

Dear Drupal Community,

we, Bauer Digital KG from Hamburg, Germany, implemented one of Bauer's numerous popular public websites using Drupal 5.8.

Compared to many other Drupal-driven websites we read about, ours is quite large. It has around 80000 nodes delivering around 5 million page impressions per month. However, our website offers only few community features, which means that most visitors use the page without being logged in.

In the course of setting up the final live system consisting of two load-balanced web servers and two MySQL database servers in master-slave replication we encountered several performance problems in spite of extensively using the caching drupal module.

As we investigated on how these problems were caused and finally managed to improve many of them, we learned a couple of things about how Drupal 5.8 can be made more efficient without much effort - not just for ourselves but also for other users and developers.

Up to our knowledge some of these issues still persist in Drupal 6.

We hereby offer our insights to the Drupal Community to assist in the further development and maturing of Drupal.

Contents

Contents	1
Optimizations	2
Taxonomy Module	2
Function taxonomy_node_get_terms()	2
Function taxonomy_get_children()	2
Block Module	3
Function block_list()	3
Pathauto Module	4
Function _pathauto_include()	4
file.inc	4
Function file_scan_directory()	4
Database	6
Additional Indexes	6
Database configuration (MySQL)	7

Optimizations

Taxonomy Module

Function `taxonomy_node_get_terms()`

Original version:

```
$result = db_query(db_rewrite_sql('SELECT t.* FROM {term_node} r
INNER JOIN {term_data} t ON r.tid = t.tid INNER JOIN {vocabulary}
v ON t.vid = v.vid WHERE r.nid = %d ORDER BY v.weight, t.weight,
t.name', 't', 'tid'), $nid);
```

Performance-optimized version:

```
$sql = 'SELECT t2.*
FROM
(
    SELECT t.tid as tid
    FROM term_node r
    INNER JOIN term_data t ON r.tid = t.tid
    INNER JOIN vocabulary v ON t.vid = v.vid
    WHERE r.nid = %d
    ORDER BY v.weight, t.weight, t.name
) as t3
INNER JOIN term_data t2
WHERE t2.tid=t3.tid';

$result = db_query(db_rewrite_sql($sql), $nid);
```

Effect: The original version usually requires around 40-60 ms on our live system, the performance-optimized version needs just 1-2 ms.

Background: The performance-optimized version doesn't require MySQL to sort a table obviously containing a TEXT column, which could only be performed on the hard disk. Thus the optimized version is performed in RAM and works much faster.

Note: The same type of optimization follows again in the next passage.

Function `taxonomy_get_children()`

Original version:

```
$result = db_query(db_rewrite_sql('SELECT t.* FROM {term_data} t
INNER JOIN {term_hierarchy} h ON h.tid = t.tid WHERE parent = %d
ORDER BY weight, name', 't', 'tid'), $tid);
```

Optimized version:

```
$sql = 'SELECT t2.*
FROM
(
    SELECT t.tid as tid
    FROM term_data t
    INNER JOIN term_hierarchy h ON h.tid = t.tid
    WHERE parent = %d
    ORDER BY weight, name
) as t3
```

```

        INNER JOIN term_data t2
        WHERE t2.tid=t3.tid';

$result = db_query(db_rewrite_sql($sql), $tid);

```

Effect & background: same as for last optimization.

Patch file: taxonomy.patch

Block Module

Function block_list()

Original version:

```

$result = db_query(db_rewrite_sql("SELECT DISTINCT b.* FROM
{blocks} b LEFT JOIN {blocks_roles} r ON b.module = r.module AND
b.delta = r.delta WHERE b.theme = '%s' AND b.status = 1 AND (r.rid
IN (". db_placeholders($rids) .") OR r.rid IS NULL) ORDER BY
b.region, b.weight, b.module", 'b', 'bid'),
array_merge(array($theme_key), $rids));

```

Optimized version:

```

$sql = " SELECT b2.*
FROM
(
  SELECT DISTINCT b.module, b.delta, b.theme, b.status,
  b.weight, b.region, b.custom, b.throttle, b.visibility,
  crc32(b.pages) as pages_crc32, b.title
  FROM blocks b
  LEFT JOIN blocks_roles r
  ON b.module = r.module
  AND b.delta = r.delta
  WHERE b.theme = '%s'
  AND b.status = 1
  AND (r.rid IN (". db_placeholders($rids) .")
  OR r.rid IS NULL)
  ORDER BY region, weight, module
) as b3
LEFT JOIN blocks b2 on
  b3.module=b2.module and
  b3.theme=b2.theme and
  b3.status=b2.status and
  b3.weight=b2.weight and
  b3.region=b2.region and
  b3.custom=b2.custom and
  b3.throttle=b2.throttle and
  b3.visibility=b2.visibility and
  b3.pages_crc32=crc32(b2.pages) and
  b3.title=b2.title;";
$result = db_query($sql, array_merge(array($theme_key), $rids));

```

Background: For MySQL to ensure the DISTINCT condition, it has to create and use a temporary hard disk table. On the other hand, the optimized statement can be executed without a temporary hard disk table.

Effect: The original version requires about 50 ms on our system, the optimized version executes in about 6-7 ms.

Note: This optimization is only nearly 100% safe. It may happen (extremely rarely) that two datasets are joined in an invalid way. However from our understanding no security-, safety-, or stability-critical issues would be involved with that. We have been taking the risk and not encountered any problems that related to this change so far.

Patch file: block.patch

Pathauto Module

Function `_pathauto_include()`

Original version:

```
$pathauto_path = drupal_get_path('module', 'pathauto');
require_once("$pathauto_path/pathauto.inc");
require_once("$pathauto_path/pathauto_node.inc");
require_once("$pathauto_path/pathauto_taxonomy.inc");
require_once("$pathauto_path/pathauto_user.inc");
```

Optimized version:

```
if (!defined('pathauto_include_performed'))
{
    $pathauto_path = drupal_get_path('module', 'pathauto');
    require_once("$pathauto_path/pathauto.inc");
    require_once("$pathauto_path/pathauto_node.inc");
    require_once("$pathauto_path/pathauto_taxonomy.inc");
    require_once("$pathauto_path/pathauto_user.inc");
    define('pathauto_include_performed', true);
}
```

Effect: Depending on what the request delivers, the optimized Drupal delivers the page 20%-40% faster than the original one.

Background: On pages with many links it seems that this function gets called once or twice *per link*. In contrast to this fact it obviously actually needs to get called a maximum of *once* per (PHP) scope, so calling it 500 to 600 times and performing database operations each time in function `drupal_get_path()` definitely is a performance issue. Eliminating these repetitive function invocations and database operations lead to the reported speed up.

Patch file: block.patch

file.inc

Function `file_scan_directory()`

Original version:

```
function file_scan_directory($dir, $mask, $nomask = array('.',
'..', 'CVS'), $callback = 0, $recurse = TRUE, $key = 'filename',
$min_depth = 0, $depth = 0) {

    $key = (in_array($key, array('filename', 'basename', 'name'))) ?
    $key : 'filename';
    $files = array();
```

```

if (is_dir($dir) && $handle = opendir($dir)) {
    while ($file = readdir($handle)) {
        if (!in_array($file, $nomask) && $file[0] != '.') {
            if (is_dir("$dir/$file") && $recurse) {
                $files = array_merge($files,
file_scan_directory("$dir/$file", $mask, $nomask, $callback,
$recurse, $key, $min_depth, $depth + 1));
            }
            elseif ($depth >= $min_depth && ereg($mask, $file)) {
                $filename = "$dir/$file";
                $basename = basename($file);
                $name = substr($basename, 0, strrpos($basename, '.'));
                $files[$key] = new stdClass();
                $files[$key]->filename = $filename;
                $files[$key]->basename = $basename;
                $files[$key]->name = $name;
                if ($callback) {
                    $callback($filename);
                }
            }
        }
    }
}

closedir($handle);
}

return $files;
}

```

Optimized version:

```

function cached_is_dir($dir)
{
    global $file_inc__is_dir_cache;
    if (!isset($file_inc__is_dir_cache[$dir]))
    {
        $file_inc__is_dir_cache[$dir]=is_dir($dir);
    }
    return $file_inc__is_dir_cache[$dir];
}

function file_scan_directory($dir, $mask, $nomask = array('.',
'..', 'CVS'), $callback = 0, $recurse = TRUE, $key = 'filename',
$min_depth = 0, $depth = 0) {
    $key = (in_array($key, array('filename', 'basename', 'name'))) ?
$key : 'filename';
    $files = array();

    if (cached_is_dir($dir) && $handle = opendir($dir)) {
        while ($file = readdir($handle)) {
            if (!in_array($file, $nomask) && $file[0] != '.') {
                if (cached_is_dir("$dir/$file") && $recurse) {
                    $files = array_merge($files,
file_scan_directory("$dir/$file", $mask, $nomask, $callback,
$recurse, $key, $min_depth, $depth + 1));
                }
                elseif ($depth >= $min_depth && ereg($mask, $file)) {
                    $filename = "$dir/$file";
                    $basename = basename($file);
                    $name = substr($basename, 0, strrpos($basename, '.'));

```

```

        $files[$$key] = new stdClass();
        $files[$$key]->filename = $filename;
        $files[$$key]->basename = $basename;
        $files[$$key]->name = $name;
        if ($callback) {
            $callback($filename);
        }
    }
}
}
}

closedir($handle);
}

return $files;
}

```

Background: During a big share of usual requests to our application there originally happened around 30000 low-level operating system file operations per request, many of them caused by invocations of "is_dir()" in function "file_scan_directory()". By caching these responses, we managed to speed up these requests by about 30%.

Note: This kind of caching may not be completely safe, depending on which file system changes happen during the request. Cache content may serve obsolete information at times or even on a regular basis if the application creates, deletes or changes files or directories itself.

Since we started using the above optimization on our live system around six weeks ago we haven't encountered any problems, though. However, the better approach from our point of view would be to stop examining the file system so many times altogether by implementing a different behaviour in the calling module or in the Drupal core. It can't be necessary to traverse the project directory tree so many times within one PI.

Patch file: file.patch

Database

Additional Indexes

We discovered that many database requests that Drupal makes could not use appropriate table indexes in the original database structure that Drupal created for itself. Thus these queries had to perform table scans instead, which showed to be performance-critical. It was a minor problem for smaller tables (like user roles) that were not used in any big joins. Other queries could be speeded up considerably and now even utilize RAM much more economically.

All of the following indexes were introduced with a good reason. After they had been added to our system, no table scans were performed any more except for queries that couldn't avoid them. All other queries now use indexes.

Please note that several-column indexes should be created with their columns in exactly the order specified in the table below..

Table	Index columns (one index per row!)
search_index	word, type
blocks	theme

	status
	theme, status
filters	format
locales__meta	isdefault
	enabled
	name
	isdefault, enabled, name
menu	path
	pid
panels__page	did
permission	perm(100)
system	type
	status
	type, status
	weight
	weight, filename
view__view	block
	url
	page
	url, page
vocabulary__node_types	type
	vid

Database configuration (MySQL)

Table Cache

It showed that one of the *most important* performance influences on our system was the possibility for MySQL to use already open tables. Thus, although it's not a programming issue that can be improved in Drupal itself, we decided to report it here.

Problem background from our understanding: If MySQL needs to perform an operation on a table (e.g. SELECT or INSERT) and it accesses that table, the table must be opened first. This seems to incorporate at least the processes of opening the table file(s) and storing the file handles somewhere.

There is by default a so-called table cache in MySQL that obviously caches handles of open tables once they are opened. Then, if a table needs to be accessed again, it doesn't need to be opened again, which saves considerable about 200 ms per access based on our research.

A typical MySQL default installation uses a table cache between 64 and 256 slots. The value in the "huge" MySQL configuration example file delivered with MySQL is 512.

A typical installation of Drupal with a couple of modules installed can easily reach 200 tables created in the database. Our application, for instance, uses 329 database tables.

On top of everything said, an *own* slot is needed for each thread if a concurrent access to the same table occurs by several database threads.

All of this means from our experience that on a heavily used live system even a table cache of about 300 would definitely *not* be enough for 200 tables.

For our 329 tables we're currently using a table cache of 1500 and it's maxed out...

If a table access doesn't find an available opened table in the table cache, it has to re-open it. In addition, it drops the least recently used entry from the cache, which then in turn would have to be re-opened at a later time.

Hence, from our experience a properly scaled table cache is a *very important* precondition for a fast-performing Drupal live system.

This can be achieved by adding or modifying a line like the following to/in your MySQL configuration file (typically my.ini or my.cnf):
table_cache=1500

Maximum Open Files

In addition, we assume that for a large table cache to work you have to increase the maximum allowed number of opened files. We yielded good results with the following setting:

open-files-limit=8192

Query Cache

MySQL's query cache caches complete SQL queries and their results. Thus, if a query is made repeatedly (this happens very often in Drupal, even within the same request), the result is already available. The query execution time in this case is usually far below 1 ms.

From our experience increasing the query cache to 300 MB with a maximum result set size of 50 MB is a good idea:

query_cache_size=300M

query_cache_limit=50M

Max Heap Table Size

This setting defines the maximum size of temporary tables if they should be kept in RAM. If a temporary table grows beyond this size, it gets written to hard disk.

As mentioned, maintaining temporary tables in RAM and avoiding hard disk access is an important issue for keeping Drupal fast.

Hence, we increased the default MySQL value to the following setting:

max_heap_table_size=103M

Key Buffer Size

In its key buffer MySQL holds in-RAM caches of database table primary keys, or indexes, respectively. In effect this means that particularly for joins and complex statements hundreds of thousands of index accesses can be done in RAM instead of reading index entries from the hard disk. Drupal frequently uses joins, so we increased the setting to 155 MB:

key_buffer_size=155M