Using cache dependencies and chains

Yii supports many cache backends, but what really makes the Yii cache flexible is the dependency and  
dependency chaining support. There are situations when you cannot simply cache data for an hour  
because the information cached can be changed at any time.

In this recipe, we will see how to cache a whole page and still always get fresh data when it is updated.  
The page will be of the dashboard-type and will show the five latest articles added and a total calculated  
for an account.

Note

Note that an operation cannot be edited as it is added, but an article can be.

Подготовка

Создайте новое yii2-app-basic приложение с помощью диспетчера пакетов Composer, как описано в официальном руководстве по адресу  
<http://www.yiiframework.com/doc-2.0/guide-start-installation.html>.   
По русски <http://yiiframework.domain-na.me/doc/guide/2.0/ru/start-installation>

1. Activate the caching component in config/web. php as follows:

return [

// ...

'components' => [

cache => ['class' => 'yii\caching\FileCache,

],

],

];

1. Set up a fresh database and configure it into config/db. php.
2. Run the following migration:

<?php

use yii\db\Schema;  
use yii\db\Migration;

class m160308\_093233\_create\_example\_tables extends Migration  
{

public function up()

{

$tableOptions = null;

if ($this->db->driverName === 'mysql') {

$tableOptions = 'CHARACTER SET utf8 COLLATE utf8\_general\_ci  
ENGINE=InnoDB';

}

$this->createTable('{{%account}}', [

'id' => Schema::TYPE\_PK,

'amount' => Schema::TYPE\_DECIMAL . '(10,2) NOT NULL',

], $tableOptions);

$this->createTable('{{%article}}', [

'id' => Schema::TYPE\_PK,

'title' => Schema::TYPE\_STRING . ' NOT NULL',

'text' => Schema::TYPE\_TEXT . ' NOT NULL',

], $tableOptions);

}

public function down()

{

$this->dropTable('{{%article}}');

$this->dropTable('{{%account}}');

}

}

1. Generate models for the account and article tables using Yii.
2. Create protected/controllers/DashboardController . php as follows:

<?php

namespace app\controllers;

use app\models\Account;  
use app\models\Article;  
use yii\web\Controller;

class DashboardController extends Controller  
{

public function actionIndex()

{

$total = Account::find()->sum('amount');

$articles = Article::find()->orderBy('id DESC')->limit(5)->all();

return $this->render('index', array(

'total' => $total,

'articles' => $articles,

));

}

public function actionRandomOperation()

{

$rec = new Account();

$rec->amount = rand(-1000, 1000);

$rec->save();

echo 'OK';

}

public function actionRandomArticle()

{

$n = rand(0, 1000);

$article = new Article();

$article->title = "Title #".$n;

$article->text = "Text #".$n;

$article->save();

echo 'OK';

}

}

1. Create views/dashboard/index . php as follows:

<?php

use yii\helpers\Html;

/\* @var $this yii\web\View \*/

/\* @var $total int \*/

/\* @var $articles app\models\Article[] \*/

?>

<h1>Total: <?= $total ?></h1>

<h2>5 latest articles:</h2>

<?php foreach($articles as $article): ?>

<h3><?= Html::encode($article->title) ?></h3>

<div><?= Html::encode($article->text) ?></div>

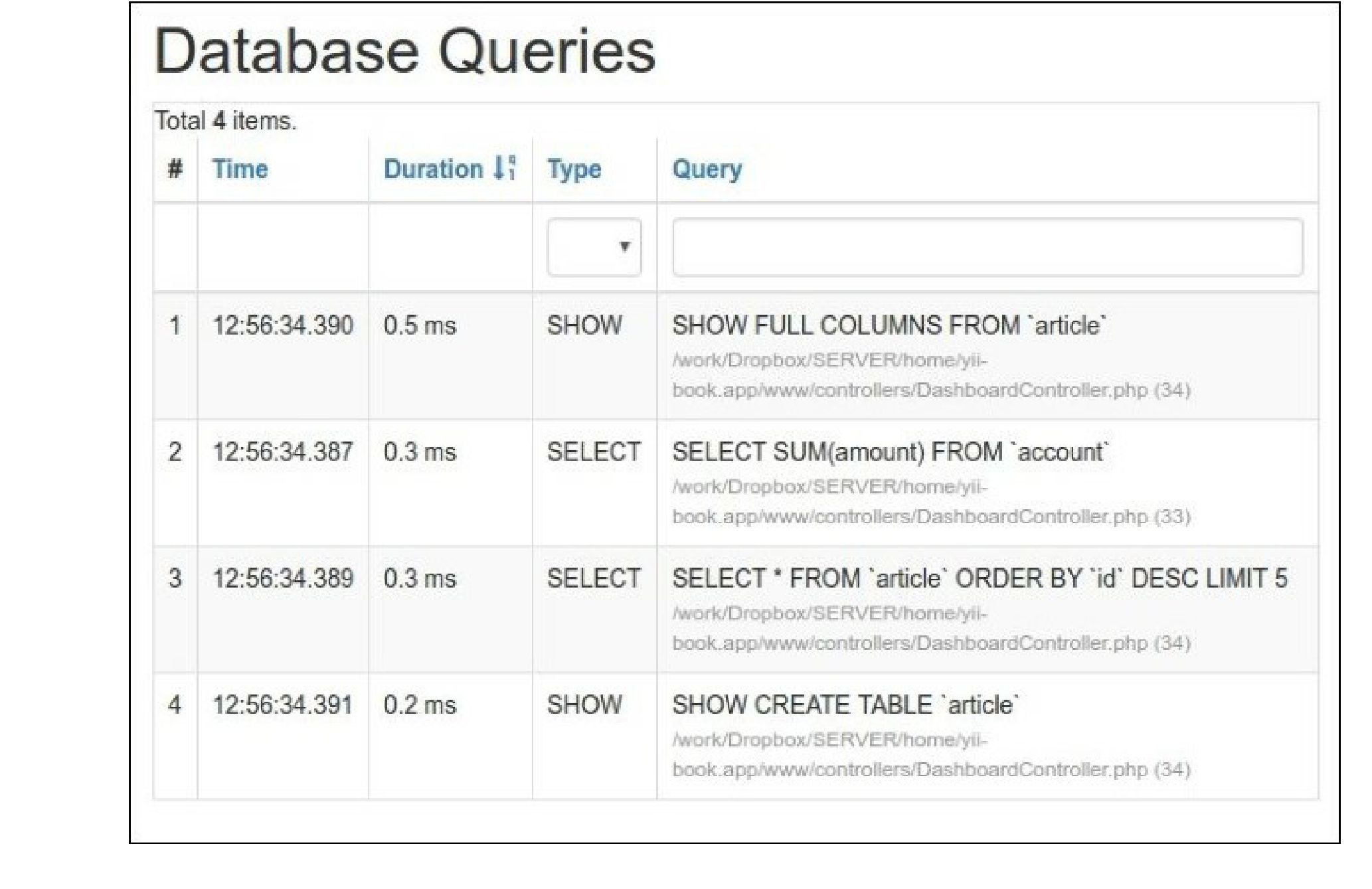
<?php endforeach ?>

7. Run dashboard/random-operation and dashboard/random-article several times. Then, run  
dashboard/index and you should see a screen similar to the one shown in the following screenshot:

8. Click on the number of database queries in the debug panel at the bottom of the page:



See a query list:



How to do it...

Carry out the following steps:

1. We need to modify the controller code as follows:

<?php

namespace app\controllers;

use app\models\Account;  
use app\models\Article;  
use yii\caching\DbDependency;  
use yii\caching\TagDependency;  
use yii\web\Controller;

class DashboardController extends Controller  
{

public function behaviors()

{

return [

'pageCache' => [

'class' => 'yii\filters\PageCache',

'only' => ['index'],

'duration' => 24 \* 3600 \* 365, // 1 year  
'dependency' => [

'class' => 'yii\caching\ChainedDependency',

'dependencies' => [

new TagDependency(['tags' =>

['articles']]),

new DbDependency(['sql' => 'SELECT MAX(id) FROM ' .

Account::tableName()])

]

],

],

];

}

public function actionIndex()

{

$total = Account::find()->sum('amount');

$articles = Article::find()->orderBy('id DESC')->limit(5)->all();

return $this->render('index', array(

'total' => $total,

'articles' => $articles,

));

}

public function actionRandomOperation()

{

$rec = new Account();

$rec->amount = rand(-1000, 1000);

$rec->save();

echo 'OK';

}

public function actionRandomArticle()

{

$n = rand(0, 1000);

$article = new Article();

$article->title = "Title #".$n;

$article->text = "Text #".$n;

$article->save();

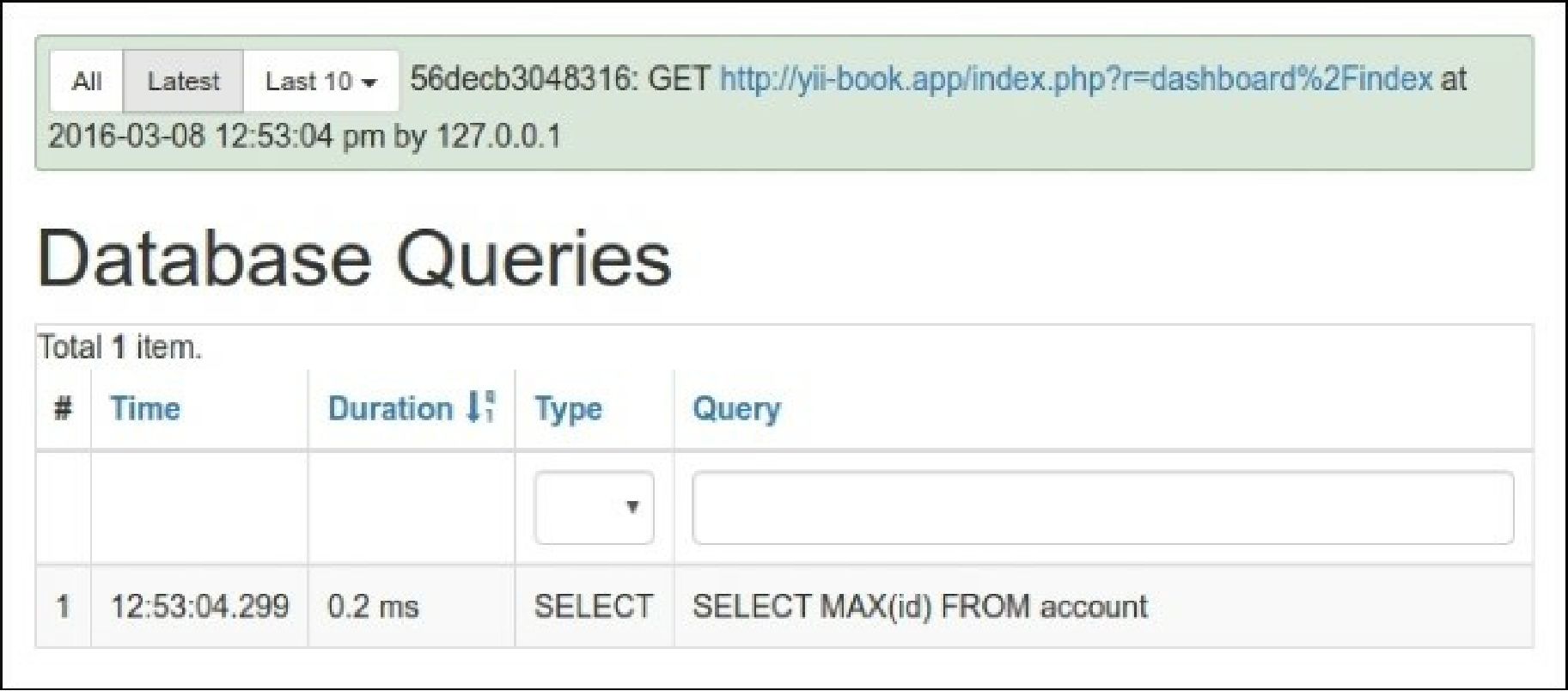
TagDependency::invalidate(\Yii::$app->cache, 'articles');  
echo 'OK';

}

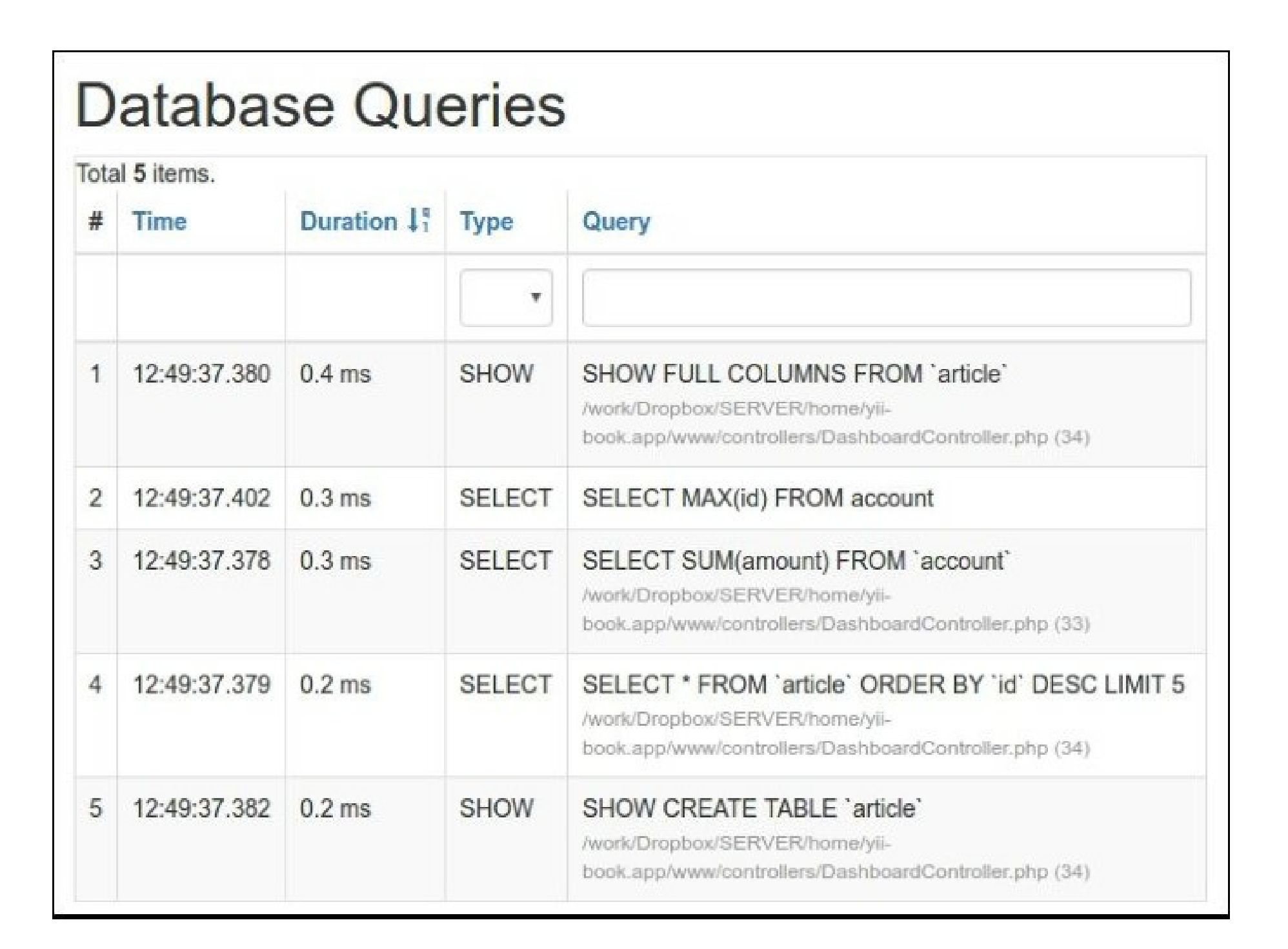
}

2. That is it. Now, after loading dashboard/index several times, you will get only one simple query in

the latest snapshot, as shown in the following screenshot:



Also, try to run either dashboard/random-operation or dashboard/random-article and refresh  
dashboard/index after that. The data should change as follows:



How it works...

In order to achieve maximum performance while doing minimal code modification, we use a full-page  
cache using a filter as follows:

public function behaviors()

{

return [

'pageCache' => [

'class' => 'yii\filters\PageCache',

'only' => ['index'],

'duration' => 24 \* 3600 \* 365, // 1 year  
'dependency' => [

'class' => 'yii\caching\ChainedDependency',

'dependencies' => [

new TagDependency(['tags' => ['articles']]),

new DbDependency(['sql' => 'SELECT MAX(id) FROM account'])

]

],

],

];

}

The preceding code means that we apply a full-page cache to the index action. The page will be cached  
for a year and the cache will refresh if one of the dependency data changes. Therefore, in general, the  
dependency works as follows:

* The first run gets the fresh data as described in the dependency, saves it for future reference, and  
  updates the cache
* It gets the fresh data as described in dependency, gets the saved data, and then compares the two
* If they are equal, it uses the cached data
* If not, it updates the cache, uses the fresh data, and saves the fresh dependency data for future  
  reference

In our case, two dependency types are used—tag and DB. A tag dependency marks data with the custom  
string tag and checks it to decide if we need to invalidate the cache, while a DB dependency uses the SQL  
query result for the same purpose.

The question that you have now is probably, “Why have we used DB for one case and tags for another?”  
That is a good question!

The goal of using the DB dependency is to replace heavy calculations and select a light query that gets as  
little data as possible. The best thing about this type of dependency is that we don’t need to embed any  
additional logic in the existing code. In our case, we can use this type of dependency for account  
operations, but cannot use it for articles as the article content can be changed. Therefore, for articles, we  
set a global tag named article which basically means that we can manually call the following when we  
want to invalidate total the article cache:

TagDependency::invalidate(\Yii::$app->cache, 'articles');

See also

In order to learn more about caching and using cache dependencies, refer to  
<http://www.yiiframework.com/doc-2.0/guide-caching-overview.html>