Conquer the 5 Most Common Magento Coding Issues to Optimize Your Site for Performance

Written by:
Oleksandr Zarichnyi

Table of Contents

Introduction

In any eCommerce implementation, system performance can mean the difference between satisfied customers and frustrated customers who shop elsewhere. In particular, PHP code performance issues can have immediate critical impact to the business. Although this type of issue is much easier to spot and fix using static code analysis than performance or scalability issues related to software design, the exact level of impact cannot be measured accurately without application profiling and monitoring, because these methods provide information on how often the code is executed and on what amounts of data.

This article provides a high-level overview of the most common issues that can impact performance and scalability of the Magento PHP code. These issues are created by Magento PHP developers during software implementation and do not relate to hardware, application or database design, or data access SQL code.

Developers who are new to Magento can use this article as a guideline on how to write optimized PHP code. This article can also serve as a checklist of reminders for experienced Magento developers.

Magento ECg

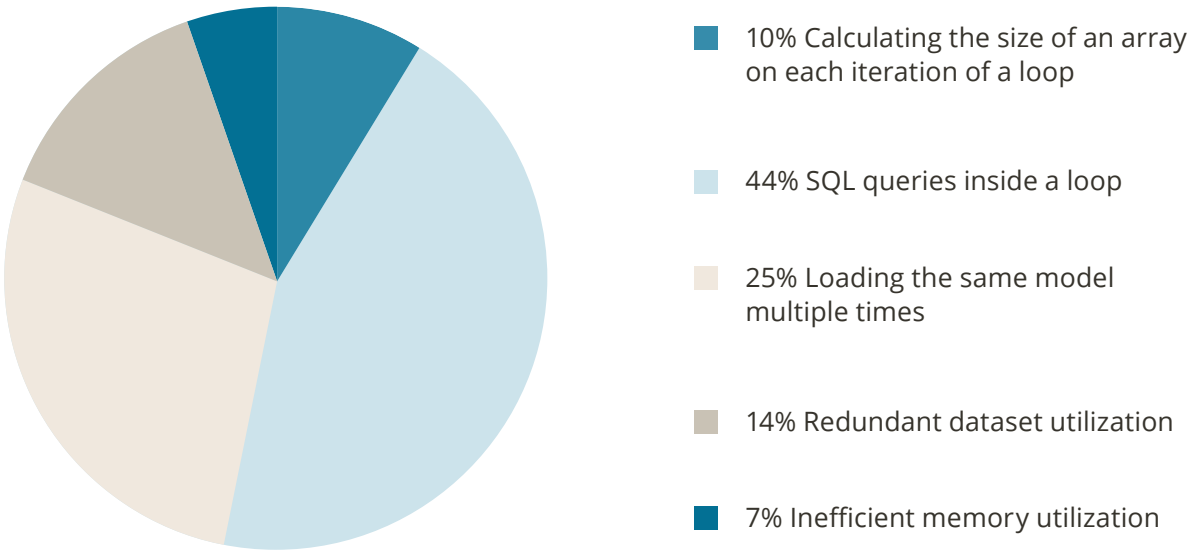
Top 5 Issues

Top 5 Issues

The Magento Expert Consulting Group (ECG) has conducted dozens of code audits of client codebases and mined about 300 common coding issues made by PHP developers. About 28% of all issues affect performance and scalability of the code. The top 5 performance coding issues represent 84% of all performance issues. They were encountered in 96% of client codebases. In this article, ECG presents detailed descriptions of the top 5 Magento PHP code performance issues along with the examples of code and recommendations on how to avoid them.

Most of the issues are related to inefficient operations, redundant or useless computations, and memory misuse. The top 5 are:

- Calculating the size of an array on each iteration of a loop
- SQL queries inside a loop
- Loading the same model multiple times
- Redundant data set utilization
- Inefficient memory utilization



Magento EC[g]

Loops

Loops

Even the slightest coding inefficiency is magnified when that code is located inside a loop. Expensive operations, such as SQL queries and redundant computations within loops, are a common culprit of performance bottlenecks in Magento code.

CALCULATING THE SIZE OF AN ARRAY ON EACH ITERATION OF A LOOP

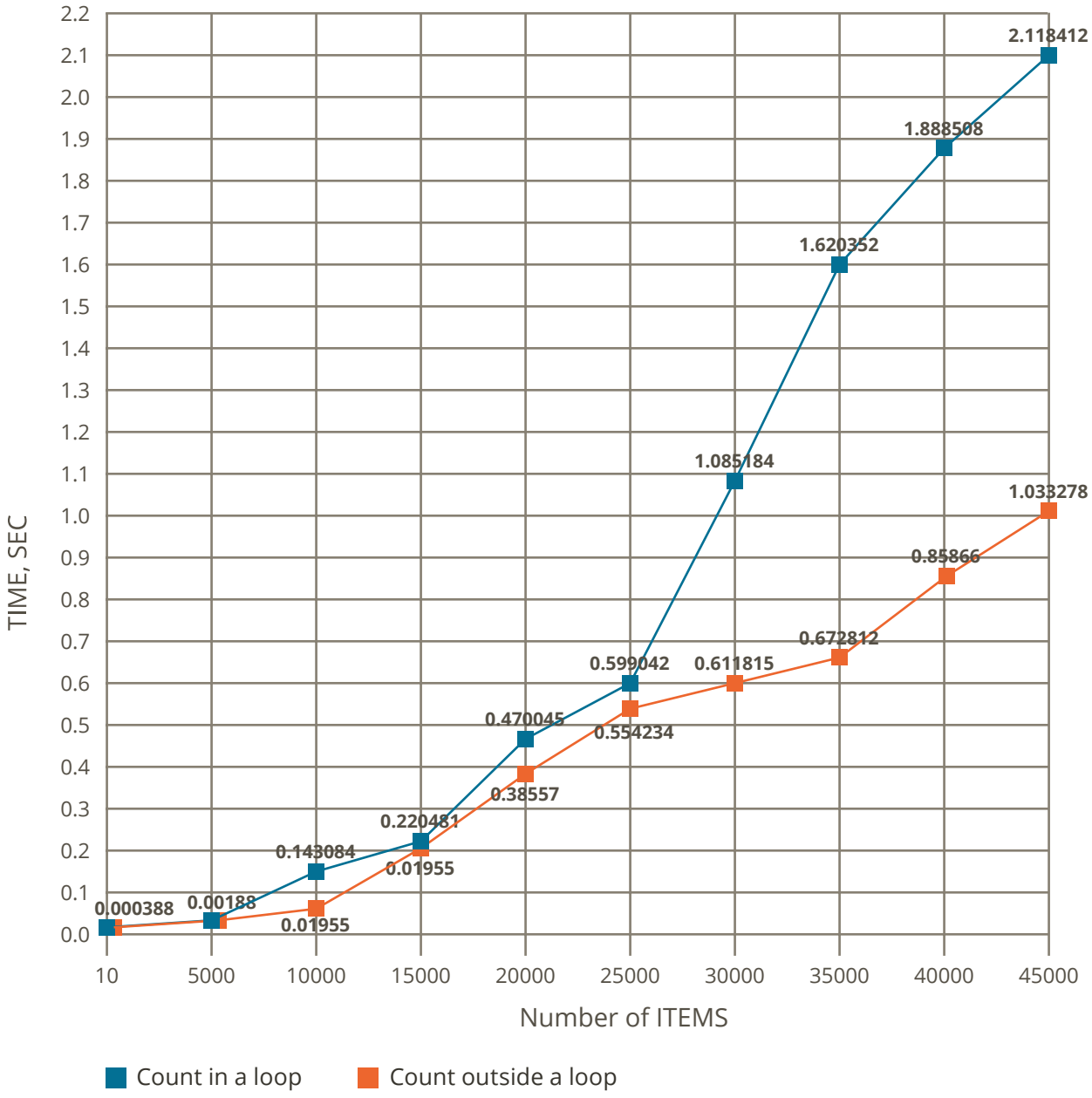
A classic example of inefficiency is calling a **count()** function inside the condition part of a for statement:

```
for ($i = 0; $i < count($rows); $i++) {
    //some code
}
```

Although **count()** is fairly fast in regular use, this changes rapidly when it is used in a loop. If the array or collection $rows contains a lot of data, this code is slowed down significantly. Because the PHP interpreter doesn't perform loop-invariant code motion automatically, a much better way is to move the **count()** call outside the loop:

```
$rowNum = count($rows);
for ($i = 0; $i < $rowNum; $i++) {
    //some code
}
```

The following chart illustrates how the time of code execution varies with the number of array items in both cases, when the **count()** call is made in a loop and outside it:



The test was conducted with an array of Magento products and was run on a dedicated server with 12 GB RAM, 16x Intel® Xeon® CPU running at 2.40GHz, and PHP 5.3.19.

In this test, in cases where the array size is less than 25,000 items, moving **count()** outside a loop can be considered as a micro-optimization, but as the array size grows further it is clear that running **count()** inside of the loop is slower; it takes twice as long or more as running it outside.

SQL Queries Inside a Loop

Execution of an SQL query is one of the most computationally expensive operations. Running an SQL query in a loop will most probably result in a performance bottleneck.

Very often developers load Magento models in a loop. For example, they iterate over the array of product IDs to load a product model and process it in the loop:

```
foreach ($this->getProductIds() as $productId) {
    $product = Mage::getModel('catalog/product')->load($productId);
    $this->processProduct($product);
}
```

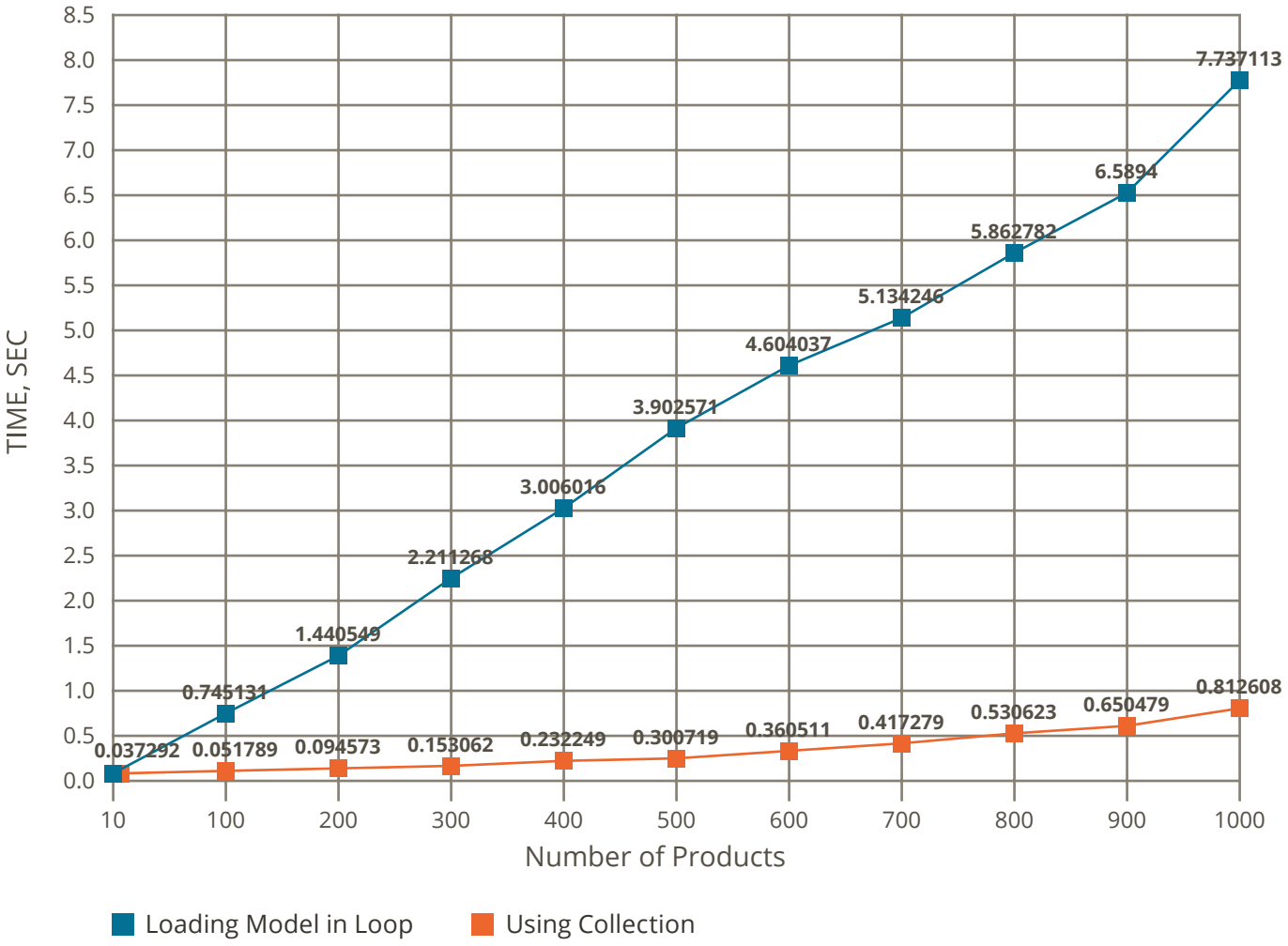
Loading an entity attribute value (EAV) model requires several heavy queries to execute. As the number of executed queries is multiplied with the number of products, we get extremely inefficient and slow code.

Instead of loading products in a loop, a Magento data collection can help to load a set of models in a very efficient manner. The following example filters the result set of a collection by an array of product IDs and adds all requested product fields to result:

```
$collection = Mage::getResourceModel('catalog/product_collection')
    ->addFieldToFilter('entity_id', array($this->getProductIds()))
    ->addAttributeToSelect(array('name'));

foreach ($collection as $product) {
    $this->processProduct($product);
}
```

To measure the performance impact, this test was run on the same environment as the previous one. The results are even more striking:



The code that uses collection is linearly faster than the code that loads models in a loop. The difference is significant from the very beginning.

Of course, when dealing with large data sets with the help of collections, you should pay attention to the amount of loaded data so that it doesn't exceed the amount of the allocated memory.

Models are often loaded merely to retrieve one or several entity attributes. An important benefit of Magento data collection is that it provides control over the loaded data fields, while loading an EAV model will request multiple tables and select all the attributes of the entity.

Model saving and deletion operations are also expensive. If the performance is critical, execute a mass save/delete query, rather than running them in a loop.

Magento EC[g]

Models

Models

LOADING THE SAME MODEL MULTIPLE TIMES

Developers sometimes do not consider that model load operation is not internally cached, and each time the **load()** method is called, one or more queries are ran against the database. Loading the same model several times causes noticeable performance degradation. For example:

```
$name = Mage::getModel('catalog/product')->load($productId)->getName();
$sku  = Mage::getModel('catalog/product')->load($productId)->getSku();
$attr = Mage::getModel('catalog/product')->load($productId)->getAttr();
```

Each model should be loaded only once (if there is a reason to load it at all) to optimize performance:

```
$product = Mage::getModel('catalog/product')->load($productId);
$name    = $product->getName();
$sku     = $product->getSku();
$attr    = $product->getAttr();
```

Another point to consider is that sometimes it is not even necessary to load the model because you don't need to work with model entity itself. The following code loads a product merely to get the product ID:

```
$product = Mage::getModel('catalog/product')->loadByAttribute('sku', $sku);
$res['id'] = $product->getId();
```

In this case you can use the native product method **getIdBySku()** that will work much faster:

```
$res['id'] = Mage::getModel('catalog/product')->getIdBySku($sku);
```

Magento EC[g]

Collections

Collections

Magento collections provide a large set of operations to work with a set of models. They are quite optimized and efficient, but there are a few things to keep in mind when working with collections. Processing large data sets requires a lot of system and network resources, so it is extremely important to restrict the results by applying proper filters and limits, to avoid fetching and processing more data than necessary.

Developers often misuse collections; the two most common issues are redundant data set utilization and inefficient memory utilization.

REDUNDANT DATA SET UTILIZATION

Collections are often used to retrieve only one item by calling the **$collection->getFirstItem()** method or returning the first item on the first iteration of the loop. A common mistake of inexperienced Magento developers is not applying a limitation on the collection's query results.

It may be not obvious that the **$collection->getFirstItem()** method does not modify the collection's query results and restrict the result to one item itself. Therefore, if no restrictions are applied before it is called, it will load all the items of the collection as follows:

```
public function getRandomItem() {
    $collection = Mage::getResourceModel('mymodule/my_collection')-
>setRandomOrder();
    return $collection->getFirstItem();
}
```

Always remember to apply the limitation in the case where the result of the query is a set of more than one item, in order to improve code performance and scalability:

```
public function getRandomItem() {
    $collection = Mage::getResourceModel('mymodule/my_collection')-
>setRandomOrder()
        ->setPageSize(1);
    return $collection->getFirstItem();
}
```

Use the **$collection->setPageSize() and $collection->setCurPage()** methods to specify the limitation and offset, respectively, or modify the collection query directly:

$collection->getSelect()->limit().

Sometimes developers want to retrieve the number of items in a particular collection, without further processing of its items. It such cases most of them use **$collection->count()** or **count($collection)** constructs, which appear to be obvious and natural solutions. However, it is most definitely the wrong way, because all the items of the collection will be loaded from the database and iterated. It is much more efficient to call the **$collection->getSize()** method instead.

INEFFICIENT MEMORY UTILIZATION

Using the database adapter method **fetchAll()** to fetch large result sets will cause a heavy demand on system and possibly network resources. Magento developers often fetch and iterate result sets as follows:

```
$rowSet = $this->_getReadAdapter()->fetchAll($select);
foreach ($rowSet as $row) {
    //process row
}
```

On large amounts of fetched data, this code will execute for a very long time and PHP will probably run out of memory.

In the following example, each database row is fetched separately using the **fetch()** method to reduce resource consumption:

```
$query = $this->_getReadAdapter()->query($select);
while ($row = $query->fetch()) {
    //process row
}
```

The database server will execute only one query and the database buffer will be used for retrieving records one by one.

Also, rather than retrieving all of the data and manipulating it in PHP, using the database server to manipulate the result sets should be considered. For example, it may be more efficient to use WHERE clauses in SQL to restrict results before retrieving and processing them with PHP.

As a general rule, to enable best code performance and scalability, developers should avoid situations when a large data set is loaded into memory before it can be processed. For example, when dealing with massive XML documents, the XMLReader PHP library is recommended rather than SimpleXML because it works with the document stream and doesn't read the entire data set into memory.

Magento EC[g]

Conclusions

Conclusions

Although some of the described issues relate to the specifics of Magento model and collection features, others are common for PHP and even other technologies. One thing Magento PHP developers should keep in mind: Always be attentive to performance-critical code, avoid useless computations inside loops, try to learn and understand the features of the framework, and use them correctly. Do not forget about updating your PHP version; fortunately, it's getting better and faster each new release.

If you're dealing with small data volumes you may not care too much about code optimization. Use common sense—sometimes it is acceptable to fetch all data or process it in a loop, but always think ahead and consider how your code will perform and scale in the future.

References

PHP performance tips – https://developers.google.com/speed/articles/optimizing-php

Dov Bulka and David Mayhew, *Efficient C++ Performance Programming Techniques*. Addison-Wesly, 1999.

Improving .NET Application Performance and Scalability. Chapter 5 — Improving Managed Code Performance http://msdn.microsoft.com/en-us/library/ff647790.aspx

http://php.net/manual/en/function.count.php

http://php.net/manual/en/class.xmlreader.php

http://php.net/manual/en/pdostatement.fetchall.php