www.phparch.com

June 2019
Volume 18 - Issue 6

php[architect]

How to Tame Your Data

Serverless PHP With Bref, Part Two

Containerizing Production PHP

Map, Filter, and Reduce in PHP

Three Interesting MySQL 8.0 Features for Developers

Free Sample Article

ALSO INSIDE

Education Station: Data Structures, Part Two

Community Corner: Philosophy and Burnout, Part Two—Logic Fails

Internal Apparatus: Memoization

Security Corner: Credentials and Secrets Management

finally{}: Conferences and Community

PHP[WORLD] 2019

Save the Date!

October 23-24

Washington D.C.

world.phparch.com

We're Hiring!

Learn more and apply at automattic.com/jobs.

Automattic wants to make the web a better place. Our family includes **WordPress.com**, **Jetpack**, **WooCommerce**, **Simplenote**, **WordPress.com VIP**, **Longreads**, and more. We're a completely distributed company, working across the globe.

AUTOMATTIC

Memoization

Edward Barnard

Memoization is another form of memory use. I see it regularly used for improving application performance, but it's also used to improve compiler performance. To learn the concept, we'll use a PHP project with database tables, then look at other ways to use the technique.

"Memoization" is a strange word. It names a technique for saving the result of expensive function calls. Where additional calls to the same function with the same input are guaranteed to produce the same output, we can save that output (we're making a "memo," which is where the name comes from). The next time we make that call, we first check for the saved result of a previous calculation (with the same inputs).

For example, if we are reading the exact same database row, assuming the row has not been updated, we should receive the same result every time. That operation is, therefore, a candidate for memoization.

Last month we looked at memory abstractions—concepts we use all the time without really thinking about them. We also introduced the Swiss Adventure project as an exercise in memory management but leaving the details for this month.

At php[tek] 2019, Samantha Quiñones[1], in *Caching on the Bleeding Edge* for the Advanced PHP track, explained memoization. It's an important technique you've likely used yourself. Compilers use the technique as well—and that's why we're here.

I prefer to work from the known to the unknown (when I can). That is, take something familiar as my starting point, before working into unfamiliar territory. I have a project that I use for teaching PHP's PDO[2] and, specifically, constructing MySQL prepared statements.

My Prepared Statements Project uses memoization as a performance enhancement—we'll focus on where memoization fits into that picture. We'll then see other applications of memoization and related techniques, including in the original Swiss Adventure.

Performance

Our project begins with a benchmark measuring the performance of various file-import techniques. See Figure 1. It's important to gather real information *before* considering optimization. As we'll see below, performance optimizations should only be considered when *proven* to be necessary.

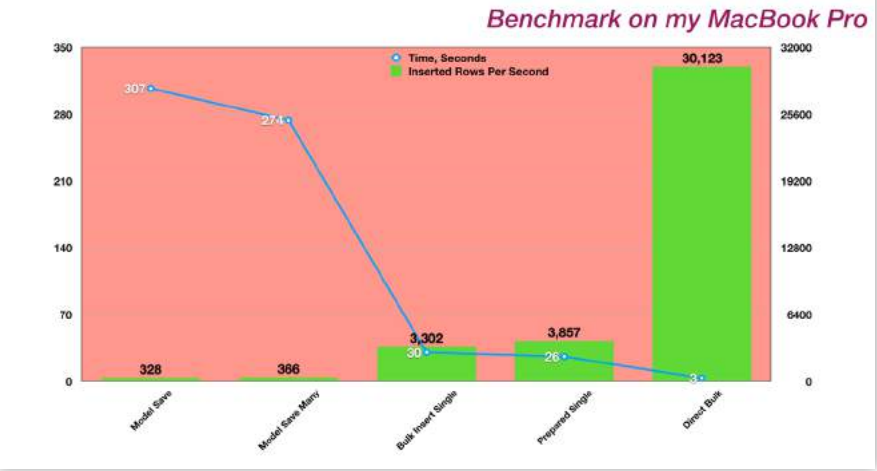
Performance improvements always come at a cost. Sometimes it's the "time versus space" trade-off as we learned last month. The more we make use of faster memory such as L1 processor cache, the faster our software can run. Other times, as here, performance improvements come from choosing different programming techniques. At the cost of greater development time, we improve run time.

A classic example is using single `'a string'` versus double `"another string"` quotes around a PHP string. Which one is faster? At what point would it make a difference? Based on this benchmark[3], we'd need more than 1,500 string manipulations to add a single millisecond to the page load time with PHP 7.0.

If we're doing 1,500 string manipulations during a web page load, we should consider different programming techniques before we worry about prematurely optimizing single and double quotes!

For the record, I use both single and double quotes. I try to make the code as readable as possible to the likely audience. I use double quotes for simple variable interpolation, and generally, use single quotes otherwise (to signal that there is no variable interpolation involved).

Figure 1. Benchmark 100,00 Row Inserts



1 *Samantha Quiñones:* https://twitter.com/ieatkillerbees

2 *PDO:* https://php.net/book.pdo

3 *this benchmark:* https://php.net/language.types.string#120160

Donald Knuth writes[4]:

> *Programmers waste enormous times thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

Meanwhile, the law of the instrument[5] warns of over-reliance on a specific skill. Abraham Maslow explained, "I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."

In this case, we're looking at various ways to import 100,000 rows from a file and insert them into a database. From the benchmark, we can see the two results on the left, "Model Save" and "Model Save Many," are an order of magnitude slower than the two middle results, and two orders of magnitude slower than the right-most result.

However, we don't normally insert 100,000 rows into a database on a normal web page load! So, we definitely don't need to convert normal web applications to use the faster techniques examined with this benchmark. We have a hammer, but not everything is a nail, so to speak.

On the other hand, many web applications do need to perform file imports from time to time. When the file import is taking too much time, impacting normal operations, that's the time to consider some type of optimization.

Let's begin optimizing by looking at the table design. The import file, in CSV format (comma-separated values), looks like Figure 2. The first row contains the column names.

We're looking at GPS data exported from my Garmin hand-held GPS device used for hiking. We imported 100,000 rows of data for the benchmark. The file-import details don't matter; we're focused on efficiency and memoization.

Note the first column, "motion," contains the same value for all rows shown. We have duplicated content—providing an opportunity for improving efficiency.

The sixth column, "nearest," shows the nearest known waypoint for that specific location. All rows again have duplicate content.

We can normalize[6] our database table. That is, we can refactor the table structure. We'll extract "motion" to be placed in its own table, and we'll extract "nearest" to become its own table. The "motion" table only needs three rows because there are only three possible motion values in the entire import file.

By converting the text column (averaging about 6 bytes per row) to a pointer to the motion table (requiring one byte per row), we've saved a few bytes of storage per row. Saving half a megabyte (5 bytes saved times 100,000 rows) doesn't, in itself, justify the added complexity of using multiple tables. But with billions of rows, backup copies, and multiple replications, and so on, the savings can add up.

Our new `motion` table looks like this:

```
CREATE TABLE `motion` (
  `id` tinyint(3) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  UNIQUE KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

Our `nearest-waypoint` table is similar:

```
CREATE TABLE `waypoint` (
  `id` smallint(5) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  UNIQUE KEY `name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

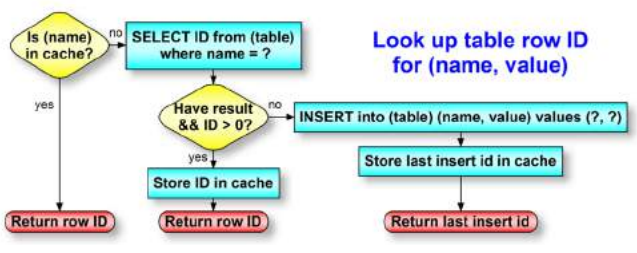
So far, we haven't seen anything unusual. We're refactoring our import table to slightly improve row-insert performance. Fewer bytes per row get sent to the main table; inserts to the other two tables are relatively rare as compared to the main table, or are they?

One approach is to insert the same value over and over again to `motion` or `waypoint`. The unique-key constraint prevents that. We could read the same value over and over again, to see if it's already there, and use its row ID in the main table. Or, we could read it once, and remember the answer (the row ID). That's where memoization comes in. See Figure 3.

Figure 2. CSV File

```
motion,lat,lon,ele,time,nearest,distance,feet,seconds,mph,climb
Drive,47.355681,-122.503923,110.09,"2017-09-04 12:37:00","Marshall House",60940,326,5,44.43,-3.06
Drive,47.353894,-122.503930,110.44,"2017-09-04 12:37:10","Marshall House",60848,652,10,44.45,0.35
Drive,47.352840,-122.503945,113.54,"2017-09-04 12:37:16","Marshall House",60799,385,6,43.69,3.10
Drive,47.351188,-122.504016,113.48,"2017-09-04 12:37:26","Marshall House",60740,603,10,41.11,-0.06
Drive,47.349998,-122.504038,109.54,"2017-09-04 12:37:33","Marshall House",60693,434,7,42.29,-3.94
Drive,47.348913,-122.504053,106.38,"2017-09-04 12:37:39","Marshall House",60653,396,6,44.98,-3.16
Drive,47.347870,-122.504062,103.11,"2017-09-04 12:37:45","Marshall House",60614,380,6,43.24,-3.27
```

Figure 3. Lookup Utility Flow



4 *writes:* https://phpa.me/wikiquote-knuth-art

5 *law of the instrument:* https://phpa.me/wikip-law-instrument

6 *normalize:* https://phpa.me/wikip-3rd-normal

Listing 1. Memoization Utility

```php
<?php

namespace App\Util;

use Cake\Database\Connection;
use Cake\Database\StatementInterface;

final class LookupUtil
{
    /** @var int Maximum cached IDs */
    private static $cacheLimit = 200;

    /** @var array One singleton per table */
    private static $instances = [];

    /** @var Connection */
    private $connection;

    /** @var StatementInterface */
    private $query;

    /** @var StatementInterface */
    private $insert;

    /** @var string Table name */
    private $table;

    /** @var array Cache of IDs given name */
    private $cache = [];

    private function __construct(Connection $connection,
                                 $table,
                                 array $dependencies) {
        $this->connection = $connection;
        $this->table = $table;
        if (count($dependencies)) {
            $this->injectDependencies($dependencies);
        }
        $this->prepareStatements();
    }

    /**
     * Testing support
     *
     * @param array $dependencies
     * @return void
     */
    private function injectDependencies(array $dependencies) {
        foreach ($dependencies as $key => $value) {
            if (property_exists(static::class, $key)) {
                $this->$key = $value;
            }
        }
    }

    private function prepareStatements() {
        if (!$this->query) {
            /** @noinspection SqlResolve */
            $sql
                = 'SELECT id FROM prepared_statements.' .
                $this->table . ' WHERE `name` = ?';
            $this->query = $this->connection->prepare($sql);
        }
        if (!$this->insert) {
            /** @noinspection SqlResolve */
            $sql
                = 'INSERT INTO prepared_statements.' .
                $this->table . ' (`name`) VALUES (?)';
            $this->insert = $this->connection->prepare($sql);
        }
    }

    public static function lookup(Connection $connection,
                                  $table, $value) {
        $instance = static::getInstance($connection, $table);
        return array_key_exists($value, $instance->cache) ?
            $instance->cache[$value] :
            $instance->runLookup($value);
    }

    /**
     * @param Connection $connection
     * @param string $table
     * @param array $dependencies
     * @return LookupUtil
     */
    public static function getInstance(Connection $connection,
                                       $table,

                                       array $dependencies = []) {
        if (!array_key_exists($table, static::$instances)) {
            static::$instances[$table]
                = new static($connection, $table, $dependencies);
        }
        return static::$instances[$table];
    }

    private function runLookup($value) {
        if (count($this->cache) >= static::$cacheLimit) {
            $this->cache
                = []; // Cache got too big; clear and start over
        }
        if (!$this->query) {
            // Should only happen when developing unit tests
            throw new \InvalidArgumentException('No query for ' .
                $this->table);
        }
        $parms = [substr($value, 0, 255)];
        $this->query->execute($parms);
        $row = $this->query->fetch('assoc');
        if (is_array($row) && array_key_exists('id', $row)) {
            $id = (int)$row['id'];
        } else {
            $this->insert->execute($parms);
            $id = (int)$this->insert->lastInsertId();
        }
        $this->cache[$value] = $id;
        return $id;
    }
}
```

Note that both tables have a column `name`. Both tables have identical structure, except the primary key uses a smaller-range data type for `motion`. Our memoization[7] technique is a form of read-through caching[8].

Our cache is a simple key-value array. The key is the `name` column from our table, and the key's value is the `id` column for that table row. We maintain one cache array for each table.

1. If our array (the cache) contains the target value ("Drive" for the `motion` table, or "Marshall House" for the `waypoint` table), return the row ID for that target value.
2. Otherwise, our value is not in cache. Attempt to read that row from the database.
3. If we received a result from the database read, we perform "read-through caching." That is, we store the just-read target value in our cache, so we remember that row ID. That's called "memoizing" the value. If we need to look up this same target value in the future, it will be available to us from the cache (as Step 1 above) and skips querying the database. Return the now-cached row ID for that target value.
4. However, if we did not receive a result from the database read, we need to add this target value to the table. Do the row insert and note the last insert ID. Memoize our target value by saving the target value, and the last insert ID, in our cache. Return the now-cached row ID for that target value.

The PHP implementation is Listing 1.

This utility uses an array of Singletons, with one singleton per table. The Singleton is generally considered an anti-pattern, because it's a way of creating global state. Software depending on global state becomes more difficult to test—and thus the singleton is generally discouraged.

However, there is one place we accept singletons—for resource connections. We use a single connection for a database, for example. Our `LookupUtil` class does something similar. Each of its singletons is memoizing the contents of a specific table. The singleton makes sense—but that doesn't make it easier to test!

This utility allows arbitrary dependency injection for testing purposes. The class constructor `__construct()` is private. Its dependency array comes from static method `getInstance()`, which is public. We can, therefore, write unit tests which pre-fill the cache, allowing us to exercise all code paths. Unit tests are online at https://phpa.me/ewb-cakefest17-tests. The cache test looks like Listing 2.

That's how memoization works. It's a way of improving performance by caching results inside our program code. It comes with a cost—added complexity. It's one more place where things can go wrong. We could, for example, run out of memory by trying to memoize a billion result rows. Listing 1 solves this by emptying the cache once it reaches 200 values. (See the top of function `runLookup()` near line 86.)

Recursion

Wikipedia[9] describes how a memoized function remembers the results corresponding to some specific set of inputs. If we know a function will return the same result for given inputs, we can memoize that result, creating some sort of cache that, given this set of inputs, we can return the previously-determined result.

Note that this is the same concept as our PHP utility. Wikipedia uses the example of recursively computing a factorial number:

```
function factorial (n is a non-negative integer)
    if n is 0 then
        return 1 [by the convention that 0! = 1]
    else
        return factorial(n - 1) times n [recursively invoke
                                         factorial with the
                                         parameter 1 less than n]
    end if
end function
```

The problem with the above algorithm is its performance. It's costly in time, and costly in space. The recursive nature means that for computing *n* factorial, we have *n* stack frames comprising the calculation in progress.

Calling a function, in PHP, is expensive relative to not calling a function. The stack frame controlling the function call must be allocated and initialized. However, that observation (that function calls are expensive) is not an excuse to indulge in premature optimization! It's far better to write well-structured, readable and maintainable, code. Only optimize when you have evidence proving it's necessary!

Listing 2

```
1. public function testAsFixture() {
2.     $table = 'fest_events';
3.     $cache = ['cache' => ['Event Two' => 2, 'Event Three' => 3]];
4.     LookupUtil::getInstance($this->connection, $table, $cache);
5.     static::assertSame(
6.         2, LookupUtil::lookup($this->connection, $table, 'Event Two')
7.     );
8.     static::assertSame(
9.         3, LookupUtil::lookup($this->connection, $table, 'Event Three')
10.    );
11. }
```

7 *memoization:* https://phpa.me/wikip-cache-memo

8 *read-through caching:* https://phpa.me/oracle-readthrough-cache

9 *Wikipedia:* https://phpa.me/wikip-memoization

To calculate six factorial, we must:

1. Multiply 6 by five-factorial,
2. Multiply 5 by four-factorial to provide five-factorial,
3. Multiply 4 by three-factorial to provide four-factorial,
4. Multiply 3 by two-factorial to provide three-factorial,
5. Multiply 2 by one-factorial to provide two-factorial,
6. Multiply 1 by zero-factorial to provide one-factorial,
7. Provide one as zero-factorial,
8. Pop the results back up the stack, ultimately returning six-factorial.

If, for some reason, we're computing various factorial numbers, we could save a lot of time and space by memoizing each result the first time it's computed. We call that amortizing the calculation cost across multiple calculations.

In fact, we could even store pre-computed factorial values in a lookup table. That lookup table could be in memory, a database, or any other long-term storage. Such tables used to be published as print books. We had books for chemistry calculations, prime numbers, random numbers, even ocean navigation[10].

The rainbow table[11] is another application of this concept. It's a precomputed table for cracking passwords. If we know the password-encrypting (hashing) algorithm, we could pre-compute the hash for a dictionary or other list of known passwords. Upon obtaining a list of "secure" but unsalted hashed passwords, we compare each hash to our rainbow table. Any time we find a match, we have obtained the plaintext password without the computing expense beyond one lookup.

This, by the way, is why it's important to add a varying salt to a password before hashing it. It defeats the rainbow-table attack. There are other possible attacks, to be sure, but the rainbow table is a great example of memoizing. Criminals can—and do—share rainbow tables.

"Memoizing" refers to short-term caching inside a function or method. However, the general concept can be implemented for the long term, such as publishing the table in a book or sharing rainbow tables as files.

Compilers

Compilers, in general, use parsers. A parser attempts to recognize program statements and syntax. A given language token, such as a left parenthesis (, could indicate one of several different parts of the PHP language. It could be part of a function declaration, an array declaration, a function call, etc.

The parser, therefore, "runs down various rabbit holes" to create the correct syntax tree for our software. The recursive descent down these rabbit holes can be assisted by memoization[12]. Any rabbit hole that, given a set of inputs, always returns the same result, can be memoized, allowing the compiler to complete its parsing steps more quickly.

Tight Memory

Another example comes from Swiss Adventure—not the PHP project, but the original assembly-language code. The original code ran on a processor with only 128 Kbytes of memory available, and therefore the program could only have small pieces resident in local memory at a time.

However, Swiss Adventure has a large table of locations, place descriptions, navigation information from one location to the next, and so on. How does it all fit?

We use a similar technique, based on the available hardware. In addition to the 128KB of local memory in the CPU, the processor had a large store of cache memory, called Buffer Memory or "MOS" in the code. "MOS" refers to a type of memory based on silicon Metal-Oxide-Semiconductor technology.

The code is online[13]. Here's the technique for getting an adventure description into local memory so it can be displayed to our hapless adventurer.

1. Allocate eight words of space in Local Memory. In modern terms, we'd call that heap space.

```
        AA = D'8
        GETMEM    AA,D1          .Allocate space for SW@ entry
```

The above code places the (decimal) value 8 into variable AA. We then call the memory allocator to provide us that much memory, placing the allocated memory address in variable D1.

My point in showing you the assembly code is that the language really doesn't matter. So long as we understand the task we're trying to accomplish, we can recognize its accomplishment in any programming language.

We're all familiar with PHP syntax. PHP and JavaScript are both "C-like" languages, meaning statements, expressions, and general structure are similar to C. Thus reading the C compiler code, given our knowledge of PHP syntax, is relatively easy. If we can follow the assembly code above, the PHP compiler will be easier!

The PHP compiler allocates space for PHP variables. The variables are called `Zval` entities, and we use `emalloc` to allocate memory for them from the heap. It's the same operation as the sample assembly code above.

2. We're looking for the text with our description location in Buffer Memory. That's the large cache memory. Once we locate that description, we copy the information into local memory.

10 *ocean navigation:* https://phpa.me/wikip-practical-navigator

11 *rainbow table:* https://phpa.me/wikip-rainbow-table

12 *memoization:* https://phpa.me/wikip-memoize-parsers

13 *online:* https://phpa.me/swiss-adv-ioplm

```
BL = ML + AA            .Absolute Buffer Memory address
BU = BU+1,C#0
AA = 2
MOSR      BU,BL,D1,AA   .Read SW@ entry
```

Buffer Memory addresses are so big that we have to hold them in two variables (BL and BU, for Buffer Memory Lower and Buffer Memory Upper). Again, the exact syntax does not matter, but the principles do. We have two structures with differing word sizes. Eight units of local memory (allocated in step one) correspond to two units of Buffer Memory. So in reading buffer (MOS) memory with `MOSR`, we're reading two units (variable `AA`), with the destination being our local memory address in variable `D1` from step 1.

In fact, we've only read 16 bytes (two 64-bit words). That's not enough for a full location description for an adventure game. What we actually read was a pointer to the text description. The assembly-language code then proceeds to allocate memory for the full text description, and read it into local memory.

We then display the description on the operator console and release the local memory. If we didn't release the memory we'd have a memory leak—not a good thing!

The original Swiss Adventure is largely an exercise in memory management. At some level that's true of nearly all software, but luckily we don't usually need to see that detail. Memory considerations are abstracted away, that is, hidden out of sight.

Summary

Memoization is an important technique. At the cost of adding complexity to the software, we can increase performance. However, it only makes sense where we're repeating the same calculation (or lookup, or other operation) many times.

The more general concept of lookup tables can persist long term—even years rather than fractions of a second with memoization. Lookup tables even used to be published as print books. We took a peek at how Swiss Adventure does a table lookup from one memory store to another.

Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching software concepts to others.
@ewbarnard

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe

php[**architect**]