

Tests et benchmark en PHP 5

par Mathieu Fernandez ([Accueil](#))

Date de publication : 3/11/2007

Dernière mise à jour :

Voilà donc un petit article sur les benchmark en PHP 5.

I - Introduction.....	3
I-A - Remerciements.....	3
I-B - Préambule.....	3
I-C - Conditions de test.....	3
II - 'Apostrophe' VS "Guillemets".....	4
II-A - L'affectation simple.....	4
II-B - L'affectation avec variable.....	4
III - L'affichage à l'écran.....	6
III-A - Affichage d'une chaîne avec une variable.....	6
III-B - "Plusieurs petits echo" VS "un seul gros echo".....	6
IV - Les variables.....	8
IV-A - Le test de nullité.....	8
IV-B - Déclaration et initialisation de variable.....	8
V - Les structures de contrôle.....	9
V-A - Les tests de condition.....	9
V-B - Les boucles.....	10
IV-C - Le "count" et les tableaux dans une boucle.....	11
VI - Les fichiers.....	12
VI-A - La lecture d'un fichier.....	12
VI-A-1 - Dans une chaîne.....	12
VI-A-2 - Dans un tableau.....	12
VI-B - L'écriture.....	13
VII - Les tableaux.....	14
VIII - Les remplacements dans les chaînes de caractères.....	15
IX - Conclusion.....	17
X - Liens utiles.....	18

I - Introduction

I-A - Remerciements

Je souhaiterais, avant toute chose, remercier tous les membres de l'équipe PHP pour l'aide qu'ils m'ont fournie dans la rédaction de cet article.

Je remercie en particulier **Yogui** et **lubito** sans qui je n'aurais pas pu faire cet article.

I-B - Préambule

Cet article est une suite de tests que j'ai réalisée sur ma propre machine et qui permet de se faire une idée sur le temps d'exécution de telle ou telle fonction PHP.

Je tiens à préciser que cet article ne vous fera pas gagner un temps considérable si d'emblée vos pages PHP sont mal codées. Ces mini-optimisations ne remplacent évidemment pas un algorithme bien pensé.

Pensez donc à améliorer les algorithmes de vos pages PHP avant même de penser à "customiser" votre code.

I-C - Conditions de test

Les tests suivants ont été réalisés sur un serveur Apache 2.2.4 (Win32) avec la version 5.2.3 de PHP.

Pour chaque test, nous avons exécuté 5 fois le même code, à savoir une boucle de 3 millions d'itérations, ceci afin d'avoir des résultats concluants.

La boucle est la suite :

```
<?php
$nb_occur=3000000;

$time_start = microtime(true);

for ($i=0 ; $i<$nb_occur; $i++)
{
    //le test
}

$time_end = microtime(true);
$time = $time_end - $time_start;

echo 'Durée : '.$time.' secondes<br/>';
?>
```

Lorsque les conditions de tests ont été différentes pour des raisons quelconques (par exemple effectuer 3 millions de fois un echo d'un millier de caractères c'est pas très intéressant :)), je l'indique dans cet article.

II - 'Apostrophe' VS "Guillemets"

Ce chapitre est inspiré de l'article de Pierre-Baptiste Naigeon **Apostrophes ou guillemets : lesquels choisir ?**. Je vous conseille donc d'y jeter un coup d'oeil si vous souhaitez avoir de plus amples informations sur la question.

II-A - L'affectation simple

Nous allons tout d'abord commencer par faire une simple affectation de chaîne de caractères à une variable pour voir la différence entre les 2 temps d'exécution.

```
//1° cas
$chaine='Ceci est une chaîne.';

//2° cas
$chaine="Ceci est une chaîne.";
```

Durée en s

1° cas	1.50106406212
2° cas	1.49910378456

Aucune différence notable, ... passons au test suivant.

II-B - L'affectation avec variable

Cette fois nous affectons, en plus de la chaîne, la variable \$i qui correspond au compteur des 3 millions d'itérations.

```
//1° cas
$chaine='Ceci est une variable : '.$i;

//2° cas
$chaine="Ceci est une variable : ".$i;

//3° cas
$chaine="Ceci est une variable : $i";
```

Durée en s

1° cas	3.96654582024
2° cas	3.97251200676
3° cas	4.95615792274

Voilà une différence et pas des moindres : le troisième cas met 25% plus de temps que le premier et le second qui, eux, sont équivalents. Pourquoi ?

La raison est simple, le serveur PHP décompose la chaîne de caractères dans le troisième cas afin de trouver et interpréter la variable \$i, chose qu'il ne fait pas dans les 2 autres cas.

En ce qui concerne l'efficacité pure et dure, la concaténation est donc préférable à l'utilisation de variable à l'intérieur même d'une chaîne de caractères : le premier et le second cas serait à ce moment là ex aequo. Pas tout à fait ...

En effet, il est conseillé d'utiliser les apostrophes car elles permettent d'avoir un code largement plus lisible quand on combine PHP et HTML :

```
//1° cas
echo '';

//2° cas
echo "<img src=\"chemin/vers/fichier/image.jpg\" alt=\"description image\" />";
```

Dans le premier cas, on utilise les apostrophes pour délimiter la chaîne de caractères ce qui permet d'éviter d'échapper les guillemets dans le code HTML (ce que nous sommes obligés de faire dans le second cas).

Pour conclure, la meilleure solution est donc d'utiliser les apostrophes et de concaténer les variables : cette méthode est non seulement plus rapide mais également plus lisible.


```
1111111111 1111111111 1111111111 1111111111 ' ;
```

Pour ce test, j'ai modifié les boucles car l'affichage 3 millions de fois de 1000 caractères est vraiment trop long. J'ai donc réalisé 2 séries : la première affichant 30 millions de caractères et la seconde affichant 1 million caractères.

	30 millions de caractères (durée en s)	1 million de caractères (durée en s)
1° cas	11.5267999172	0.39734005928
2° cas	10.7492429256	0.329221200943

La théorie est respectée : le second cas est plus rapide bien que la différence ne soit pas si importante. Par contre, outre le fait que ce soit légèrement plus rapide d'afficher tout dans un gros bloc, c'est surtout bien plus lisible que de faire un echo par ligne comme on le voit souvent :)

Une solution alternative et très efficace est d'utiliser **ob_start** couplé à **ob_end_flush**. Il faut appeler ob_start au début de vos pages PHP et ob_end_flush à la fin.

Par exemple sur notre première série (affichage de 30 millions de caractères), la durée d'exécution est de 4.90419578552 secondes soit un gain de temps de plus de 50% !!!

Ceci s'explique car ob_start permet d'envoyer toutes les données, qui devraient être affichées à l'écran, dans le tampon de sortie et c'est seulement lors de l'appel de ob_end_flush que les données sont restituées au navigateur.

Cela permet d'éviter les allers-retours incessants entre le serveur et le navigateur dus à la répétition des echos d'où un gain de temps plus que conséquent.

IV - Les variables

Voyons maintenant différents tests sur les variables. Tout d'abord, 2 façons de tester si une variable est nulle puis 2 façons de faire une affectation identique à 2 variables.

IV-A - Le test de nullité

Souvent les gens utilisent la fonction `is_null` afin de savoir si une variable est NULL ou pas. Est-ce vraiment la meilleure solution ?

```
//1° cas
$i===NULL;

//2° cas
is_null($i);
```

Durée en s

1° cas	1.1594440937
2° cas	2.51942610741

Comme vous pouvez le voir la fonction `is_null` est 2 fois plus lente que l'opérateur binaire `===`.

L'utilisation d'une fonction est généralement plus gourmande en mémoire qu'un opérateur : ceci à cause de l'obligation de sauvegarder l'environnement. Donc, en plus de la perte de performances, l'utilisation de la fonction `is_null` consomme de la mémoire vive.

Attention tout de même à ne pas confondre une variable nulle et une autre inexistante : les 2 tests ci-dessus renverront tous les 2 des notices si vous testez des variables non définies auparavant dans votre code. Pour pallier ce problème, il faut utiliser les fonctions `isset` ou `empty`.

IV-B - Déclaration et initialisation de variable

Voyons maintenant les différentes manières de déclarer et d'initialiser plusieurs variables avec la même valeur.

```
//1° cas
$var1 = 'une chaine';
$var2 = 'une chaine';

//2° cas
$var1 = 'une chaine';
$var2 = $var1;

//3° cas
$var1 = $var2 = 'une chaine';
```

Durée en s

1° cas	2.16453216553
2° cas	2.09558109283
3° cas	2.16578411102

Le temps d'exécution est quasi identique dans les 3 cas. Le second et le troisième cas sont en revanche plus propres au niveau du code (en effet, si la chaîne change il suffit de la modifier à un seul endroit).

Il est à noter que le fait de transtyper une variable augmente légèrement le temps d'exécution.

V - Les structures de contrôle

V-A - Les tests de condition

Nous avons testé ici les 3 différentes manières de traiter les conditions : la structure if/elseif/else, le switch et enfin l'opérateur ternaire.

Nous avons réalisé 2 jeux de tests : un jeu avec seulement 2 possibilités et un autre avec 8 possibilités. Le test consiste simplement à faire un modulo sur le compteur.

```
//1°cas
if($i%2 == 1) $test=0;
else $test=1;

//2°cas
switch($i%2)
{
case 1:
    $test=0;
    break;
default:
    $test=1;
}

//3° cas
$test = (($i%2 == 1) ? 0 : 1);
```

Durée en s

1° cas	0.887945795059
2° cas	1.14216017723
3° cas	1.089978790283

À la vue du premier jeu de tests, on pourrait dire que l'opérateur classique if/else est le plus rapide mais voyons voir si on rajoute des possibilités.

```
//1°cas
$res=$i%8;
if($res == 0) $test=0;
elseif($res == 1) $test=1;
elseif($res == 2) $test=2;
elseif($res == 3) $test=3;
elseif($res == 4) $test=4;
elseif($res == 5) $test=5;
elseif($res == 6) $test=6;
else $test=7;

//2°cas
$res=$i%8;
switch($res)
{
case 0:
    $test=0; break;
case 1:
    $test=1; break;
case 2:
    $test=2; break;
case 3:
    $test=3; break;
case 4:
    $test=4; break;
case 5:
    $test=5; break;
case 6:
    $test=6; break;
```

```

default :
    $test=7;
}

//3° cas
$res=$i%8;
$test = ($res == 0) ? 0 : (
    ($res == 1) ? 1 : (
    ($res == 2) ? 2 : (
    ($res == 3) ? 3 : (
    ($res == 4) ? 4 : (
    ($res == 5) ? 5 : (
    ($res == 6) ? 6 : 7))))));
    
```

	Durée en s
1° cas	1.92930579185
2° cas	1.83621811867
3° cas	2.26550005913

Si nous rajoutons des possibilités, nous pouvons voir que le switch est plus rapide que les 2 autres solutions. Ceci nous permet de conclure qu'il vaut mieux utiliser l'opérateur classique if/else quand il y a peu de possibilités et le switch quand il y en a plus.

Il est à noter que l'opérateur ternaire n'est le plus rapide dans aucun des 2 jeux de tests et cela tombe bien étant donné que son code est relativement illisible.

V-B - Les boucles

Nous testons dans ce paragraphe, à travers 2 séries de tests, 3 manières de faire des boucles en PHP 5 : for, while et foreach + range.

Nous avons remplacé la boucle que nous utilisons pour tous les tests par les boucles suivantes :

```

//1° série : $cpt=3000000;
//2° série : $cpt=100000;

//1° cas
for($i=0; $i <$cpt; $i++) $test=$i;

//2° cas
$i=0;
while($i<$cpt)
{
    $test=$i;
    $i++;
}

//3° cas
foreach(range(0,$cpt) as $i) $test=$i;
    
```

	1° série (durée en s)	2° série (durée en s)
1° cas	1.52313993454	0.0494940280914
2° cas	1.34025406837	0.0419161319733
3° cas	Fatal error: Allowed memory size	0.119401931763

Le while est comme prévu la façon la plus rapide de faire une boucle mais il est évident qu'au niveau de la lisibilité du code, le for est mieux.

Pour ce qui est du foreach + range, il est à oublier pour principalement 2 raisons :

- si vous souhaitez faire de grosses boucles, vous allez tomber sur une "fatal error" (j'ai augmenté le memory_limit jusqu'à 128Mo mais rien n'y change).
- c'est long, trop long ... environ 3 fois plus qu'un while

IV-C - Le "count" et les tableaux dans une boucle

Très souvent on voit des bouts de code avec un count inséré dans le for, c'est mal ! Voyez par vous-même dans ce qui suit.

```
//1° cas
$k=0;
$tableau=array('a', 'b', 'c', 'd', 'e');
$n=count($tableau);
for($i=0; $i<$n; $i++) $k++;

//2° cas
$k=0;
$tableau = array('a', 'b', 'c', 'd', 'e');
for($i=0; $i<count($tableau); $i++) $k++;
```

Durée en s

1° cas	17.314617157
2° cas	27.6772232056

Le résultat est sans appel, il faut toujours calculer la taille du tableau avant de faire une boucle. Dans le cas contraire, cette taille est recalculée à chaque tour de boucle, ce qui est une perte de temps considérable.

VI - Les fichiers

VI-A - La lecture d'un fichier

Pour la lecture nous avons réalisé, à chaque fois, le même test qui consiste à ouvrir, lire puis fermer un fichier 3000 fois. Nous avons réalisé ce test sur 3 fichiers : le premier de 1 octet, le second de 1Ko et le troisième de 1Mo.

VI-A-1 - Dans une chaine

Voici les 2 codes comparés :

```
//1° cas
$fichier=file_get_contents('test.txt');

//2° cas
$fp = fopen('test.txt', 'r');
$fichier = fread($fp, filesize('test.txt'));
fclose($fp);
```

	Fichier 1 octet (durée en s)	Fichier 1 Ko (durée en s)	Fichier 1 Mo (durée en s)
1° cas	0.314026117325	0.323743124008	10.8227910995
2° cas	0.27210401535	0.285511016846	9.0904991627

Nous pouvons d'ores et déjà tirer 2 conclusions : lire un fichier "a la mano" est plus rapide qu'en utilisant la fonction **file_get_contents**.

Le temps d'exécution s'envole lorsque les fichiers à lire deviennent gros. En effet entre un fichier d'un seul octet et un 1000 fois plus gros, le temps est quasiment le même. Par contre, entre un fichier d'un Ko et un d'un Mo, le temps d'exécution est à peu près 30 fois plus long.

VI-A-2 - Dans un tableau

```
//1° cas
$fichier=file_get_contents('test.txt');
$lignes = split('[\n\r]', $fichier);

//2° cas
$lignes=file('test.txt');

//3° cas
$fp = fopen('test.txt', 'r');
$fichier = fread($fp, filesize('test.txt'));
fclose($fp);
$lignes = split('[\n\r]', $fichier);
```

	Fichier 1 octet (durée en s)	Fichier 1 Ko (durée en s)	Fichier 1 Mo (durée en s)
1° cas	0.385547161102	0.524013004303	124.7746181493
2° cas	0.379977817535	0.385583868027	11.8984890079
3° cas	0.366604846954	0.483832120895	113.7731289865

Le résultat est sans appel : la lecture d'un fichier dans un tableau est largement plus expéditive avec l'utilisation de la fonction **file**, surtout lorsque le fichier est "gros".

En effet la solution utilisant la fonction file est, environ, 10 fois plus rapide que les 2 autres sur de gros fichiers.

VI-B - L'écriture

Dans cette partie, nous avons écrit de différentes façons dans un fichier.

Tout comme pour la lecture, nous avons réalisé 3 séries : écriture de 1 octet, 1 Ko et enfin 1 Mo dans un fichier que l'on crée auparavant.

Nous avons réalisé 3000 fois chacune de ces opérations. La variable `$a_ecrire` contient la chaîne de caractères à écrire soit un caractère dans la première série, 1024 dans la seconde et 1048576 dans la troisième et dernière.

```
//1° cas
file_put_contents('test.txt', $a_ecrire);

//2° cas
$fp = fopen('test.txt', 'w');
fwrite($fp, $a_ecrire);
fclose($fp);
```

	Fichier 1 octet (durée en s)	Fichier 1 Ko (durée en s)	Fichier 1 Mo (durée en s)
1° cas	1.641685075759	1.919305665971	22.27384757997
2° cas	1.568504238128	1.911750719071	18.38060402871

Les résultats restent donc très similaires à la lecture dans un fichier : le temps d'exécution est plus long lorsque l'on utilise la fonction `file_put_contents` et il augmente exponentiellement avec la taille des données à écrire.

VII - Les tableaux

Nous allons maintenant voir ce qui semble être le mieux pour parcourir un tableau.

Pour cela nous allons réaliser les test sur 2 tableaux : un de 10000 cases et un de 5. Le tableau utilisé dans ces test est parcouru 1000 fois pour le test du tableau de 10000 cases et 1000000 de fois pour le test du tableau de 5 cases :

```
//1° série : $nb_cases=10000;
//2° série : $nb_cases=5;

$stab = range(1,$nb_cases);
srand((float)microtime()*1000000);
shuffle($stab);
```

Et voici les différentes façons de parcourir le tableau testé :

```
//1° cas
for ($i=0;$i<10000;$i++) $test=$stab[$i];

//2° cas
foreach($stab as $cle => $valeur) $test=$valeur;


//3° cas
foreach($stab as $valeur) $test=$valeur;

//4° cas
while(list($cle, $valeur) = each($stab)) $test=$valeur;
reset($stab);

//5° cas
$i=0;
while ($i<$nb_cases) $test=$stab[$i++];
```

	Tableau de 10000 cases (durée en s)	Tableau de 5 cases (durée en ms)
1° cas	6.04399681091	2.54988670349
2° cas	4.66101288795	2.15983390808
3° cas	3.88768601418	1.75404548645
4° cas	22.2336660919189	12.6601020813
5° cas	5.73187505722	2.24590301514

Dans tous les cas, on s'aperçoit que l'utilisation du foreach est nettement plus rapide que les autres, tout particulièrement sur des grands tableaux.

 **Attention à son utilisation tout de même : il travaille sur une copie du tableau spécifié, et pas sur le tableau lui-même. Par conséquent, le pointeur de tableau n'est pas modifié, comme il le serait avec la fonction each(), donc les modifications faites dans le tableau ne seront pas prises en compte dans le tableau original.**

VIII - Les remplacements dans les chaînes de caractères

Il existe de multiples manières pour remplacer des caractères dans une chaîne en PHP mais laquelle est la plus rapide ? Tout d'abord, si nous regardons la documentation PHP officielle de **ereg_replace**, celle-ci dit : *"preg_replace(), qui utilise la syntaxe des expressions rationnelles compatibles PERL, est une alternative plus rapide de ereg_replace()."*

Donc sans faire de tests, nous savons déjà que preg_replace va plus vite que ereg_replace.

Ensuite, comme l'indique encore une fois la documentation, preg_replace utilise des expressions régulières tandis que **str_replace** prend une chaîne de caractères simple en paramètre. La fonction str_replace est donc plus rapide que preg_replace.

Voyons donc str_replace en oeuvre.

Nous avons réalisé 2 séries de tests : la première consiste en remplacer "abcxyz" par "bbcyyz" tandis que la seconde remplace "Hello ! abcdefghijklmnopqrstuvwxyz" par "BOnjOUr ! AbCdEfGhIjKlMnOpQrStUvWxYz".

Pour la seconde série nous n'avons réalisé qu'un million d'itérations au lieu des 3 millions habituels car les temps étaient trop élevés.

Voici les 4 utilisations de str_replace testées :

1° série

```
// $a = 'abcxyz';

// 1° cas
$a = str_replace('x', 'y', $a);
$a = str_replace('a', 'b', $a);

// 2° cas
$a = str_replace('x', 'y', str_replace('a', 'b', $a));

// 3° cas
$a = str_replace(array('x', 'a'), array('y', 'b'), $a);

// 4° cas
$a = strtr($a, 'ax', 'by');
```

2° série

```
// $a = 'Hello ! abcdefghijklmnopqrstuvwxyz'

// 1° cas
$a = str_replace('Hello', 'Bonjour', $a);
$a = str_replace('a', 'A', $a);
$a = str_replace('c', 'C', $a);
$a = str_replace('e', 'E', $a);
$a = str_replace('g', 'G', $a);
$a = str_replace('i', 'I', $a);
$a = str_replace('k', 'K', $a);
$a = str_replace('m', 'M', $a);
$a = str_replace('o', 'O', $a);
$a = str_replace('q', 'Q', $a);
$a = str_replace('s', 'S', $a);
$a = str_replace('u', 'U', $a);
$a = str_replace('w', 'W', $a);
$a = str_replace('y', 'Y', $a);

// 2° cas
$a = str_replace('a', 'A', str_replace('c', 'C',
    str_replace('e', 'E',
        str_replace('g', 'G',
            str_replace('i', 'I',
                str_replace('k', 'K',
                    str_replace('m', 'M',
                        str_replace('o', 'O',
                            str_replace('q', 'Q',
                                str_replace('s', 'S',
                                    str_replace('u', 'U',
                                        str_replace('w', 'W',
                                            str_replace('y', 'Y',
```

2° série

```

str_replace ('Hello','Bonjour', $a))))))));

//3° cas
$a = str_replace(
    array('Hello','a','c','e','g','i','k','m','o','q','s','u','w','y'),
    array('Bonjour','A','C','E','G','I','K','M','O','Q','S','U','W','Y'),
    $a);

//4° cas
$a = strtr(str_replace('Hello','Bonjour', $a), 'acegikmoqsuwy', 'ACEGIKMQSUWY');
    
```

	1° série (durée en s)	2° série (durée en s)
1° cas	8.0842359066	57.0145347117
2° cas	7.71057006836	17.7155649662
3° cas	10.6429069042	15.9177130699
4° cas	4.14906597137	3.26361989975

Aucun doute à la vue de ces résultats : la fonction **strtr** est bien plus rapide que la fonction `str_replace` tout particulièrement lorsqu'il s'agit de modifier de nombreux caractères.

Un petit bémol cependant : `strtr` permet de modifier des caractères par d'autres caractères mais il ne peut pas remplacer de "mots" d'où l'utilisation couplé de `strtr` et de `str_replace` dans la seconde série.

Juste pour avoir un ordre d'idée, voyons quels résultats nous aurions obtenus avec `preg_replace`. Prenons le code suivant :

```

$a = preg_replace('/([acegikmoqsuwy])/e', "strtoupper('\\1')", $a);
    
```

Ce code affiche le temps d'exécution suivant : *Durée : 30.3204350471 secondes* ... sans commentaires.

Nous pouvons donc conclure que le meilleur moyen de remplacer des caractères dans une chaîne de caractères est l'utilisation de `strtr` couplé à `str_replace` s'il y a nécessité à remplacer des "mots" entiers.

IX - Conclusion

Comme vous avez pu le voir tout au long de l'article, il y a des techniques de programmation afin d'optimiser son code et de le rendre plus performant.

Toutes ces astuces de programmation ne pallieront jamais le manque de conception dans vos programmes ou bien tout simplement des algorithmes bâclés.

Pensez également qu'en optimisant le code à outrance pour gagner quelques microsecondes, il risque de devenir totalement illisible ce qui peut entraîner de sérieux problèmes de maintenance dans le cadre d'un travail en équipe. Il est évident que ces tests ne sont en rien des preuves et permettent seulement de se faire une idée sur certaines questions récurrentes.

X - Liens utiles

- **PHP Benchmark tests**
- **Apostrophes ou guillemets : lesquels choisir ?**
- **PHP Bench**