be
- Write code to accept an "event" from supplier, emit DTO

- Write validation code
- Write serialization / deserialization code

Store and compare

- Normalised objects can be serialised and persisted anywhere
- We used Elasticsearch because we have it, and dated indexes work well
- Create new index on each import, stream all record in
  - Good built in tools to manage old data
- Point-in-time snapshots – see what their data was in the past
- Diff via streaming – scroll query works well

Matching Algorithm

- From `esdiff`
- Doesn't require holding both sets in memory
- Generalisable to any data source with ordering by primary keys
- Write a generator for any DB

- Step over two streams together, sorted by PK
- Same id = same document, compare properties (jsondiff)
- Left lower = right deleted
- Left higher = right created
- Advance until back in sync

Matching Algorithm

Left	Right	Result
1	1	diff
2	2	diff
3	4	3 was deleted
4	4	diff
6	5	5 was created
6	6	diff

Pipeline Class

- Chain together classes without nesting recursively
- Meta methods – `limit()` – `peek()`
- Supports full classes and functions
- Type hinting generators is hard
- https://www.hughgrigg.com/posts/simple-pipes-php-generators/

Right Tool For The Job

vs

Bad Workman Blames Their Tools

vs

Run Less Software

=

Build From Composable Parts

Summary

- Designing Data Intensive Applications is worth reading for any backend or storage engineer
- Generators and streaming parsers make data applications more testable
- Composable data storage parts helps you create Reliable, Scalable, and Maintainable software
- Designating a System Of Record help you design and control data flow
- CDC and a Log database can provide you with full historical rebuild for all derived data