

Amazon Cloud Recipes

Gil Hildebrand
Chief Engineer / Squidoo.com

Squidoo.com

- Founded in 2005
- At that time, not many sophisticated frameworks or ORMs. Cloud computing wasn't mainstream yet.
- Custom PHP framework with traditional backend consisting of four physical servers: 2 web, 1 db, 1 staging
- Relied heavily on managed hosting provider for sysadmin

Squidoo Today

- 22 million pageviews per month
- 8 physical servers: 4 web, 3 db, 1 staging
 - Current hardware leaves plenty of room for growth
- 6 employees, no full time sysadmin on staff , and little interaction with hosting provider

Enter Amazon

- AWS provides framework for solving scaling problems
 - Break large, general components into many small, specialized components
 - Specialized components achieve greater efficiency
- Amazon has much more experience with these problems than I do
- Outsource the pain of worrying about uptime and reliability
- Set it and forget it (if architected that way)
- Not an all-or-nothing approach



Three Recipes

1. Quick and easy database backups to S3
2. Effortless static content hosted on CloudFront
3. Scaling traffic analytics with SimpleDB

AWS Tools

- Amazon CLI tools
- AWS Mgmt console (EC2 only - more on way)
- Firefox addons (S3Fox, Elasticfox, etc)
- Rightscale.com - full featured AWS GUI
- Tarzan wrapper lib for PHP

Database Backups

- Initially relied on mysqldump and gzip
- Managed hosting provider stored backups and maintained off-site copies
- BUT hosts can charge \$1 to \$3 per GB per month, and restores require tech support - can take hours, especially if backup is off-site

Enter S3

- Simple Storage Service is SIMPLE
- By definition files are stored off-site
- Pricing peaks at \$0.15 per GB stored, and bandwidth costs for backups are minimal
- No tech support call required

Automated Backups - S3 and Maatkit

PROBLEMS:

- S3 storage limit of 5 GB per file means a single tar archive won't work
- Single-threaded mysqldump is SLOW and BORING

SOLUTION:

- mk-parallel-dump FTW
 - One table per file
 - If files are still greater than 5GB, use --chunksize
 - Faster on many datasets

mk-parallel-dump

- mk-parallel-dump connects to a MySQL server, finds database and table names, and dumps them in parallel for speed.
- --tab flag uses SELECT INTO OUTFILE. Tab backup is generally faster (esp. on localhost) and also makes it possible to restore views and triggers.
- --stopslave issues STOP SLAVE to ensure data is not changed during backup. Issues START SLAVE once backup is complete.



Warning

The following recipe is designed to run on a slave that can tolerate a replication break while the backup is being taken.

One option, if you have the disk space, is to keep a slave running on your staging or utility server.

Here's what you'll need

1. AWS account with access key & secret key
<http://aws.amazon.com>
2. An S3 bucket named "my_s3_backup"
3. s3sync - rsync-like interface to S3 (requires ruby $\geq 1.8.4$)
<http://s3sync.net>
4. Maatkit - MySQL server management tools
<http://www.maatkit.org>

Database Backup Script

```
#!/bin/bash
```

```
BACKUP_DIR="/tmp/$(date +%F)-mysql-backup"  
mkdir $BACKUP_DIR
```

```
mk-parallel-dump --tab --stopslave --basedir \  
$BACKUP_DIR
```

```
s3sync.rb --recursive $BACKUP_DIR my_s3_backup:
```

```
rm -rf $BACKUP_DIR
```

```
echo "Backup complete\n"
```

Make It Better

- If you have the disk space, archive backups locally for a short period of time
- Build a completely independent tool to verify that new backup files make it to S3 every day
- ALWAYS have a restore script ready to go

Database Restore Script

```
#!/bin/bash
```

```
BUCKET="my_s3_backup"
```

```
BACKUP_DB="my_db"
```

```
TABLES="table1 table2 table3 table4"
```

```
if [ "$1" == "" ]; then
```

```
    echo "Missing argument: date of backup file to use (YYYY-MM-DD)"
```

```
    exit
```

```
fi
```

```
TEMPDIR=/tmp/mysql_restore
```

```
mkdir -p ${TEMPDIR}/${BACKUP_DB}
```

```
for t in $TABLES; do
```

```
    s3cmd.rb get \
```

```
        ${BUCKET}:${1}-mysql-backup/default/${BACKUP_DB}/${t}.000000.txt.gz \
```

```
        ${TEMPDIR}/${BACKUP_DB}/${t}.000000.txt.gz
```

```
done
```

```
mk-parallel-restore --disablekeys --noforeignkeys --binlog 0 --truncate \
```

```
    --local --tab ${TEMPDIR}
```

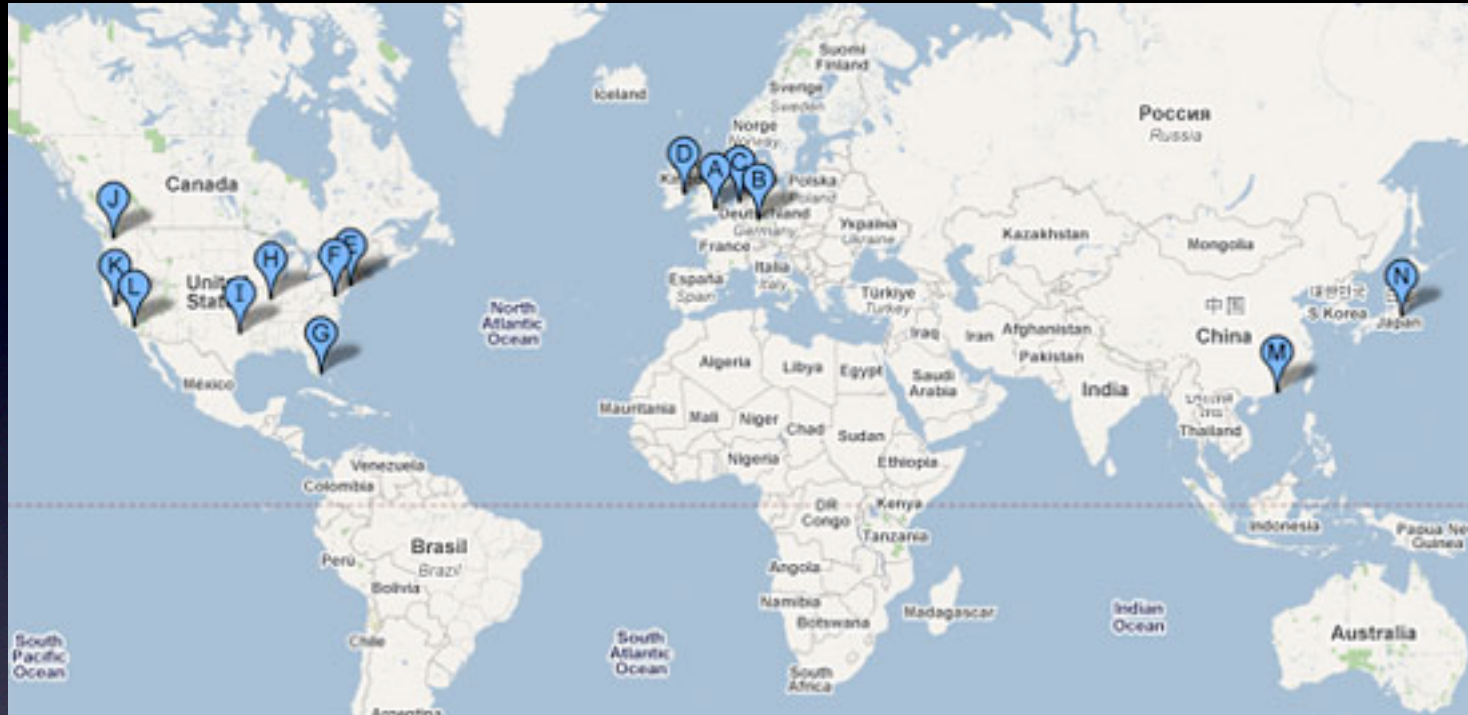
```
rm -rf ${TEMPDIR}
```

```
echo "Tables restored to ${BACKUP_DB} database!\n"
```

Effortless Static Content

- Images, Javascript, CSS
- Using Apache to host static content = FAIL
- Let your app servers do what they are born to do - serve dynamic content only
- Static content is bandwidth intensive, so fast downloads and aggressive caching are highly desirable

Enter CloudFront



- Content delivery network - competes with Akamai, Limelight
- No annoying salesman, no bandwidth estimating, no proposals
- US and EU bandwidth pricing peak at \$0.17 per GB
- In our application, per-request costs are 50% of bandwidth costs

How Expensive Is It?

- In general, more expensive than hosting it yourself
 - At Squidoo, approximately 30% more expensive than small load balanced EC2 instances running lighttpd
- However, “set and forget” is valuable too
- Research from Google suggests faster load times = more revenue

Aggressive Caching

- In a perfect world, static content stays cached on the client forever.
 - Faster load times
 - Lower bandwidth costs
- To accomplish this, filenames must be unique. Otherwise, clients with local cache will not always have latest copy.
- Doubly important since CloudFront edge locations cache data too.

CloudFront in 5 Easy Steps

1. Create S3 bucket *my_s3_bucket* to host your static content
2. Create CloudFront distribution *myapp* and link it to the S3 bucket
3. Update ``, `<script>`, and `<style>` tags to use Amazon-provided CloudFront domain
4. Add versioning support for static files
5. Deploy versioned static files to S3

Versioning Support

To make sure code updates are never hidden in a caching abyss, we need unique filenames like so:

<http://myapp.cloudfront.net/styles/core.v12.css>

Here's one way to do it...

Versioning Support

Step 1: create a version config file
(conf/staticversions.php)

```
<?php
$_REQUEST['STATIC_CSS_VERSIONS'] = array(
    'core' => 12,
    'nav' => 4
);
?>
```

- Each array item corresponds to a CSS file in the /styles dir
- Developers increment the version number any time an update is made
- Possible to integrate with SVN, but much trickier

Versioning Support

Step 2: update your srcs and hrefs

```
<?php
// this line normally defined in my controller
$this->styles = array('core', 'nav');

// everything below goes in my HTML header template
require('conf/staticversions.php');

foreach ($this->styles as $stylesheet) {
    if (array_key_exists($stylesheet, $_REQUEST['STATIC_CSS_VERSIONS']))
    {
        $version = $_REQUEST['STATIC_CSS_VERSIONS'][$stylesheet];
        $stylesheet = "{$stylesheet}.v{$version}";
    }
    printf('<link href="http://%s/styles/%s.css" rel="stylesheet"
type="text/css" />', 'myapp.cloudfront.net', $stylesheet);
}

?>
```

Versioning Support

Step 3: deployment script hook (Capfile)

- FTP = dark ages
- One-step script to deploy code to production servers is an absolute must
- Capistrano (Rails project) works really well for us
 - Here's the Cap way...

```
after "deploy", "myapp:deploy_s3"

namespace :myapp do
  task :deploy_s3 do
    system "svn -q export file:///usr/local/svn/myapp/trunk/
site /tmp/cloudfront"
    system "php deploy_to_s3.php /tmp/cloudfront"
    system "rm -rf /tmp/cloudfront"
  end
end
```

Versioning Support

Step 3.5: deploy versioned files to S3 (deploy_to_s3.php)

```
<?php
```

```
$release_dir = $argv[1];  
if (!file_exists($release_dir)) {  
    die('Please include the base path to this release');  
}  
  
// path to the version log we created in step 1  
require("$release_dir/conf/staticversions.php");  
  
foreach ($_REQUEST['STATIC_CSS_VERSIONS'] as $sheet => $version) {  
    copy("$release_dir/styles/$sheet.css", "$release_dir/styles/$sheet.v  
$version.css");  
}  
  
// push files to an S3 bucket named my_s3_bucket  
// notice the super long cache expiration  
exec("s3sync.rb --public-read --cache-control='max-age=315360000' --  
recursive $release_dir/www/styles my_s3_bucket:");
```

```
?>
```

Scaling Analytics

- Squidoo users have access to in-house traffic analytics
- A "unique" is defined as an IP address that has not been to this specific page in the past 7 days
- Each pageview requires a lookup to determine uniqueness
- Impossible to scale many simultaneous reads and writes on such a small amount of data

MySQL Couldn't Keep Up

- Relational DBs aren't optimal for high-concurrency analytics processing
 - INSERT DELAYED works great up to a point
 - Purging old granular data leads to even more locking issues
- Summary tables help, but what if you need granular data too?

Enter SimpleDB

- Non-blocking, fast querying
- Schema-less w/ auto-indexing
- Suitable for high-concurrency environments

SimpleDB Gotchas

- Difficult to estimate costs, but there is a free tier
- Eventual consistency
- No NULL values and data must be url encoded
- A few of the many limits:
 - 10 GB per domain
 - Attributes (aka columns) 1024 bytes
 - 5s max execution time - NextToken to continue

SDB Definitions

(very loosely translated)

Domain = database

Item = row

Attribute = column

Value = single cell of data*

* attributes can hold multiple values

ex: attribute "color" can contain both blue *and* red

Using SDB

Constructing Queries

SDB uses a custom query language. Each **predicate** is evaluated independently against a single attribute, and you may request the **intersection** or **union** of the results. No types - everything is a string.

```
$uniqueness_query =  
    '['page_id' = '$page_id'] intersection "  
    '['date_saved' > '$one_week_ago'] intersection "  
    '['ip_address' = '$ip'] intersection "  
    '['unique' = 'Y']";
```

Using SDB

Item Naming

Each item (aka row) must be assigned a **unique name**. In our case, we choose a combination of `ip_address`, `page_id`, and `date`.

```
$item_name = $ip . "-" . $page_id . "-" . time();
```

Recording a Pageview

```
// construct the query to determine uniqueness
$uniqueness_query =
    '['page_id' = '$page_id'] intersection " .
    '['date_saved' > '$one_week_ago'] intersection " .
    '['ip_address' = '$ip'] intersection " .
    '['unique' = 'Y']";

$uniquess_result = $sdb->query($domain, array('MaxNumberOfItems' => 1), $query);

$is_unique = 'N';

// if no visit found in past 7 days, assume unique
if (!isset($uniqueness_result->body->QueryResult->ItemName)) {
    $is_unique = 'Y';
}

// name the item we're about to create
$item_name = $ip . "-" . $page_id . "-" . $timestamp;

// save it
$put = $sdb->put_attributes($domain, $item_name, array(
    'page_id' => $page_id,
    'referrer_url' => urlencode($referrer),
    'date_saved' => $timestamp,
    'ip_address' => $ip,
    'unique' => $is_unique
));
```

Make It Better

- Use memcache for faster uniqueness querying. Make the cache key a combination of `page_id` and `ip_address`, with a 7 day expiration.
- For large datasets or extremely high concurrency, create multiple domains. Use a hash algorithm to store items in a consistent domain.
- When using multiple domains, `curl_multi` is your friend.

Daily Traffic Summary Table

```
$query = "[ 'date_saved' >= '$yesterday_start' ] intersection " .
        "[ 'date_saved' <= '$yesterday_end' ] intersection " .
        "[ 'unique' = 'Y' ]";

$pages = array();
$next_token = null;

while (1 == 1) {
    $params = array(
        'AttributeName' => 'page_id',
        'NextToken' => $next_token
    );
    $result = $sdb->query_with_attributes($domain, $params, $query);

    foreach($result->body->QueryWithAttributesResult->Item as $visit) {
        $page_id = (string) $visit->Attribute->Value;
        if (!array_key_exists($page_id, $pages)) {
            $pages[$page_id] = 0;
        }
        $pages[$page_id] += 1;
    }

    $next_token = $result->body->QueryWithAttributesResult->NextToken;

    if (!$next_token) break; // stop looping if there are no more results
}

foreach ($pages as $page_id => $visits) {
    // insert into MySQL
}
```

Massive Improvements Coming to EC2

- **Load Balancing** to automatically balance incoming requests and distribute traffic across multiple Amazon EC2 instances
- **Auto-scaling** to grow and shrink usage of EC2 based on capacity thresholds
- **Monitoring** of operational metrics

Other AWS Goodies

- **Elastic MapReduce** offers a managed interface to Hadoop
- **Mechanical Turk** helps you automate tasks that computers aren't good at (content moderation, quality control, market research)
- More tools on the way

General AWS Tips

- Expect failure. Retry failed queries using an exponential backoff. Worst case scenario, log failures for later processing.
- API calls will fail if system time is correct. Keep ntpd (timeserver daemon) running always.

The End

Gil Hildebrand

gil@squidoo.com