Service-Oriented Data Denormalization for MediaWiki

Master thesis by

Peng Li

Student Number

1824507

Under the supervision of

Guillaume Pierre

Vrije Universiteit Amsterdam

Department of Computer Science

June 29, 2010

Abstract

Many techniques have been proposed to scale web applications. However, the data interdependencies between the database queries and transactions issued by the applications limit their efficiency. To overcome the limitation, service-oriented data denormalization technique restructures a centralized database into several independent data services, and reorganizes the application into a service-oriented architecture. Each data service handles specific business logic, and has exclusive access to its own database. This project aims at applying service-oriented data denormalization technique to MediaWiki. First of all, I have split MediaWiki's data into many independent data services by studying the data access pattern in the code of MediaWiki extensively. Secondly, I have defined and developed these data services in terms of Web services in PHP. Each Web service has its own database, and exchanges data via HTTP and JSON (JavaScript Object Notation). After denormalizing MediaWiki, I have scaled individual Web service by applying special techniques on each Web service according to its own characteristics. Finally, I have studied the scalability of the denormalized MediaWiki, and demonstrated that the maximum sustainable throughput grows linearly with the number of hosting servers used.

Contents

1. Introduction

The World-Wide Web has profoundly reshaped people's daily life. People rely increasingly on Web applications. They are demanding for more accessible functionalities and higher performance on Web applications, which urges business and public organizations to make great efforts to enrich their Web applications. As a result, these organizations require more and more scalable Web hosting architectures which can support unpredictable and wildly fluctuating levels of workload. It is also necessary for Web hosting architectures to support the addition of new functionality and easily handle more business needs without a noticeable loss of performance [1]. So far this problem is well understood for static content [5, 7, 8, 11], but it is still a challenge to provide scalable architectures for hosting dynamic Web applications.

Dynamic Web applications are widely used to deliver personalized and frequently updated information to individual customer. These Web applications generate dynamic content according to each customer's specific requests, which issues any queries to an underlying database. These queries do not only retrieve data from the database, but can also perform Update, Delete, and Insert operations (UDI queries). The diversity of these queries is also known as the complexity of the workload. Many techniques have been proposed to scale dynamic Web applications, such as data caching, computation distribution and database replication. But there is no single best technique for dynamic Web applications [2]. Application administrators need to study their application's workload to decide on the best-suited solution which is often a conjunction of several techniques.

Each of these main techniques, data caching, computation distribution and database replication, improves scalability efficiently in specific situations. Data caching such as fetched pages or results of database queries is effective only when the workload has high temporal locality. Computation distribution, for example the Edge Computing of Akamai, dispatches user requests among several edge servers, thereby distributes computation workload across multiple edge servers. Computation distribution can be very efficient depending on the application, but its scalability is always limited by the back-end database as a bottleneck of the entire system. Then database replication can be introduced to eliminate the bottleneck only when the workload includes few UDI queries. However, with large number of UDI queries, the system performance can be degraded significantly by the cost of committing UDI queries among all database replicas. Another option is partial replication. Instead of replicating all data items at all database replicas, partial replication assigns copies of a subset of data items to some database replicas. By this approach, UDI queries involving the subset of data items can be performed only on a limited number of database replicas. However, partial replication in turn creates a real difficulty because read queries may span different database partitions.

To overcome the limitations of techniques mentioned before, service-oriented data denormalization restructures a centralized Web database into several independent data services, and reorganizes the application into a service-oriented architecture [3]. Each data service handles specific business logic, and has exclusive access to its own database. The workload can be dispatched among these data services, which reduces the complexity of

workload. The benefit of this loosely coupled architecture is that each data service can be applied any special scaling techniques according to its own characteristics. The data denormalization technique has already been applied to three mainstream benchmark Web applications: TPC-W, RUBiS and RUBBoS, which demonstrates that the maximum sustainable throughput grows linearly with the number of hosting resources used.

Although the data denormalization technique is effective when applied to these simple benchmark Web applications, there is still a question about how well it applies to the real-world applications. Compared with these benchmark Web applications, a real-world application can have more data items in its databases, more complex code with Object-Oriented features and more complex workload. Each real-world application also has its particular features to meet specific business needs. All these differences can make real-world applications more difficult to be denormalized than simple benchmark applications. To prove the potential of the data denormalization technique, we need to apply it on an application which has most common characteristics of real-world applications such as being used widely and having high complexity of its database, code and workload.

This project aims at applying service-oriented data denormalization technique to MediaWiki, a real-world application used by the popular Wikipedia [22]. MediaWiki is a good sample for illustrating how data denormalization technique improves scalability of a real-world Web application under a real workload. Originally, MediaWiki is a web-based wiki software written in PHP for use on Wikipedia which has more than 15 million articles written by volunteers who collaborate around the world. MediaWiki is implemented in a typical three-tier architecture relying on a centralized database. It supports master-slaver replication technique on its back-end databases, and caching techniques that includes object caching on application servers and data caching on front-end servers. Its workload consists mostly of read queries. On the other hand, almost all of Wikipedia's articles can be edited by anyone, hence there is also a large number of update and insert queries in the workload (the Delete function is disabled by default).

The contribution of this project is as follows. First of all, I have split MediaWiki's database into independent data services by studying the data access pattern in the code of MediaWiki extensively. Secondly, I have defined and developed these data services in terms of Web services in PHP. Each Web service has its own database, and exchanges data via HTTP and JSON (JavaScript Object Notation) [14]. After denormalizing MediaWiki, I have scaled individual Web services by applying special techniques on each Web service according to its own characteristics. Finally, I have studied the scalability of the entire denormalized MediaWiki, and demonstrated that the maximum sustainable throughput grows almost linearly with the number of hosting servers used.

The rest of this thesis is organized as follows: Section 2 presents the related work. Section 3 describes the process of data denormalization for MediaWiki. Section 4 details the design and implementation of denormalized MediaWiki as well as scaling techniques involved in. Section 5 presents performance evaluations. Section 6 concludes. Finally section 7 discusses future work.

2. Related Work

Existing techniques to scale dynamic web applications can be broadly classified as computation distribution, database replication and data caching.

Data Caching

Data caching can be categorized into database caching and web caching. The web caching is the most straightforward one, by which web objects such as, entire HTML pages, JSP results, or page fragments are cached in the web servers, a proxy server or an edge server. While reuse of complex web objects may save some processing cost, web caching does not work very efficiently for dynamically generated and personalized contents. The database caching can be implemented in two different ways: content-blind data caching and content-aware data caching [2]. In case of content-blind data caching (CBC), the query results are cached independently and not merged together [6]. Since data are stored as result sets, CBC may save processing cost and return results immediately when a hit occurs. With content-aware data caching (CAC), query records returned from the back-end database is cached in local databases [15, 16]. Instead of storing result sets independently, CAC stores query records in tables with the entire or partial back-end schema in local databases. An incoming query is checked to see if it can be satisfied from a local database. If the check result is positive, then the query can be executed locally. Otherwise, the query must be passed to the back-end database. Then the result returned by the back-end database can be inserted in the local database for future use. CAC offers high reusability of cached data since the same rows in tables might be used for different queries especially when these queries have overlapping predicates. But it also induces high processing cost as checking every incoming query with all previously cached queries. To reduce this cost, CAC systems exploits the fact that a web application's workload often consists of a small set of read and writes query templates [17]. A query template is a set of SQL queries whose selection predicates share the same structure and vary only in a few numeric or string arguments. All instances of one template are aggregated and their predicates merged together, using data structures, which is called Merged Aggregate Predicates (MAPs). MAPs are parameterized with indexes. Thus query containment check is converted into an index-based data search, achieving significant speedups. However, both CAC and CBC work well under the workloads with high temporal locality and few UDI queries.

Computation distribution

With computation distribution, user requests are dispatched among several edge servers running the same code, thereby distributing the computational workload across multiple servers. Although computation distribution offloads the computational load on back-end servers to many edge servers, the back-end databases still have to handle all database requests from edge servers. Then the back-end database can be the bottleneck of the system.

Data partitioning

Data partitioning techniques are desirable for performance improvement of a database,

which have been widely used in the design process of the distributed databases [20, 21]. There are two kinds of partitioning: horizontal and vertical, which partition tables either vertically or horizontally into smaller fragments. Then each fragment can be stored at the location where it is most frequently used, so that most operations are local and network traffic is reduced. The difficulty in the data partitioning is that ensuring the fragments independence, and operations on these fragments are efficient when the distributed database environment changes.

Database replication

Database replication is a common technique to improve the performance of a back-end database system. Typically, an entire database is replicated across multiple "slave" databases which can be located on edge servers or organized within a cluster [2, 18]. The read query workload can be distributed evenly across all the database replicas, thereby improving the throughput of the database system. However, to handle UDI queries and maintain the consistency of all database replicas is difficult. Each change caused by UDI queries needs to be propagated to all the database replicas to keep them in consistency, which incurs a significant cost in terms of performance and network traffic overhead. Instead of replicating complete databases, partial replication replicates a subset of data items on some database replicas to limit the number of replicas involved in certain UDI queries. However, the difficulty in this approach is handling complex queries that span different database partitions.

Service-oriented data denormalization

In [3], the authors propose a new approach to scale dynamic web applications, called service-oriented data denormalization. With service-oriented data denormalization, web applications can be restructured into multiple independent web services, each with a very simple data access pattern. Then every web service can be applied the best-suited scaling technique according to its characteristics. The authors have applied the technique to TPC-W [19] to demonstrate the effectiveness of this technique. They have denormalized TPC-W into 8 independent web services, each of them has its own exclusive database and handles specific user requests. After denormalization, the workload of each web service can be easily characterized. Some of them have read-only or read-dominant workload. These services can be scaled by database replication or caching data techniques. Other services have to handle more UDI queries. These services can be scaled using partial replication or data partitioning techniques. Finally, the authors have evaluated the scalability and performance of the denormalized TPC-W and shown that the maximum sustainable throughput of the denormalized TPC-W grows linearly with the quantity of hosting resources used.

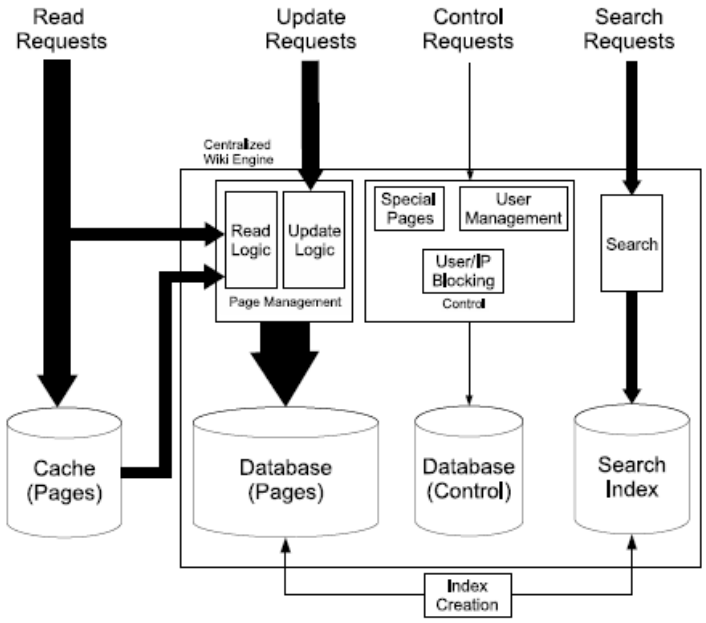

Figure 1. Current Wikipedia Architecture (adapted from [23])

3. MediaWiki Data Denormalization

3.1. MediaWiki and Wikipedia Architecture

MediaWiki is a popular free web-based wiki software application written in PHP and based on a centralized architecture. Originally, MediaWiki is implemented as a set of PHP scripts that access a central database.

The best implementation of MediaWiki is Wikipedia. It contains about 15 million articles (or pages) written collaboratively by volunteers around the world on Wikipedia. Almost all of these articles can be edited by anyone with access to the site. To handle these massive data and the increasing number of requests, Wikipedia is deployed in a special way to improve its scalability and performance in practice.

As shown in Figure 1, Wikipedia can be divided into two parts: cache (pages) and the centralized wiki engine. Cache (pages) can be located on front-end servers to handle most of the read requests, which reduce the response time and the workload on the web and database server. In the centralized wiki engine, there are three basic components: page management, control and search. The page management component and control component have the access to a centralized database containing pages and control information, but the search component has its own database. In particular, there is a specific component: index creation which is used to update the search index database according to the centralized database (pages).

The page management component is the core of Wikipedia. It handles all update requests and a fraction of read requests. The database (pages) stores most of the information provided by Wikipedia (e.g., encyclopedic articles, user information and discussions) in the form of wiki pages. Each page has a unique identifier consisting of a character string. Pages can be created, read and modified by any user. Once a page is modified, a new revision is created

Components in MediaWiki Engine	Accessed Table sets in MediaWiki Database
Page Management component	Page text and associated information Images and media Caching tables Miscellaneous
Search component	Search index
Control component	User accounts and privileges and watchlist Blocks Statics and logging Page text and associated information Images and media
Index Creation component	Page text Search index

Table 1. MediaWiki Engine Components and Accessed Table sets in Database

and associated with this page. Therefore, each page has a revision history containing all editions of this page. A page can also be configured to redirect all its read requests to another page as a link. But only privileged users have the option to rename, delete and protect pages from being edited.

The search component allows users to look wiki pages up via keyword search in the search index database and receive a list of links to the related wiki pages. The search index database is isolated from the centralized database. Then all search requests are handled by the search component, which reduces the workload from the centralized database.

The control component provides the rest of the functions. It consists of user management, User/IP blocking and special pages. Special pages are generated by the Wikipedia to perform some specific functions. For example, a special page named *export pages* can be used to export wiki pages from the Wikipedia into local XML files. User/IP blocking allows administrators to block certain IP addresses or user accounts from editing wiki pages. User management allows users to authenticate with the system and choose individual preferences, watchlist and contributions information.

3.2. MediaWiki Database Layout

In this project, we have chosen to denormalize the current version of MediaWiki 1.15.2. There are 41 tables in MediaWiki database [24] (more details are described in Appendix 9.1). These tables can be classified into following table sets: page text and associated information, search index, images and media, user accounts and privileges and watchlist, blocks, statics and logging, caching tables and miscellaneous. As described in Table 1, each table set can be used by one or more components in MediaWiki engine.

The page text and associated information table set can be considered the “core of the MediaWiki database”. Typically, it contains the *page*, *revision* and *text* tables. Each page (article) in a MediaWiki implementation has an entry in the *page* table which identifies it by title (within a certain namespace) and contains some essential metadata. All the page's

revisions are stored in the *revision* table, except the real text of these revisions which are stored in the *text* table respectively. To retrieve the text of a page, MediaWiki first searches for *page_title* in the *page* table. Then, *page_latest* is used to search the revision table for *rev_id*, and *rev_text_id* is obtained in the process. The value obtained for *rev_text_id* is used to search for *old_id* in the text table to retrieve the text. This table set also contains other tables which store associated information about MediaWiki pages, such as *archive, pagelinks, templatelinks, externallinks, category, categorylinks, langlinks, recentchanges, redirect, page_restrictions, protected_titles and page_props*. These tables can be used in the control component to generate extra information about MediaWiki pages. For example, the *recentchanges* table contains information about the latest modifications made to the MediaWiki pages, and is used to generate the pages about the recent changes pages, related changes pages, watchlists and the list of new pages by special pages.

The search index table set contains only one table of *searchindex*. The *searchindex* table contains IDs and titles of pages from *page* table, and indexed text from *text* table. It is used to provide full text search from both titles and texts of pages by the search component.

The images and media table set consists of *image*, *imagelinks*, *oldimage* and *filearchive* tables. The *image* table describes images and other uploaded files. In the *imagelinks* table, an image link on the MediaWiki pages is linked to its entry in the *image* table. The *oldimage* table holds information about old revisions of images and will be filled when one uploads a new version of an existing image to the MediaWiki. The *filearchive* table stores all the media that has been deleted, similar to the *archive* table's job for text. This is the table that makes image undeletion possible.

The user accounts and privileges and watchlist table set is used for user administration, for example the *user* table stores basic account information about user (name, password, "My Preferences" setting, email address, etc). The blocks table set stores details of IP addresses and users who have been blocked from editing. The statics and logging table set contains some aggregate information on the state of MediaWiki, and basic log information. The caching tables are used for some generic cache operations, for instance, the *querycache* table is used for caching expensive grouped queries. The miscellaneous table set contains all other tables, such as the *job* table stores all jobs performed by parallel apache threads or a command-line daemon, the *trackbacks* table stores TRACKBACKS from external sites – provides notification when someone links to one of the MediaWiki's pages, and the *interwiki* table stores the interwiki prefixes with their targets.

3.3. MediaWiki Database Access

In Table 2, all available wrapper functions for database access in MediaWiki are listed. By default, all read-only queries are sent to slave databases, and UDI queries and transactions are sent to the master database. Updates are executed on the master database at first, and then they are propagated to the slave databases. In case only one database is used in MediaWiki, the slave database and master database are configured to the same one.

One special design in MediaWiki is that MediaWiki always sends a BEGIN command as the first query, and a COMMIT command as the last query before the HTML output is sent. However transactions cannot be nested in MySQL. Once a new transaction is called by issuing a BEGIN command at first, the BEGIN command causes an implicit COMMIT

Query Types	Functions

Read-only	$dbr = &wfGetDB(DB_SLAVE);
	function select($table, $vars, $conds = '', $fname = 'Database::select', $options = array());
	function selectRow($table, $vars, $conds = '', $fname = 'Database::select', $options = array());

UDI	$dbw = wfGetDB(DB_MASTER);
	function insert($table, $a, $fname = 'Database::insert', $options = array());
	function insertSelect($destTable, $srcTable, $varMap, $conds, $fname='Database::insertSelect', $insertOptions=array(), $selectOptions=array());
	function update($table, $values, $conds, $fname = 'Database::update', $options = array());
	function delete($table, $conds, $fname = 'Database::delete');
	function deleteJoin($delTable, $joinTable, $delVar, $joinVar, $conds, $fname= 'Database :: deleteJoin ');

Transaction	$dbw = wfGetDB(DB_MASTER); $dbw->begin(); /* Do queries */ $dbw->commit();

Table 2. MediaWiki Database API: wrapper functions for database access [25]

operation as if one had done a COMMIT command and then started the new transaction. So when ACID requirements are vital, a transaction can be used explicitly as the syntax described in Table 2. For example, the update operation on pages is handled in a transaction in the *article* class in MediaWiki. All explicit transactions are discussed in Section 3.4.1.

MediaWiki also provides a query() function for raw SQL, but it is rarely called in MediaWiki core scripts. It is only used in other scripts about Special pages and extensions.

3.4. Data Denormalization

In this project, we have denormalized MediaWiki by following two types of constraints: transactions and query data overlap [3].

3.4.1. Denormalization According to Transactions

First of all, all data accessed by one transaction must be inside a single database, and therefore inside a single data service. In MediaWiki, a transaction is defined in the syntax as described in Table 2. We have aggregated all tables or columns named in queries between *$dbw->begin()* and *$dbw->commit()* into a preliminary data service, as described in Table 3.

Then, many preliminary data services can be gained according to transactions. If two preliminary data services contain the same data items (tables or columns), we combine these two data services into a single data service. Similar to transactions, the columns that are updated in UDI queries must be local to the same data service to be able to provide atomicity (as listed in Table 4). Columns that are only read by the UDI queries can reside on other data services.

Data Services	Transactions	Functions	Tables/Columns
Article Management	Update page	Article->doEdit()	Page text revision redirect recentchanges
	Create a new page	Article->doEdit()	
	Delete page	Article->doDeleteArticle()	
Image Management	Image upload	LocalFile->recordUpload2()	Image oldimage imagelinks filearchive
	Image delete	LocalRepo->cleanupDeletedBatch()	
Watchlist Management	Notification on page changes	UserMailer->notifyOnPageChange()	Watchlist
Site Statistics	Update site statics	SiteStatsUpdate->doUpdate()	site_stats

Table 3. Transactions in MediaWiki and Associated Preliminary Data Services

3.4.2. Denormalization According to Read Queries

Second, no query data should overlap among multiple data services. To this end, we have rewritten complex database queries into multiple, simpler queries. For example, a common search query is of the form:

```
SELECT page_id, page_namespace, page_title FROM `page`,`searchindex` WHERE page_id=si_page AND MATCH(si_text) AGAINST('+searchterm' IN BOOLEAN MODE) AND page_is_redirect=0 AND page_namespace IN (0) LIMIT 20;
```

The search query can be rewritten into two simple queries:

```
Query 1: SELECT si_page FROM `searchindex` WHERE MATCH (si_text) AGAINST ('+searchterm' IN BOOLEAN MODE);
```

```
Query 2: SELECT page_id, page_namespace, page_title FROM `page` WHERE page_id IN ('+Query 1 result ') AND page_is_redirect=0 AND page_namespace IN (0) LIMIT 20;
```

Then the returned result of the first query to search data service (a data service as concluded in Table 4) is used as input for the second one and the final result is returned by the second query to article management data service.

3.5. Data services of the denormalized MediaWiki

As shown in Table 4, we have denormalized the centralized MediaWiki into as many independent data services as possible.

In this project, we focus on the data services of article management, image management and search. The article management and image management data services can be accessed by the core component of page management in MediaWiki engine (as described in Figure 1). The search data service matches the database in the component of search.

We have combined the data services of object cache and JobQuery into the portal data service, and therefore as the portal web service. The portal web service works as a MediaWiki User Interface, and supports generic object caching and query job caching. Although the data

Data services	Data Tables	Functions
Article Management	page text revision redirect recentchanges	getRedirectTarget(), insertRedirect(), updateRedirectOn(), pageData(), ageDataFromTitle(), pageDataFromId(), getCount(),insertOn(), updateRevisionOn(), updateIfNewerOn(), doEdit(), getCascadeProtectionSources(), loadRestrictionsFromRow(), LinkBatch::doQuery(), fetchFromConds(), replaceInternal(), addLinkObj(), SearchMySQL->queryMain(), getTimestampFromId(), getContributors(), LinkCache->addLinkObj(), doDeleteArticle()
Image Management	image oldimage imagelinks filearchive site_stats(ss_images)	LocalFile->recordUpload2(), LocalFile->loadFromDB(), LocalRepo->cleanupDeletedBatch()
Search	searchindex	SearchMySQ->LsearchText(),SearchMySQ->searchTitle(),SearchMySQ->newQueryMain(),SearchMySQ->queryMain(),SearchMySQ->update()
User Management	User user_newtalk	getContributors(), Pager::reallyDoQuery(),checkNewtalk( )
Watchlist Management	watchlist	notifyOnPageChange(), insertNewArticle(),doEdit(),
Trackbacks Management	trackbacks	addTrackbacks(), deletetrackback()
Category Management	category categorylinks	Category::refreshCounts()
Blocklist Management	ipblocks	Block->load(), Block->loadRange()
Counter	page(page_counter) hitcounter	incViewCount(),
Object Cache	objectcache	BagOStuff::set(), delete(), expireall(), deleteall(), get(), keys(),
JobQueue	job	pop(), batchInsert(),
Site Statistics	site_stats	SiteStats::doUpdate()
Logging Statistics	Logging	Pager::reallyDoQuery(),

Table 4. Data Services of the Denormalized MediaWiki

service of object cache is independent and can be abstracted as a single data service, the performance gained by this way is negative. If the object cache data service is separated as a single data service, all requests should be forwarded to this data service at first. The extra cost in the network traffic is expensive because each data service is hosted on the different machine.

Other data services will be implemented in the future work.

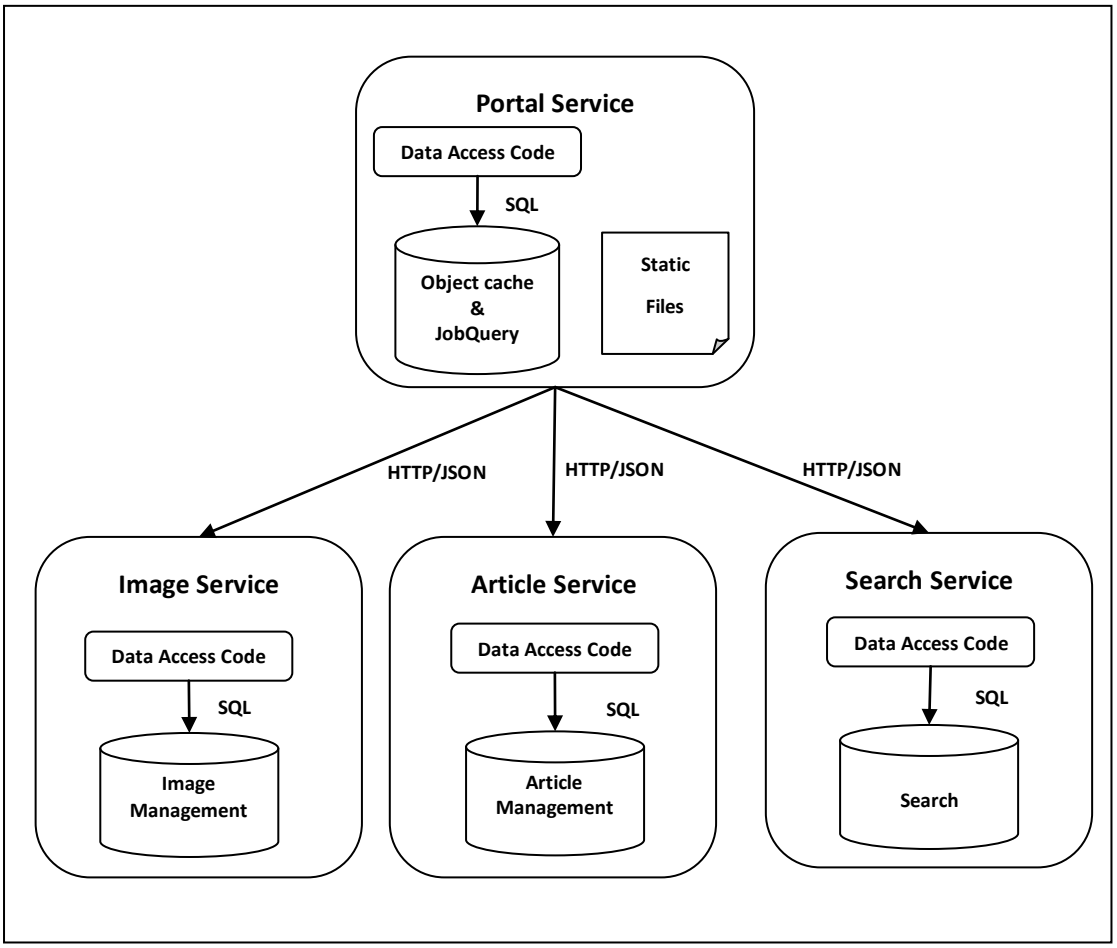

Figure 2. Service-Oriented Design Model of Denormalized MediaWiki

4. Service-Oriented Design, Implementation and Scaling

4.1. Web Services Organization

After denormalizing the MediaWiki, four main data services can be abstracted: article management, image management, search and portal (consisting of object cache and JobQuery) data service. As shown in Figure 2, four web services can be designed according to these data services： portal service, article service, image service and search service respectively. Details of these web services will be discussed in the following sections.

We have chosen HTTP as the exchange protocol and JSON (JavaScript Object Notation) as the data-interchange format. JSON is an open, lightweight, human-readable and platform-independent data format. It can be parsed by PHP implementations with incredible ease. In PHP 5 (>=5.2.0), any type except a resource can be encoded as JSON format by using *json_encode()* function. For example, an array as ('a'=>1,'b'=>2,'c'=>3,'d'=>4,'e'=>5) can be encoded as {"a":1,"b":2,"c":3,"d":4,"e":5} in form of a collection of name/value pairs delimited by curly braces {}. Simply the function *json_decode()* returns the value encoded in JSON in appropriate PHP type.

```
function TitleHash($str, $rank){
        $sum = 0;
        for ($c=0; $c < strlen($str); $c++)
                $sum += ord($str[$c]);
        return $sum%$rank;
}
```

Figure 3. A Simple Hash Function

4.2. Portal Service

The portal service works as a MediaWiki User Interface, which generates the entirety of the output HTML code of a wiki page. It gathers all information about a wiki page from other web services, and parses this information into HTML code according to the MediaWiki Markup Language. Then the portal service caches the PHP objects as well as a default expire time into the *objectcache* table. It also handles some static resource of MediaWiki, for example, image files for MediaWiki skins and JavaScript files for client side.

Since the portal service handles most load of rendering HTML pages and all requests for static files, we have chosen computation distribution and data caching techniques to scale the portal service. The code of portal service and static files in MediaWiki can be quite simply replicated among several servers (machines). These portal servers are running the same code to generate HML pages.

In addition, many data caching techniques can be applied on the portal servers, such as HTML page caching, object caching, and PHP caching [26]. HTML page caching and object caching are supported by MediaWiki without extra software. To enable HTML page caching, $wgUseFileCache, $wgFileCacheDirectory and $wgShowIPinHeader variables should be set up in *LocalSettings.php*. As a result, the rendered HTML page for each page of the MediaWiki is to be stored in an individual file on the file system. Any subsequent requests from anonymous users are met not by rendering the page again, but by simply sending the stored HTML version if its modification time is more than the corresponding *page_touched* in *page* table (that means the HTML file is cached after the last update of the page).

The object caching is enabled by setting the $wgMainCacheType variable as CACHE_DB, and then MediaWiki uses a database for caching. The $wgMainCacheType also can be set as CACHE_ACCEL or CACHE_MEMCACHED to use third-party caching software for object cache. Furthermore, APC (Alternative PHP cache), PHP accelerator and eAccelerator can be used.

4.3. Article Service

The article service is the core web service in the denormalized MediaWiki, which is in charge of MediaWiki page management. It performs all operations about MediaWiki pages including three transactions: update page, create a new page and delete page.

As described in chapter of MediaWiki database layout, each MediaWiki page is identified by *page_title* and *page_namespace*. Therefore, we mark each row of tables: *page*, *text*,

Web services	Scaling Techniques
Portal service	Computation Distribution Data Caching
Article service	Data Partitioning
Image service	Data Partitioning
Search service	Data Replication

Table 5. Web Services and Applied Scaling techniques

revision, *redirect* and *recentchanges* with a unique *page_title* (in namespace 0). Then we can partition the database vertically into smaller fragments which can further be hosted by different database servers. To determine which database server should be responsible for a MediaWiki page, we use a simple string hash function as described in Figure 3. The function calculates the sum (*$sum*) of ASCII values of all letters in the *page_title* ($str), and return *$N=$sum%$rank* (where $rank is the number of database servers). Then the $N database server should have the entire data about the MediaWiki page named as *page_title*. To reduce the calculation cost, we cache the hash result of the form: ($str, $N) in an array.

4.4. Image Service

The image service manages the tables of *image*, *imagelinks*, *oldimage* and *filearchive*， which store the metadata of images. But All image files are stored in an image folder (set by $wgUploadPath) on a file system. Basically the image service handles the queries for the path (URIs) to images, and performs two transactions: upload image and delete image.

The image service does not handle the requests for real image files, but generates the URIs to these files. In this project, these URIs are handled by the portal service which can access the image folder on a shared file system as well. Furthermore it is possible to access image files stored in foreign repositories, without needing to upload them to the MediaWiki (by setting the $wgForeignFileRepos array).

Similar to MediaWiki pages, each image file has a unique *img_name* (but in namespace 6 as file namespace). Once an image is to be uploaded, the storage path of the image is calculated from the *img_name* (using the sha1 hash function in this case). So the straightforward approach to scale the image service is partitioning the tables: *image*, *imagelinks*, *oldimage* and *filearchive* vertically, as the way to scale article service above.

4.5. Search Service

The search service provides full text search. It searches the keywords both in the titles and the content of pages. Whenever a page is created or modified, a *SearchMySQL->update()* function is called to update the appropriate row in the *searchindex* table.

In practice, the *searchindex* database is isolated from the other databases. It is updated periodically according to the content of pages by index creation component as shown in Figure 1. Similarly the search service can be scaled by replicating the *searchindex* database. Then these replicas are updated periodically to keep up to date.

In Table 5, all web services are listed as well as applied scaling techniques.

5. Performance Evaluation

5.1. Experimental Environment

We performed all experiments on the DAS-3, an 85-node Linux-based server cluster [27]. Each node has a dual-CPU / dual-core 2.4 GHz AMD Opteron DP 280, 4 GB of memory and 250 GB of local disk space. Nodes are connected to each other with a gigabit LAN such that the network latency between the servers is negligible. We used Apache 2.2.12, MySQL 5.1.37 and PHP 5.3.0 as web servers, and Pound 2.4 as load balancers to distribute HTTP requests among multiple web servers.

5.2. Populating the Databases

Before experiments, we populated the database with 115,420 wiki pages (with the current revisions). Because we performed the experiments by using a Wikipedia trace file which contains a trace of 10% of all user requests issued to Wikipedia (in English) during one hour. It is not necessary to import all wiki pages into the database because only 115,420 wiki pages were requested in the trace file.

To extract these 115,420 wiki pages from a Wikipedia snapshot XML file, we used MWDumper first. The MWDumper supports filtering a Wikipedia snapshot XML file according to a title list and outputting back to SQL statements to add these filtered pages directly to a database. We used *sed* command to make a list of all titles of pages that are requested in the trace file (no duplicated titles). Then these titles can be used as inputs to extract associated pages from the Wikipedia snapshot XML file. Sometimes it is not good enough using MWDumper alone. We tested all requests for wiki pages (requests beginning with '/wiki/' in the trace file) one by one under no time limit. There were still some wiki pages missed. Then we used Special:ExportPages on Wikipedia site (English) to export these missed pages into XML files, and imported them into the database from these XML files.

5.3. Workload Generation

We used TraceBench [12] to generate workloads for experiments. TraceBench can read a real-world Wikipedia trace file (10% of all user requests issued to English Wikipedia during one hour), and reduce the traffic intensity of this trace. By this way, TraceBench creates very realistic workloads with different intensities ranging from very low up to the original traffic intensity of the trace file. In addition TraceBench can add POST data, obtained from a database, to the trace file. Then the generated workloads can be issued on the denormalized MediaWiki system by WikiBench.

5.4. Experimental Procedures

As described in Figure 4, a controller in WikiBench reads the workload trace file (generated by TraceBench) and distributes trace lines to several workers in WikiBench. The workers iteratively get a trace line from the controller, parse the URL and timestamp from that

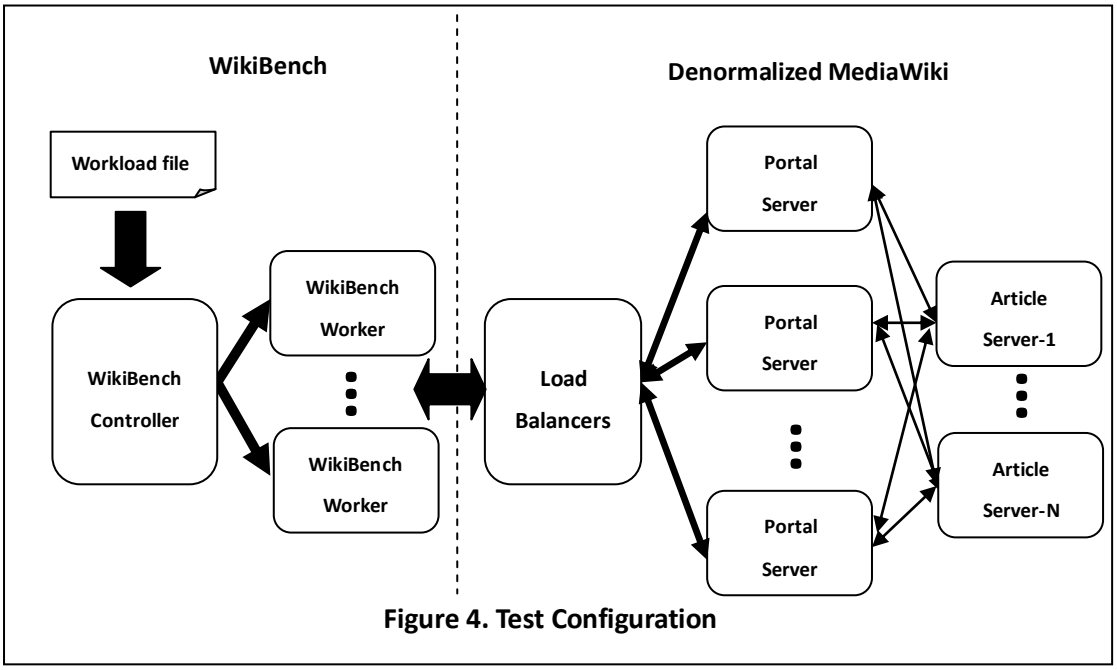

Figure 4. Test Configuration

line, and perform a GET or POST HTTP request to the load balancers. The load balancers distribute the HTTP requests among multiple portal servers (running the same code and caching HTML page). Each portal server communicates with many article servers (scaled by data partitioning technique) to retrieve the data about wiki pages.

To perform the experiments on the DAS3, we used the *prun* command to reserve the requested number of nodes and execute a script on all nodes parallelly to generate the configuration files and then run appropriate code on each node in a certain order. In the script, each node is identified by the value of $PRUN_CPU_RANK and designated a role for each test, for example as a portal server. Before starting a web server on a node, the script copies the needed code and database files from the file server into the node's local disk, and checks the required configuration file whether it is available. As described in Figure 4, the article servers should start first, and then save their IP addresses into a PHP configuration file on a shared disk in DAS3. The PHP configuration file can be accessed by the portal servers. The portal servers also should save their IP addresses in to a Pound configuration file. Then the load balancers read the Pound configuration files, and start working. The controller of WikiBench checks a flag file (created by the load balancers), start reading a trace file, and waits for workers. Similarly, the workers check a flag file but created by the controller, and then start reading the requests from the controller and forwarding them to the load balancers.

5.5. Result

We started the experiments by using one portal server and one article server. We then added a portal server or an article server to measure the maximum throughput of the entire denormalized MediaWiki system until 9 servers were used in total. In all cases, we deliberately over-allocated the number of WikiBench workers to make sure that all the requests from the WikiBench controller should be forwarded (no missed requests in WikiBench). We used a number of trace files created by the TraceBench with different traffic intensities, ranging from 0.1% to 10.0% of the original sample of a Wikipedia trace. Since

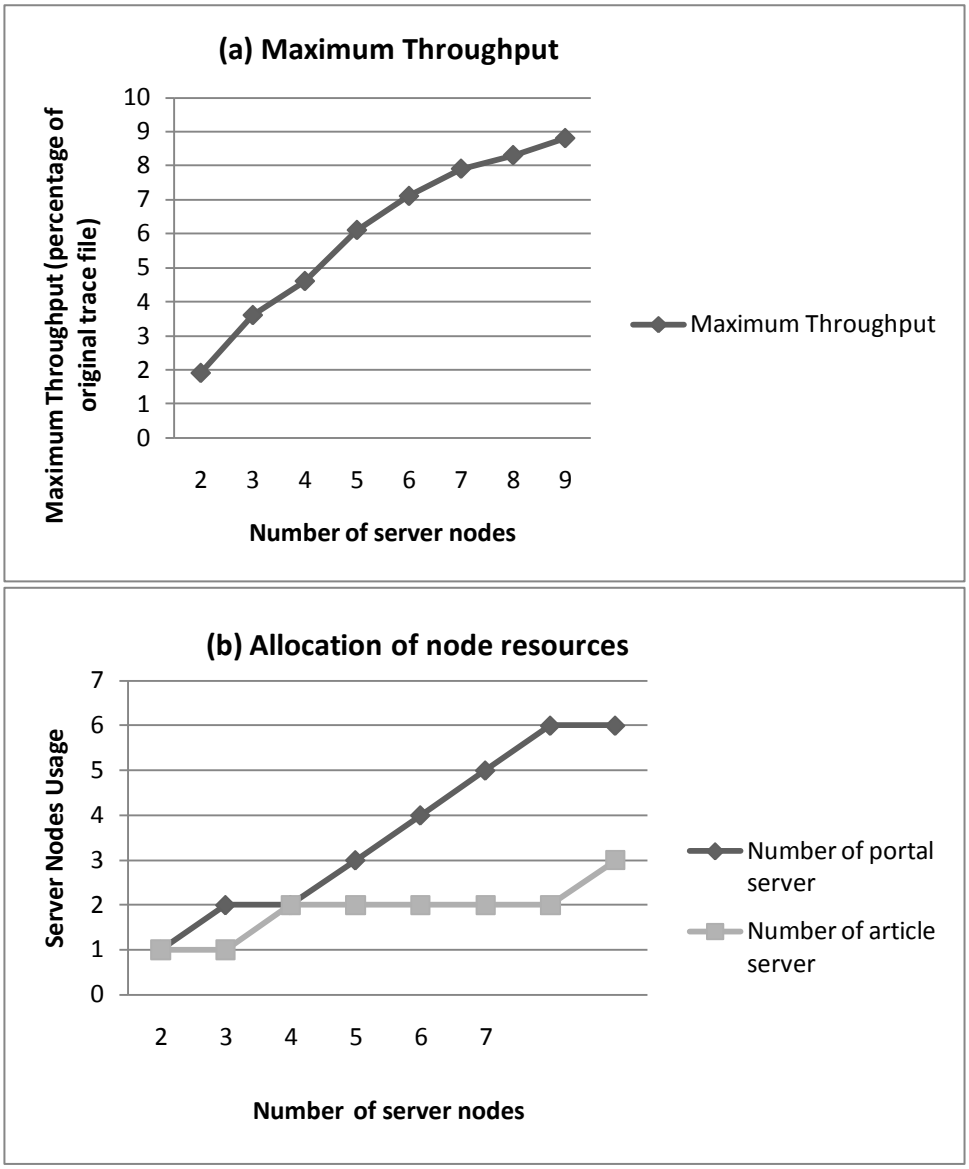

Figure 5. Scalability of Denormalized MediaWiki

almost 50% of requests could be for static files (non-wiki page requests), we defined a SLA more strictly to that 90% of web interactions of wiki page requests must complete under 500 ms. We measured the maximum throughput of the entire denormalized MediaWiki system by increasing 0.1% intensity per time until the SLA was not respected, then we added a portal server or an article server to continue.

As shown in Figure 5-a, the denormalized MediaWiki shows near linear scalability. It handles 9% requests of original trace file when using 9 server nodes (6 portal server and 3 article server). In Figure 5-b, it shows that adding more portal servers is the first option to scale the system, which significantly improves the scalability. That makes sense because most portions of the original requests are for the static files in the portal servers (non-wiki page requests). When more wiki page requests are issued, more article servers are needed, as shown in the Figure 5-b. Due to the time and resource limit, we did not test the system with more nodes and workload. But we believe that the denormalized MediaWiki can easily be

scaled further.

6. Future Work

As mentioned before, only the core of the denormalized MediaWiki has been developed and tested in this project. For the future work more web services are to be designed and implemented, in order to make the denormalized MediaWiki fully work as the original MediaWiki but with more scalability. Especially, when the original MediaWiki is updated for new features or added more data items in the database, it is important to make sure the denormalized MediaWiki can also support the new features or extend to handle the new data items in the data services.

As shown in the experiments, improving the scalability of the portal service can significantly improve the scalability of the entire denormalized MediaWiki system. More scaling techniques can be applied, for example, caching the static files to reduce the non-wiki page requests on the portal servers, or even split a new web service from the portal service to handle all requests for static files.

7. Conclusion

In this project we have applied service-oriented data denormalization technique to MediaWiki. First we have denormalized MediaWiki into many independent data services, thereby developed the corresponding web services. Each web service has clear semantics and can be easily named, and be scaled by applying special techniques according to its own characteristics. Finally, the maximum sustainable throughput of the entire denormalized MediaWiki grows almost linearly with the number of used hosting servers based on the current experimental environment. As a result, this project has demonstrated the procedure of applying service-oriented data denormalization technique to MediaWiki, and proved the efficiency of the denormalization technique in improving the scalability of MediaWiki.

8. Reference

[1] B.Clifford Neuman. Scale in Distributed System. Readings in Distributed Computing Systems, IEEE Computer Society Press, 1994.

[2] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. Analysis of caching and replication strategies for web applications. IEEE Internet Computing, 11(1):60-66, January-February 2007.

[3] W. Zhou, D. Jun Jiang, G. Pierre, Chi-Hung Chi, and M. van Steen. Service-oriented data denormalization for scalable web applications. In Proceeding of the 17th international conference on World Wide Web, pages 267-276, 2008.

[5] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. IEEE Internet Computing, 6(5), September-October 2002.

[6] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands, 2006.

[7] M. Freedman, E. Freudenthal, and D. Mazi`eres. Democratizing content publication with Coral. In Proc. Symp. on Networked Systems Design and Implementation, pages 239 – 252, San Francisco, CA, USA, March 2004.

[8] M. Rabinovich and A. Aggarwal. RaDaR: a scalable architecture for a global web hosting service. In Proc. Intl. WWW Conf., May 1999.

[9] B. J. McKenzie, R. Harries, T. Bell. Selecting a Hashing Algorithm. SP&E 20(2):209-224, Feb 1990.

[10] Simple hash function. http://www.jonasjohn.de/snippets/php/simple-hash.htm

[11] G. Pierre and M. van Steen. Globule: a collaborative content delivery network. IEEE Communications Magazine, 44(8):127 – 133, August 2006.

[12] E. van Baaren, G. Pierre, G. Urdaneta. WikiBench: A distributed, Wikipedia based web application benchmark.

[13] http://www.mediawiki.org/wiki/ How_to_become _a_MediaWiki_hacker.

[14] JSON. http://www.php.net/manual/en/intro.json.php

[15] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proc. ICDE*, Mar. 2003.

[16] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. Data Engineering, 27(2):11-18, June 2004.

[17] K.Amiri,S.Park, R.Tewari, and S.Padmanabhan. Scalable template-based query containment checking for web semantic caches. In ICDE Conference.2003.

[18] E. Cecchet. C-JDBC: a middleware framework for database clustering. Data Engineering, 27(2):19-26, June 2004.

[19] W. D. Smith. TPC-W: Benchmarking an ecommerce solution. White paper, Transaction Processing Performance Council.

[20] C.J.Date. An Introduction to Database Systems (Seventh Edition). Addison-Wesley.

[21] L. Kazerouni and K. Karlapalem. Stepwise redesign of distributed relational databases. Technical Report HKUST-CS97-12, Hong Kong Univ. of Science and Technology, Dept. of Computer Science, Sept. 1997.

[22] http://en.wikipedia.org/wiki/Wikipedia

[23] G. Urdaneta, G. Pierre, M. van Steen. A decentralized wiki engine for collaborative Wikipedia hosting, in: Proceedings of the Third International Conference on Web Information Systems and Technologies (WEBIST), March 2007, pp. 156 – 163.

[24] http://www.mediawiki.org/wiki/Database_layout

[25] http://www.mediawiki.org/wiki/Manual:Database_access

[26] http://www.mediawiki.org/wiki/Manual:Cache#Localization_caching

[27] DAS3: The Distributed ASCI Supercomputer 3.http://www.cs.vu.nl/das3/

[28] http://www.mediawiki.org/wiki/Manual:MWDumper

9. Appendix

9.1. MediaWiki Database Layout

Category	Table Names	Comments
Page text and associated information	page	contains entries of all wiki pages and related metadata
	revision	stores metadata about the revision, and a reference to the text storage backend.
	text	holds text of individual page revisions
	archive	holds area for deleted articles, which may be viewed or restored by admins through the Special:Undelete interface.
	pagelinks	track page-to-page hyperlinks within the wiki
	templatelinks	track template inclusions
	externallinks	track links to external URLs
	category	track all existing categories
	categorylinks	track category inclusions
	langlinks	track interlanguage links
	recentchanges	contains some additional info on edits from the last few days
	redirect	contains exactly one row defining its target for each redirect
	page_restrictions	used for storing page restrictions
	protected_titles	contains nonexistent pages that have been protected
	page_props	contains Name/value pairs indexed by page_id
Search index	searchindex	used to provide full text searches
Images and Media	image	contains metadata of uploaded images and other files
	imagelinks	track links to images
	oldimage	contains metadata of previous revisions of uploaded files
	filearchive	contains record of deleted file data

Category	Table Names	Comments
User Accounts and Priviledges and Watchlist	user	contains basic account information, authentication keys, etc
	user_groups	maps the users in a particular MediaWiki installation to their corresponding user rights
	watchlist	stores sets of pages every user has selected to monitor for changes
	user_newtalk	stores notifications of user talk page changes, for the display of the "you have new messages" box
Blocks	ipblocks	used to block a vandal or troll account
Statics and Logging	site_stats	contains a single row with some aggregate info on the state of the site
	logging	stores logging information to be displayed on Special:Log
	updatelog	used to log updates, one text key row per update
	hitcounter	stores an ID for every time any article is visited
Caching tables	objectcache	used for a few generic cache operations
	math	used by the math module to keep track of previously-rendered items
	transcache	caches interwiki transclusion
	querycache	used for caching expensive grouped queries
	querycache_info	stores details of updates to cached special pages
	querycachetwo	used for caching expensive grouped queries that need two links
Miscellaneous	trackbacks	stores info for tracking back
	job	contains jobs performed by parallel apache threads or a command-line daemon
	interwiki	stores recognized interwiki link prefixes
	valid_tag	contains a list of defined tags, to be used by Special:Tags
	tag_summary	used to pull a LIST of tags simply without ugly GROUP_CONCAT that only works on MySQL 4.1+
	change_tag	tracks tags for revisions, logs and recent changes