Table of contents

Introduction

Started as Perl CGI script running on single server in 2001, site has grown into distributed platform, containing multiple technologies, all of them open. The principle of openness forced all operation to use free & open-source software only. Having commercial alternatives out of question, Wikipedia had the challenging task to build efficient platform of freely available components.

Wikipedia's primary aim is to provide a platform for building collaborative compendium of knowledge. Due to different kind of funding (it is mostly donation driven), performance and efficiency has been prioritized above high availability or security of operation.

At the moment there're six people (some of them recently hired) actively working on internal platform, though there're few active developers who do contribute to the open-source code-base of application.

The Wikipedia technology is in constant evolution, information in this document may be outdated and not reflecting reality anymore.

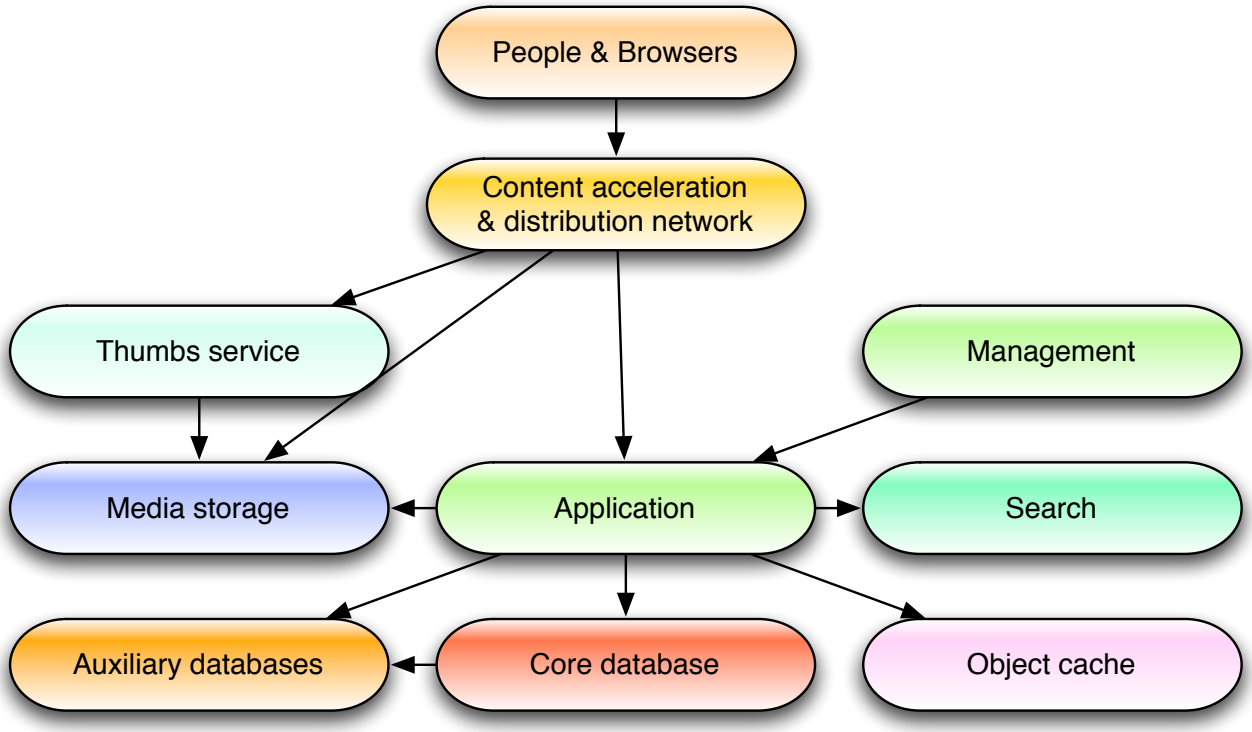
The big picture

Generally, it is extended LAMP environment - core components, front to back, are:

- Linux - operating system (Fedora, Ubuntu)
- PowerDNS - geo-based request distribution
- LVS - used for distributing requests to cache and application servers
- Squid - content acceleration and distribution
- lighttpd - static file serving
- Apache - application HTTP server
- PHP5 - Core language
- MediaWiki - main application
- Lucene, Mono - search
- Memcached - various object caching

Many of the components have to be extended to have efficient communication with each other, what tends to be major engineering work in LAMP environments.

This document describes most important parts of gluing everything together - as well as required adjustments to remove performance hotspots and improve scalability.



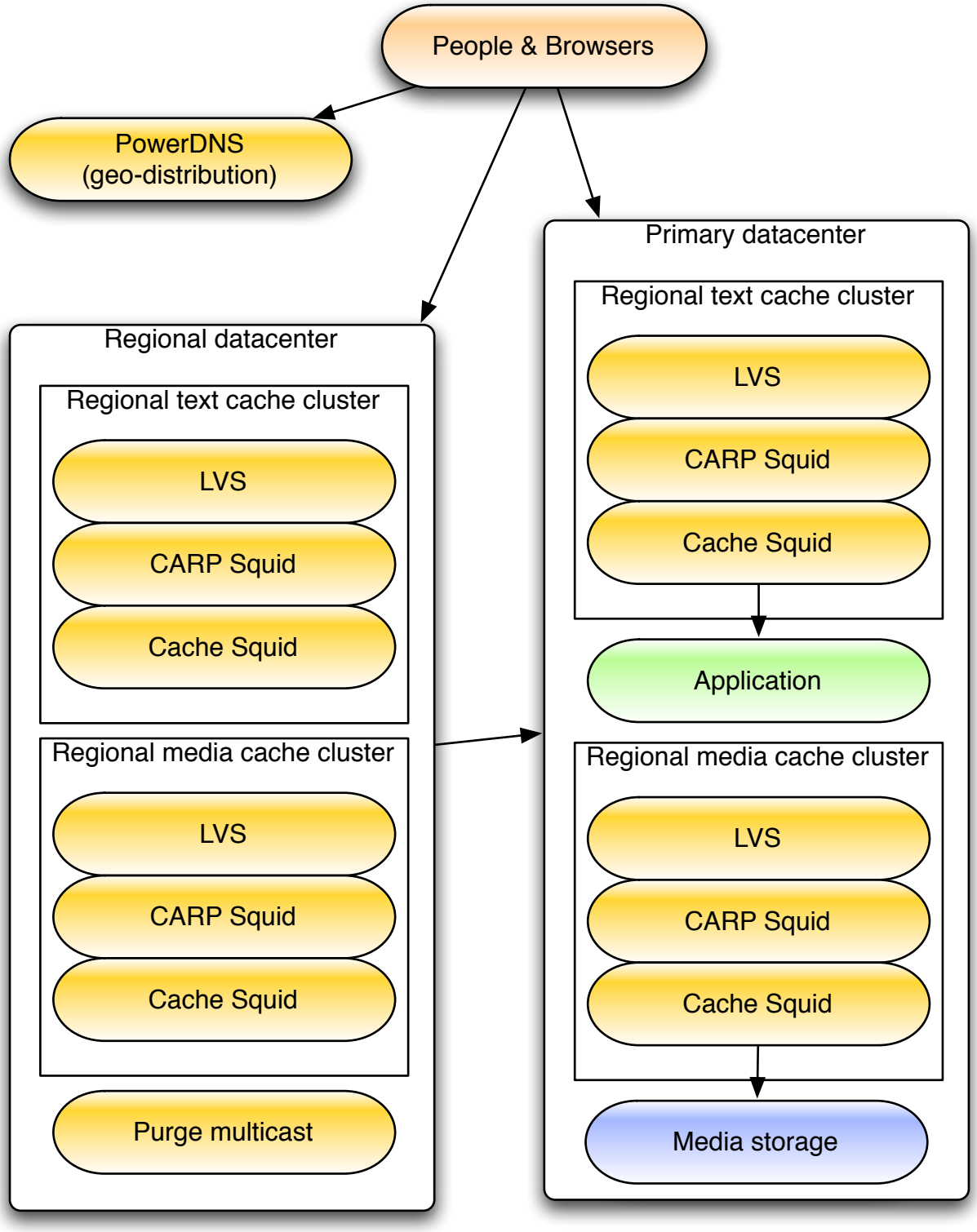
As application tends to be most resource hungry part of the system, every component is built to be semi-independent from it, so that less interference would happen between multiple tiers when a request is served.

The most distinct separation is media serving, which can happen without accessing any PHP/Apache code segments.

Other services, like search, still have to be served by application (to apply skin, settings and content transformations).

The major component, often overlooked in designs, is how every user (and his agent) treats content, connections, protocols and time.

Content delivery network



Content delivery network is the ‘holy grail’ of performance for Wikipedia. Most of pages (except for logged in users) end up generated in such a manner, where both caching and invalidating the content is fairly trivial.

Cacheable content
Key rules for such setup are:

- There're no unaccounted dynamic bits on a content page (if there are, the changes are not invalidated in cache layer, hence causing stale data)
- Every content page has strict naming, with single URI to the file (good for having uniform linking and not wasting memory on dupe cache entries)
- Caching is application-controlled (via headers) (simplifies configuration, more efficient selection of what can and cannot be cached)
- Content purging is completely application-driven (the amount of unpredictable changes in unpredictable areas would render lots of stale data otherwise)
- Application must support lightweight revalidations (If-Modified-Since requests)

Example of random page response header en-route from application to CDN:

```
HTTP/1.1 200 OK
...
Vary: Accept-Encoding,Cookie
Cache-Control: s-maxage=2678400, must-revalidate, max-age=0
Last-modified: Tue, 03 Apr 2007 17:50:55 GMT
```

Squid would cache such request for a month, unless purged or user has a cookie.
The request for the very same page to CDN would give slightly different cache-control headers:

```
HTTP/1.0 200 OK
Date: Tue, 03 Apr 2007 19:36:07 GMT
...
Vary: Accept-Encoding,Cookie
Cache-Control: private, s-maxage=0, max-age=0, must-revalidate
Last-Modified: Tue, 03 Apr 2007 17:50:55 GMT
...
Age: 53020
```

All non-managed caches must revalidate every request (that usually consists of sending IMS request to upstream), though ones in CDN have explicit purges coming in.

Due to current limitations in Squid only complete headers can be replaced, not their portions, so all external requests get 'private' Cache-Control bit, which forbids the caching. Ideally, this should be fixed, but requires refactoring of that particular Squid code (or upgrade to Squid3...)

We had to extend Squid to handle HTCP purges properly, and also multicast is employed - every invalidation is sent out from application once, but then is delivered to every managed cache server.

Until recently every CDN datacenter used to have just single tier of cache servers, each trying to coordinate their caches with their neighbors. Squid's ICP and HTCP protocols are not designed for a big cluster of servers, so not much of efficiency is achieved, though number of problems is still increased.

Cache efficiency

Neighbor coordination doesn't yield perfect results, so multiple-tier approach was chosen for every data-center, and cache clusters were split into task-oriented groups, as content serving has different access/communication patterns than media serving.

The major recent change was adding URI-based distribution layer in front, utilizing CARP (Cache-Array-Routing-Protocol). It's major advantage is hash-based distribution of requests and very efficient handling of node failures - the requests for content handled by failing node are redistributed across pool of active machines according to their weights.

Such request distribution reduces the number of object copies, and requests for especially active objects can still handled by front-most layer.

Our caches use COSS (cyclical object storage) - the really efficient on-disk storage of cached blobs.

CDN Configuration files

We do generate per-server configuration files from a master configuration.

The closest to application servers pool (tier1 cache nodes):

```
# no negative caching
negative_ttl 0 minutes
half_closed_clients off
pipeline_prefetch on
read_timeout 1 minute
request_timeout 1 minute
cache_mem 3000 MB
maximum_object_size 600 KB
maximum_object_size_in_memory 75 KB
# Purge groups
mcast_groups 239.128.0.112

cache_dir coss /dev/sda6 8000 max-size=524288 max-stripe-waste=32768 block-size=1024
cache_dir coss /dev/sdb 8000 max-size=524288 max-stripe-waste=32768 block-size=1024
..
client_db off
digest_generation off
# No logging for CARP-balanced caches, that will be done by the frontends
cache_access_log none
# First some handy definitions
acl all src 0.0.0.0/0.0.0.0
acl localsrc src 127.0.0.1/255.255.255.255
acl purge method PURGE
# defining managed hosts -
acl tiertwo src 66.230.200.0/24                    # pmtpa
acl tiertwo src 145.97.39.128/26                   # knams
acl tiertwo src 211.115.107.128/26                 # yaseo
acl tiertwo src 10.0.0.0/16                        # pmtpa internal
# Upstream - Application
cache_peer        10.0.5.3    parent 80 3130 originserver no-query connect-timeout=5 login=PASS
# Only connect to the configured peers
never_direct allow all
# Allow coordination requests
htcp_access allow all

# Only internal nodes allowed to connect to caching CDN nodes
http_access allow tiertwo
http_access deny all
```

Such kind of CDN node does not do any direct interactions with clients, that is the job for front-end node:

```
# no negative caching
negative_ttl 0 minutes
cache_mem 10 MB
maximum_object_size 600 KB
maximum_object_size_in_memory 75 KB

# No cache directories for CARP squids
cache_dir null /tmp
client_db off
digest_generation off

# Send out logs to log collectors
logformat wikimedia sq20.wikimedia.org %sn %ts.%03tu %tr %>a %Ss/%03Hs %<st %rm %ru %Sh/%<A %mt
%{Referer}>h %{X-Forwarded-For}>h %{User-Agent}>h
cache_access_log udp://1.2.3.4:5555 wikimedia

# Only connect to the configured peers
never_direct allow all

cache_peer        66.230.200.129  parent        3128 0 no-query connect-timeout=5 login=PASS carp
weight=10 monitorurl=http://en.wikipedia.org/wiki/Main_Page monitortimeout=30
cache_peer        66.230.200.130  parent        3128 0 no-query connect-timeout=5 login=PASS carp
weight=10 monitorurl=http://en.wikipedia.org/wiki/Main_Page monitortimeout=30
cache_peer        66.230.200.131  parent        3128 0 no-query connect-timeout=5 login=PASS carp
weight=10 monitorurl=http://en.wikipedia.org/wiki/Main_Page monitortimeout=30
...

# lots of various ACLs :-)
...

# css and js files
acl wpstatic urlpath_regex :.*/.*\.(css|js)
acl wpstatic urlpath_regex action=raw
# wiki content pages
acl wpcontent url_regex -i http://.*/wiki/.*
acl wpcontent url_regex -i http://.*/w/.*

#important: replace cache headers, s-maxage has to be 0!
#this won't be needed with Squid3, there s-maxage is 0 anyway and lifetime
#on our Squid is controlled via Surrogate-Control
header_access Cache-Control allow wpstatic
header_access Cache-Control allow tiertwo
header_access Cache-Control deny wpcontent
header_replace Cache-Control private, s-maxage=0, max-age=0, must-revalidate
```

Additionally, some kernel configuration parameters are set in sysctl.conf on these servers:

```
# increase TCP max buffer size
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
# increase Linux autotuning TCP buffer limits
# min, default, and max number of bytes to use
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216
# don't cache ssthresh from previous connection
net.ipv4.tcp_no_metrics_save = 1
# recommended to increase this for 1000 BT or higher
net.core.netdev_max_backlog = 2500
# Increase the queue size of new TCP connections
net.core.somaxconn = 1024
net.ipv4.tcp_max_syn_backlog = 4192
```

There are some differences for media caching cluster, for example for media cache server such differences exist:

```
# Override reloads from clients (for upload squids)
refresh_pattern .       60      50%     527040  ignore-reload

# Different destinations may be served by different image servers/clusters

acl commons_thumbs      url_regex       ^http://upload.wikimedia.org/wikipedia/commons/thumb/
acl de_thumbs           url_regex       ^http://upload.wikimedia.org/wikipedia/de/thumb/
acl en_thumbs           url_regex       ^http://upload.wikimedia.org/wikipedia/en/thumb/

# amane.pmtpa.wmnet
# Default destination
cache_peer        10.0.0.33    parent 80 3130 originserver no-query connect-timeout=5 login=PASS
cache_peer_access 10.0.0.33    deny commons_thumbs
cache_peer_access 10.0.0.33    deny de_thumbs
cache_peer_access 10.0.0.33    allow all

# bacon.wikimedia.org
cache_peer        66.230.200.200    parent 80 3130 originserver no-query connect-timeout=5 log-
in=PASS
cache_peer_access 66.230.200.200    deny de_thumbs
cache_peer_access 66.230.200.200    allow commons_thumbs
cache_peer_access 66.230.200.200    deny all

# srv6.wikimedia.org
cache_peer        66.230.200.198    parent 80 3130 originserver no-query connect-timeout=5 log-
in=PASS
cache_peer_access 66.230.200.198    deny commons_thumbs
cache_peer_access 66.230.200.198    allow de_thumbs
cache_peer_access 66.230.200.198    deny all
```

CDN Notes

It is very important not to expose too many hostnames to client browsers, as then persistent connections are not possible (and keep-alive fails).

Keep-alive becomes especially important in long-distance connections.

There have been keep-alive failures because of short-comings of HTTP/1.0 - sometimes simple issues have caused serious performance impact.

Many bugs have been hit on the way of building such environment, most of them fixed in Squid 2.6 - Wikipedia's Tim and Mark are in AUTHORS list, and Adrian and Henrik from Squid team were especially helpful too.

Additional stuff:

http://svn.wikimedia.org/viewvc/mediawiki/trunk/udpmcast/ - UDP-multicast bridge (for purges)

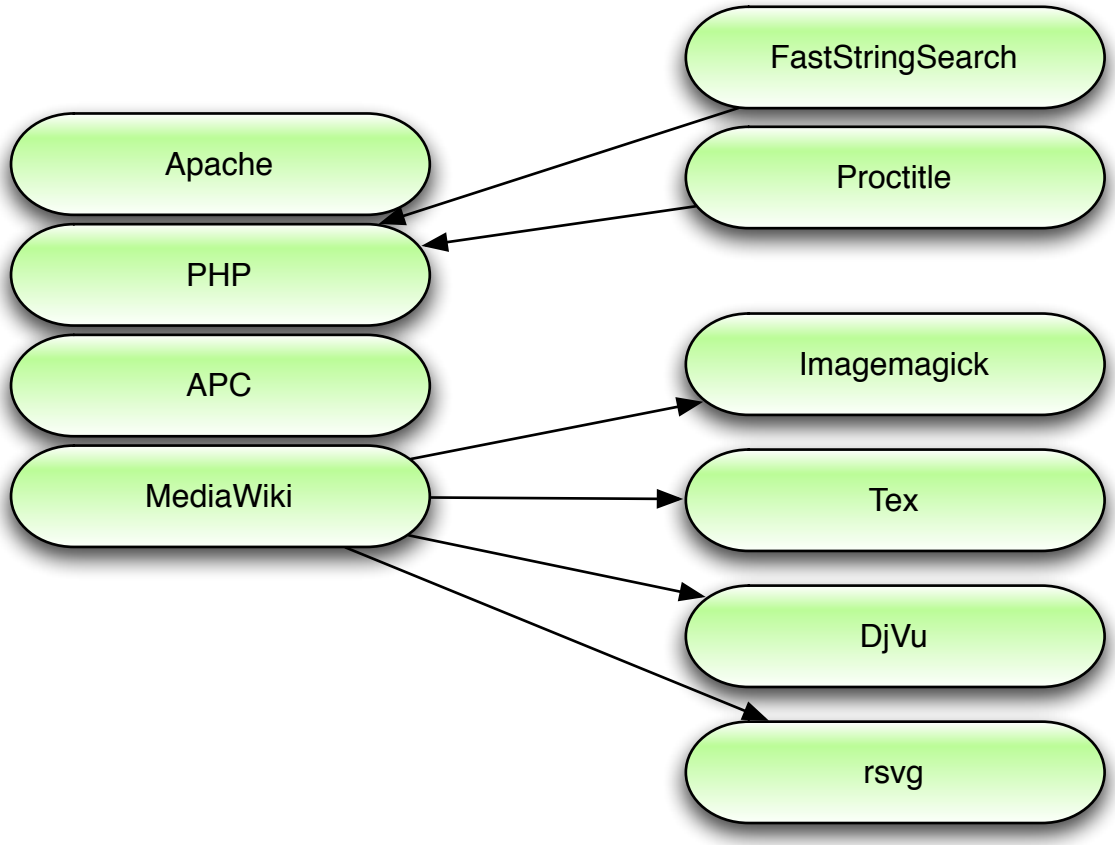
http://svn.wikimedia.org/viewvc/mediawiki/trunk/udplog/ - UDP logging code

http://svn.wikimedia.org/viewvc/mediawiki/trunk/HTCPpurger/ - HTCP purging for app

Application

Components

Though MediaWiki, Wikipedia's in house software, could be run on standard LAMP platform, quite a few external extensions are required:



APC, the bytecode cache for PHP, should be treat as very important member of a stack, as running PHP applications without it (or alternatives) is a resource waste.

For handling richer content external libraries had to be used, though we had to roll out our own packages for them, as many requirements for big website are different from regular users.

- Imagemagick does most of thumbnailing work. Though PHP often comes with GD included, it may lack some of features - quality control included (sharpening thumbnails tends to provide more attractive results). Imagemagick though has multiple modes of conversions, resulting in different math applied to internal calculations. Also, pthread-enabled distribution package did not fail gracefully in case of memory boundaries, causing deadlocks.
- Ocaml-written filter calls Tex libraries to render formulae and other scientific content. Output to PNG format was added to Tex libraries just recently, and some minor glitches in older versions of dvipng force to maintain our own packages.
- rsvg support was hacked out of 'test' utility coming with librsvg. Usually the library is used for desktop applications.

- Fast String Search is replacement module for PHP's strtr() function, which utilizes Commentz-Walter style algorithm for multiple search terms, or a Boyer-Moore algorithm for single search terms. License collisions (GPL code was used for it) do not allow this to be included in PHP. Using proper algorithm instead of foreach loops is incredible boost for some applications.

- Proctitle extension to PHP allows seeing which portion of code is executed by Apache child at the moment simply using 'ps':

```
20721 ?        SNs    0:00 httpd: php_init
20722 ?        SN     0:01 httpd: requestfinish
20723 ?        SN     0:01 httpd: LuceneSearchSet::newFromQuery-contact-10.0.5.10 [frwiki]
20724 ?        SN     0:01 httpd: requestfinish
20725 ?        RN     0:00 httpd: MessageCache::addMessages [hewiki]
```

- WikiDiff provides faster facility for comparing articles.

- APC used to be constant pain with new PHP5 features actively used (like accessing parent class' static arrays, autoloading, early and late code binding combined, etc). The support we had from APC people helped as well.

- We tend to hit some of issues regularly, whereas other sites see them only as exceptions. Cleanup of PHP tends to trigger recursive free(), causing deadlocks with pthread-enabled Apache/PHP stack. This of course causes confusion in load balancing - servers without any CPU use end up not accepting connections.

- Migration to PHP5 caused some unhappy voices among 3rd party MediaWiki users. We tend to use fresh PHP releases, and often - new features (like code autoloading)

- MediaWiki itself is home-grown application that is widely used by 3rd party users as well. This often forces use of modular design with some features replaced by more efficient components. This leads to situation, where performance-oriented use of software requires disabling some features and enabling more complex distributed backends for features, that can be handled by trivial solutions on regular locations.

As components are quite heavy, we tend to have powerful application servers (nowadays - 8-core) - it allows more efficient resource usage than trying to load-balance heavy requests across zillions of nodes.

Caching

The common mistake is to believe that database is too slow and everything in it has to be cached somewhere else. In scaled out environments reads are very efficient, and difference of time between efficient MySQL query and memcached request is negligible - both may execute in less than 1ms usually).

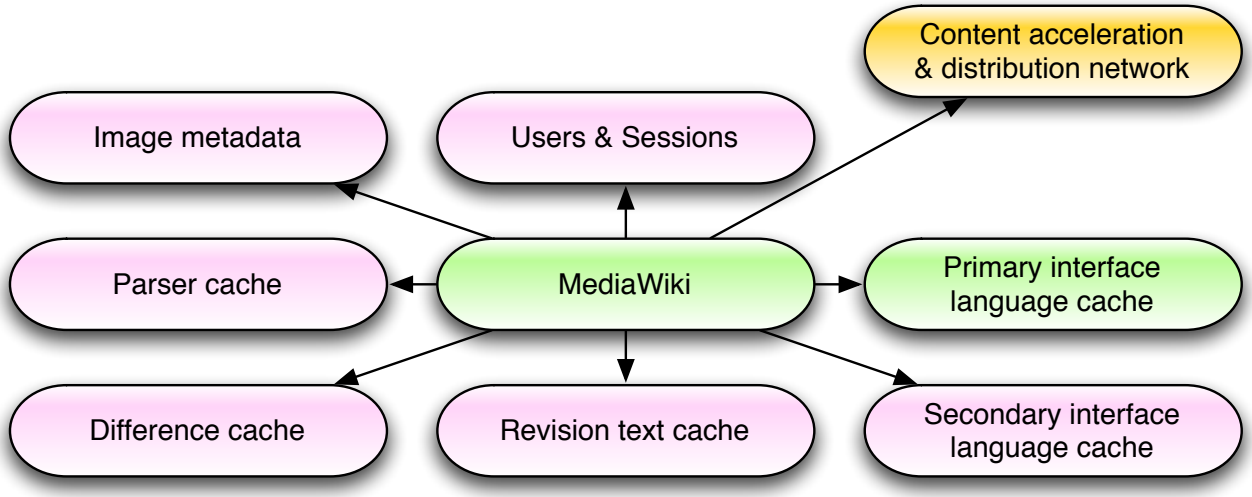
Sometimes memcached request may be more expensive than database query simply because it has to establish connection to a server, whereas database connection is usually open.

There're important questions to be asked before deciding, to cache or not to cache:

- How often the object will be read? How often will it be invalidated?
- How often will the object be written?
- Does it have to stay persistent?
- Does it have to be replicated all over the database cluster?
- What will be the hitrate?
- Can it be cached locally?
- What are the invalidation rules?
- Can individual cache objects be grouped together?

There have been situations where the remote cache request would always miss, then application would fetch the object from database and would populate the cache, hence the operation of 'get small bit' would get 200% overhead.

These are basic shared cache groups in Wikipedia:



The memcached memory pool consists of 30 servers, 2GB each (um, 60GB!)

Remote caches use LRU expiration policy, which is quite efficient for most uses.

- Parser cache is most critical for application performance - it stores preprocessed HTML output. The content language, wikitext, allows integrating multiple content sources, pre-processing clauses and depend on many other content objects. Page views by users with same settings would not cause re-parsing of content.
- 'Compare articles' output is cached in difference cache, and it is especially efficient with 'recent changes patrol' - people reviewing changes.
- Interface language cache combines dynamic interface changes, together with messages from source code. Even parsing the PHP file is quite expensive, so having this serialized or directly in APC cache helps a lot.
- Revision cache was introduced after revision texts (hundreds of gigabytes of content) were moved away from core databases to bigger but slower distributed storage. Some operations do need source text, and hitting the storage backend may be much slower, and having this content in shared cache helps.
- Image metadata keeps simple information about images, that doesn't have to go to databases, and may be forgotten.
- Session information may be especially dynamic but has no long-term value - writing into database would result with replication pollution, even though on single server it could be quite efficient.
- There're quite a few other small bits cached, like pre-parsed interface parts, information about database server states, etc.
- If application requests more than two objects from same cache type, it is already important to reconsider the caching. In past there were >50 accesses for simple kind of information, and that added more than 50ms to each request of a popular page. Simply storing such kind of cache as .db file on every application server caused immediate performance improvements.

Code inside application is usually pretty trivial - removing a content node from all caches, including CDN is just calling Title::invalidateCache() method, same goes for User objects.

Of course, more direct approach to accessing caches is provided, with simple hash-database-like interface (get/set/add/etc) - eventually that is mapped to a real backend, like APC, memcached, MySQL or Fake (though mostly just memcached used on Wikipedia).

Memcached has much cheaper connection costs, therefore it is not that expensive to maintain persistent connections (200 app servers with 20 apache instances on each would cause _just_ 4000 efficiently managed connections). MySQL would create thread for each connection, but in this case event-based engine handles it without sweating.

The example of our memcached configuration file (still maintained by hand):

```
<?php
/*
 * Before altering the wgMemCachedServers array below, make sure you planned
 * your change. Memcached compute a hash of the data and given the hash
 * assign the value to one of the servers.
```

```
 * If you remove / comment / change order of servers, the hash will miss
 * and that can result in bad performance for the cluster !
 *
 * Hashar, based on dammit comments. Nov 28 2005.
 *
 */
$wgMemCachedPersistent = true;
$wgUseMemCached = true;
$wgMainCacheType = CACHE_MEMCACHED;

$wgMemCachedInstanceSize = 2000;

$wgMemCachedServers = array(
    # ACTIVE LIST
    # PLEASE MOVE SERVERS TO THE DOWN LIST RATHER THAN COMMENTING THEM OUT IN-PLACE
    # If a server goes down, you must replace its slot with another server
    # You can take a server from the spare list

# SLOT      HOST
    0   => '10.0.2.55:11000',
    1   => '10.0.2.102:11000',
    2   => '10.0.2.119:11000',
...
    25  => '10.0.2.118:11000',

/**** DOWN ****
    XX  => '10.0.2.62:11000',
    XX  => '10.0.2.59:11000',
...
***** SPARE ****
    XX  => '10.0.2.111:11000',
...
*************/

);

# vim: set sts=4 sw=4 et :
?>
```

There’re quite a few warnings in configuration, asking not to insert or delete entries - making so confuses the distribution algorithm (which generally is rand() % numberofservers), and all 60GB of cache are rendered useless.

Profiling

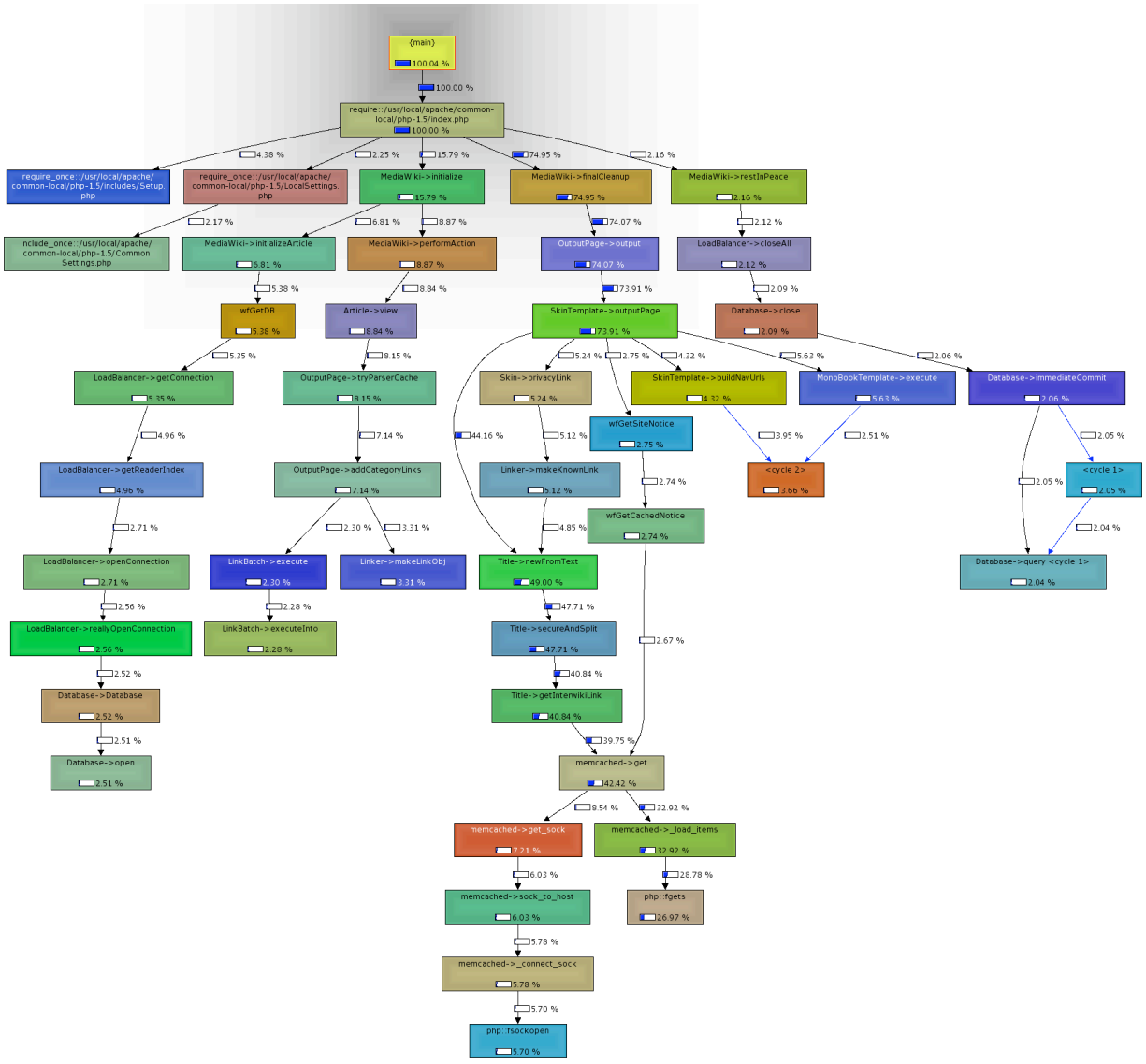
For general stack profiling sometimes tools like gprof and oprofile help, but application-level profiling may need additional tools.

MediaWiki has enclosed many major sections with wfProfileIn() and wfProfileOut() calls, which map into either database-backed profiler, or can send aggregated data out to a collecting agent, that later provides visualization. Example of such real-time information can be seen at:

http://noc.wikimedia.org/cgi-bin/report.py

The code for reporting agent is at
http://svn.wikimedia.org/viewvc/mediawiki/trunk/udpprofile/

Additionally very useful duet of tools is xdebug profiling combined with kcachegrind - it may provide visualized tree of calls with execution times and also allows to dive deeper into various execution trees:



http://flake.defau.lt/pics/mediawiki.png

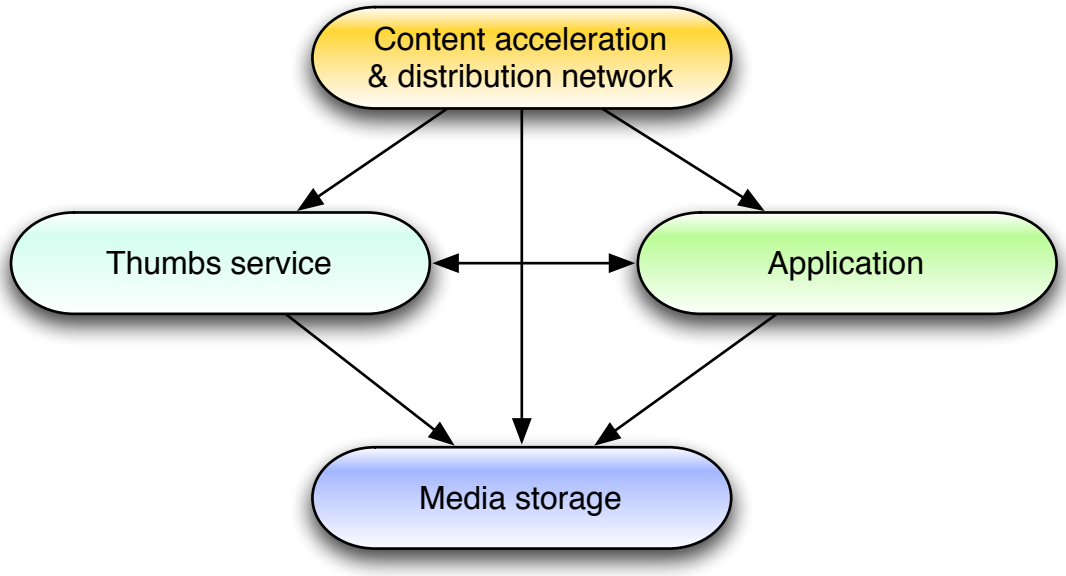
In this particular example immediate optimization urge could be seen for Title::getInterwikiLink(). Other interesting information can be revealed - that in this case there was less time spent opening database connections than opening memcached connections, or that adding links to categories did cause separate links resolution.

Of course, every profiler has overhead, and there constantly exists a struggle to have as accurate data as possible at lower costs.

Profilers can help a lot in detecting site problems - slow connections to particular systems pop up immediately on reports. Combined with ‘proctitle’ module, it makes big site administration somewhat easier.

Media storage

For now it is the major single point of failure in whole Wikipedia cluster. Servers are not redundant, backups are less than perfect, etc:



Most of media delivery is already done by CDN, but the actual storage and thumbs generation has to go into inner systems.

At the CDN level requests for thumbnails are delivered to distinct servers, each having a single portion of whole thumbnail set (now split by project). In case file does not exist, thumbs HTTP server (lighttpd) traps the 404-file-does-not-exist request and sends thumb generation request to application cluster, which then uses NFS to read data from main media storage to thumb nodes.

Lighttpd simply has such special configuration statements:

server.error-handler-404 = "/scripts/thumb-handler.php"

```
fastcgi.server             = ( ".php" => ( "localhost" => (
                                 "socket" => "/tmp/php-fastcgi.socket",
                                 "bin-path" => "/usr/local/bin/php",
                                 "bin-environment" => (
                                   "PHP_FCGI_CHILDREN" => "32"
                                 ) ) ) )
server.max-fds = 8192
server.max-worker = 8
server.network-backend = "writev"
server.max-keep-alive-requests = 128
server.max-keep-alive-idle = 30
server.max-read-idle = 30
server.max-write-idle = 30
$HTTP["url"] == "/fundraising/2006/meter.png" {
        setenv.add-response-header  = ( "Cache-Control" => "max-age=300,s-maxage=300" )
}
expire.url = ("/fundraising/2006/meter.png" => "access 5 minutes")
```

The speed of lighttpd (and lack of image server failures) has been one of reasons the more scalable solution has not been designed so far. Currently there is work to build more scalable and redundant solution, employing CARP, replication and distribution.

Currently main image server is again under serious I/O contention.

Database

Database balancing

LoadBalancer.php: >600 lines of connection balancing code :)

```
/*
 * Get a Database object
 * @param integer $db Index of the connection to get. May be DB_MASTER for the
 *                master (for write queries), DB_SLAVE for potentially lagged
 *                read queries, or an integer >= 0 for a particular server.
 *
 * @param mixed $groups Query groups. An array of group names that this query
 *              belongs to. May contain a single string if the query is only
 *              in one group.
 */
function &wfGetDB( $db = DB_LAST, $groups = array() ) {
        global $wgLoadBalancer;
        $ret = $wgLoadBalancer->getConnection( $db, true, $groups );
        return $ret;
}
```

Every database use in code has to acquire Database object via wfGetDB(), and it is always good idea to separate queries that require master and queries that do not.

LoadBalancer needs array of servers for initialization, and per-project arrays are built by our separate configuration file, db.php:

```
# Define which projects go into what server groups (separate replication clusterS):
$sectionsByDB = array(
        'enwiki' => 's1',
        'commonswiki' => 's2',
        'fiwiki' => 's2',
        'nlwiki' => 's2',
        'nowiki' => 's2',
...
        'dewiki' => 's2a',
        'frwiki' => 's3a',
        'jawiki' => 's3a',
);

$sectionLoads = array(
        's1' => array(
                'db2'       => 0,
                'db3'       => 200,
                'ariel'     => 0, // watchlist only
                'db4'       => 200,
                'db6'       => 200,
                'db7'       => 70, // used for batch jobs
        ),
        's2' => array(
                'db8'     => 0,
                'lomaria' => 150,
                'ixia'    => 200,
        ),
        's2a' => array(
                'db8'     => 0,
                'ixia'    => 200,
                'lomaria' => 0,
                'holbach' => 2000, // replicating dewiki only
        ),
        /* s3 */ 'DEFAULT' => array(
                'thistle' => 0,
                'adler'   => 200,
                'samuel'  => 200,
                'db1'     => 200,
```

```
                'db5'     => 80,
        ),
        's3a' => array(
                'thistle' => 0,
                'samuel'   => 200,
                'db1'     => 200,
                'db5'     => 0,
                'webster' => 2000, // replicating this section only
        ),
);

# Falling back to default section
if ( isset( $sectionsByDB[$wgDBname] ) ) {
        $section = $sectionsByDB[$wgDBname];
} else {
        $section = 'DEFAULT';
}
$loads = $sectionLoads[$section];

/*if ( $section != 's1' ) {
        $wgReadOnly = 'Routine maintenance, should be back in a few minutes';
}*/

$commonsServers = array(
        'db8'       => 100,
        'lomaria'   => 100,
        'ixia'      => 100,
);

// Make sure commons servers are in the main array
foreach ( $commonsServers as $server => $load ) {
        if ( !isset( $loads[$server] ) ) {
                $loads[$server] = 0;
        }
}

$enwikiWatchlistServers = array(
        'ariel' => 100,
);

$enwikiContributionServers = array(
        'db6' => 1000,
);
$enwikiDumpServers = array(
        'db7' => 1000,
);

# Miscellaneous settings

$dbHostsByName = array(
        'ariel'    => '10.0.0.2',
        'holbach'  => '10.0.0.24',
        'webster'  => '10.0.0.23',
...
        'db9'      => '10.0.0.242',
);

$wgDBservers = array();
$templateServer = array(
        'dbname'      => $wgDBname,
        'user'        => $wgDBuser,
        'password'    => $wgDBpassword,
        'type'        => 'mysql',
        'flags'       => DBO_DEFAULT,
        'max lag'     => 30,
        'max threads' => 100,
        'groupLoads'  => array(),

);
```

```
foreach( $loads as $name => $load ) {
        $server = array(
                'host'     => $dbHostsByName[$name],
                'hostname' => $name,
                'load'     => $load,
        ) + $templateServer;

        if ( isset( $commonsServers[$name] ) ) {
                $server['groupLoads']['commons'] = $commonsServers[$name];
        }

        if ( $wgDBname == 'enwiki' && isset( $enwikiWatchlistServers[$name] ) ) {
                $server['groupLoads']['watchlist'] = $enwikiWatchlistServers[$name];
                $server['groupLoads']['recentchangeslinked'] = $enwikiWatchlistServers[$name];
        }
        if ( $wgDBname == 'enwiki' && isset( $enwikiContributionServers[$name] ) ) {
                $server['groupLoads']['contributions'] = $enwikiContributionServers[$name];
        }
        if ( $wgDBname == 'enwiki' && isset( $enwikiDumpServers[$name] ) ) {
                $server['groupLoads']['dump'] = $enwikiDumpServers[$name];
        }

        $wgDBservers[] = $server;
}


# The master generally has more threads running than the others
$wgDBservers[0]['max threads'] = 200;
```

Such kind of application-level balancing flexibility allows efficient database use, at the cost of having that inside single application.

LoadBalancer does some additional magic - checking server lags, skipping delayed servers, and does put site read-only in case of ‘all lagged’ event.

Additionally it has few other useful commands:

LB::commitAll() - doesn’t use 2PC, but with pessimistic locking by InnoDB it is not that needed

LB::closeAll() - closes all connections open

LB::pingAll() - keeps all sessions up

LB::waitFor() - asks for slave servers to catch up with replication position specified

LB::saveMasterPos() - saves master position into user session, so that subsequent requests would not get out-of-date operation, usually used for rw requests

LB::loadMasterPos() - retrieves master position from session and waits for it if needed

LB::getLagTimes() - retrieves lag times from servers or cached variable

We never use persistent (outside web-request scope) connections for databases, as connect costs are negligible, but having thousands of threads may be not.

Database API

Database class used to be simple DBMS-specific query call abstraction, but eventually ended up as query abstraction tool too.

Use of Database class forces to treat changes as method calls rather than text.

Example of trivial select:

```
$res = wfGetDB(DB_SLAVE)->select(
          /* FROM    */ array( 'page', 'categorylinks' ),
          /* FIELDS  */ array( 'page_title', 'page_namespace', 'page_len', 'cl_sortkey' ),
          /* WHERE   */ array( $pageCondition,
                               'cl_from          =  page_id',
                               'cl_to'           => $this->title->getDBKey()),
          /* COMMENT */ __METHOD__,
          /* OPTIONS */ array( 'ORDER BY' => $this->flip ? 'cl_sortkey DESC' : 'cl_sortkey',
                               'USE INDEX' => 'cl_sortkey',
                               'LIMIT'    => $this->limit + 1 ) );
```

Every literal passed is properly escaped (and MySQL tends to be efficient with quoted numbers) - no more space for SQL injection errors. Additionally, due to dynamic nature of language, various magic expansion happens, if needed:

```
$count = wfGetDB()->selectField('job','job_id',array('job_id'=>array(5,6,7,8)));
```

Would be expanded into:

```
SELECT /* Database::selectField */ job_id FROM job WHERE job_id IN('5','6','7','8') LIMIT 1
```

As lists and arrays inside application are already passed around as arrays, actual calls to Database class methods look much more elegant, than any attempts to build queries.

Object-relational mapping is usually avoided as it brings less control over batch updates or retrievals. The need to specify fields not only reduces information sent over the wire, but also more often employs use of covering indexes.

Reading that data can be done in quite usual PHP-like method:

```
               while( $row = $dbr->fetchObject ( $res ) ) {
                      $title = Title::makeTitle( $row->page_namespace, $row->page_title );
                      $this->addPage( $title, $row->cl_sortkey, $row->page_len );
               }
```

Additionally, entries can be wrapped into ResultWrapper:

```
$r = new ResultWrapper($dbr, $res);
```

Which allows passing single result object around.

Database class can be used stand-alone, without LoadBalancer, though there might be references to other MediaWiki classes.

Writing methods are easy too:

```
wfGetDB(DB_MASTER)->insert('table', array( 'name' => $this->name, 'parent' => $this->parent));
```

Array of arrays can be passed as an argument too, resulting in multiple-row insert.

Other write operations (DELETE, UPDATE) are combinations of INSERT and SELECT syntax:

```
wfGetDB(DB_MASTER)->update('table',
      /* SET */ array('modified' => 1, ...) ,
      /* WHERE */ array('id' => $ids)
);
```

or

```
$wfGetDB(DB_MASTER)->delete('table','*'); // delete all rows
```

There're lots of helper functions for manual query building too.

Usually the interface does all required transformations (like adding prefixes to table names, if needed), but for manual query building various helpers can be used:

```
      $nameWithQuotes = $dbr->escape($name);
      extract($dbr->tableNames('user','watchlist'));
      $sql = "SELECT wl_namespace,wl_title FROM $watchlist,$use
            WHERE wl_user=user_id AND wl_user=$nameWithQuotes";
```

There is also support for deadlock-prone queries, where a retry logic is done.

On top of such base database class there're few higher-level wrappers, like Pager (and subclasses).

It allows to write efficient index-based offsets pager (instead of ugly LIMIT 50 OFFSET 10000). An example of use:

```
<?php

class CategoryPager extends AlphabeticPager {
        function getQueryInfo() {
                return array(
                        'tables' => array('categorylinks'),
                        'fields' => array('cl_to','count(*) AS count'),
                        'options' => array('GROUP BY' => 'cl_to')
                        );
        }

        function getIndexField() {
                return "cl_to";
        }

        function formatRow($result) {
                global $wgLang;
                $title = Title::makeTitle( NS_CATEGORY, $result->cl_to );
                return (
                        '<li>' . $this->getSkin()->makeLinkObj( $title, $title->getText() )
                        . ' ' .  wfMsgExt( 'nmembers', array( 'parsemag', 'escape'),
                                        $wgLang->formatNum( $result->count ) )
                        . "</li>\n" );
        }
}

?>
```

It produces a list based on simple specification of dataset and parameters in request (offset, order and limit). Additional call to AlphabeticPager::getNavigationBar() provides with a resultset browsing interface. There's more sophisticated TablePager class as well.

Database servers

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2917 mysql     15   0 14.4g  14g 4016 S  2.3 91.0  78838:10 mysqld
```

Though we currently use 16GB-class machines (RAM size is main database server power metric for us), we still consider ourselves scale-out shop.

The main ideology in operating database servers is RAIS:

Redundant Array of Inexpensive/Independent/Instable[sic] Servers

- RAID0. Seems to provide additional performance/space. Having half of that capacity with an idea that a failure will degrade performance even more doesn't sound like an efficient idea. Also, we do notice disk problems earlier :-) This disk configuration should be probably called AID, instead of RAID0.
- innodb_flush_log_at_trx_commit=0, tempted to do innodb_flush_method=nosync. If a server crashes for any reason, there's not much need to look at its data consistency. Usually that server will need hardware repairs anyway. InnoDB transaction recovery would take half an hour or more. Failing over to another server will usually be much faster. In case of master failure, last properly replicated event is last event for whole environment. No 'last transaction contains millions' worries makes the management of such environment much easier - an idea often forgotten by web applications people.
- Complete datacenter crash = loss of just a few transactions. We'd have more stuff to do than worry about those.
- Software crashes just don't happen. We run MySQL 4.0 which is completely rock-solid. Didn't see software-related crashes for many years! :)
- Well, hardware doesn't crash too much either. In case of serious master crash we can just switch site read-only, and most users won't even notice (except few - "oh no! the edit button disappeared!"). We're used to have core database uptimes over a year:

```
samuel:          08:09:57 up 354 days, 21:38,  0 users,  load average: 0.67, 0.71, 0.77
ixia:            08:09:57 up 354 days, 21:38,  1 user,  load average: 0.96, 1.08, 1.08
lomaria:         08:09:57 up 354 days, 21:38,  0 users,  load average: 1.97, 1.17, 1.27
thistle:         08:09:57 up 45 days,  9:38,  0 users,  load average: 0.18, 0.15, 0.15
webster:         08:09:57 up 354 days, 19:43,  1 user,  load average: 1.86, 1.81, 1.56
db1:             08:09:58 up 184 days,  4:48,  1 user,  load average: 0.66, 0.55, 0.55
db2:             08:09:57 up 229 days,  1:46,  3 users,  load average: 0.29, 0.39, 0.50
db4:             08:09:57 up 118 days, 13:07,  0 users,  load average: 0.30, 0.45, 0.51
...
```

- Rebuilding the server after some serious crash is not that expensive. Copying data at gigabit speeds (>80MB/s) is usually less than an hour even for largest datasets.
- RAIS mysql-node configuration:

```
back_log=1000
skip-name-resolve
slave-skip-errors=0,1213,1158,1053,1007,1062
innodb_buffer_pool_size=12000M
innodb_log_file_size=500M
innodb_flush_log_at_trx_commit=0
innodb_lock_wait_timeout=10
query_cache_size=64M # Actually, not used
set-variable    = table_cache=6000
set-variable    = thread_cache=300
```

External Storage

In Wikipedia context ‘External Storage’ doesn’t mean SAN or NAS or NFS. It was quick hack that worked to keep most of our data outside of database servers.

The revision storage used to be in ‘text’ table in main database:

```
CREATE TABLE `text` (
  `old_id` int(8) unsigned NOT NULL auto_increment,
  `old_text` mediumtext NOT NULL,
  `old_flags` tinyblob NOT NULL,
  UNIQUE KEY `old_id` (`old_id`)
) TYPE=InnoDB
```

We did hit multiple issues with that:

- It would take space on disk
- It would make resyncs to other nodes longer
- It would consume cache
- It would... have the usual trouble of having too much data

The solution (hack?) was pretty simple - abusing the flags field by putting ‘external’ flag into it, and treating old_text as locator rather than text itself, then migrating the data record-by-record into other locations.

The easiest way to implement other location was to deploy lots of small mysql replication setups across application servers (which always come with big disks, just in case, ATA/SATA is cheap anyway):

```
/* db.php External Store definitions */
$externalTemplate = $templateServer;
$externalTemplate['flags'] = 0; // No transactions

$wgExternalServers = array(

        'cluster1' => array(
                array( 'host'=> '10.0.2.30', 'load' =>1)+$externalTemplate , // Master
                array( 'host'=> '10.0.2.28', 'load' =>1)+$externalTemplate ,
                array( 'host'=> '10.0.2.29', 'load' =>1)+$externalTemplate ,
        ),
        'cluster2' => array(
                array( 'host'=> '10.0.2.27', 'load' =>1)+$externalTemplate , // Master
                array( 'host'=> '10.0.2.25', 'load' =>1)+$externalTemplate ,
                array( 'host'=> '10.0.2.24', 'load' =>1)+$externalTemplate ,
        ),
      /* up to ... */
      $wgDefaultExternalStore = array( 'DB://cluster11', 'DB://cluster10', 'DB://cluster12' );
```

Once ExternalStore finds out it needs to fetch something, it can call just:

```
$db = LoadBalancer::newFromParams( $wgExternalServers[$cluster])->getConnection(DB_SLAVE);
```

The logic in script is slightly more extended, as there’s in-script cache of LoadBalancer objects.

Database queries

All database interaction is optimized around MySQL's methods of reading the data.

Some of requirements for every query are obvious, some are luxury, but:

- Every query must have appropriate index for reads - traversing a thousand rows even in an index to return one hundred is already bad table access efficiency, though of course, it still happens in some operations (where more criteria is silently slipped into reporting functions, without managing edge cases).
- Every query result should be sorted by index, not by filesorts. This means strict and predictable path of data access. This can not yet be possible by few major functions (timeline of changes for multiple objects, like watchlist, 'related changes', etc). Now such operations are sent to a dedicated server in order not to have filesort-caused I/O contention on other operations on 'high-speed' core servers.
- Paging / offsetting should be done just by key positions than by resultsets (no LIMIT 10 OFFSET 100, just WHERE id>... LIMIT 10) - this becomes a problem when some spider hits 'next 5000 results' too often.
- No sparse data reads should be done, except for hottest tables. This means covering indexes for huge tables, allowing to do reads in different range directions/scopes. Narrow tables (like m:n relation cross-indexes) usually always have covering indexes to both directions.

Some fat-big-tables have extended covering indexing just on particular nodes, like this index on revision:

```
KEY `usertext_timestamp` (`rev_user_text`,`rev_timestamp`)
```

is extended to:

```
KEY `usertext_timestamp` (`rev_user_text`,`rev_timestamp`,`rev_user`,`rev_deleted`,
                          `rev_minor_edit`,`rev_text_id`,`rev_comment`)
```

on nodes handling 'Contributions for a user' task. As operation would usually hit lots of cold data, it is much more efficient to keep it clustered this way at least on single server.

More of such effect is described at
http://dammit.lt/2007/01/26/mysql-covering-index-performance/

As well, this effect can be seen in any proper benchmark - range scans from indexes are extremely efficient in MySQL, as all data is always clustered together, by one key or another.

- Queries prone to hit multiversioning troubles have to be rewritten accordingly - as various slow tasks can hold 'purging' lock. An example can be job queue pop, where:

```
SELECT * FROM job LIMIT 1
```

has to be rewritten into:

```
SELECT * FROM job WHERE id>${lastid} LIMIT 1
```

in order not to hit lots of deleted, but still not purged rows.

Splitting
Multiple database servers allow splitting in many different patterns:

- Move read load to slaves

At mediocre or low write rates this always helps - scaling read operations becomes especially cheap (everybody knows this nowadays anyway).

If replication positions are accounted in application, the inconsistencies are barely noticed.

- Partitioning by data segments

It can be done just for slaves (different slaves handle different datasets), or for masters as well (splitting database clusters into multiple clusters). This method allows to reuse outdated database hardware to extremes (say, sending German of French requests to crippled-by-4GB database server). Slower disks or less storage can be resolved this way too. The only problem remains the inter-segment (in our case - interlanguage) operations, which we do none (nearly). We had to remove some of cross-database access hacks and to maintain separate connections or RPC to handle some of accesses.

Of course, more dense data observes much higher cache hits, with much less stress on disk subsystem. The only database which cannot be optimized this way for now is English Wikipedia (it is the only database having it's own dedicated database cluster).

- Partitioning by tasks

Though some tasks can have extended slaves (in terms of indexing or added tables), some features have to go to separate instances running modified or different software.

The External Store can be an example of such partitioning.

The more radical approach was to move search to Lucene-based daemon (we used to have dedicated database servers for search, using MySQL fulltext search).

Some indexing methods (like fuzzy text search, intersections, queues, data streams, etc) could/can be handled by other solutions too.

- Partition by time

Some operations may be done delayed, in serialized manner, hence not clogging replication or I/O. Job queues help for that too. Of course, having multiple layers of data persistence (squid, memcached, APC, mysql core, external storage) automatically puts most active data to front of the system, and archival data lives least touched in slowest storage.

Some changes can be propagated in completely different cycles from replication or application updates - the information that is distributed to application nodes as on-disk hash databases can be delayed and doesn't need that high consistency requirements, though read performance is improved magnificently.

Data itself

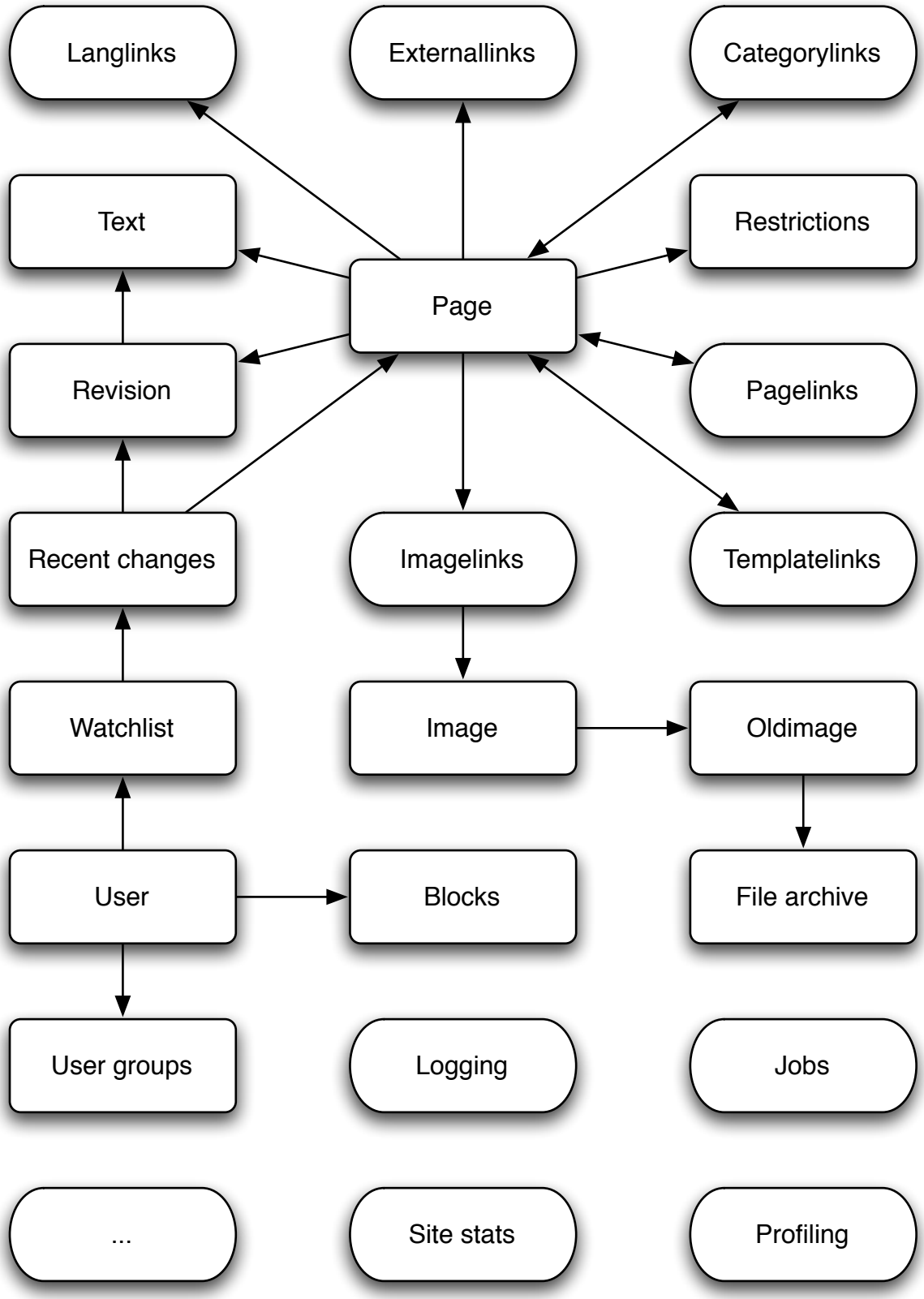

Langlinks
Externallinks
Categorylinks
Text
Restrictions
Page
Revision
Pagelinks
Recent changes
Imagelinks
Templatelinks
Watchlist
Image
Oldimage
User
Blocks
File archive
User groups
Logging
Jobs
...
Site stats
Profiling

Table	Rows (counts)	Data (bytes)	Index (bytes)
page	8m	1g	1g
revision	110m	17g	40g
pagelinks	12m	12g	8g
categorylinks	13m	1.5g	2g
imagelinks	7m	700m	450m
image	1m	450m	100m
text	110m	13g	0
watchlist	26m	2.3g	1.3g
user	3m	5g	300m
recentchanges	8m	2g	2.5g
externallinks	15m	4g	4.5g

compare to numbers last year:

Table	Rows	Data	Index
revision	25m	8g	11g
page	4m	0.5g	0.5g
pagelinks	45m	4.5g	3g
text	25m	0.3t	-
watchlist	8m	1g	9.5g
user	1m	1.2g	60m
recent	5m	1.3g	1g

Many tables have more indexing than data (even considering, that PK is index too, though not included into the ‘index’ column). Usually extended indexing is used (composite PKs for data clustering, then included into other indexes for covering reads:

Revision PK is (Page ID, Revision ID) - every page has it’s own tree of data

Watchlist PK is (User, Namespace, Title) - containing all fields, this makes every other index fully covering, even if it is on less fields.

Pagelinks PK is (from, namespace, title) - having all fields too, making (ns,title) index covering.

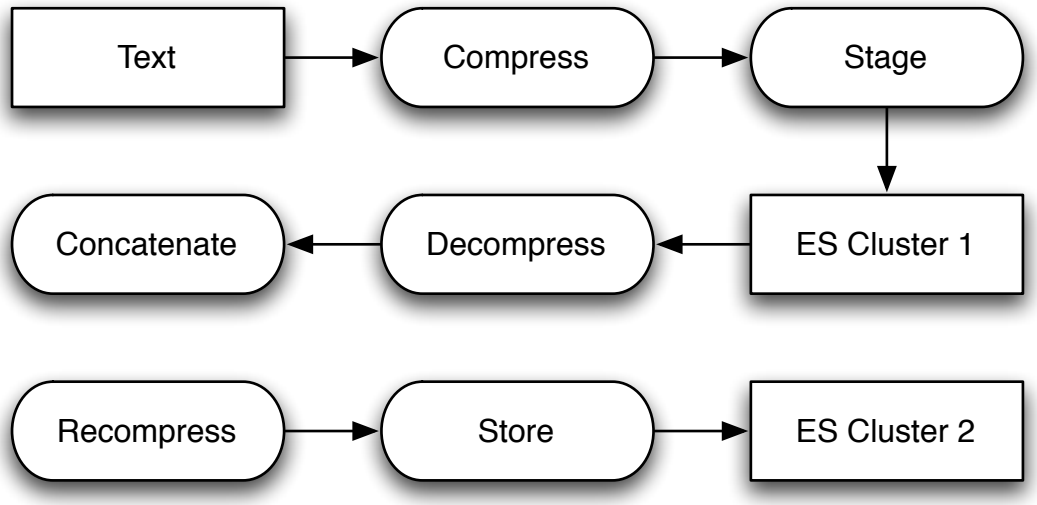
Same is for most tables - no crossindex tables have their own PKs, they just provide really efficient way to connect multiple entities - usually possible just with single block read.

Compression

One of easy ways to save resources is actually by using compression wherever data compresses and takes more than a kilobyte. Of course, sending compressed information over internet makes page loads faster, but even for internal communications, having various blobs compresed helps in:

- Less fragmentation (less data is split off from row to another blocks in database)
- Less bytes to send over the network
- Less data for DB server to work on - better memory/cpu efficiency, etc.
- Less data for DB server to store

And all this comes out from really fast zlib (gzip) library, which is often treated as too expensive. It takes a fraction of millisecond to compress ten kilobytes of text, resulting in few times smaller object to keep around.



For external store extreme efficiency is achieved by recompressing - concatenating bunches of old revisions, then applying compression again. Efficient algorithms take similarity of revisions into account, ending up with 5%-10% of original size, instead of 30%-50% for single-pass compression.

Of course, such efficiency is required just for long-term storage, and short term storage in cache (all objects are compressed there) usually does not need that.

Of course, the availability of cheap partitioned/distributed storage on application servers does allow to slack and some text still waits for the re-compression stage.

Unfortunately, most of media formats are already compressed - there huge wins are just for text data.

Search

The major component for it is Lucene framework (and it's ports). Due to licensing issues Wikipedia did not run Java Virtual Machine from Sun Microsystems (non-free software is not matching free content ideals), so alternatives were chosen - at first GCJ-based solution, afterwards .net Lucene port was used on top of Mono .NET framework.

The 'mwdaemon' is very simple HTTP interface to Lucene indexes.

The major components for search are:

- LuceneSearch.php - the MediaWiki extension, utilizing mwdaemon - it is just simple proxy of requests to external HTTP-backed agent
- MWDaemon - the daemon itself. For now the only way to scale it is to run multiple copies of same index on different machines, though the platform supports 'parallel queries' - those would have to hit all index segments, but cache efficiency on machines would be higher. For now simply splitting by language and adding more copies of index helps.
- Index Builder - offline batch builder of index. The incremental updates used to leak memory, so only batch solution has been enabled.
- Mono (or GCJ.. or JVM... depends on mood - we have support for all).
- Balancing. Currently done by LVS.

Current solution is closely integrated with MediaWiki internals - it is not possible as generic solution at the moment.

Lucene project though has included a simple generic search server in their latest releases - probably using it directly may be a way to go for this platform.

LVS: Load balancer

LVS (Linux Virtual Service) is critical component, which allows efficient load balancing in front of CDN, between CDN and Application, and between Application and Search.

The key advantages against the competition are:

- Kernel-level
- Director gets just inbound traffic, outbound traffic is sent directly from processing node with proper source IP
- Efficient

We have written tools for monitoring, pooling and depooling servers (so that connections don't hit black holes), as well as managing weights (same pool of servers has both 1-cpu and 8-core machines).

As we're using balanced connection count based distribution, any server that refuses to work properly may start getting more load, thus slowing down the site serving.

Monitoring of extremes in connection counters is always important:

```
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  upload.pmtpa.wikimedia.org:h wlc
  -> sq10.pmtpa.wmnet:http        Route   10     2          151577
  -> sq1.pmtpa.wmnet:http         Route   10     2497       1014
  -> sq4.pmtpa.wmnet:http         Route   10     2459       1047
  -> sq5.pmtpa.wmnet:http         Route   10     2389       1048
  -> sq6.pmtpa.wmnet:http         Route   10     2429       1123
  -> sq8.pmtpa.wmnet:http         Route   10     2416       1024
  -> sq2.pmtpa.wmnet:http         Route   10     2389       970
  -> sq7.pmtpa.wmnet:http         Route   10     2457       1008
```

In this case sq10 needs gentle touch.

The script used to monitor and pool servers can be found at:

http://svn.wikimedia.org/viewvc/mediawiki/trunk/pybal/

Previous generation of monitoring script is called Lvsmon and is kept at:

https://wikitech.leuksman.com/view/Lvsmon

Administration

The site-ops personnel is tiny, so the tools are not that sophisticated...

NFS

Lots of bits not used in constant site operation are kept in shared directory. Shared directories are good. Except when shared directory server goes down - this has caused few downtimes in past.

Of course, nowadays shared directory server is not used for many additional tasks, what used to be the case, but still it is SPOF, though with less impact.

dsh

The key tool used by many administration scripts is quite simple perl program called ‘dsh’.

SSH connection is established for each node in nodelists (could be read from file) and commands are executed.

This is used together with deployment scripts. Most ‘sync’ scripts dsh into bunch of boxes and ask to copy in data from shared directory to local file systems.

The sync-scripts make backups of files, do verifications of PHP code (against occasionally slipped in syntax errors), even may announce synced file paths on IRC channel (it is main communication media for distributed team).

Nagios

The monitoring software is in constant race, will it be the first to announce system failure, or will that be hundreds joining Wikipedia IRC channels.

Maintaining and automatic configuration of Nagios has been constant issue. Having it in usable state has helped few times to have accurate diagnosis of what went wrong.

Can be found at http://nagios.wikimedia.org/

Ganglia

Has been tremendous help in understanding site load usage - monitoring resource usage and trends is quite important for planning of hardware acquisitions. Some problems can be immediately spotted by comparing different server load patterns.

http://ganglia.wikimedia.org/

People

It is tremendous pleasure to build anything being in such a nice team. Come see us at:

#wikimedia-tech on Freenode (irc.freenode.net)

Your ideas will be always helpful. Especially if implemented already ;-)