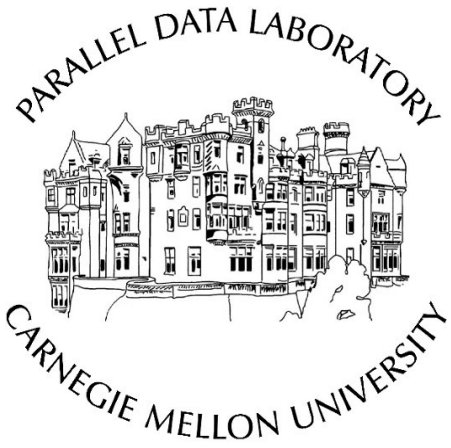
7-2009

No Downtime for Data Conversions: Rethinking Hot Upgrades (CMU-PDL-09-106)

Tudor Dumitraş
Carnegie Mellon University, tudor@cmu.edu

Priya Narasimhan
Carnegie Mellon University, priya@cs.cmu.edu



No Downtime for Data Conversions: Rethinking Hot Upgrades

Tudor Dumitraş and Priya Narasimhan
Carnegie Mellon University
Pittsburgh, PA 15217
tudor@cmu.edu priya@cs.cmu.edu

CMU-PDL-09-106
July 2009

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Unavailability in enterprise systems is usually the result of planned events, such as upgrades, rather than failures. Major system upgrades entail complex data conversions that are difficult to perform on the fly, in the face of live workloads. Minimizing the downtime imposed by such conversions is a time-intensive and error-prone manual process. We present Imago, a system that aims to simplify the upgrade process, and we show that it can eliminate all the causes of planned downtime recorded during the upgrade history of one of the ten most popular websites. Building on the lessons learned from past research on live upgrades in middleware systems, Imago trades off a need for additional storage resources for the ability to perform end-to-end, enterprise upgrades online, with minimal application-specific knowledge.

Acknowledgements: We would like to thank Alan Downing, Jim Stamos and Byron Wang of Oracle for their feedback during the early stage of this research project.

1 Introduction

Software upgrades are unavoidable in enterprise systems. For example, business reasons sometimes mandate switching vendors; responding to customer expectations and conforming with government regulations can require new functionality. Moreover, many enterprises can no longer afford to incur the high cost of downtime [14] and must perform such upgrades online, without stopping their systems. While fault-tolerance mechanisms focus almost entirely on responding to, avoiding, or tolerating unexpected faults or security violations, system unavailability is usually the result of planned events, such as upgrades. A 1998 survey of 426 sites with high-availability applications, using servers from IBM, Sun, Compaq, HP, DEC, and Tandem, showed that 75% of nearly 6000 outages were due to planned hardware and software maintenance and that such planned outages typically lasted twice as long as unplanned ones [12].

Enterprise-system upgrades often require complex data conversions for changing the data schema or for migrating to a different data store. Such upgrades are costly; for example, upgrading enterprise resource planning (ERP) systems can cost up to 30% of the price of the original implementation ($40M – $240M) [3]. Because some data conversions are difficult to perform on the fly, in the face of live workloads, and owing to concerns about overloading the production system, *upgrades that involve computationally-intensive data conversions currently necessitate planned downtime*, ranging from tens of hours to several days. Conversely, system administrators often avoid complex upgrades, which might impose an unacceptable downtime, and preserve database schemas that provide suboptimal performance and that cannot support new, user-requested features. Despite previous work on schema evolution, determining the leading causes of planned downtime in enterprise systems remains an open question.

There is a tension between the upgrade atomicity and the system availability during the upgrade. In practice, enterprise-system administrators often favor the atomicity, by upgrading inter-dependent components and data objects together, and introduce planned downtime [8]. Previous approaches for online upgrades [5, 11] favor the availability and introduce mixed, interacting versions, which synchronize their states in the back-end. Ensuring the correctness of the mixed-version interactions requires a time-consuming and error-prone manual process. Both these approaches assume that an upgrade must be performed in-place, replacing the existing system. We show that, at the expense of additional storage and computational resources, the tension between upgrade atomicity and availability can be resolved.

We believe that an upgrade mechanism that requires careful manual intervention and in-depth knowledge of the application's interaction with the data store is much less likely to become successful. We especially wish to avoid the need to establish correctness constraints for the mixed-version interactions or to track the old version's data dependencies. Therefore, a meta-goal for our research is to assess the effort required to implement and coordinate a complex online-upgrade and to evaluate how close we can get to the zero-downtime ideal.

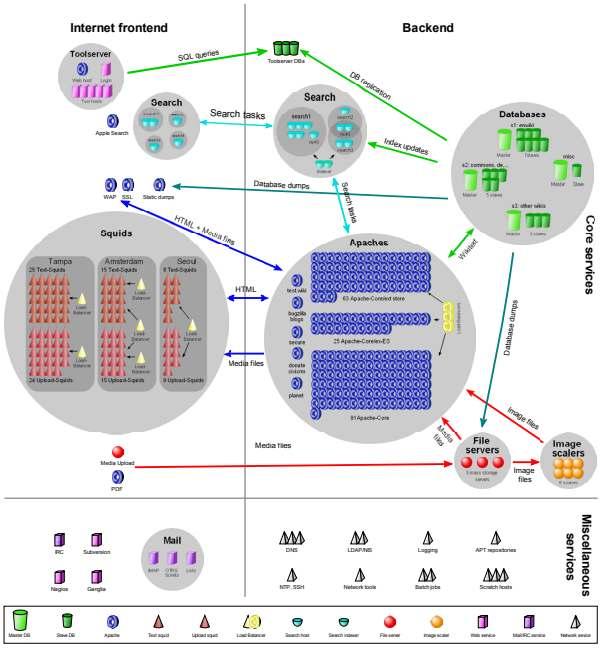
We have two goals in this paper:

- Identify the common causes of planned downtime from the upgrade history of Wikipedia, one of the ten most popular websites to date;
- Rethink online upgrades in order to eliminate the leading causes of planned downtime and to reduce the effort required for ensuring that the persistent data will be correctly converted into the new format.

2 Causes of planned downtime

Wikipedia (`http://www.wikipedia.org`) is one of the ten most popular websites to date,[1] providing a multi-language, free encyclopedia. The English-language Wikipedia has 2.9 million articles, with content stored in a 1 TB database and 850,000 files (*e.g.* images). The web site is supported by an infrastructure (Figure 1) running on 352 servers, including 23 MySQL database servers configured for master/slave replication. The master database receives the write queries and propagates the updates to the slave, which handle read-only queries. A wiki

[1] According to `http://www.alexa.com`. Wikipedia handles peak loads of 70,000 HTTP requests/s.



Source: `http://meta.wikimedia.org/wiki/Wikimedia_servers`

Figure 1: Wikipedia architecture.

engine called MediaWiki, implemented as a set of PHP scripts, accesses the database and generates the content of the articles. In addition to Wikipedia, MediaWiki drives over 15,000 other wikis,[2] which makes it a representative case study for enterprise-system upgrades.

We study the upgrade history of Wikipedia by combining data from a rigorous study of Wikipedia's schema evolution [7] with information from design documents and archived discussions. Between 2003 and 2008, MediaWiki's database schema has gone through 171 evolution steps [7] in the main development branch. During this time, the project has released eleven versions (1.1 – 1.11) of the wiki engine. Minor versions (*e.g.*, the 1.4.x series) do not introduce schema changes; upgrading to a new version within the same release series requires only replacing the PHP code on the application servers. However, upgrading to a different major version (*e.g.* from 1.4 to 1.5) can require database changes.

Example of planned downtime. Figure 2 illustrates a simple schema upgrade proposed for MediaWiki 1.4 that was rejected because it would have imposed downtime on upgrade (we adapted this example to simplify its presentation). After ① dropping column `a` from table `X`, some queries issued by the old MediaWiki version will produce SQL errors. Furthermore, if ② the schema upgrade also adds a column `X.b`, the new MediaWiki version will likely be unable to operate on the old schema. To prevent exposing the clients to these errors, the upgrade should be performed in two steps: (i) add column `X.b` and upgrade the wiki engine, in an atomic operation; (ii) drop column `X.b`. During the first step, the clients must not access the system to avoid producing errors.

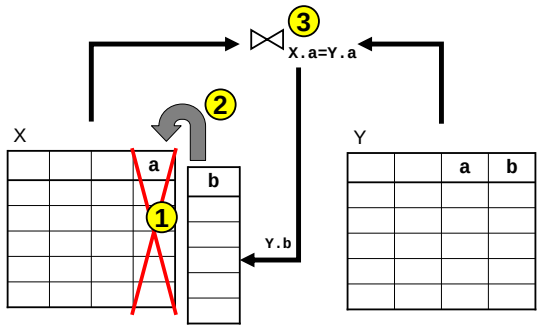


Figure 2: Replacing a column ① with another one ②, initialized with data from a different table ③.

Instead of atomically upgrading the schema and the business logic, several approaches have been proposed [5, 11] for allowing mixed versions to coexist during the upgrade. However, dependencies among the old and new data further complicate the upgrade procedure. For example, ③ let the new column `X.b` be initialized with the contents of a column from a different table, `Y.b`, selected by joining tables `X` and `Y` on the column `a`. The offline schema upgrade uses three SQL commands:

```
ALTER TABLE X ADD COLUMN b INT(8) NOT NULL;
UPDATE X,Y SET X.b=Y.b WHERE X.a=Y.a;
ALTER TABLE X DROP COLUMN a;
```

However, in an online upgrade, the clients can access the system during these operations. The online-upgrade procedure must take into account the effect of the `INSERT/UPDATE/DELETE` queries issued by the live workload and must synchronize the data in the new column, `X.b`. This requires reevaluating the join condition `X.a=Y.a` for each live query to determine if the value of `X.b` must be updated, in the row changed by the query or in some other row of

[2] According to `http://s23.org/wikistats/`.

table X. This example illustrates that synchronizing multiple versions during an online upgrade requires additional development effort, is error-prone due to its complexity and can impose a high run-time overhead on the production system.

Figure 3 illustrates the most complex schema change, introduced in MediaWiki 1.5. Prior to this version, the cur table stored the content and metadata of the current revisions of Wikipedia articles, and table old stored the previous revisions. The 1.5 upgrade moved the article-specific metadata into the page table, the revision-specific metadata into the revision table and the content of the revisions into the text table. The goals of this major restructuring were to improve performance (*e.g.*, by separating metadata from content to allow faster aggregation) and to support new features (*e.g.* renaming articles without having to modify all their past revisions). This change was implemented by five developers over a period of one year. During the upgrade, Wikipedia was locked for editing, and the schema was converted to the new version in about 22 hours [1].

Conversely, complex database changes have often been avoided because they might impose downtime. For instance, the MediaWiki developers have repeatedly rejected requests for features that would require major schema reorganizations [2], and have rolled back schema changes that would impose downtime on upgrade (*e.g.*, the change described in Figure 2). Such rollbacks account for 8.8% of the changes from MediaWiki's schema evolution [7].

Current upgrade approaches. With a single database server in the back-end, downtime is hard to avoid during an upgrade. In some situations, it is possible to allow read-only access concurrently with the upgrade procedure. MediaWiki provides a configuration parameter ($wgReadOnly) that places a site in read-only mode for such planned-maintenance activities.

When the wiki uses a replicated database, it is possible to avoid downtime through a *rolling upgrade* [4, 13]. In Wikipedia, a rolling upgrade removes slave nodes one-by-one from replication group, applies the schema changes, and then restarts the replication [2]. The rolling upgrade swaps database masters before completing the schema upgrade, to avoid re-applying the changes through the replication protocol. The application servers are upgraded in a similar fashion, in a wave rolling through the data center.

To support rolling upgrades, the database replication mechanism must allow source and target tables that do not have identical definitions. In MySQL, for instance, a table on the master can have more or fewer columns than the slave's copy, or the corresponding columns on the master and the slave can use different data types. In general, rolling upgrades require the upgrade to be backwards-compatible [4], excluding changes such as the one from Figure 2.

Index and data-type changes. Index changes, implemented for performance-tuning purposes, account for 40.3% of the schema changes in MediaWiki [7]. While some commercial database servers support online index-definition, MySQL locks a table in order to rebuild its index. These common changes impose downtime for wikis with a single database node. In Wikipedia, however, such changes are performed online, through a rolling-upgrade, because they do not require application modifications. Moreover, rolling upgrades allow simple data-type conversions (12.8% of schema changes [7]), such as increasing the size of a numeric type (*e.g.*, INT(8) $\mapsto$ INT(16)).

Schema changes. Curino *et al.* show that the 171 evolution steps of MediaWiki's database schema can be modeled using 11 schema-modification operators (SMOs) [7]. These SMOs, defined in Table 1, specify how the upgrade converts the database to a new schema. Four SMOs, DROP TABLE, RENAME TABLE, MERGE TABLE, RENAME COLUMN, create new schemas that prevent restarting the MySQL replication during a rolling upgrade, and, therefore, impose downtime. Five additional SMOs cannot be supported during a rolling upgrade because (i) the old application would be unable to query the new schema (DROP COLUMN, MOVE COLUMN); (ii) UPDATE queries would attempt to access rows that do not exist anymore (DISTRIBUTE TABLE); or (iii) data dependencies would be broken because changes would be applied only to the source column (COPY COLUMN, MOVE COLUMN). The most common operator, ADD COLUMN, is usually compatible with rolling upgrades because it inserts constant values into

Database-schema definition

Old schema

```
CREATE TABLE cur (
* cur_id INT(8) UNSIGNED
*   NOT NULL auto_increment,
* cur_namespace TINYINT(2) UNSIGNED
*   NOT NULL default '0',
* cur_title VARCHAR(255) binary
*   NOT NULL default '',
‡ cur_text MEDIUMTEXT NOT NULL default '',
† cur_comment TINYBLOB NOT NULL default '',
† cur_user INT(5) UNSIGNED
†   NOT NULL default '0',
...
  PRIMARY KEY cur_id (cur_id)
);

CREATE TABLE old (
  old_id INT(8) UNSIGNED
    NOT NULL auto_increment,
* old_namespace TINYINT(2) UNSIGNED
*   NOT NULL default '0',
* old_title VARCHAR(255) binary
*   NOT NULL default '',
‡ old_text MEDIUMTEXT NOT NULL default '',
† old_comment TINYBLOB NOT NULL default '',
† old_user INT(5) UNSIGNED
†   NOT NULL default '0',
...
  PRIMARY KEY old_id (old_id)
);
```

New schema

```
CREATE TABLE page (
* page_id INT(8) UNSIGNED
*   NOT NULL auto_increment,
* page_namespace TINYINT NOT NULL,
* page_title VARCHAR(255) binary NOT NULL,
  page_latest INT(8) UNSIGNED NOT NULL,
...
  PRIMARY KEY page_id (page_id)
);

CREATE TABLE revision (
  rev_id INT(8) UNSIGNED
    NOT NULL auto_increment,
† rev_page INT(8) UNSIGNED NOT NULL,
† rev_comment TINYBLOB NOT NULL default '',
† rev_user INT(5) UNSIGNED
†   NOT NULL default '0',
...
  PRIMARY KEY rev_page_id (rev_page,
                           rev_id)
);

CREATE TABLE text (
  old_id INT(8) UNSIGNED
    NOT NULL auto_increment,
‡ old_text MEDIUMTEXT NOT NULL default '',
...
  PRIMARY KEY old_id (old_id)
);
```

Upgrade pseudocode

DO_SCHEMA_RESTRUCTURING

1 Check duplicate <title, namespace> entries in cur and remove all but the most recent ones
2 Create page and revision tables
3 Lock page, revision, old, cur tables for writing
4 *maxold* ← MAX(old.old_id)
5 Insert contents of cur, other than cur_id, into old (old_id is auto-incremented)
6 Insert cur $\bowtie_{title,\ namespace}$ old into revision († fields from old; rev_id ← old_id; rev_page ← cur_id)
7 Insert cur $\bowtie_{cur\_id=rev\_page,\ rev\_id>maxold}$ revision (*i.e.* the rows that originated from cur) into page (* fields from cur; page_latest ← rev_id)
8 Unlock tables
9 Rename table old to text

* columns moved to the page table; † columns moved to the revision table; ‡ columns moved to the text table.

Figure 3: MediaWiki schema restructuring from version 1.5.

the new column. However, in a few cases, ADD COLUMN adds data dependencies, by inserting values based on other columns from the same table, or creates columns with values incremented automatically, which might not produce the same ordering on the master and the slave.

In a sequence of schema modifications (*e.g.*, to describe a MediaWiki upgrade from V1.4 to V1.5), a SMO can cancel the effects of a previous one; for instance, if CREATE TABLE X precedes DROP TABLE X, these changes do not impose downtime during the upgrade. While Wikipedia always uses the most recent MediaWiki version and has deployed all the past versions of the wiki engine sequentially, in practice software upgrades often skip versions. We consider the SMO sequences that define all the possible upgrades among MediaWiki versions V1.1 – V1.11 (we do not consider downgrades, which would require the inverse operations).

Figure 4 shows the likely outcome of these upgrades. 38 out of the 55 upgrades would introduce changes that prevent restarting the MySQL replication, and in 12 additional cases the changes would prevent a rolling upgrade. These upgrades impose

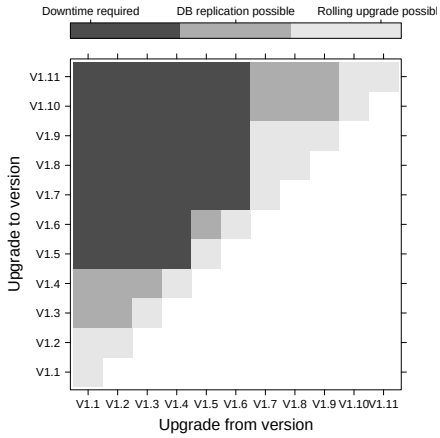


Figure 4: Downtime imposed by MediaWiki upgrades.

downtime. Only 5 MediaWiki upgrades can be performed online, through a rolling upgrade. These situations typically correspond to upgrades between subsequent versions of MediaWiki. However, the major changes introduced in versions 1.5 and 1.7 are incompatible with the database replication. Additionally, versions 1.4 and 1.10 introduced auto-incremented columns, version 1.3 dropped a column, and version 1.6 added a column with depen-

Table 1: Schema Modification Operators (SMOs).

SMO (%)	Description	Rep	RU
`CREATE TABLE` (4.9%)	Creates new, empty table.	Y	Y
`DROP TABLE` (1.8%)	Remove existing table.	N	N
`RENAME TABLE` (0.6%)	Rename table, without affecting the data.	N	N
`DISTRIBUTE TABLE` (0%)	Distribute rows of a source table into two new tables, according to a condition.	Y	N
`MERGE TABLE` (0.8%)	Create a new table as the union of two tables with the same schema.	N	N
`COPY TABLE` (1.2%)	Duplicate existing table.	Y	Y
`ADD COLUMN` (21.2%)	Add column and populate it with values generated by a constant or a function.	Y	Y/N
`DROP COLUMN` (14.5%)	Remove existing column.	Y	N
`RENAME COLUMN` (8.8%)	Change column name, without affecting the data.	N	N
`COPY COLUMN` (0.8%)	Copy column into another table, according to a join condition.	Y	N
`MOVE COLUMN` (0.2%)	`COPY COLUMN`, then drop the original.	Y	N

Rep = Supported by MySQL replication (Yes/No).
RU = Supported during a rolling upgrade (Yes/No).

dencies on other columns.

Data conversions. In MediaWiki's version history, one upgrade has required converting the text from all the article revisions recorded in the database. Starting from version 1.5, MediaWiki supports only the UTF-8 character set, and older wikis using Latin-1 were required to convert their data to the new encoding. This is a long-running operation, which competes with the live workload for querying and modifying the database and which can impose a significant performance overhead.

Competitive upgrades. Sometimes, instead of switching to a newer version, the upgrade aims to replace the existing system with a completely different one, which provides similar or equivalent functionality. These upgrades occur when an enterprise changes vendors, and they usually impose downtime because of incompatibilities between the two systems. Wikipedia has performed two such competitive upgrades: when it switched from UseModWiki to a custom-built wiki engine, remembered as "Phase II," and when this code base was rewritten and became MediaWiki.

Summary. We summarize our findings as follows:

- Database indexes cannot be redefined online in some open-source databases, but this common change is supported through rolling upgrades;
- Incompatible schema changes (*e.g.*, renaming tables in the database) prevent rolling upgrades and require upgrading the schema and the application in an atomic step;
- Data dependencies (*e.g.*, resulting from table joins) are hard to synchronize in response to updates issued by the live workload and can impose a high runtime overhead;
- Long-running data conversions compete with the live workload and might overload the database;
- Competitive upgrades require data conversions and typically impose downtime.

3 Online upgrades with Imago

We present Imago (Figure 5), a system for performing online upgrades with complex data conversions. Imago installs the new version in a parallel universe $\mathbf{U_{new}}$– a logically distinct collection of resources, realized either using different hardware or through virtualization.

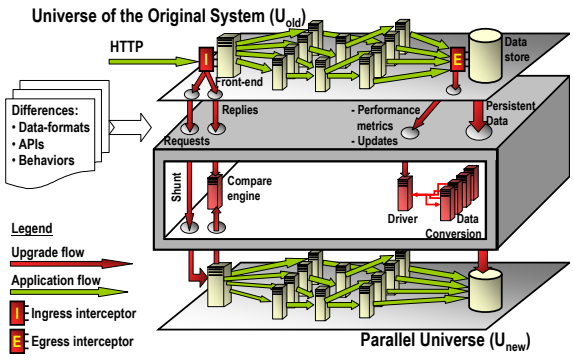


Figure 5: Online upgrades with Imago.

Imago uses a procedure with five phases: bootstrapping, data-transfer, termination, testing, and switchover [10]. Imago opportunistically transfers the persistent data from the production system in $\mathbf{U_{old}}$ to the new version in $\mathbf{U_{new}}$, performs data conversions, monitors the updates reaching $\mathbf{U_{old}}$'s data store, and identifies which objects it must re-transfer in order to prevent data-staleness. The live workload accesses $\mathbf{U_{old}}$'s data store concurrently with the data-transfer process. To avoid locking objects, Imago queries $\mathbf{U_{old}}$'s data store under snapshot isolation. Imago monitors $\mathbf{U_{old}}$'s data-store activity to ensure that all of the updated or new data objects are eventually (re)-transferred to $\mathbf{U_{new}}$. Moreover, Imago monitors the load and the performance of $\mathbf{U_{old}}$'s data store and regulates its data-transfer rate to avoid disrupting the live workload.

The upgrade will eventually terminate if the transfer rate exceeds the rate at which $\mathbf{U_{old}}$'s data is modified (this is easily achieved for read-mostly workloads). Imago also supports a series of iterative testing phases, which do not interfere with the production system from $\mathbf{U_{old}}$. After adequate testing, the upgrade can be rolled back, by simply discarding the $\mathbf{U_{new}}$ universe, or committed, by making $\mathbf{U_{new}}$ the production system, completing the atomic switchover to the new version. By reducing the risk of breaking hidden dependencies, Imago reduces the unplanned downtime due to major upgrades [9].

Data conversions supported. Imago also reduces the planned downtime by performing an online upgrade in the presence of data conversions. Unlike a rolling upgrade, Imago does not upgrade the system in place. Because the application never interacts with a data schema belonging to a different version, Imago trivially supports the `DROP TABLE`, `RENAME TABLE`, `DROP COLUMN`, and `RENAME COLUMN` schema changes. Moreover, the live workload accesses either the old or the new version, but not both, which simplifies the data conversion. Figure 6 shows how Imago handles the other schema changes that commonly impose downtime. For new auto-incremented columns, Imago assigns the new value during the data transfer. For new columns initialized with a function based on other columns from the same table, Imago monitors the updates to the table and applies the updates correctly, to the source and destination columns.

```
DISTRIBUTE TABLE R INTO S with condition, T
1  if condition
2     then Apply change to S
3     else  Apply change to T
MERGE TABLE S,T INTO R
1  Apply update to new table R
ADD COLUMN C auto-increment INTO R
1  Assign auto-incremented value during data transfer
ADD COLUMN C FUNC(A) INTO R
1  if A is updated
2     then C ← FUNC(A)
COPY COLUMN C FROM R INTO S WHERE join-cond
1  if transfer of R and S is complete
2     then Compute R ⋈join-cond S at the destination
3     else  Save the update and apply it later
```

Figure 6: SMO handlers in Imago.

`COPY COLUMN` is a more complex transformation because it joins two tables to determine which values are copied into the new column. If the transfer of tables R and S from $\mathbf{U_{old}}$ is ongoing, Imago saves the stream of updates for applying it later; otherwise, Imago applies the update and re-evaluates the set of values that must be copied into the new column. We implement these schema transformations using the GORDA API [6], which provides a uniform reflective interface for several database servers. For instance, we monitor the data updates performed by the live workload by retrieving the object-sets from the executor stage of $\mathbf{U_{old}}$'s database.

Similarly, Imago can perform computationally-intensive data transformations during the online upgrade, because these operations do not interfere with the production data store. These facilities can also be exploited for performing competitive upgrades. In the future, we plan to reduce Imago's spatial overhead (additional hardware and storage requirements) through virtualization. Additional storage space and compute cycles could be hired, for the duration of the upgrade, from existing cloud-computing infrastructures (*e.g.* Amazon's EC2). This suggests that Imago is the first step toward an upgrade-as-a-service model, making complex upgrades practical for a wide range of enterprise systems.

4 Discussion

Imago builds upon (and, in some sense, goes beyond) our observations and experiences with a previous online-upgrade approach, Eternal [11], nearly a decade ago.

Eternal was developed first as a transparent replication middleware for CORBA and Java applications. Eternal used this infrastructure for providing online upgrades, leveraging the presence of replicas to upgrade a component, one replica at a time, without disrupting its service or its availability to the clients. A pre-processor front-end automatically parsed the old and the new versions of each component's source-code, and then created an intermediate version that supported both the old and the new interfaces of the the component. This intermediate version acted as a staging area, where the old and the new versions could co-exist and where state could be transferred between them, in the presence of a live workload.

As automated as the pre-processor could be, there were some inevitable hurdles that required careful manual intervention. Eternal orchestrated an atomic upgrade of the old version's replicas, along with all their dependencies. However, some of these dependencies only manifested dynamically, or at runtime. In such cases, the pre-processor was unable to proceed, without manual assistance (*e.g.*, an in-depth pointer analysis), to understand the nature and depth of the dependency chain. Recent commercial products for rolling upgrades continue to exhibit a similar problem, by requiring the application developers to determine if the interactions among mixed versions are safe [13].

Imago attempts to take the goal of transparency even further. While mixed versions operating on the same database save storage space because the upgrade is only concerned with the parts of the data schema that change between versions, they present the risk of breaking hidden dependencies [9]. In contrast, Imago requires additional resources in order to simplify the preparatory phase of resolving and understanding dependencies, and it eliminates mixed-version interactions. Imago trades off spatial overhead for a data-conversion procedure that makes it easier to implement online upgrades correctly. Moreover, Imago prevents overloading the production system by performing the computationally-intensive data conversions downstream, on the target nodes.

These design choices are based on the intuition, from our previous experience with Eternal, that new hardware costs less than the process of planning an in-place, online upgrade. Enterprises sometimes take advantage of a software upgrade to renew their hardware as well [8, 15]. This practice is supported by the fact that the new functionality included in software upgrades usually imposes higher demands on the infrastructure.[3] Moreover, we note that Imago requires additional resources only for implementing and testing the online upgrade. Once the upgraded system is in place and has been tested adequately, the additional resources can be freed up once more, which suggests that our upgrading approach could be integrated with existing cloud-computing infrastructures. Thus, the lessons learned from developing Eternal have informed and improved our approach with Imago, while retaining the essential ingredients (*e.g.*, the atomic switchover) that are useful in both approaches. Imago is likely to be more practically usable, less error-prone and better suited to fast upgrade cycles.

5 Conclusions

We study upgrade the history of Wikipedia to determine the causes of planned downtime. Incompatible schema changes and data conversions often impose downtime, because of the need to perform upgrade the schema and the application in one atomic step, and owing to concerns about overloading the production system. We describe the design of Imago, which trades off spatial overhead (additional hardware and storage resources) for a data-conversion procedure that makes it easier to implement online upgrades correctly. We show that Imago can help avoid, with minimal operator interventions, the common causes of planned downtime.

[3] For example, a Gartner study has found that upgrading SAP R/3 version 3 to version 4 requires 87% more CPU cycles, 72% more memory and 33% more storage space [3].

References

[1] MediaWiki 1.5 upgrade. http://meta.wikimedia.org/wiki/MediaWiki_1.5_upgrade, 2005.

[2] Wikipedia village pump #46. http://en.wikipedia.org/wiki/Wikipedia:Village_pump_(technical)/Archive_46#Watchlist_individual_sections_of_article_and_talk_pages.3F, 2008. Discussion among Wikipedia contributors about the technical reasons for not implementing the perennial request of being able to watchlist individual sections of a page.

[3] R. C. Beatty and C. D. Williams. ERP II: best practices for successfully implementing an ERP upgrade. *Communication of the ACM*, 49(3):105–109, Mar 2006.

[4] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[5] C. Boyapati et al. Lazy modular upgrades in persistent object stores. In *Object-Oriented Programing, Systems, Languages and Applications*, pages 403–417, Anaheim, CA, Oct 2003.

[6] N. Carvalho, A. C. Jr., J. Pereira, L. Rodrigues, R. Oliveira, and S. Guedes. On the use of a reflective architecture to augment database management systems. *Journal of Universal Computer Science*, 13(8):1110–1135, 2007.

[7] C. A. Curino, H. J. Moon, L. Tanca, and C. Zaniolo. Schema evolution in Wikipedia: toward a Web information system benchmark. In *International Conference on Enterprise Information Systems*, Barcelona, Spain, Jun 2008.

[8] A. Downing. Oracle. Personal communication, 2008.

[9] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it? In *Middleware'09, submitted*.

[10] T. Dumitraş, J. Tan, Z. Gho, and P. Narasimhan. No more HotDependencies: Toward dependency-agnostic upgrades in distributed systems. In *Workshop on Hot Topics in System Dependability*, Edinburgh, Scotland, Jun 2007.

[11] L. Moser et al. Eternal: fault tolerance and live upgrades for distributed object systems. In *Information Survivability Conference and Exposition*, pages 184 – 196, Hilton Head, SC, Jan 2000.

[12] D. Lowell, Y. Saito, and E. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. *ACM SIGOPS Operating Systems Review*, 38(5):211–223, 2004.

[13] Oracle Corporation. Rolling upgrades of stateful J2EE applications in Oracle Application Server. White Paper, Aug 2005.

[14] D. Patterson. A simple way to estimate the cost of downtime. In *Large Installation System Administration Conference*, pages 185–188, Philadelphia, PA, Nov 2002.

[15] I. Zolti, Accenture. Personal communication, 2006.