Scalable PHP Architecture

Jack Steadman
Manager of Architectural Development
Smarter Travel Media LLC

Agenda

- About STM
- Publishing Platform Overview
- Challenges and Solutions

About Smarter Travel Media

Company details

- Founded in 1998 as Smarter Living, Inc.
- Publishing site URL changed in 2005 to reflect singular focus on travel
- Located in Charlestown, MA
- 85 employees, 18 developers
  - 1 year ago: 60 employees, 11 developers
- Development teams focused on applications, architecture, SEO, and QA
  - Applications developers specialized by product
  - Other functional areas are cross-product
- Since February 2007, part of TripAdvisor Media Network and Expedia, Inc. (company name officially changed as part of acquisition)

Our products

Two completely different sites:

smartertravel.com

- Traditional content site
- Articles, blogs, airfares, destinations, and associated metadata
- Several e-mail newsletter products
- Administered primarily by a team of editors

bookingbuddy.com

- Travel search tool, plus paid deal listings
- Search forms, paid search placements, deep-linking adapters
- Two e-mail newsletter products promoting deal listings
- Administered primarily by a team of ad operations coordinators

Some technical details

- Terrible measures of infrastructure complexity:
  - 180,000 lines of PHP code
  - 1,100 classes
  - 8,000 functions and methods
  - Future ground-up rewrite not an option
- 3M visitors per month
- Traffic comes in large spikes during mailings
- Steady growth over past few years

Publishing Platform

History

- First iteration: perl/MySQL-based CMS plus periodic publishing process to write flat HTML files (no PHP at all)

History

- First iteration: perl/MySQL-based CMS plus periodic publishing process to write flat HTML files (no PHP at all)
- Second iteration: legacy perl CMS, sites a mix of PHP and flat files
  - PHP classes written to access existing MySQL DB
  - Flat files still used to present data not modeled in PHP/MySQL
  - Static content (image, CSS, javascript files) served by separate, lightweight servers
  - Adding new data types was difficult: parallel perl/PHP development, need to duplicate a lot of code

History

- First iteration: perl/MySQL-based CMS plus periodic publishing process to write flat HTML files (no PHP at all)
- Second iteration: legacy perl CMS, sites a mix of PHP and flat files
  - PHP classes written to access existing MySQL DB
  - Flat files still used to present data not modeled in PHP/MySQL
  - Static content (image, CSS, javascript files) served by separate, lightweight servers
  - Adding new data types was difficult: parallel perl/PHP development, need to duplicate a lot of code
- Third (current) iteration: ground-up rewrite focusing on data and relationship modeling, plus PHP CMS based on database reflection
  - Can add data types by adding tables (no code changes necessary)
  - Can add relationships without code or database changes
  - More complex data types require only a few PHP wrappers for custom functionality

The database tells you a lot

- Field types and sizes provide basic validation
  - e.g. tinytext limited to 255 characters; int unsigned limited to 0-2^32; date field turns into /^\d{4}-\d{1,2}-\d{1,2}$/
- Field names can be treated as custom types
  - e.g. field called 'url' in any table is validated as a url
- Names and types also drive rendering of fields for CMS UI
- Everything you need to add a simple new data type can be gleaned from the database
- Reflection is expensive, but you don't need to know all this when serving your site – only for CMS

Other design considerations

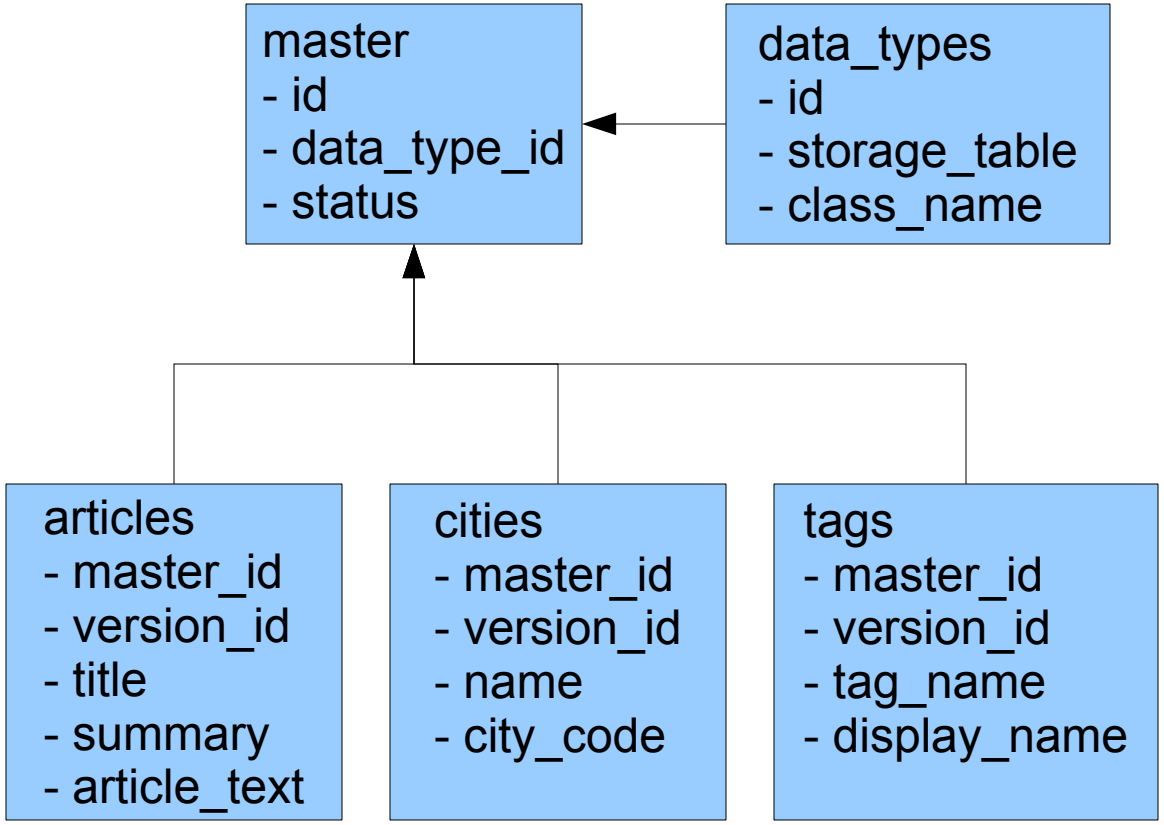
- Versioning: must maintain version history for web content
- Preview site: all changes must be made first to a preview site, then released to production

Overall goal: model all existing data, and support addition of new, fully-integrated types with a trivial amount of effort.

Why build our own platform?

- Existing frameworks are good for getting up-and-running quickly
- Over the long term, rolling your own pays dividends
  - Features targeted directly to needs
  - Sacrifice generality for performance
  - Institutional knowledge of internals
  - Can easily incorporate the best of any other framework, while not being locked into one in particular
    - STM uses components from PEAR, Zend, Horde, and Xaraya
- Building doesn't always make sense: we've integrated MediaWiki and phpBB installations to take advantage of mature applications
- Other downsides: no community support, development resources limited to what's in-house

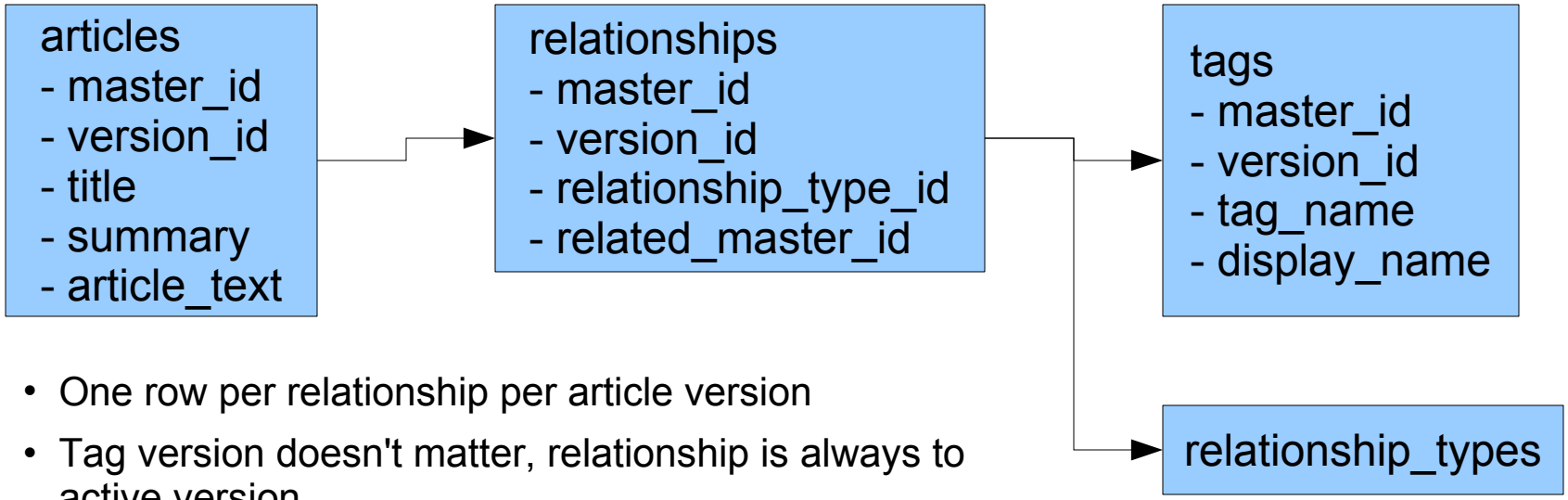
Platform basics: data storage



- All data gets a global ID
- Data type controls storage and handling
- Status can be live, archived, deleted

- All data tables have master and version ID fields
- One row for each version
- Must know data type to access data

Platform basics: relationships



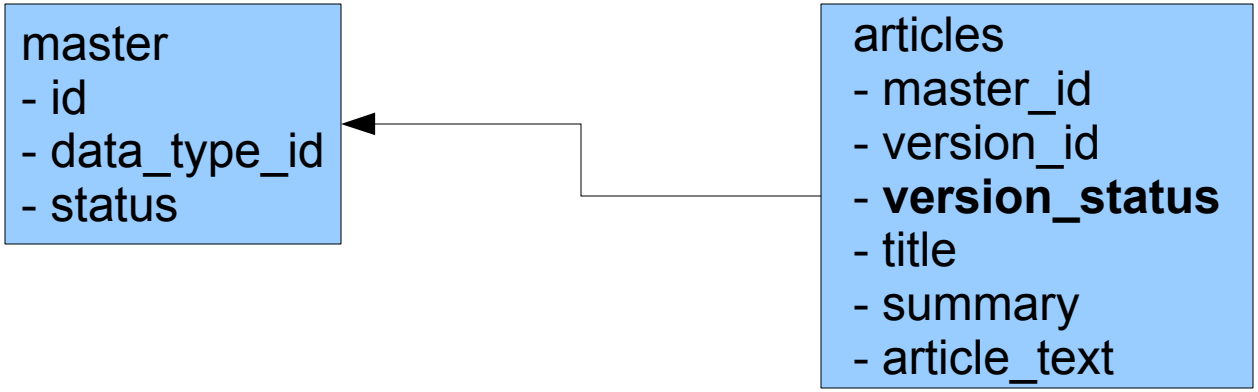
- One row per relationship per article version
- Tag version doesn't matter, relationship is always to active version
- Relationship types provide data-type-like control over the nature of the relationship

Example feature: previewing changes

How to determine which version is public and which to preview?

Example feature: previewing changes

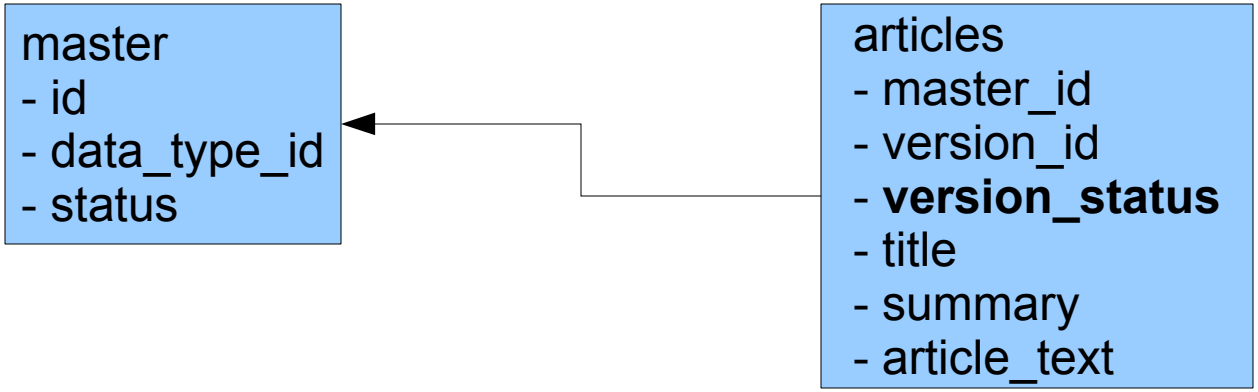
How to determine which version is public and which to preview?



Version status identifies live, preview, and archived versions.

Example feature: previewing changes

How to determine which version is public and which to preview?



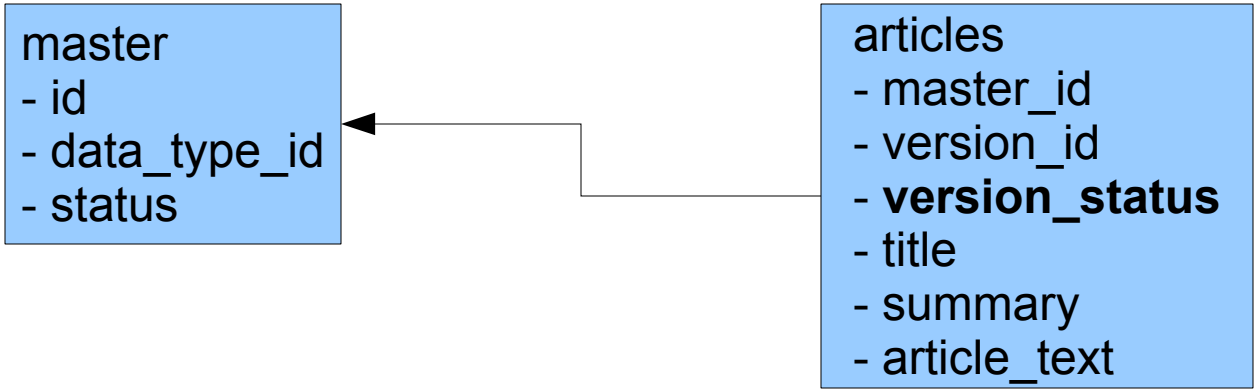
Version status identifies live, preview, and archived versions.

SQL:

```
SELECT m.id, m.status, a.* FROM articles a
INNER JOIN master m ON a.master_id=m.id
WHERE a.version_status='preview'
AND m.status='live' AND m.id=12345;
```

Example feature: previewing changes

How to determine which version is public and which to preview?



Version status identifies live, preview, and archived versions.

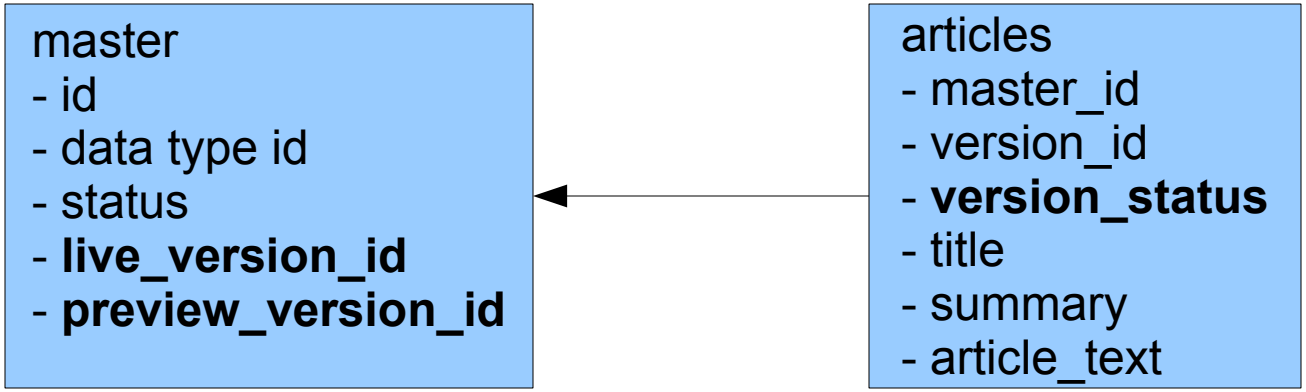
SQL:

```
SELECT m.id, m.status, a.* FROM articles a
INNER JOIN master m ON a.master_id=m.id
WHERE a.version_status='preview'
AND m.status='live' AND m.id=12345;
```

Problem: more versions, slower join.

Example feature: previewing changes

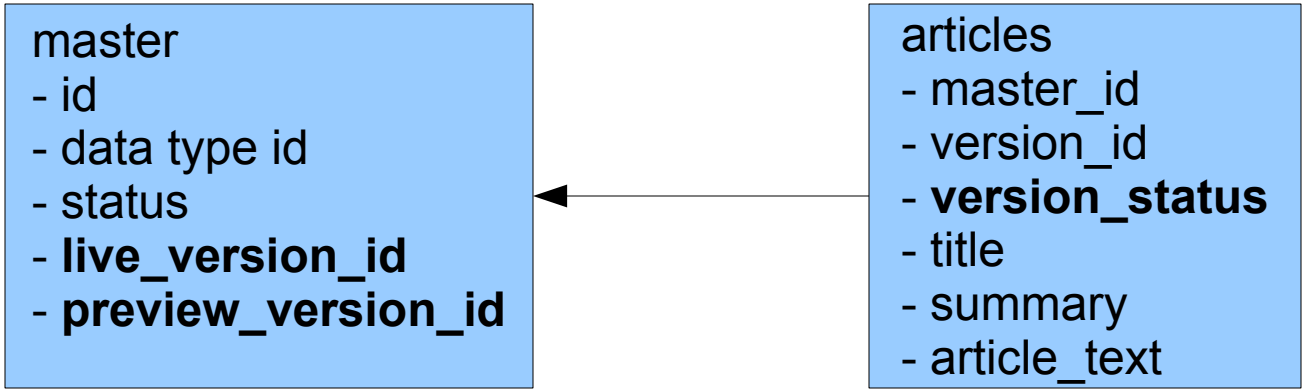
How to determine which version is public and which to preview?



Better idea: master stores a copy of the live and preview version IDs.

Example feature: previewing changes

How to determine which version is public and which to preview?



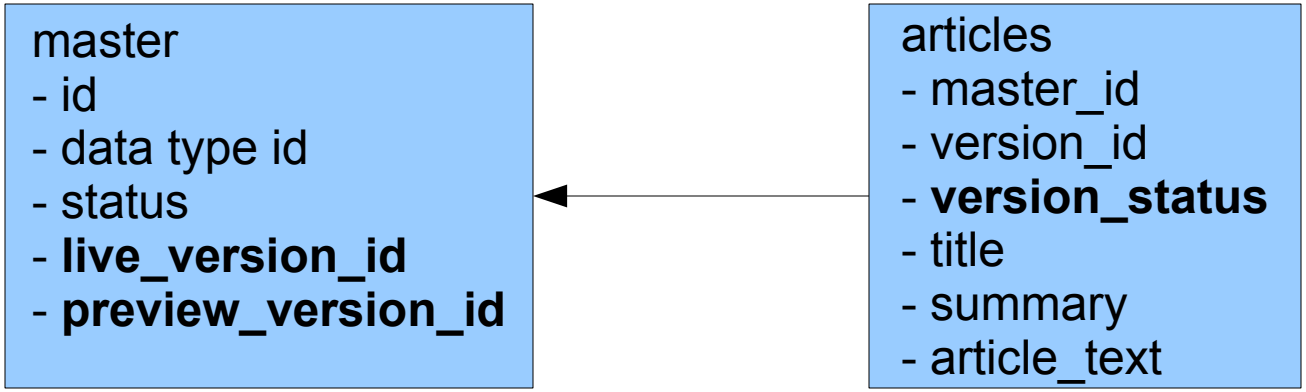
Better idea: master stores a copy of the live and preview version IDs.

SQL:

```
SELECT m.id, m.status, a.* FROM articles a
INNER JOIN master m ON a.master_id=m.id
AND a.version_id=m.preview_version_id
WHERE m.status='live' AND m.id=12345;
```

Example feature: previewing changes

How to determine which version is public and which to preview?



Better idea: master stores a copy of the live and preview version IDs.

SQL:

```
SELECT m.id, m.status, a.* FROM articles a
INNER JOIN master m ON a.master_id=m.id
AND a.version_id=m.preview_version_id
WHERE m.status='live' AND m.id=12345;
```

With proper indexing, this is very fast even with many versions.

Challenges and Solutions

PHP itself doesn't make things easy

- Circular references cause memory leaks (PHP response: not a bug)
  - Solution: regular apache restarts to free memory (or code around it; we chose the former)
- Backward-breaking changes between versions
  - Object/reference handling
  - Return values of built-in functions (e.g. strtotime)
  - Case-sensitivity of class and function names
  - Took STM years to upgrade to PHP5, and when we did...

Undeclared static method performance

- PHP5 adds 'static' keyword for declaring that a class method should be called statically (coming from PHP4, no STM classes used this keyword)
- After upgrade to PHP5, STM sites slowed down considerably
- Frantic code profiling ensued, finally leading to this discovery:

Undeclared static method performance

- PHP5 adds 'static' keyword for declaring that a class method should be called statically (coming from PHP4, no STM classes used this keyword)
- After upgrade to PHP5, STM sites slowed down considerably
- Frantic code profiling ensued, finally leading to this discovery:

```
<?php
class A {
    function test() {}
}

for ($i = 0; $i < 100000; $i++) {
    A::test();
}
```

```
<?php
class B {
    static function test() {}
}

for ($i = 0; $i < 100000; $i++) {
    B::test();
}
```

Undeclared static method performance

- PHP5 adds 'static' keyword for declaring that a class method should be called statically
- After upgrade to PHP5, STM sites slowed down considerably
- Frantic code profiling ensued, finally leading to this discovery:

```
<?php
class A {
    function test() {}
}

for ($i = 0; $i < 100000; $i++) {
    A::test();
}
```

Runs in .11 seconds

```
<?php
class B {
    static function test() {}
}

for ($i = 0; $i < 100000; $i++) {
    B::test();
}
```

Runs in .04 seconds

Undeclared static calls run 275% slower!

Worst of all, this issue is **completely undocumented**

Static method fallout

- Scramble to update entire code base
- Still had thousands of calls to PEAR::isError
- No desire to edit PEAR.php

Static method fallout

- Scramble to update entire code base
- Still had thousands of calls to PEAR::isError
- No desire to edit PEAR.php
- Solution: write our own standalone function to do the same thing:

```
function isPEARError($e) {
        return $e instanceof PEAR_Error;
}
```

- Mass-replace PEAR::isError calls
- Site performance back to normal

Zend Platform helps, but it has issues

- Newer features unreliable under load (Job queue server crashes, version incompatibilities, HA session clustering causes 10% performance hit)
- Output cache is file-based; no benefit across multiple servers
- Inexplicable churning of CPU when Zend Central server unavailable
- Systems admins say it's a nightmare to upgrade
  - Zend support standard bug report response: upgrade to latest version
- Code acceleration still very beneficial despite problems
- Free solutions can provide same or greater benefits

Data structures require complex SQL

Data structures require complex SQL

```
(SELECT DISTINCT(C.id),C.content_type_id,S.version_id,"25" AS selector_group,C.create_time AS
selector_sort FROM content C INNER JOIN stories S ON C.live_version_id = S.version_id AND C.id =
S.content_id INNER JOIN content_relationships CSC_LH2_CR0 ON CSC_LH2_CR0.content_id = S.content_id AND
CSC_LH2_CR0.content_version_id = S.version_id AND CSC_LH2_CR0.relationship_id = 5 INNER JOIN
index_nested_set_location_hierarchy_production CSC_LH2_NSI0 ON CSC_LH2_NSI0.content_id =
CSC_LH2_CR0.related_content_id INNER JOIN content CSC_LH2_C0  ON CSC_LH2_C0.id = CSC_LH2_NSI0.content_id
WHERE (C.status  = "live" AND C.content_type_id IN (209,46,210,46,232,46,46) AND C.site_id IN (3, 0)) AND
(CSC_LH2_NSI0.left_val >= 36891 AND CSC_LH2_NSI0.right_val <= 36892) AND (CSC_LH2_C0.status IN ('live'))
AND (CSC_LH2_C0.content_type_id IN (8,5))) UNION ALL (SELECT
DISTINCT(C.id),C.content_type_id,S.version_id,"25" AS selector_group,C.create_time AS selector_sort FROM
content C INNER JOIN blog_entries S ON C.live_version_id = S.version_id AND C.id = S.content_id INNER
JOIN content_relationships CSC_LH2_CR0 ON CSC_LH2_CR0.content_id = S.content_id AND
CSC_LH2_CR0.content_version_id = S.version_id AND CSC_LH2_CR0.relationship_id = 5 INNER JOIN
index_nested_set_location_hierarchy_production CSC_LH2_NSI0 ON CSC_LH2_NSI0.content_id =
CSC_LH2_CR0.related_content_id INNER JOIN content CSC_LH2_C0  ON CSC_LH2_C0.id = CSC_LH2_NSI0.content_id
WHERE (C.status  = "live" AND C.content_type_id = 159 AND C.site_id IN (3, 0)) AND (CSC_LH2_NSI0.left_val
>= 36891 AND CSC_LH2_NSI0.right_val <= 36892) AND (CSC_LH2_C0.status IN ('live')) AND
(CSC_LH2_C0.content_type_id IN (8,5))) UNION ALL (SELECT
DISTINCT(C.id),C.content_type_id,S.version_id,"25" AS selector_group,C.create_time AS selector_sort FROM
content C INNER JOIN slideshows S ON C.live_version_id = S.version_id AND C.id = S.content_id INNER JOIN
content_relationships CSC_LH2_CR0 ON CSC_LH2_CR0.content_id = S.content_id AND
CSC_LH2_CR0.content_version_id = S.version_id AND CSC_LH2_CR0.relationship_id = 5 INNER JOIN
index_nested_set_location_hierarchy_production CSC_LH2_NSI0 ON CSC_LH2_NSI0.content_id =
CSC_LH2_CR0.related_content_id INNER JOIN content CSC_LH2_C0  ON CSC_LH2_C0.id = CSC_LH2_NSI0.content_id
WHERE (C.status  = "live" AND C.content_type_id = 157 AND C.site_id IN (3, 0)) AND (CSC_LH2_NSI0.left_val
>= 36891 AND CSC_LH2_NSI0.right_val <= 36892) AND (CSC_LH2_C0.status IN ('live')) AND
(CSC_LH2_C0.content_type_id IN (8,5))) UNION ALL (SELECT
DISTINCT(C.id),C.content_type_id,S.version_id,"25" AS selector_group,C.create_time AS selector_sort FROM
content C INNER JOIN podcasts S ON C.live_version_id = S.version_id AND C.id = S.content_id INNER JOIN
content_relationships CSC_LH2_CR0 ON CSC_LH2_CR0.content_id = S.content_id AND
CSC_LH2_CR0.content_version_id = S.version_id AND CSC_LH2_CR0.relationship_id = 5 INNER JOIN
index_nested_set_location_hierarchy_production CSC_LH2_NSI0 ON CSC_LH2_NSI0.content_id =
CSC_LH2_CR0.related_content_id INNER JOIN content CSC_LH2_C0  ON CSC_LH2_C0.id = CSC_LH2_NSI0.content_id
WHERE (C.status  = "live" AND C.content_type_id = 224 AND C.site_id IN (3, 0)) AND (CSC_LH2_NSI0.left_val
>= 36891 AND CSC_LH2_NSI0.right_val <= 36892) AND (CSC_LH2_C0.status IN ('live')) AND
(CSC_LH2_C0.content_type_id IN (8,5))) ORDER BY selector_sort DESC LIMIT 6 OFFSET 0
```

Data structures require complex SQL

- Under good conditions, example query runs in .17 seconds
- Fast for its complexity, but not fast enough
- MySQL query cache is helpful, but...
  - Every change touches master and relationships tables
  - Query cache completely wiped out
- More significant improvement: framework-level query result caching
  - Serialize array of rows returned from a query
  - Store in cache using hash of query as cache key
  - Can be implemented at the DB access layer
  - On cache hit, no DB interaction required

Beyond SQL issues

If a page takes hundreds of queries to generate, optimizing database access only gets you so far.

- smartertravel.com home page: 443 queries
- bookingbuddy.com home page: 1129 queries (!)

Beyond SQL issues

If a page takes hundreds of queries to generate, optimizing database access only gets you so far.

- smartertravel.com home page: 443 queries
- bookingbuddy.com home page: 1129 queries (!)

Solution: partial page caching

- Business requirements prevent full-page caching
- Site designed around small, reusable modules
- Each module has its own cache settings (TTL, key generation, etc.)
- Staggered TTLs mean modules on a page don't all expire at once
- Full output string of each module is cached
- No serialize/unserialize required

Caching as bottleneck

- Caching a large number of small pieces of data
  - Disk I/O using file-based cache
  - Network access using memcached
  - Can move file-based cache to memory to mitigate, but only if you have lots of extra memory
- Caching large data structures
  - CPU overhead of serializing and unserializing
- Know how much space you need for your cache
  - Churn means lower hit rate

Disk woes

- With a large and complex code base, require_once is a necessity – but slow
  - Can be improved with a simple wrapper:

```
function load_once($file) {
        static $loaded = array();
        if (isset($loaded[$file])) return;

        require_once $file;
        $loaded[$file] = true;
}
```

Other best practices:

- Keep include paths short
- Put more-commonly-accessed paths first

Disk woes

- Most important: avoid loading and compiling code you don't need
- __autoload gets a bad rap, but it simplifies things immensely
- Even if it's slower than explicit includes, you may save enough overhead to offset by including only what's necessary
- Combine with load_once function:

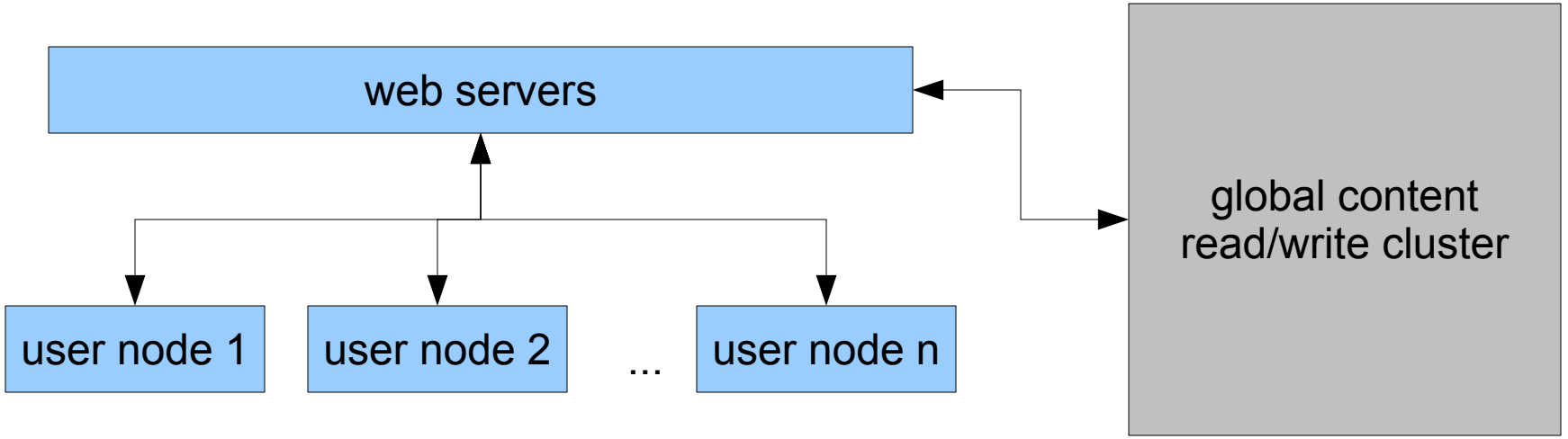
```
function __autoload($class_name) {
        $class_file = get_class_file($class_name);
        load_once($class_file);
}
```

- Must be able to create a file name from a class name

Database writes

- Content is easy: read-write ratio very high
- Reads scale well enough with added hardware and MySQL replication
- Interactive applications require many more writes
- Look to a PHP application with much larger traffic than ours: LiveJournal

User data gets dedicated storage



- User ID and node ID written to global content write master at signup
- Content read servers used to look up user's node ID
- All other read/write of user data happens on nodes
- More users + more traffic + more writes = more nodes

User data storage: overkill?

- Build it once
- The architecture isn't all that complex
- Three developers, 3 months to build from ground up, including conversion from existing database and UI for managing user data
- Keep the details simple:
  - Randomized node assignment, no attempt to intelligently load-balance
  - Use existing content database servers for lookups and to maintain key uniqueness
  - Can't query across entire user base, but that's ok
    - Push updates to specialized databases for things like reporting, mailings, etc.
    - Distributed storage is for web interaction only

Upcoming challenges

We're hiring!

If any of this sounds interesting to you, send us your resume.

http://www.smartertravel.com/us/careers.html
careers@smartertravelmedia.com

STM is a fantastic place to work, and not just for the free trip once a year.

Questions?