Upgrades as a Service

Eetu Korhonen, eeko@iki.fi

February 16, 2012

http://eeko.iki.fi/

http://github.com/Eeko/mediawiki_uaas/

Abstract

This report examines several concepts of performing upgrades for real-time service applications without usage downtime by leveraging the possibilities of using on-demand computing resources provided by Infrastructure as a Service -providers. Common example of such application would be a public web-service requiring high availability for user, yet utilizing centrally maintained server architechture for operation. Within the study we produced a feature-limited prototype of a a tool intended for providing a live-upgrade from MediaWiki 1.4 to 1.5. Which originally required over 22 hours of write-locking to the system when it was applied into the English Wikipedia in 2005. In addition of building the tool, we explored a set of possible approaches to easen the availability problem during the upgrade and the innate challenges with implementing and using said approaches.

1 Introduction

This document is the end report for an independent research project performed for EURECOM[1] semester project done between July 2011 - January 2012. The purpose of the project was to research and demonstrate possibilities to leverage flexible cloud-infrastructure to provide online service updates with very little or no downtime for the end-user. The project was directed by Dr. Tudor Dumitras from Symantec Research Labs and supervised by Prof. Marc Dacier from EURECOM.

In the study, we first define the problem of performing service upgrades without downtime and introduce the concept of using on-demand computing resources for providing flexible capability to trade system downtime to temporary computing resources. We also examine the case of Wikipedia 1.4 to 1.5 upgrade in 2005, where the significant table-restructurings originally required a 22 hour write-lock in operations.

[1] http://www.eurecom.fr/

For this problem, we present a prototype for a software performing capable of tackling a small subproblem of the said upgrade without requiring a write-lock in the original system. Namely we present a program which translates an article upgrade in the old 1.4 schema to the new 1.5 schema running in a replicated computing environment. The implementation showed a number of limitations, such as the requirement of translating much of the application logic to a format understandable by the translator and the need for reasoning logic due to the limited information in the data sources.

After describing the software created, we examine some of the alternate approaches explored along the project and the issues encountered with them. Much of the focus with these approaches are beyond just upgrading an individual application (MediaWiki), but in viewing them in greater generalization and why they can be infeasible for larger, more comlex and less robust programs.

2 Problem

The high level issue under research is the possibility of performing system-updates in scale without affecting the availability of service to the end-user. With system updates changing the functionality of software, the usual case requires some unavailability period while modifications to the system are made. For this study, we reviewed the Wikipedia upgrade 1.5 from 2005, which required a 22 hour write lock due to a significant database-schema change requiring a re-write of the entire article database.[2]

There are two conventional methods to avoid availability breaks in a distributed system. One is to perform the upgrade as a switch-over, where the system is split in two halves. First one part of the system gets updated upgraded while the other part serves the clients. When the update is completed, the updated system is switched to be the client-serving end and the other part applies the update in turn. The second way is to perform the upgrade as a "rolling wave". Here the upgrade is applied to individual nodes of the distributed system in successive order. This allows for a greater accuracy in failure localization and reduces risks of failures as the entire system (or significant parts of it) do not get compromised for upgrade-errors.

However, neither of these approaches allow a downtimeless upgrade if the upgrade causes backwards incompatibilities. Any updates into the non-updated systems should reflect into the updated system as well. In this study, such incompatibility appears with the significant database-schema change of MediaWiki 1.5.

2.1 Leveraging elastic computing resources for updates

The examined method of avoiding the incompatibility issues with downtimeless upgrade is to use external computing resources to flexibly clone the existing service into a "parallel universe", where the upgrade can be applied without

[2] `http://meta.wikimedia.org/wiki/MediaWiki_1.5_upgrade`

touching the existing system providing service to the clients. When the upgrade is successfully applied to a machine cloned from a corresponding existing resource, the system performs some kind of catch-up with the changes inserted into the original client-serving machine and starts routing the client requests into itself. [2]

Modern cloud computing infrastructures provide us with a flexible platform for creating and utilizing external resources as needed. Applications running within Infrastructure as a Service (IaaS) providers such as Amazon EC2, Rackspace Cloud Servers or OpenNebula are by default running on virtualized hardware and are thus very easily replicable without large and permanent investments in big hardware.

2.2 MediaWiki 1.4 to 1.5 upgrade

The June 2005 update to MediaWiki 1.5 was primarily introduced to perform the schema change examined in this project. Few of the new features, such as logging the page-rename history and revised permalinking are dependant on the new schema. But those features do not interfere with online-upgrading, since the relevant tables can be re-generated separately from the existing database if needed.

In the 1.4 version of the schema, individual articles contained entries in two tables. The "cur"-table contained the most recent revisions of the articles in their entirety. The "old"-table contained the corresponding article history, usually listing up several related (old wikipage revisions) entries for a single article in cur-table. The entries in the tables are connected by the cur-table unique id of "cur_namespace + cur_title". A completely new article appears as a new insertion into the cur-table and a modification creates a new entry to the old-table where the contents of the previous cur-entry are copied before it gets updated.

In the 1.5 version, the tables are split into three tables. The new "revision" table is formed by combining the previous old- and cur tables and is intended to represent the relevant metadata for all article insertions and updates of the system. Such as the user who edited the page, the relevant timestamps and comments of the new update. A revision-entry is connected to a relevant entry in "text"-table, which contain the articles themselves. An entry in revisions also connect to an entry in the "page"-table, which represents an individual article with a revision-history in the revision-table. The most current article is stored in the "page_latest" pointer towards a singular entry in revision-table.

3 Solution

We built a small prototype of a software-stack capable of reading the MySQL query logs in real time from the system providing service to the end-user. Whenever it detects an update to the article-tables under update, it would create a translation of those queries compatible with the new schema. After the standard, non-modified 1.4 to 1.5 update is applied to the parallel universe clone of

Figure 1: Database Schema Changes in MediaWiki 1.5 [5]

the system, we use an external program hooking into the new database. This program uses the recorder modifications to mimic inserting article updates to the updated database. When the system under update has reached a synchronous state with the live-system, we can shut down the old systemand the upgrade programs and route all traffic to the updated system.

3.1 Implementation details

The original test-system under study is a small-sized[3] Amazon EC2-instance running a software stack[4] capable of running MediaWiki 1.4 with a custom test-database for a set of test articles.

[3] `http://aws.amazon.com/ec2/instance-types/`

[4] Amazon Linux 2011.2 with PHP 5.2, Apache 2 and MySQL 5.1

3.1.1 System replication tools

To automate the system replication procedure, we developed a series of bash-scripts leveraging the Amazon EC2-tools and knowledge of the details of the system under upgrade. Mainly we require the Amazon instance running details (instance number, hostname) and the application information (database name, host, username and password) for running the replication stack. The system is designed in a way, that we can use an external node with ssh- and EC2-tools access to the Amazon Instances to dowload the necessary programs from repository and start performing the upgrade process centrally, without touching the running instance providing service for the clients.

The main scripts to initiate the upgrading process are as follows:

- configs.conf – A sourcable configuration file to set the required environmental variables in the bash-scripts. Requires manual modifications to point to the EC2-node to be replicated and for the necessary database knowledge.
- prepare_for_cloning.sh[5] – Intended for installing a necessary stack of software to the node to be replicated. Such as Python 2.7 and mysql-python required by the updater software. Should also ensure that the required program-versions are available for the updating scripts at the locations specified in them.
- create_aws_replica.sh – Initiates the cloning process by copying the targeted node disk-image into a Amazon AMI (Requiring a brief shutdown of the said instance.) and starting a new identical instance with the said image. Creates a modified. configs.replica -file to include the necessary instance details of the replicated instance needed by the rest of the scripts.
- setup_replica.sh – Copies the necessary scripts and programs into the new instance.
- start_update.sh – Makes some necessary database-access modifications into the query-translating programs and launches an SSH-pipe into the original MediaWiki 1.4 node to stream the query log into a file to be readable by the local transaction-catchup programs. This is to be run in the new, replicated instance.
- std_update_mwiki14-15.sh – This script contains tools to download the newer MediaWiki 1.5 version and for running the standard upgrade-procedure to create a new copy of MediaWiki 1.5 running on top of the restructured database. This is to be run in the new, replicated instance and requires superuser access for the necessary Apache configuration and reboots.

3.1.2 Query parser, translator and mapper

The rest of the software is a series of python-programs used by update_mediawiki.py. Update_mediawiki.py requires the path to the file where the original systems

[5]Not implemented yet as of February 16, 2012.

query-log has been streamed as an argument and eventually writes all updates detected for existing articles into the new database schema within the parallel universe.

The program components are as follows:

- update_mediawiki.py – The entry-point of the program. Contains a main() method executing the translator-program from translator.py.
- translator.py – Contains the logic needed to use the query-log parser program (parser.py), how to interpret its returns and to translate them into SQL-queries writable by the database-hookup component. (mysql_connect.py)
- parser.py – Contains the logic required to detect and parse relevant INSERT and UPDATE queries from MySQL query-logs. The lines we wish to detect are the ones making article-updating modifications into an original MediaWiki 1.4 database.
- mysql_connect.py – Helper methods used to interact with a MySQL database.

Developing the software stack from scratch in its current form took from a single pre-intermediate programmer approximately 80 hours of research and development time. Though it is to be noted that most of the time was spent in getting familiar with the less understood portions of the toolset and the system undergoing the upgrade. Most of the time allocated for this project (totalling to approximately 250 hours) was spent exploring the problem and the other described approaches for it.

I estimate that doing a similar deployment with full feature set within similarly limited application could be done in less than 300 hours with distributed work and decent professionals. Much of the work required would be with testing against other use-cases and developing rules for the architecture to parse. Most of my proof-of-concept programming were about the necessary infrastructure in parsing relevant queries and working the communication between the systems. The individual translations and query-detection still needed can benefit from a robust infrastructure, reducing the amount of programming work in the end parts. Much of the labour could also substitute the work needed to do the regular upgrade. The standard upgrading schema could be made to utilize translation rules compatible with the real-time updating procedure.

3.2 Problems with the implementation

3.2.1 Case-specific

Our query-detector and translator approach mainly requires us to understand and re-implement large portions of the application logic within an external framework. This requires a significant amount of manual labor and would intuitively be more suitable to be integrated directly into the standard upgrading mechanisms instead of providing an external framework. More notably,

individual upgrade-instructions are not recyclable for other upgrades; neither can we leverage the existing SQL-upgrade instructions to automate the logic-programming.

3.2.2 Query-logs as a data source

Another issue are the limits of the data extractable from the query logs. Much of the details ending up to the database can be programmed and computed to be performed by the database itself without necessarily revealing them in the query logs E.g. generating entries via database-triggers and the auto-incrementations of id's are sometimes done within the database and can't be read from default query-logs. In the lower levels the database may perform optimizations or transaction aborts not necessarily visible to the logs. Or they can be insufficiently hard to predict and react for in large scale parsing. For adequate understanding of the workings of the database, the visible plain-text query-logs are likely insufficient. The approach would be more suited to be done by using the existing database replication infrastructures and binary-logs, which reliably reveal the internal database-actions in detail.

Though due to the elasticity of computing resources and low cost of upgrade-failures, failed upgrades can be tried again as often as needed. This opens us for possibilities of performing a less-refined probabilistic upgrade, where we only need a chance for individual upgrade to succeed and a number of computers performing the upgrade. After a unit passes tests for schema-equivalence, one can use standard database-replication suites to duplicate the relevant infrastructure to match the existing system.

3.2.3 Inefficiency and fault tolerance

The implementation looks into the updates as individual transaction one at a time, which is necessary as the program simulates a working application performing similar actions in a live use scenario. This is hardly efficient for larger data sets and is somewhat error prone; should there be unexpected modifications (such as manual inserts) to the database not detectable by the developed application.

Another way would be to use the query-logs to create records of data requiring action after a stage of upgrade has been completed. For example, two updates to the same article could be marked as a single entry to a table of "touched" article-id's. Then we make a external query to the original database to stream the necessary changes into the parallel universe. This does not free us from implementing some application logic, as actions such as deleting rows or modifying their unique id's would have to be represented in the tracking logic of tainted-entries. Neither is it granted that the query-logs available present us with enough data to identify the tainted items. For example, an INSERT-query might enter their unique id as NULL and auto-increment it in the database or application-logic. Such incrementation based on the MAX(ID)-value of the new flattened text-table of MediaWiki 1.5 was required in our implementation.

3.2.4 Limitations of virtualization technology

It is necessary to note that the current implementation does not manage fully without downtime, since creating an identical real-time duplicate in Amazon EC2 requires a downtime to make a copy of the image. However, it should be possible to make relevant replications in virtualized production environments, since equivalent copies of the database can be made without shutting down, by using the standard redundancy replication procedures provided by every major RDBMS.

4 Alternate approaches

During the course of the study, several other methods of performing the online-upgrade were speculated of and experimented with.

4.1 Using existing database-replication

One approach we experimented with was trying to leverage an existing database-reflection infrastructure. Namely, the GORDA[6][7] database replication architecture. GORDA is a set of extensions for a range of RDBMS's offering an external API-access and for the inner workings of the databases. Most common RDBMS's offer a number of reflective interfaces, such as the used query-logger and described binary-loggers – but they often fall short with certain atomicity features in distributed setups.[8] [1]

GORDA would help us by providing us the necessary infrastructure for reflecting the database interactions [9] and it would provide a programmable interface capable of instructing the parallel database.[10] This would all be done within the extended databases where the atomicity and reflection-reliability issues would be handled by GORDA. [4]

However, there proved to be a number of issues with this approach. First of all, the most mature implementation of GORDA and with MediaWiki was one hooking up into a PostgreSQL database. For MediaWiki 1.4, the PostgreSQL -support was considered to be "experimental" and it was recommended to use MySQL for production database. After the schema upgrade in 1.5, the PostgreSQL-support was officially discarded, though the unmodified components were still within the source code. Making the software run on PostgreSQL (or Apache Derby, which is "default" database for GORDA) requires a number of extra modifications.

Secondly, GORDA itself is still on a prototype stage. Orienting into it and extending it to fit the system at stake proved to be infeasible within the scope

[6] GORDA Open Replication of DAtabases

[7] `http://gorda.di.uminho.pt/`

[8] Such as the visibility of commit-order and capabilities to view and influence the client commits.

[9] Which was done by the query-parser of our implementation.

[10] Done in the translator and connector classes of our solution.

and skill level involved with the project. The most time-consuming issues being that the current prototype showed to lack support for a number of features[11] and necessary documentation. This project did not manage to examine the GORDA system enough to give estimates on the difficulty and time requirements for continuing with the approach. Some issues figured out could be circumvented by reconfiguring the application but at the stage of abandoning the approach[12] the risk probability and time required for additional unknown obstacles were considered too high.

4.2 Using similarity in application calls

Another idea to provide upgrades as a service for an ongoing database upgrade would be to move the upgrade-synchronization entirely away from the database-layer. If an upgrade touches only the underlying database layer and the application interface connecting to it, one could cache and re-route identical application calls to a parallel-universe backend replicating similar functionality within different schema. This kind of upgrade would naturally suit a typical 3-tiered web-application, where the user-transactions provided by web-server are separated from a dynamic content engine and data storages.

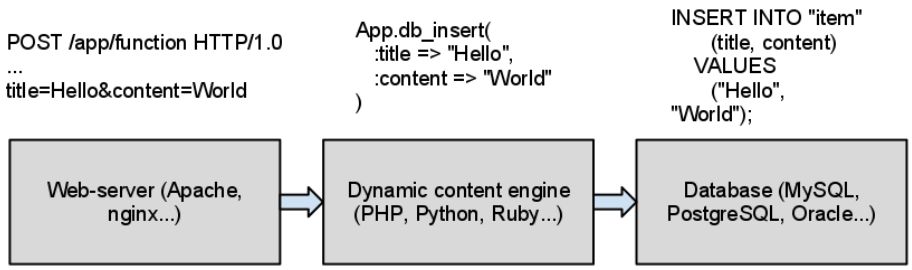

Figure 2: A typical 3-tiered web application

Should the interface to the frontend-engine stay identical, this kind of approach would provide a fairly easy and elegant solution for making a seamless transition to new software version without requiring any adaptations or query mappings between the systems. In the case of MediaWiki 1.4 to 1.5 upgrade, the main focus of the upgrade was this database-schema change. The upgrade could have been split into parts only affecting the application & database layers whilst keeping the user facing web server interface the same. Should an upgrade provide any UI modifications, those can be made in an another upgrade-package keeping the database untouched.

[11] Such as namespace support in table naming.
[12] After approximately 120 hours of development.

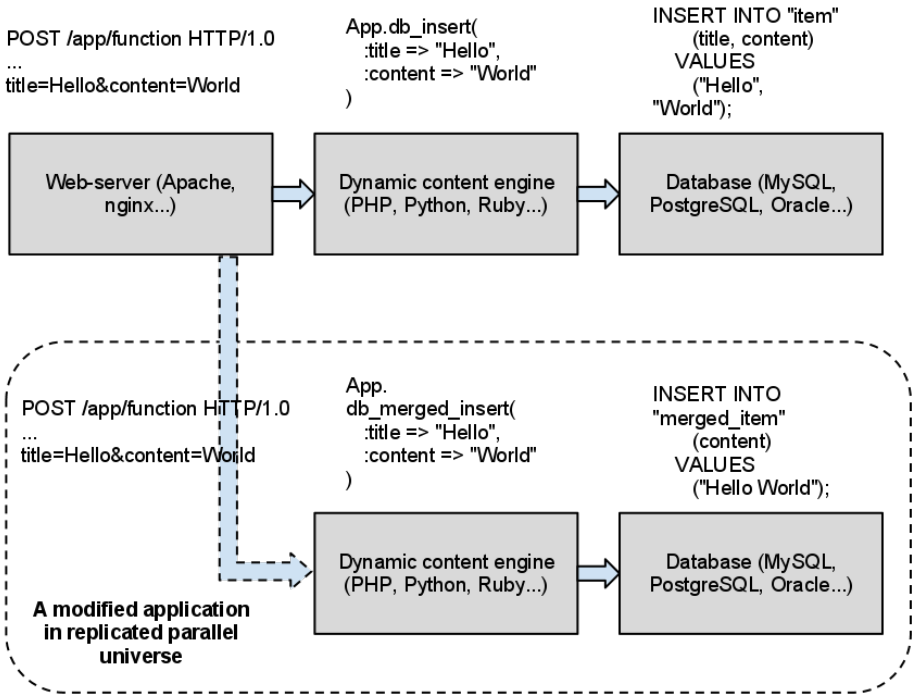

Figure 3: A 3-tiered web application routing interface-queries to original and upgraded backends

Not every kind of software update would work with this approach. Several upgrades implement new functionality with a need for wide modifications to every layer of the software. Though often such upgrades can be split into incremental parts where the heavy database-modifying and writing operations can be done before the corresponding modifications are introduced to the frontend. As this requires additional engineering-consideration for upgradeability, it might not be feasible to provide upgrading as an external service for application-upgrades modifying the entire stack.

Other issue would be the lack of guarantees for consistency in the entire stack. Should there be failures in the middle- or database tier of the live-system, a dumb frontend-replicator would not detect them nor present adequate information to account the inconsistency in the parallel universe. Such issues could be coped with most standard redundancy & reliability techniques utilized in the system. The redundancy technology can be made to monitor and require confirmation of commits from the backend or from redundancy replication interfaces. Or with cheap replicable virtual hardware, we could settle with eventual consistency where we just restart an upgrade-procedure until the backend passes sufficient tests of equivalence.

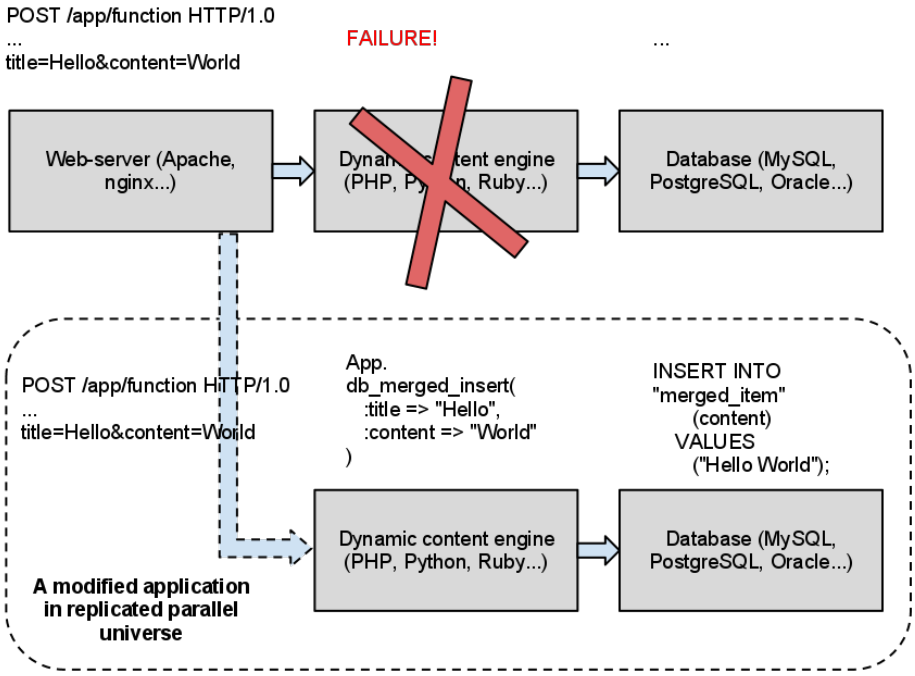

Figure 4: A faulty parallel-run leaving the two systems in inequivalent states

4.3 Using the existing upgrade tools to decreasing database subsets

One of the more intriguing approaches would be to create a framework to be able to read the existing schema-upgrade scripts available and deduct the upgrade logic and resulting table from those. Since the target tables are expanding leading to increasing time-requirements of re-applying the updates to complete tables, we need to be able to divide the work into smaller subsets as new items and updates get inserted during the online-upgrade.

An intuitive way for such division would be to split the dataset under update by their timestamps, so that we only re-run the standard upgrade script for new items inserted after the last known item in the databases under upgrade was received. However, under some upgrades this will provide and incompatible and possibly broken consistency due to the unpredictability of the live-system updates.

MediaWiki 1.4 used two separate tables to represent articles (cur-table) and their revision history (old-table). The tables contain mostly similar information allowing the history to be formed by mostly copying the old contents of cur-table to be archived in old-table. The upgrade in MediaWiki 1.5 system combines these and splits them into three tables containing different logical fractions of the same data.

In our example, we have generalized the first combining operation into a higher abstraction of state-entries (“cur”, as in MediaWiki 1.4) and history-entries

("old", as in MediaWiki 1.4). Every history entry has a foreign key pointer to a corresponding current state. Performing incremental joins within this kind of database leads into an inconsistent state compared to a singular batch operation done with similar joins.

cur_id	cur_content
1	1_Fourth_state
2	2_Second_state

Table 1: Current states -table (cur-table)

old_id	old_content	old_link_to_cur
1	1_First_state	1
2	1_Second_state	1
3	2_First_state	2
4	1_Third_state	1

Table 2: Old states history -table (old-table)

Suppose, that a schema upgrade would flatten these said tables into one table containing both the current state of the items and the given history of said items. An upgrade would be done with the following SQL-code:

```
-- Note, that the primary id's of the table are auto-incremented.
-- This is similar to how Mediawiki 1.4 to 1.5 upgrade handles the
-- database-flatten operation.
INSERT INTO "old" (old_content, old_link_to_cur)
        SELECT (old_content, old_id)
        FROM "old";
```

After the update, the new table would look like this:

old_id	old_content	old_link_to_cur
1	1_First_state	1
2	1_Second_state	1
3	2_First_state	2
4	1_Third_state	1
5	1_Fourth_state	1
6	2_Second_state	2

Table 3: Merged old-table

However, suppose that we receive a third state to the original-system during the time taken by the upgrade of the system and it receives several updates for it. We have sufficient translation logic in place to only apply the INSERT-queries for items entered to the database after we begun merging our previous entries. The code would work something like this:

```
-- We first apply the modifications from old-table
INSERT INTO "old" (old_content, old_link_to_cur)
        SELECT (old_content, cur_id)
        FROM "olddb.old"
        WHERE timestamp > last_update_time;

-- Then we flatten the items from the table of current states
INSERT INTO "old" (old_content, old_link_to_cur)
        SELECT (old_content, old_link_to_cur)
        FROM "olddb.cur"
        WHERE timestamp > last_update_time;
```

Then when we would receive the following rows into the database:

old_id	old_content	old_link_to_cur
5	3_First_state	3

Table 4: New row in old-table inserted during the update

cur_id	cur_content
3	3_Second_state

Table 5: New row in cur-table inserted during the update

We would end up with a merged table looking like the following:

old_id	old_content	old_link_to_cur
1	1_First_state	1
2	1_Second_state	1
3	2_First_state	2
4	1_Third_state	1
5	1_Fourth_state	1
6	2_Second_state	2
7	3_First_state	3
8	3_Second_state	3

Table 6: Merged old-table after incremental upgrade

If the upgrade would have been done with write-locks and the two tables would be merged after all 8 commits were received in the same sequential order as in our online-example, the resulting table would look like this:

old_id	old_content	old_link_to_cur
1	1_First_state	1
2	1_Second_state	1
3	2_First_state	2
4	1_Third_state	1
5	3_First_state	3
6	1_Fourth_state	1
7	2_Second_state	2
8	3_Second_state	3

Table 7: Merged-old table without online-upgrading

The id's for items in the different upgrade-approaches are not equivalent. If it is used as an external-id in somewhere else in database-logic without the necessary modifications, we will encounter in faulty behaviour. To fix this, we need to implement some amount of application- or database-logic to upgrade corresponding tables with history_id references.

This would likely not be a problem within the MediaWiki, since the id's are supposedly used only for separating primary id's from each other and the supposed order carries no significance. However, inconsistency of incrementation might be an issue in some programs. Such as ones requiring predictable consistency with FETCH FIRST [N] -queries[3] for unordered data or non-standard SQL such as TOP or LIMIT -syntaxes.

5 Future work

Although the approach with GORDA proved to be unfeasible in this scope, different methods to leverage the reflections from database-internals to provide upgrades as a service are still to be examined. Other possibilities to get sufficient data could be to hook up into existing database replication protocols or into the binary-logs used by the said replication protocols.

And even if the other introduced methods to perform upgrades externally in replicated environments are not generalizible, we have still to examine whether they might be sufficient for individual cases such as the MediaWiki upgrade presented. Especially given the simplicity of rerouting high-level application calls for replicated cloud-servers, designing upgrades to support it could prove to be a decent engineering practice for online-services.

Though this study does not help much with re-trying the GORDA-route, one could examine the costs of extending the database replication protocols to suit dynamic modifications to the queries and for presenting more detailed views on the interactions to support an upgrade-compatible replication procedure.

6 Conclusions

The implementation provided falls short with the initial goal of performing an online upgrade in a scale comparable to a real-life use scenario. (I.e. the entire

Wikipedia and updates performed to it during the upgrade.) Though we manage to explore illustrate several approaches for the task and the challenges involved. Mostly relating with the need to model much of the application logic within the mapper and the unreliabliness of the used data-source. For which the former would be solvable by being able to utilize the upgrading schema in a suitable framework and applications and for the latter

In addition, we have examined a number of other approaches capable of providing a non-downtime upgrade-procedure in a replicated system and shown via contradiction, why some of them would be inadequate for selected, common database-modifying procedures. Though used as an initial approach for the upgrade, we were unsuccessful in utilizing the most promising method of leveraging database-reflection interfaces.

References

[1] Nuno Carvalho, José Pereira, Rui Oliveira, Luís Rodrigues, and Susana Guedes. On the use of a reflective architecture to augment database management systems.

[2] Tudor Dumitras and Priya Narasimhan. Toward upgrades-as-a-service in distributed systems. In Fred Douglis, editor, *Middleware (Companion)*, page 29. ACM, 2009.

[3] ISO/IEC 907-3:2008 – Information technology – Database languages – SQL – Part 3: Call-Level Interface (SQL/CLI), 2008.

[4] Alfrânio Correia Jr., José Pereira, Luís Rodrigues, Nuno Carvalho, Ricardo Vilaça, Rui Carlos Oliveira, Susana Guedes, and Susana Guedes. Gorda: An open architecture for database replication. In *NCA*, pages 287–290, 2007.

[5] MediaWiki. Proposed database schema changes/october 2004 — mediawiki, the free wiki engine, 2007. [Online; accessed 2-February-2012].