Home-grown Data Warehousing with Cake 3 ORM

Agenda

1. On being data-driven
2. A short talk about curiosity
3. Your very first data warehouse
4. The MP in LAMP
5. From data to insights
6. Stars and rollups

On being data-driven

"It is a capital mistake to theorize before one has data."

Arthur Conan Doyle

On being data-driven

- You can only collect data with the product you have, no the one you wish you had.
- Data is only history.
- More data is not necessarily more information.

A short talk about curiosity

- It is better to be curiosity-driven than data-driven.
- Your data should be structured in a way that this curiosity can be satisfied.

A short talk about curiosity

- There's currently an explosion of data storage and analysis tools.
    - Elastic search, logstash and kibana (ELK) is one of the newer and stronger ones.

- Document based storage is great for unstructured data of dubious value.
- Unstructured data is difficult and very tedious to query.
- Relational databases (using SQL) are very well suited for curiosity-driven people.

Your very first data warehouse

A data warehouse is the name give to system used for reporting and data analysis.

What alternatives have I explored?

- Pentaho: Very flexible and complete BI suite.
- Mondrian: An open-source OLAP server, owned by Pentaho. Only the server, no visualization tools.
- Saiku: A very good and intuitive interface for Mondrian. I highly recommend this.

Disadvantages

- They all require training or experience to get them running correctly.
- Full of data warehousing terminology (cubes, hyper cubes, dimensions, metrics...)
- Require knowledge of yet another query language (MDX)

Why I like SQL

- Can be learnt both by technical an non-technical people.
- It is ubiquitous.
- Runs very fast on the adequate hardware.
- Can be composed to virtually answer any question.
- Almost any programmer have decent knowledge of it.

Disadvantages

- Building certain queries can be daunting or just plain difficult.
- Big and complex queries are difficult to read and understand by others.
- SQL is difficult to parametrize without extra tooling.

The MP in LAMP

Leverage the abilities your team already has

Why use PHP and a relational database?

- PHP and MySQL are ubiquitous in the web develoment world
- If your team already uses this stack, there is no need to bring in additional skills to start your warehouse.
- Favour simple architectures over complex stacks.
- Cheaper.
- The great majority of companies and startups won't need anything fancier for a long time.
- I chose CakePHP 3 as the tooling for creating my own warehouse.

Why Cake 3?

- Very flexible ORM and query builder, suitable for complex analytical queries.
- Very little configuration required, only point to the database it needs to use.
- Utilities for working with millions of rows as a stream of data.
- Plentiful of options for extending it when the available API falls short for creating the queries I need.

From data to insights

Let's pick a table

Pick a table

- Find a table you're curious about.

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `company_id` int(11) NOT NULL,
  `email` varchar(200) DEFAULT NULL,
  `gender` varchar(10) DEFAULT NULL, -- We will talk about this later
  `created` datetime DEFAULT NULL,
  `unsubscribed_time` datetime DEFAULT NULL,
  ...
);
```

Ask questions about it

- How often do I get a new user per company?
- What are the most common email ISPs of my users?
- What's the median lifetime after they subscribe?

Creating the metrics

```php
// UsersTable.php

private function getMetrics()
{
  return [
    'subscription_count' => 'count(*)' // The most simple metric possible
  ]
}

public function findMetrics(Query $query, $options)
{
  $metrics = array_flip($options['metrics']);
  $valid = array_intersect_key($this->getMetrics(), $metrics);

  return $query->select($valid);
}
```

```php
$subscriptions = $users->find('metrics', ['metrics' => ['subscription_count']]);
```

```sql
SELECT COUNT(*) AS subscription_count FROM users; -- resulting SQL
```

Filtering by date range

```
public function findTimeRange(Query $query, $options)
{
 $option += ['withDateGroup' => true];
 $date = $query->func()->date(['created' => 'indentifier'])

 $query->where(function ($exp) use ($date, $options) {
    return $exp->between($date, $options['from'], $options['to']);
 });

 if ($options['withDateGroup']) {
  $query->select(['date' => $date]);
  $query->group([$date]);
 }

 return $query;
}
```

```
$subscriptions = $users
  ->find('metrics', ['metrics' => ['subscription_count']])
  ->find('timeRange', ['from' => new DateTime('-1 month'), 'to' => new DateTime()]);
```

```
SELECT DATE(created) as date, COUNT(*) AS subscription_count FROM users
WHERE DATE(created) BETWEEN ? AND ?
GROUP BY DATE(created);
```

Displaying the data

```
// MetricsController.php

public function chart()
{
  $data = $this->Users
    ->find('metrics', ['metrics' => explode(',', $this->request->query('metrics'))])
    ->find('timeRange', ['from' => new DateTime('-4 days'), 'to' => new DateTime()]);
    ->through(new PlotlyFormatter()) // reformat the results for plot.ly
    ->toList();

  $this->set(compact('data'));
}
```

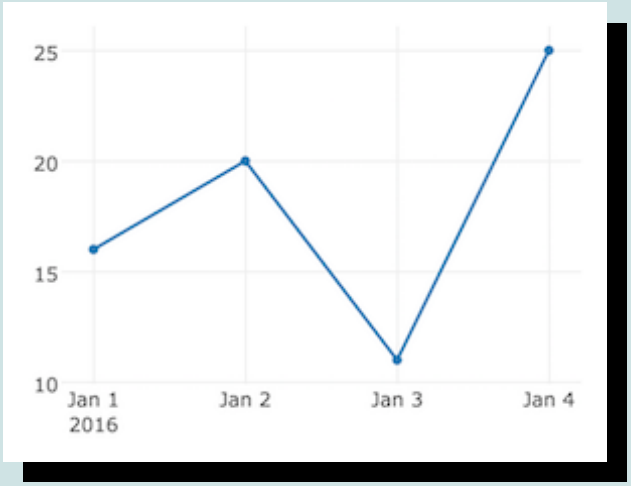
```
curl -X GET https://mysite.com/metrics/chart.json?metrics=susbscription_count
```

```
{
"xAxis": ['2016-01-01', '2016-01-02', '2016-01-03', '2016-01-04'],
"metrics": [
  ["subscription_count", [5, 11, 30, 25]] // Count of users
]
}
```

Displaying the data

```
var traces = [];
ajaxResponse.forEach((metric) => {
  traces.push({
    x: ajaxResponse.xAxis,
    y: metric[1],
    type: 'line'
  });
});

Plotly.newPlot('myDiv', traces);
```



Some Problems

With performance

- Using DATE() in mysql will not use any indexes, this makes things slow

With complex metrics

- Expressing countable metrics of of other columns is difficult
  - How do we get the count of users per email ISP?

With empty values

- Dates where no value is available should still be present in the result
  - For example, a day with no unsubscribes

Stars and Rollups

- We need a type of table where each row can contain one or more pre-calculated metrics, per the level of granularity we want to query on

```
CREATE TABLE user_agg (
  fact_date DATE NOT NULL,
  subscribed_count INT,
  unsubscribed_count INT,
  total_count INT
);
-- This is called an aggreagate table
```

- Metrics should be addition friendly (only counts and sums, no rates!)
- All other columns in the table should either be a foreign key to another table, or a value that is available in a controlled set of possible values

```
CREATE TABLE user_agg (
  ...
  company INT NOT NULL
  email_isp VARCHAR(20) NOT NULL,
  PRIMARY KEY (date, company_id, emails_isp) -- All non-metric columns should be here
);
```

Creating the facts table

In order to create an aggreagete we need a facts table. This is a table where each row represents a single metric. We can then create our aggregate table out of it.

Creating the facts table

We want to convert something like this:

id	company_id	email	created	unsubsbscribed_time
1	10	foo@gmail.com	2016-01-01	2016-03-20
2	10	bar@yahoo.com	2016-01-02	NULL
3	20	baz@gmail.com	2016-01-02	2016-03-21
4	30	another@yahoo.com	2016-03-04	NULL

Into this:

date	isp	company	event
2016-01-01	gmail	10	subscribe
2016-01-02	yahoo	10	subscribe
2016-01-02	gmail	20	subscribe
2016-03-20	gmail	10	unsubscribe
2016-03-04	yahoo	30	unsubscribe

Creating the facts table

```
// UsersShell.php

private function createFacts()
{
  $factsTable = new Schema\Table('user_facts', [
    'date' => ['type' => 'date'],
    'email_isp' => ['type' => 'string', 'length' => 40],
    'company' => ['type' => 'integer'],
    'event_type' => ['type' => 'varchar', 'length' => '10']
  ]);

  $factsTable->temporary(true); // It's up yo you if you want to persist this
  $sql = $factsTable->createSQL();

  foreach ($sql as $query) {
    $connection->execute($query);
  }
}
```

Populating the facts table

- Time to insert the events in the facts table

```
private function populateFacts()
{
  $users = $this->loadModel('Users');
  $query = $users->find()

    ->select(function ($query) {
      return [
        'date' => 'COALESCE(uDate.date, sDate.date)',
        'email_isp' => calculateIsp('email') // extract the IPS form the email
        'company' => 'company_id',
        'event' => 'IFNULL(sDate.date, "subscribe", "unsubscribe")'
      ];
    })

    // Use the dates table for mathing users with the same subscription time
    ->leftJoin(['sDate' => 'dates'], ['sDate.date = DATE(users.created)'])

    // Use the dim_date table for mathing users with the same unsubscription time
    ->leftJoin(['uDate' => 'dates'], ['uDate.date = DATE(users.unsubscribed_time)']);

    // populating the table
    $connection->newQuery()->insert('user_facts')->values($query)->execute();
}
```

Generating the aggregate

- At the end we want a table looking like this

date	isp	company	subscribed_count	unsubscribed_count
2016-01-01	gmail	10	1	0
2016-01-02	yahoo	10	1	0
2016-03-04	yahoo	30	0	1
...	...	...	...	...

It will contain the total metric values for each unique combination of the other columns.

Generating the aggregate

- We count now count the events in the facts table to generate the aggregate

```
public function createAggregate()
{
  $this->createFacts();
  $this->populateFacts();
  $facts = $this->loadModel('user_facts')
  $agg = $this->loadModel('user_agg');

  $query = $facts->find()
    ->select($agg->schema()->primaryKey())
    ->select([
      'subscribed_count' => 'sum(case event when "subscribe" 1 else 0)',
      'unsubscribed_time' => 'sum(case event when "subscribe" 0 else 1)',
    ])
    ->group($agg->schema()->primaryKey());

  $connection->newQuery()->insert('user_agg')->values($query);
}
```

You can easily parametrize the previous code using date ranges if for doing incremental updates in the aggregate table

Putting it all together

We can now adapt the code we had before, but this time using the aggregate table

```
// UserAggTable.php


private function getMetrics()
{
  return [
    'subscription_count' => 'sum(subscribed_count)' // remember addition friendly?
    'unsubscription_count' => 'sum(unsubscribed_count)',
    'churn_rate' => 'sum(unsubscribed_count) / sum(subscribed_count)'
  ]
}

public function findTimeRange(Query $query, $options)
{
  $option += ['withDateGroup' => true];
  $date = 'date'; // We use the date value from the agg table directly
  ...
}
```

We can also use any of the columns in the table to do super fast finds, since it is part of the primary key for the agg table.

Filtering by other columns

```
public function findFiltered(Query $query, $options)
{
  $valid = array_flip($this->primaryKey());
  $filters = array_intersect_key($options['filter'], $valid);

  return $query->where($filters);
}
```

Getting the global churn-date for the last month in Gmail users

```
$userAgg
  ->find('filtered', [
    'filter' => [
      'isp' => 'gmail',
      'company' => 10
   ]
  )
  ->find('timeRange', ['from' => new DateTime('-1 month'), 'to' => new DateTime])
  ->find('metrics', ['metrics' => ['churn_rate']]);
```

Next steps...

Automating it

- Make the populating script delete the agg table first for the specified rage before populating
- Using cron to populate the agg table every day

Using associations for filtering

- Since columns in the agg table are foreign keys, you can filter by pretty names in associations

Questions?