A Semi-automated Framework for Migrating Web applications from SQL to Document Oriented NoSQL Database

Rahma S. Al Mahruqi
Queen's University
Kingston, Canada
13rsha@queensu.ca

Manar H. Alalfi
Ryerson University
Toronto, Canada
manar.alalfi@ryerson.ca

Thomas R. Dean
Queen's University
Kingston, Canada
tom.dean@queensu.ca

ABSTRACT

Non-relational databases, such as *NoSQL*, play a major role enabling web applications to manipulate flexible data. Non-relational databases have different strengths from relational databases and are optimized to scale dynamic data such as that used by posts by users in social media, producing terabytes of data. As businesses scale, they need to migrate legacy relational applications to use *NoSQL* techniques. There are two parts to the migration: the migration of the schema and data, and the migration of the application source code. In this paper, we present a semi-automated approach to solve the second part. We migrate and optimize the embedded *SQL* queries to interact with the new database API and change the application code to use the new queries.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Software design tradeoffs*; *Automatic programming*.

KEYWORDS

NoSQL, Application Migration, SQL, MongoDB

ACM Reference Format:
Rahma S. Al Mahruqi, Manar H. Alalfi, and Thomas R. Dean. 2019. A Semi-automated Framework for Migrating Web applications from SQL to Document Oriented NoSQL Database. In *CASCON '19: 29th Annual International Conference on Computer Science and Software Engineering, Nov 04–05, 2019, Toronto, ON*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Many web applications start out small, and typically use relational databases such as *MySQL* [33], even for unstructured data such as user comments and user posts. Once the applications become popular, they may scale to a size in which relational databases impact the performance of the application. In recent years, more attention has been credited to the *NoSQL* data stores that were developed to address the issues identified through the emerging Web 2.0 data landscape and cloud technology, namely, the need for storage solutions to support higher availability and greater scaleability. As a point of clarification, the class of *NoSQL* databases typically refers to the non-relational storage of the data rather than the lack of the *SQL* query language [26]. Migration of the underlying database technology for these mature applications can be difficult. There are two parts to this process. The first part is the migration of the database itself which includes formulating the decision of which parts of the database to migrate, the design of the new *NoSQL* database and the migration of the data. The second part to change to the application to use the new *NoSQL* documents. Considerable research already exists for schema and data migration [9]. The migration of the code has largely been left for manual migration efforts. This has become a motivating factor to examine how difficult and laborious it is to move an existing, regularly used application, based on the relational environment to a non-relational data structure. The difficulty of carrying out a database migration process, the scope of changes, which would have to be done in the existing source code and the efficiency of an application while using new data structure, will be considered in our work.

In this paper, we propose a semi-automated approach to migrate a highly dynamic web application that uses a relational database such as *MySQL* to one that uses both a relational and a document oriented *NoSQL* database such as *MongoDB*. *MongoDB* is known for its easy and quick setup and its features such as high-performance, high availability, automatic scaling, and the ability to support fast iterations [17]. There are two parts to this approach: the migration of schema and data, and the migration of the actual application code. Our approach provides contributions to the second part, migrating and optimising the embedded *SQL* queries to interact with the new *NoSQL* database system and changing the application code to use the translated queries. The contributions of this paper are:

- A framework for the automated migration of relational database web applications to a mixture of relational and document oriented *NoSQL* applications.
- A semi-automated tool that realizes the proposed framework.
- An experiment that evaluates our framework and tool on production web applications of various sizes.

Section 2 describe our approach and contributions in more detail.

2 APPROACH

Fig.1 shows the overall structure of our approach. It is composed of four stages. The first stage deals with the migration of the schema. Not all data in a web application should be stored in *NoSQL* databases. Structured data such as users, administrators, roles and access control should remain in a relational database. Data that is very dynamic in nature such as the time-line in Facebook, user posts in bulletin board systems, or user posts in an application such

CASCON '19, Nov 04–06, 2019, Toronto, ON

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

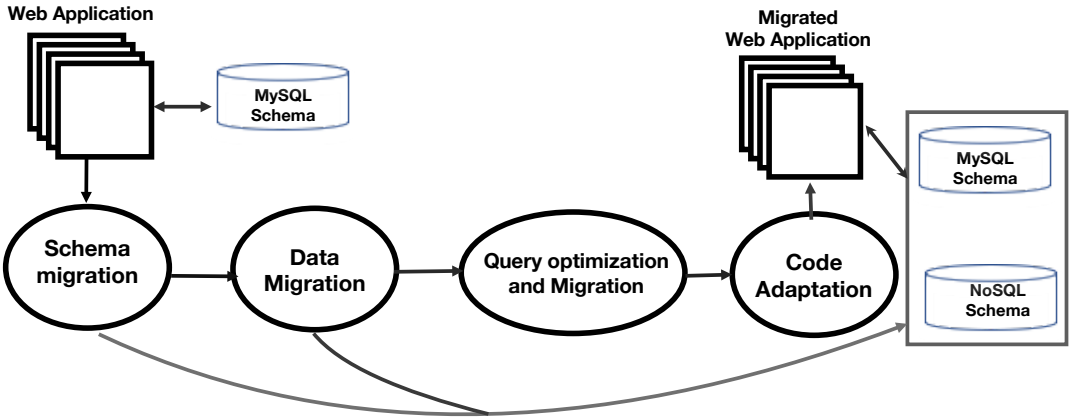

Figure 1: Proposed Migration Approach

as Twitter are the type of data that should be migrated to a *NoSQL* database. This stage identifies the part of the database schema that should be migrated and creates the *MongoDB NoSQL* data model.

The second stage migrates the data in the tables that were identified in the first stage to the newly created *NoSQL* database.

The third stage of the approach uses a combination of backward slicing [32] and the technique used by Alalfi et al[3] to extract the set of *SQL* statements executed by the application. Those queries that involve tables that were migrated in the first stage are translated. One key component of this translation is query optimization. Existing *SQL* database engines have built in query planners that optimize the order of operations in complex queries. Most *NoSQL* databases require the application programmer to specify the order of operations. A naive translation of the *SQL* query may result in sub-optimal queries. While the queries may be optimized manually, applying some database query optimization techniques during the translation may result in better *NoSQL queries*.

The last stage of our approach is to use the migrated queries as a template to modify the code of the web application to generate the modified queries. This involves a backwards slice to identify the statements that assemble each of the *SQL* statements migrated in the third stage. These statements are migrated to generate the appropriate *NoSQL* query. The main contribution of this paper is the migration of the application code, which is composed of the last two stages.

To test and validate our approach, we collected three applications to migrate to *NoSQL*. These were a small application *SCARF* (15.5 Mb) [27] and two intermediate applications *PHPBB v2 and v3* (221.1 Mb, 1362.6 Mb respectively [22]. The smallest application allows us to explore migration strategies, while the intermediate applications allow us to evaluate in a larger context. We chose the application based on this type pf transformation from *SQL* to document oriented *NoSQL* that suites this type of applications; bulletin boards and forums systems, and content management systems that require horizontal scaling of thousands of nodes as demanded when handling huge collections on structured and unstructured data sets. Also, all the applications under test are PHP based application. Applications were first installed and exercised to generate initial test data to be migrated. SCARF is a conference paper submission and organization web application. We created 10 users, submitted 25 papers and 67 posts. PHPBB2 and PHPBB3 are bulletin board applications that allow users to post comments about a variety of topics. For PHPBB2, we created 47 users, 815 topics and made 1017 posts to populate the data-set. For PHPBB3, we created 10333 users, 23931 topics and 28174 posts.

We now examine each of the steps in more detail.

3 SCHEMA AND DATA MIGRATION

The structure of the relational databases is usually more complex than *NoSQL*, due to the normalization process in which data is split into multiple related tables [12]. In contrast, *NoSQL* databases store data in a de-normalized unstructured or semi-structured way. Thus, the transformation from relational databases to the *NoSQL* ones is not straight forward. *NoSQL* does not have an explicit JOIN operations as if the table to be migrated is involved in a join relationship with another table, either both tables must be migrated to a single *NoSQL* collection, or the code for the *NoSQL* equivalent of a join must be included in the application.

In the relational model, the data is stored in tables where attributes represent *forum, posts, users and topics* information and rows represent *posts and users IDs* as example of PHPBB application. In the document model, the data transformed into JavaScript Object Notation syntax (JSON) files.

As part of the first two stages of application migration, we evaluated two published approaches for migrating the database schemes to *MongoDB*. Arora et al. [4] transform a relational database into a *MongoDB* document database. They propose an interface to design the database schema from *MySQL* to *MongoDB* and then applying *Pentaho Data Integration* (PDI) tool [6] to migrate the data. The algorithm applies data transformations (embedding documents) and generate the *NoSQL* structure design. Jia et al. [12] model the *NoSQL* database, *MongoDB*, with relational algebra. They propose an approach to do model transformation, and they present their tool that aids in the design of the new data structure. Their approach is similar to [4], but they give some guidelines to choose when to embed documents. Their model transformation algorithm

only optimizes specific tables instead of the entire database of the relational database to avoid data redundancy.

In these approaches, developers extract an Entity Relationship (ER) model from the existing relational database. If developers are not satisfied with the result of model transformation, they can modify the result of model transformation, such as removing some collections, forcing documents to be embedded, or removing some fields.

Since implementations of the two schema migration approaches were not publicly available, we reproduced their algorithms in Java. We then tested our implementation on the migration of the schemas for *SCARF*, *PHPBB2* and *PHPBB3* . Our implementation connects to the source database to obtain the schema: the names of the tables, their attributes and relationships. The information on the relationships can be retrieved using the primary constraint and foreign keys for each table. Our implementation provides an interface that allows the user to select the starting tables and columns and, using information about the *NoSQL* target database server, and the algorithms described in the previous research, generates the text files used as input for the data migration tools.

An effective data models support the application needs, and the key consideration for the structure of the documents is the decision to either embed or to use references. Embedded data models allow applications to store related pieces of information in the same database record [14]. As a result, applications may need to issue fewer queries and updates to complete common operations. In our schema design, we apply embedding for the one-to-many relationships as an array of sub-documents. In these relationships the *many* or child documents always appear with or are viewed in the context of the *one* or parent documents. We apply reference documents by their *_id* value to represent more complex *many-to-many* relationships.

We used the *Pentaho* data integration tool (PDI) [6], an open source data integration tool to migrate the data in our sample applications. The tool has the flexibility to either embed the related tables to*MongoDB* collections automatically or to allow the user to select the columns for joins. We successfully applied *Pentaho* tool to migrate the data residing in the *MySQL* database of the three applications under test to the generated *MongoDB* schema using the parameters generated by our tool.

We converted 7 tables out of 7 *MySQL* tables to *MongoDB* in the *SCARF* application. In *PHPBB2*, 26 tables out of 30 tables were converted and in *PHPBB*3 64 tables out of 70 tables were converted to *MongoDB*. The result was two independent installations of each test system. One with only *MySQL*, the other with both *MySQL* and *MongoDB*. The data verification shows that data migration process from *MySQL* relational database to *MongoDB NoSQL* database was performed successfully by applying the proposed methodology and the data migration tool. *MongoDB* also performs all of the basic operations like INSERT, UPDATE, DELETE and SELECT, which are identical to *MySQL*.

4 QUERY MIGRATION AND OPTIMIZATION

There are two steps to the query migration process in our approach. The first step is to inventory the queries made by the application. The second step is to migrate each of the queries. The first step extracts the queries and classifies them based on the use of tables from the schema. Our migration is implemented using source transformation in TXL [8]. We use a *MySQL* grammar adapted from SQL2XMI [1], our own *MongoDB API* grammar, and the official *PHP* grammar from the TXL website[31].

4.0.1 Query Extraction. The first phase in the query migration process is the query extraction phase. We start by extracting the queries from the the web application by instrumenting calls to the *mysql_query* function. The final computed string that is passed to this call is logged along the location (file and line) of the call. This provides an inventory of all the queries executed by the application.

4.0.2 Query Classification and Migration. We classify the extracted *SQL* statements into four categories. The first category(*Unchanged*) are *SQL* queries that only use tables that are not part of the migration process and remain in the relational database. An example might be the table that identifies users and their account preferences. These queries are obviously left alone. The Second category of queries (*Single*) are queries that involve only a single table that has been migrated to *MongoDB*. This represent a simple translation from the *SQL* to the equivalent API for *MongoDB*. The third set of queries(*Double*) are queries that involve more than one table, all of which have been migrated in the previous stage. These *SQL* queries are also migrated from *SQL* to the equivalent for *MongoDB*, but are more complex than a single table. The last class of queries (*Dual*) are queries that include both relational and *MongoDB* tables. This is the most complex migration as it requires a combination of *SQL* and the *MongoDB API*. The classification process allows us to apply different migration rules. This allows us to have specific migration rules for queries with all tables non migrated (classified as Unchanged), and queries with a mixture of migrated and non-migrated tables (Dual). Fig.2 illustrates our query classification and translation process.

A single program is used to read a list of tables that has been migrated and process each of the extracted *SQL* queries in turn. Separate translation rules were written for each case. We started with direct naive translations, drawn from the *SQL* to *MongoDB* mapping table on the PHP website [21]. For example, the following *SQL* select pattern, matches columns tables, where, order and limit elements of a SELECT statement. The following pattern example about SELECT with a single table and selecting a list of columns are translated to *MongoDB find* operation. If there is an ORDER BY clause, a sort() call is added to the find action and the pattern supports limit clause.

SQL Statement

```
pattern:
SELECT
[list select_Expr1]
FROM [table_references]
[opt whereClause]
[opt orderbyClause]
[opt limitClause]
```

An example of the type of translation from *SCARF* application is shown below. We show the original *SQL* statement in a PHP comment and the equivalent *MongoDB* action.

SQL Statement and translated MongoDB action

```
// SELECT * FROM users WHERE email = '$email'
db.users.find({email : $email});
```

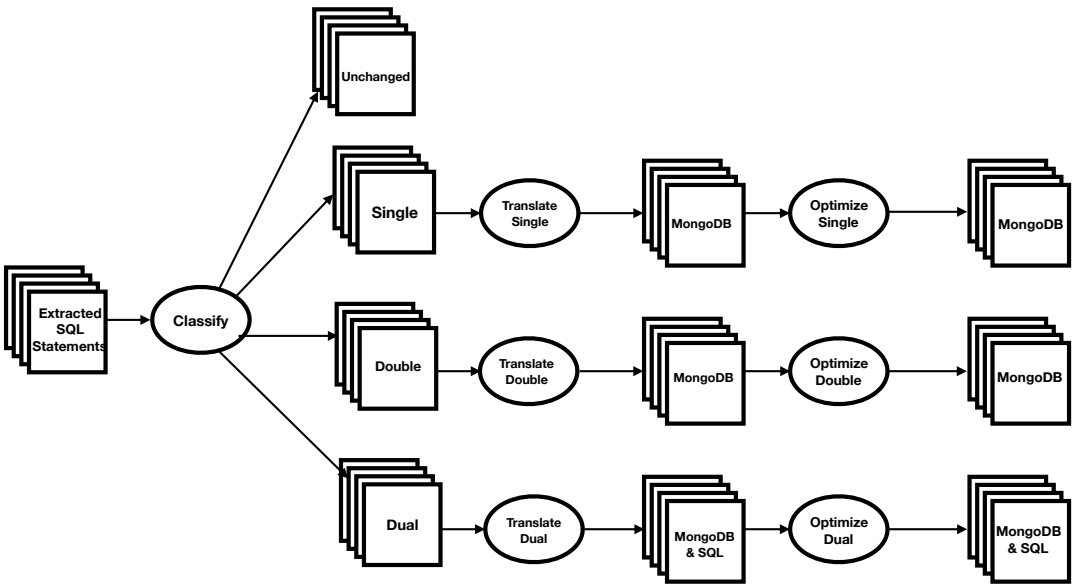

Figure 2: Query Migration Process

This is the simplest case of a select all with a simple where clause. It is turned into a *find* operation with the search fields as the parameter.

The following query from the SCARF application is an example of one that is not changed. The tables authors and users in the SCARF application are not sufficiently dynamic to benefit from the use of *MongoDB*.

SQL Statement

```
SELECT firstname, lastname
     FROM authors LEFT
     JOIN users USING (user_id)
     WHERE paper_id = '$row2[paper_id]';
```

The translation applies a different set of rules for each category of query. The migrated statements are then stored in a mapped directory. Output for statements without mapping is stored in the *unchanged* directory. If the statement is migrated, the output will be the original *SQL* statement followed by the result *MongoDB* operation.

SQL Statement

```
SELECT p.post_id FROM phpbb_posts p
WHERE p.poster_id = $param1 AND p.post_visibility = $param2
ORDER BY p.post_time DESC L
LIMIT $param3"$
```

MongoDB translated Statement

```
db.phpbb_posts.find({poster_id : $param1, post_visibility : $param2}).sort({
      post_time:-1}).limit($param3);$
```

This example illustrates a single table with select, where, order and limit clauses.

Our current implementation covers the *SQL* language including SELECT, INSERT, UPDATE, DELETE, and CREATE TABLE statements. There are some complex cases involving built in *SQL* functions which are not fully covered, however the cases in the *MongoDB* mapping table on the PHP website [21] are covered, as are the special cases found in our test applications.

From a total of 1098 queries, the classifications process of PHPBB3 gives us the following categorized queries: 80 unchanged queries, 968 single, 63 dual, and 74 double queries. SCARF: 64 single and 22 double queries. For PHPBB2: 254 single queries, 29 double queries, 36 dual queries and 72 unchanged queries stayed in MySQL database. Table. 1 shows the migration statistics of the three applications.

Some queries specify an order to the results using computed values. The *MongoDB find* operation can only sort by fields, not by computed value, resulting in a different order. To produce the same order as the original *SQL* statement, we translate the *SQL* statement to a *MongoDB* aggregation action. This creates a new computed field which can be used for sorting as shown in the following example:

SQL Statement

```
SELECT t.topic_id FROM (phpbb_topics t)
WHERE t.topic_last_post_time >= $param4
ORDER BY t.topic_type DESC, LOWER (t.topic_title) ASC LIMIT $param5
```

MongoDB Translated Statement

```
db.phpbb_topics.aggregate([ {"$match" : {topic_last_post_time : {$gte : $param4
      }}}, $projection : {topic_id : 1,computed1 : {$toLower "$topic_title"}}},
       {"$sort" : {topic_type : -1, computed1 : 1}},
{"$limit" : $param5}]);$
```

The aggregate function *Lower* is implemented in the result by the new field *computed1* which is computed using the built in function $toLower. The field *computed1* is used in the sort clause of the translated *MongoDB* statement.

Some queries (involve to migrated tables) have multiple aggregation functions (e.g. max, min, count) as shown in the following example from PHPBB3 :

SQL Statement

```
SELECT MAX(post_id) AS last_post, MIN(post_id) AS first_post, COUNT(post_id) AS
      total_posts FROM phpbb_posts WHERE topic_id = $id$
```

This statement will be translated using migrate select aggregation rule to generate a *MongoDB* aggregation pipeline with the aggregate functions.

MongoDB translated Statement

```
db.phpbb_posts.aggregate([ {$projection : {post_id : 1, post_id : 1, post_id :
      1}}, {"$match" : {topic_id : $id}}, {"$group" : {"_id" : null,
      total_posts : {"$sum" : {"$cond" : [{"$ifNull" : ["$post_id", false]}, 1,
       0]}}, first_post : {"$min" : "$post_id"}, last_post : {"$max" : "$
      post_id"}}
```

The min and max are implemented as simple functions at the end of the pipeline. The count aggregation is implemented as a combination of the functions $sum, $cond and $ifNull.

Some double queries have a join that is not a simple field equality such as the following *SQL* statement:

SQL Statement

```
SELECT t.topic_id
FROM phpbb_topics t, phpbb_topics t2
WHERE t2.topic_id = $topic_id AND t.forum_id = t2.forum_id
AND t.topic_moved_id = 0 AND t.topic_last_post_id > t2.topic_last_post_id
ORDER BY t.topic_last_post_id ASC LIMIT 1$
```

In this example, the join filter includes an inequality to sort topics by the id of the last post in the topic. The lookup operator in *MongoDB* can only join two collections with a single field equality. So the result has to be implemented as two *MongoDB* statements, one for each of the field equality operators This statement has been migrated to two *MongoDB* actions, one for each collection.

MongoDB translated Statement

```
db.phpbb_topics.find({topic_moved_id : 0});
db.phpbb_topics.find({"t.topic_id" : $topic_id, forum_id : {$in : $list_forum_id
      }});$
```

*PHPBB*3 also uses array elements in select statements as in the following example:

SQL Statement

```
SELECT * FROM phpbb_forums WHERE id = $row[id]
```

SQL statements are also used in other constructions such as object properties:

SQL Statement

```
SELECT * FROM phpbb_forums WHERE parent_id = $this->parent_id
```

Dual queries are *SQL* sentences that use some tables migrated to *MongoDB* with other tables that have not been migrated. They request data from two unconnected databases. So, there is one operation done in *MySQL*, and another in *MongoDB* to get all data for the join. Since the databases cannot merge this data, it has to be merged in the application code. For example, in *includes/functions_users.php*, there is this query definition:

Dual query definition

```
SELECT r.post_id, p.topic_id
FROM ' . REPORTS_TABLE . ' r, ' . POSTS_TABLE . ' p
WHERE ' . $db->sql_in_set('r.user_id', $user_ids) .'
AND p.post_id = r.post_id'
```

Which joins table *REPORTS* (not migrated) with *POSTS* table (migrated). It is transformed to this:

PHP code of Dual query

```
"\$res=\$_sql('SELECT * FROM phpbb_reports'); \$rows=array();while(\$row=
      mysql_fetch_assoc(\$res))\$rows[]=\$row;
\$rows0=\$rows;
\$list_post_id=array_column(\$rows0,'post_id');foreach(\$list_post_id as &\$a){\
      $a=(int)\$a;};
\$rows=\$db->phpbb_posts->find([post_id => ['\$in' => \$list_post_id]]);
\$rows=\$s2m->join_external(\$rows0,'post_id',\$rows,'post_id',true); \$cols=[\"
      post_id\",\"topic_id\"]; \$proj=[];"
```

This automatically generated php code does a query with *SQL* to get data from *REPORTS* table, and then a *MongoDB* call to get *POST* information for these records, and lastly it merges information from two databases with *join_external*.

Our naive approach is able to generate *MongoDB* actions for the *SQL* queries from our test applications which we tested against the migrated tables in our experimental installations of the applications. The *MongoDB* actions produced the same results as the original *SQL* queries. However, the performance of the naive translation is an issue. These performance results will be shown in the next section.

4.0.3 Query Optimization: There are many factors that can affect database performance and query execution time including the use of indexes, query structure, data models and schema design. Applying some of the database query optimization techniques like creating indexes and changing table order during the translation can result in better performance.

MongoDB supports pipeline optimization where the developer can specify the order of operations. One clear optimization is to place the match filter as early in the aggregation pipeline as possible [18]. Because the filter limits the total number of documents in the aggregation pipeline, it minimizes the amount of processing later in the pipe. As such, in this step, we move all match conditions to as early in the pipeline as as possible. For example in the naive approach, the match is later in the pipeline:

SQL Statement

```
SELECT DISTINCT p.post_id FROM phpbb_posts p, phpbb_topics t WHERE p.poster_id =
       $param1 AND (p.post_visibility = $param2 OR p.forum_id IN $list) AND t.
      topic_id = p.topic_id GROUP BY t.topic_id, t.topic_last_post_time ORDER
      BY t.topic_last_post_time DESC$
```

In the optimized version the match is at the beginning of the pipeline as shown:

MongoDB optimized Statement

```
db.phpbb_posts.aggregate([{@"$match"@ : {$and : [{poster_id : $param1}, {$or :
      [{post_visibility : $param2}, {forum_id : {$in : $list}}]}]}}, {"$lookup"
       : {"from" : "phpbb_topics", "localField" : "topic_id", "foreignField" :
      "topic_id", "as" : "t"}}, {"$unwind" : {"path" : "$t", "
      preserveNullAndEmptyArrays" : false}}, {"$project" : {"post_id" : 1, "t.
      topic_last_post_time" : 1}}, {"$sort" : {"t.topic_last_post_time" : -1}},
       {"$group" : {"_id" : {topic_id : "$t.topic_id", topic_last_post_time : "
      $t.topic_last_post_time"}}}, {"$group" : {"_id" : {post_id : "$post_id"
      }}}])$
```

In the above example, there are two collections, *phpbb_posts* and *phpbb_topics*. The query is implemented as an aggregate pipeline on *phpbb_posts*. The collection *phpbb_posts* is the main collection of the query. A double table query where the main collection is filtered is more likely to be faster. So, if the main collection of the query is unfiltered, changing the order of the collections so that the main collection with a filtered collection may be faster. However, it will not be always the case, for example, a main collection can be unfiltered but has fewer documents than the result of filtering the other collection. For example, we have a table with posts and another with topics. Each post has a field with its topic id.

SQL Statement

```
SELECT phpbb_topics.topic_title
FROM phpbb_topics, phpbb_posts
WHERE phpbb_posts.post_id = 1
AND phpbb_posts.topic_id = phpbb_topics.topic_id
```

If we run a query to retrieve the topic title for the post id = 1, *SQL* database query planner chooses what table is best to read first, but *MongoDB* does not. Without the table order optimization program, *MongoDB* query looks first the topic table. It has to read ALL topics and ALL their posts. Then, it will discard all posts and retrieve the post with id = 1 only. With the optimization, the post table would be used first because it is filtered. *MongoDB* will have to read only one post, searching by its id and for that post it will read only its topic.

Based on recommendations of the *NoSQL* community[18], our optimization process will change the table order in the join chain. The first tables in the operation are tables that are filtered by their key fields. Next are tables filtered by non-key fields, and finally tables with no filters. The program will produce only one *MongoDB* output for each double *SQL* query. This query will use the table order with the highest score. In most cases, the optimization program produces a query with the lowest query execution time. Here is an example of an *SQL* query and its equivalent optimized *MongoDB* action:

SQL Statement

```
SELECT COUNT (p.post_id) AS total FROM phpbb_posts p, phpbb_topics t WHERE p.
      forum_id IN $list AND p.post_visibility = $param1 AND t.topic_id = p.
      topic_id AND t.topic_visibility <> p.post_visibility
```

MongoDB optimized Statement

```
db.phpbb_posts.aggregate([{"$match" : {$and : [{forum_id : {$in : $list}}, {
      post_visibility : $param1}]}}, {"$lookup" : {"from" : "phpbb_topics", "
      localField" : "topic_id", "foreignField" : "topic_id", "as" : "t"}}, {"$
      unwind" : {"path" : "$t", "preserveNullAndEmptyArrays" : false}}, {"$
      project" : {"post_id" : 1, computed1 : {$ne : ["$t.topic_visibility", "$
      post_visibility"]}}}, {"$match" : {computed1 : true}}, {"$group" : {"_id"
       : null, total : {"$sum" : {"$cond" : [{"$ifNull" : ["$post_id", false]},
       1, 0]}}}}])
```

4.1 Query Optimization Analysis

We evaluate the impact of our query optimizations on the performance of the migrated queries using the test data migrated in section 3. The evaluation was conducted on an Ubuntu 14.04.4 server, with *MySQL 5.5.47* and *MongoDB 3.2.10*, with the following collections from our PHPBB3 database test set: 28174 posts , 23931 topics and 10333 users. Queries that were measured are the SELECT queries that were classified as SINGLE or DOUBLE and migrated to *MongoDB*. Tests were run on both the *MySQL* and *MongoDB* systems using the same data sets. One test was done on the original non-optimized queries and the other with the optimized queries, and the time is recorded in seconds.

Fig. 3 shows a comparison between the execution time of the non-optimized migrated *MongoDB* queries (green),the execution time of the optimized migrated *MongoDB* (red), and the original *SQL* queries (blue). For example the un-optomized execution time for query 1 is over 5 seconds while the optimized and original *SQL* execution times were a small fraction of a single second. We can observe that the optimization stage of our approach did drastically improved the performance of the migrated *MongoDB* queries to a level that is comparable to the performance of the *SQL* queries. The lower graph in Fig. 3 shows a closer look at the two bottom lines in the upper graph. The larges difference is in query 6 which the optimized *MongoDB* is 0.1 sec vs a very small *SQL* execution time.

Since the test size of the test collections are still relatively small, the advantage of *MongoDB* for these types of collections is not immediately obvious. However, it is clear that our approach does not introduce a significant performance penalty while enabling the use of *MongoDB*.

5 WEB APPLICATION MIGRATION

The last stage of our approach is altering the application to use the translated queries. This process is the converse to the *SQL* queries extraction stage. We apply a backward slice combined with the approach used by Alalfi et al. [2].

We modify a backwards slice to do markup instead of removing statements. We start from the location where each of the migrated *SQL* statements is executed (a call to the function *mysql_select*). This is the function used to launch *SQL* queries in SCARF and both PHPBB versions. The *SQL* statement may be a literal string, in which case the migration is done at the site of the call. Otherwise, we move backwards from the execution of the *SQL* statement in the program tracking the construction of the string literal that contains the *SQL* statement.

While it is possible to build up a PHP data structures and lambda functions[21] to execute the equivalent *MongoDB* action, we have chosen an intermediate transform to start. We construct the *MongoDB* query in a string literal the same way the *SQL* query is constructed. We transform each PHP statement that adds a component to the *SQL* stringlit to add a transformed component to the *MongoDB* stringlit. The resulting string contains *MongoDB* actions in PHP code that is then executed using the *eval* function.

Our first example is a simple *SQL SELECT* statement that retrieves the forum id for a specific topic. The result of the query is a dictionary that has a single element ,'forum_id'.

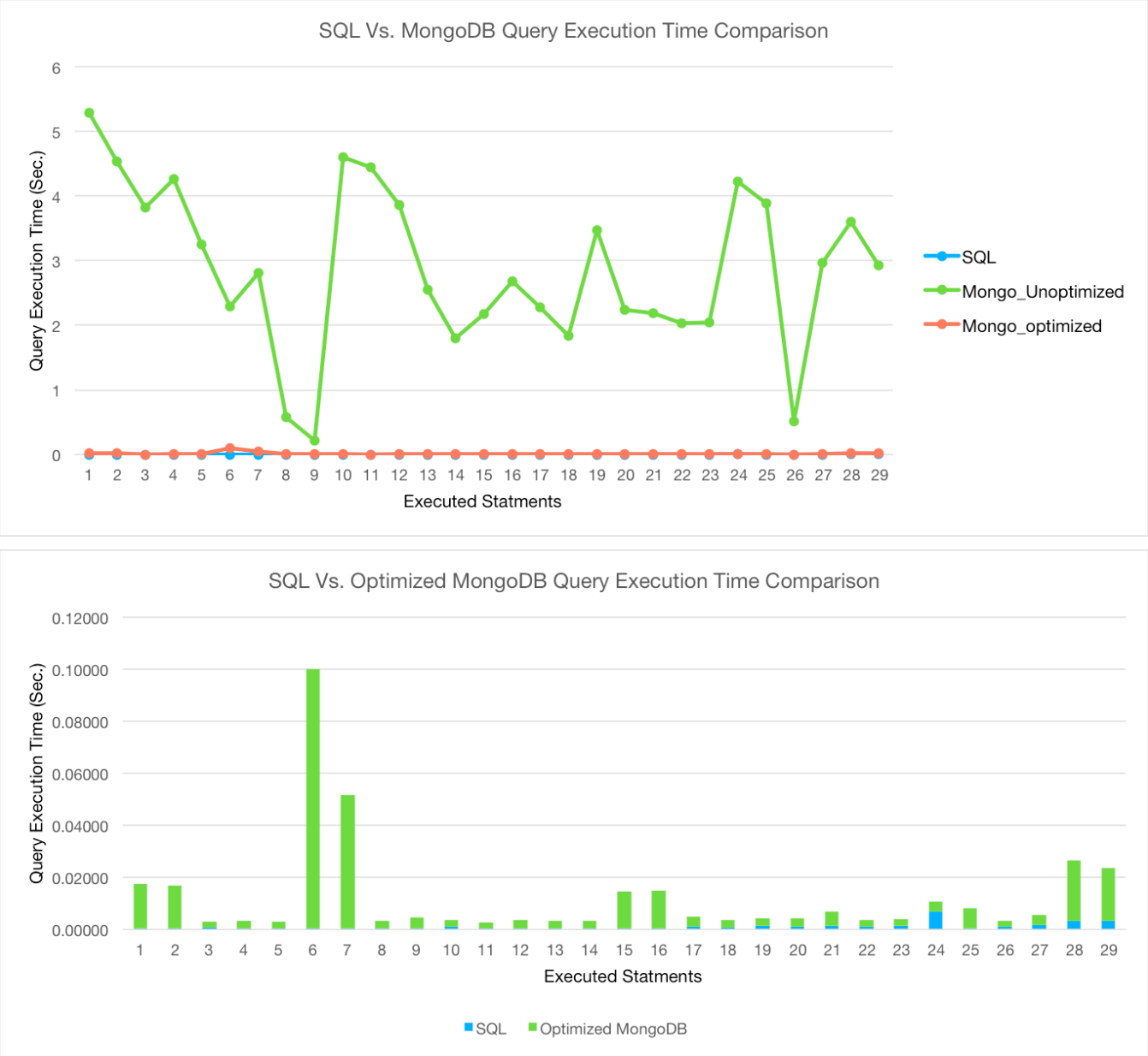

Figure 3: Query Optimization Analysis

Simple SQL SELECT statement in PHP

```
$sql = 'SELECT forum_id FROM ' . TOPICS_TABLE
    . ' WHERE topic_id = ' . $topic_id;
$result = $db->sql_query($sql);
$forum_row = $db->sql_fetchrow($result);
$db->sql_freeresult($result);
if (!$forum_row)
{
    trigger_error('NO_TOPIC');
}
```

In the PHPBB3 application, all calls to *mysql_query* are done through the *sql_query* function provided by the driver. Using the known values of the system constants in *constants.php*, the *SQL* statement concatenation is translated to the equivalent *MongoDB find* action. We then execute the *MongoDB* action:

Simple MongoDB find action in PHP

```
$sql = "\$rows = \$db->phpbb_topics->find(['topic_id'=> intval ($topic_id)],
     ['forum_id'= > 1 ] ) ;\$cols=[\"forum_id\"];"
$result = db->sql_query($sql)
$forum_row = $db->sql_fetchrow($result);
$db->sql_freeresult($result);
if (!$forum_row){
    trigger_error('NO_TOPIC');
}
```

Our migrated version of the *sql_query* function checks the query. If the query is an un-migrated *SQL* query, the standard *MySQL* function is called, and the result is returned as normal. If it s a *MongoDB* action, then we evaluate the string and check the results:

Elided Migrated Version of *sql_query*

```
try {
    ...
        $res = eval($out);
    ...
} catch (MongoException $e) {
    ... error handling ...
    return null;
} catch (Throwable $t) {
    ... error handling ...
    return null;
}
...
if (isset($rows)) {
    $res = $rows;
}
```

Our approach also instruments calls to the rest of the *MySQL API*. We have provide implementations of the *MongoDB* actions that are similar to the semantics of the *SQL* operations to simplify conversions, and modified versions of the *MySQL API* that work for both the *SQL* and the *MongoDB* queries and results. This simplifies the translation by allowing us to focus on the translation of the query.

Next is an example of an UPDATE Statement.

SQL UPDATE statement in PHP

```
$sql = 'UPDATE ' . POSTS_TABLE . '
       SET post_reported = 1
       WHERE post_id = ' . $post_id;
$db->sql_query($sql);
```

This sets the *post_reported* field to 1 on the post identified by *$post_id*. *POST_TABLE* is a PHP constant defined in *constants.php*. It is used to mark a post as reported. Note that in *MySQL*, *post_reported* is a boolean field, the value 1 is converted to TRUE. PHPBB does not check the result of the query.

MongoDB update action in PHP

```
$sql = "\$db->phpbb_posts->update(['post_id' => intval ($post_id)],
                        ['\$set' => ['post_reported' => boolval (1)]], [multi =>
                               1])";
$db->sql_query($sql);
```

As mentioned above this is an intermediate step. While simpler constructions that use only constants and user input like the example above can be optimized, *SQL* statements that are constructed in pieces are not easily optimized. The main advantage of this approach is that it provides a transparent executable migration. The *MongoDB* actions are directly visible in the application code. Developers can manually tune the string construction based on the optimized statements produced in the previous section and their knowledge of the structure and semantics of their data. A second stage of translation that translates the strings to direct PHP code is also possible in the future.

5.0.1 Application Migration Statistics . We have migrated the *SCARF* and both versions of *PHPBB*. In the case of *SCARF*, all queries were successfully translated, and the application was manually tested to ensure the migrated version had the same functionality. We checked that the migrated application has the same output with an equivalent database on all pages. The current implementation of our approach does not fully migrate the two versions of *PHPBB*. In *PHPBB*2 15% of queries are not fully migrated, and 25% of the queries in *PHPBB*3 are not fully migrated. The missing parts are a result of missing elements of the backwards markup and because of the dynamic nature of the queries that could not get a complete sentence by the static migration. Also the limitation of the backward slicing is the query may not exist in the source code. Backwards slicing is a well understood problem, and completion is a matter of dealing with many language details of PHP. We finished the migration of three PHPBB3 web pages manually. We only manually migrated the untranslated queries and did not modify the queries that were migrated automatically. This allowed us to test the functionality of the automatically translated queries in these pages (which were correctly executed). We log all *SQL* queries executed in a PHPBB3 session, testing as many pages as possible by instrumenting PHPBB3 to write all executed queries to a file. Then going through all pages in the web browser with an anonymous user, and with an identified user. With that we got a list of complete PHPBB3 queries to test our procedures and to improve our migration rules.

A total of 86 queries in SCARF are processed automatically, with *100%* coverage. In PHPBB2, of a total of 460 queries, 391 processed automatically, with (*85%*) coverage and in PHPBB3 with a total of 1572 queries, 1185 processed automatically with (*75%*) coverage. Some examples of the more esoteric queries migrated from PHPBB3 pages are:

Migrated SQL Query to MongoDB of viewforum.php page

```
 if (! $forum_id)
    { $sql = (((" 
// SQL SELECT Statement
// \"SELECT forum_id FROM phpbb_topics WHERE topic_id = $topic_id\"
// Translated find MongoDB Action
\$rows=\$db->phpbb_topics->find(['topic_id' => intval ($topic_id)], ['forum_id'
       => 1]);
\$cols=[\"forum_id\"];
")));
    track_strlit ($sql, "SELECT forum_id
 FROM phpbb_topics
 WHERE topic_id = $topic_id");
    $result = $db -> sql_query ((($sql)));
    track_call_local2 ('sql_query', $sql);
    $forum_id = (int) $db -> sql_fetchfield ('forum_id');
               $db -> sql_freeresult ($result);
```

An Example of migrating *SQL Select* to *MongoDB aggregate* function of the *viewtopic.php* page:

Migrated SQL Query to MongoDB aggregate function of viewtopic.php page

```
// SQL Statement
"SELECT topic_last_post_id AS post_id, topic_id, forum_id
 FROM phpbb_topics WHERE topic_id = $topic_id"
// Translated MongoDB Action
$rows=\$db->phpbb_topics->aggregate([['\$project' => ['post_id' => '\$
       topic_last_post_id', 'topic_id' => 1,
                                    'forum_id' => 1]], ['\$match' => ['topic_id'
                                          => intval ($topic_id)]]]);
$cols=[\"post_id\",\"topic_id\",\"forum_id\"]"));
```

6 RELATED WORK

Existing tools and techniques target the problem of data and schema migration without consideration to migrating the applications that

Table 1: Experiment Results

Application Name	PHP files	Modified files	Queries	MigratedQueries	Single	Double	Dual	Not changed
SCARF	19	16	86	86	64	22	0	0
PHPBB2	71	49	460	391	254	29	36	72
PHPBB3	2906	1561	1572	1185	968	74	63	80

interacts with the data. For example, the ETL *Extract-Transform-Load* tool implements the interface to some mainstream *NoSQL* database systems to do data transformation, such as Cassandra, *MongoDB*, and *HBase* [12]. However, ETL tools cannot automatically map the existing database to the target database. ETL tools can only complete the data migration process, requiring the user to design their own mapping strategies between different databases. When designing a data model for *MongoDB*, the key consideration is to decide when to use embedding or references between different documents [14]. This determines the structure of documents, the performance and the data redundancy. Kanade et al. [14] discuss the pros and cons of embedded data and references. Experiments were performed to find the extent of normalization and embedding to reduce query execution time of *MongoDB*.

Chung [7] and Li [15] propose two approaches to convert schema from relational databases to *HBase*. Chung uses a JackHare Map-Reduce framework that converts every table in a relational database into a single *HBase* table. After the conversion, each table becomes a column family in *HBase*. Li uses three guidelines to guide the schema conversion where related tables will be nested as a whole and transformed into a table in HBase. The three guidelines include grouping correlated data in a column family, adding foreign key references if one side needs to access the other side's data, and merging the attached data tables to reduce foreign keys. However, the latter does not take multilevel nesting into consideration. Schram [28] proposes an abstract layer that allows software applications to access data in the *NoSQL* model transparently, without the need of changing the existing queries in the applications.

Roijackers [25] creates an abstraction layer between *SQL* and *NoSQL* databases. According to the authors, certain data sets show better results when processed by relational databases, and others are better running on *NoSQL* databases. Requests are analyzed to determine which of the two models would be ideal for processing each request. We adopt this approach to automate the process of deciding which table to migrate to *NoSQL* and which one to keep as relational.

Sellami et al. [29] develop PaaS, *ODBAPI*, a streamlined and REST-based API in order to execute the CRUD operations, i.e., create, read, update and delete, on *SQL* and *NoSQL* databases.

Ellison et al. [11] introduce a two-stage approach which accurately estimates the migration cost, migration duration and cloud running costs of relational databases. The first stage of the approach obtains workload and structure models of the database to be migrated from database logs and the database schema. The second stage performs a discrete-event simulation using these models to obtain the cost and duration estimates.

Serrano et al. [10] describe a methodology for migrating applications relying on relational databases to *HBase* back-ends. The authors describe how to create the *NoSQL* tables for best performance, and to query the database. The paper describes a methodology; it is not an automated tool. To apply it, developers have to redesign the database following the guidelines, and manually change all source code that accesses the database.

Rocha et al. [24], propose a product to run *MySQL* applications with a *MongoDB* database, using run-time query migration. This is a proxy approach, where queries are migrated at run-time and the original application source code remains untouched. The work divided into two parts: First is a process to migrate all data from *SQL* database to a *NoSQL* database. The second part is query mapping, where the *SQL* sentences are transformed to *MongoDB* actions. They use a custom *MySQL* driver based on *MySQL* Proxy [20] that intercepts the *SQL* queries at run-time. It communicates with a web service developed in Java which uses JSQLParser" [13] to parse and execute the queries. They use *MongoDB* operations implemented in Java to execute the equivalent query components, using information from the original *SQL* schema. The data recovered from the *MongoDB* operations is transformed to *MySQL* format and returned to the caller application. The work is similar to "MongoDB Connector for BI product" [5], but it seems to support more use cases. There are two products: one with very basic *SQL* support that translates sentences directly to *MongoDB*, and the other that manages a virtual database with better *SQL* support.

In our framework, we only migrate the dynamic document based data to *MongoDB* using the *Pentaho* data migration tool. The *SQL* quereies are migrated in the source code. After migration, the transformed quereies can be further optimized by the developers that have better understanding of the data and queries.

There are some industrial products and tools that help in migrating *SQL* queries to *NoSQL* ones. Unity driver product[19] uses JDBC, which is an API used in Java to access databases, which is equivalent to PHP Data Objects (PDO)[23]. These products are aimed to use *SQL* applications with a *MongoDB* database and not to migrate the applications. This makes sense commercially, since it can be used also with applications where source code is not available. Unity driver [19] shows *MongoDB* equivalences for *SQL* sentences, which can be an aid to learn *MongoDB*, but it does it only for very basic *SQL* sentences, other sentences are processed by its virtual database.

Liao et al. [34] propose a product that allows an application to use both *SQL (MySQL)* and *NoSQL (HBase)* databases at the same time. It consists of two parts: a DB Converter to migrate data and asynchronization process to maintain the databases. They use *Sqoop* [30], a data converter, to transform bulk data between *Relational* Database and *NoSQL* database. The application uses *SQL* to interact with both databases. *SQL* commands are migrated to *NoSQL* at runtime. It is a middle-ware using *C#* with *ANTLR* as an *SQL* parser and *SQL* grammar based on *Macroscope*, a .NET library [16]. The application has to be changed to use its interface, but it can use

SQL to access all databases. It uses the *MySQL* JDBC driver as an RDB connector, and a *SQL* query parser and *Apache Phoenix* as a *SQL* translator to connect *HBase*.

Our work is focusing on migrating and optimizing the embedded *SQL* queries to interact with the new database system and changing the application code to use the translated queries.

7 CONCLUSION AND FUTURE WORK

In this paper, we present an approach for semi-automating the migration of highly dynamic relational-based (e.g. MySQL) web applications to ones that use document oriented *NoSQL* databases such as *MongoDB*. We demonstrate the approach on the migration of subset of queries of the PHPBB3, a bulletin board web application. The approach was tested on three applications, *SCARF*, *PHPBB2*, *PHPBB3*. The approach successfully migrated the applications into fully working systems which uses *MongoDB NoSQL* databases and partially interacting with some non migrated *SQL* tables. We conducted an experiment to evaluate our optimization phase and the evaluation suggest that our optimization was instrumental in improving the performance of the migrated system with a performance almost equivalent to the original non-migrated application.

Since the test size of the test collections are still relatively small, the advantage of *MongoDB* for these types of collections is not immediately obvious. However, it is clear that our approach does not introduce a significant performance penalty while enabling the use of *MongoDB*.

We automate the extraction of the *SQL* statements by applying a backward slicing and the approach suggested by Alalfi et al [3]. We also provide an automated way to migrate the application source code to use the translated queries. We are working on verifying the generality of the approach by applying it to the last application *WordPress*. Future work is to integrate the optimization from the query migration into the application migration.

The suggested framework will eliminates or minimizes the effort of rewriting the application code when the back-end data storage system is changed. Further, the proposed transformation framework will reduce the effort of maintaining data portability between the different databases models. The framework transforms the design into compatible code to support the ability of moving between the different web applications and database systems.

REFERENCES

[1] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. 2008. SQL2XMI: Reverse Engineering of UML-ER Diagrams from Relational Database Schemas. In *Proceedings of the 15th Working Conference on Reverse Engineering ,WCRE 2008, 15-18 October 2008, Antwerp, Belgium.* 187–191.

[2] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. 2009. WAFA: Fine-grained dynamic analysis of web applications. In *Proceedings of the 11th IEEE International Symposium on Web Systems Evolution, WSE 2009, 25-26 September 2009, Edmonton, Alberta, Canada.* 141–150.

[3] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. 2010. Automating Coverage Metrics for Dynamic Web Applications. In *Proceedings of the 14th IEEE European Conference on Software Maintenance and Re-engineering, 15-18 March 2010, Madrid, Spain.* 51–60.

[4] P. Rupali Arora and Rinkle Aggarwa. 2013. An Algorithm for Transformation of Data from MySQL to NoSQL (MongoDB). In *International Journal of Advanced Studies in Computer Science and Engineering, IJASCSE, Volume 2, Special Issue 1.*

[5] bi-connector [n. d.]. MongoDB Documentation Website. https://www.docs.mongodb.com/bi-connector/master//

[6] M. Casters, R. Bouman, and J. Van Dongen. 2010. *Pentaho Kettle Solutions:Building Open Source ETL solutions with Pentaho Data Integration.* John Wiley and Sons.

[7] W. C. Chung, H. P. Lin, S. C. Chen, M. F. Jiang, and Y. C. Chung. 2014. Jackhare: a framework for SQL to NoSQL translation using Map-Reduce. In *Automated Software Engineering, vol. 21, no. 4.* 489–508.

[8] James R. Cordy. 2006. The TXL source transformation language. *Sci. Comput. Program.* 61, 3 (2006), 190–210.

[9] Cosmin Sebastian IOSIF Cristina Bazar. 2014. The Transition from RDBMS to NoSQL, A Comparative Analysis of Three Popular Non-Relational Solutions: Cassandra, MongoDB and Couchbase. *Database Systems Journal* 4, 2 (2014), 49–59.

[10] Eleni Stroulia. Diego Serrano. 2016. From relations to multi-dimensional maps: a SQL-to-HBase transformation methodology. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON '16), Blake Jones (Ed.). IBM Corp., Riverton, NJ, USA.* 156–165.

[11] Radu; Paige Richard F Ellison, Martyn; Calinescu. 2018. Evaluating Cloud Database Migration Options Using Workload Models. *Journal of Cloud Computing: Advances, Systems and Applications archive, Article No. 108* 7, 1 (2018).

[12] Tianyu Jia, Xiaomeng Zhao, Zheng Wang, Dahan Gong, and Guiguang Ding. 2016. Model Transformation and Data Migration from Relational Database to MongoDB. In *International Congress on Big Data, IEEE Computer Society.*

[13] jsqlparse [n. d.]. Jsqlparse Website. http://www.jsqlparser.sourceforge.net//

[14] A. Kanade, A. Gopal, and S. Kanade. 2014. A study of Normalization and Embedding in MongoDB. In *IACC 2014, in Advance Computing Conference (IACC), 2014 IEEE International. IEEE.* 416–421.

[15] C. Li. 2010. Transforming relational database into HBase: A case study. In *Software Engineering and Service Sciences (IC-SESS), 2010 IEEE International Conference on. IEEE.* 683–687.

[16] MacroScope [n. d.]. macroscope Website. http://macroscope.sourceforge.net//

[17] Alza A. Mahmood. 2018. Automated Algorithm for Data Migration from Relational to NoSQL Databases. *Al-Nahrain Journal for Engineering Sciences (NJES.* 21, 1 (2018), 60–65.

[18] MongoDB [n. d.]. MongoDB Manual Website. https://docs.mongodb.com/manual/core/aggregation-pipeline-optimization//

[19] mongojdbc [n. d.]. unityjdbc Website. http://www.unityjdbc.com/mongojdbc/mongosqltranslate.php//

[20] MySQL Proxy [n. d.]. Proxysql Website. https://www.proxysql.com//

[21] PHP [n. d.]. The PHP Website. https://secure.php.net/manual/en/mongo.sqltomongo.php/

[22] Phpbb [n. d.]. PhpBB Website. https://www.phpbb.com//

[23] phppdo [n. d.]. php manual Website. http://php.net/manual/en/book.pdo.php//

[24] Leonardo Rocha. 2015. A framework for migrating relational data-sets to NoSQL. In *Procedia Computer Science 51.* 2593–2602.

[25] J. Roijackers and G.H.L Fletcher. 2013. *On Bridging Relational and Document-Centric Data Stores. Big Data. BNCOD 2013. Lecture Notes in Computer Science, Vol. 7968.* Springer, Berlin, Heidelberg.

[26] R.P.Padhy, M. R. Patra, and S. C. Satapathy. 2011. RDBMS to NoSQL:Reviewing Some Next-Generation Non-Relational DatabaseâĂŹs. In *International Journal of Advance Engineering Sciences and Technologies,Vol. 11, Issue No. 1, 015-030.*

[27] scarf [n. d.]. SCARF - Stanford Conference And Research Forum Website. http://scarf.sourceforge.net//

[28] Aaron Schram and Kenneth M. Anderson. 2012. MySQL to NoSQL: Data modelling challenges in supporting scale-ability. In *ACM SPLASH.* 19–20.

[29] R. Sellami, S. Bhiri, and B. Defude. 2014. ODBAPI: A Unified REST API for Relational and NoSQL Data Stores. In *Proceedings of IEEE International Congress on Big Data (BigData Congress.* 653–660.

[30] sqoop [n. d.]. sqoop.apache Website. https://www.sqoop.apache.org//

[31] TXLPHPGrammer [n. d.]. TXL Website. https://www.txl.ca/txl-resources.html//

[32] M.D. Weiser. 1979. *Program slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* University of Michigan, Ann Arbo.

[33] M. Widenius and D.A. 1999. MySQL introduction. *Linux Journal.* 1999, 673 (1999).

[34] C.H. Lu S.C. Chen C.H. Hsu W. Chen M.F. Jiang Y.C. Chung Y.T. Liao, J. Zhou. 2016. Data adapter for querying and transformation between SQL and NoSQL database. *Journal of Future Generation Computer Systems.* (2016).