Assignment 1
Data Mashup
F21BD

Boris MOCIALOV
ID: bm4
URL: refs: (Mocialov 2015)
MSc Software Engineering

February 18, 2015
1 Introduction

Job of a database managers/administrators consists primarily of taking care of databases in companies that employ one or another organisation system for a set of data that is used to support businesses of these companies.

2 Context

This report is produced by a hypothetical database manager/administrator in a company called Movie Rankings R Us, which has just taken over a competitor - Killer Movies. The job of imaginary manager/administrator is to integrate datasets of the both companies. Currently, dataset of Movie Rankings R Us is supported by relational database system. Lately, this have been a problem for clients of the company and a decision had been made to move over to a NoSQL system, which is supposed to improve response times and make datasets integration job easier. Therefore, a proof of concept has to be performed before the actual migration. For the proof of concept, test-data will be used - MovieLens 100k Dataset (subset of dataset used by Movie Rankings R Us) will be integrated together with MovieKill Dataset.

3 Requirements

Overall project for described over situation is broken down into following tasks:

- Choose NoSQL system
- Prepare NoSQL system
- Create new data model
- Clean both datasets
- Integrate both datasets into a single one that uses new data model
- Import integrated dataset into the fresh NoSQL system
- Construct a set of queries to test new data model
- Execute testing queries
- Confirm correctness of the new dataset

4 Assumptions

A set of assumptions had to be made before the integration due to the nature of fuzzy requirements. Assumptions were negotiated with the individuals, responsible for the requirements document and are presented in the resolution column of the assumptions table given below.
<table>
<thead>
<tr>
<th>Assumption</th>
<th>Resolution (Rationale)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration will be made only once</td>
<td>MovieKill dataset will be either removed or backed-up after the integration and should not be used after the integration</td>
</tr>
<tr>
<td>New data model should be as simple as possible</td>
<td>Data models for NoSQL systems are usually presented in one table without external dependencies. Therefore no need to be concerned about normalisations of the relations</td>
</tr>
<tr>
<td>Analysis queries should test integrated dataset</td>
<td>Queries must incorporate information from both datasets to test if integration succeeded</td>
</tr>
<tr>
<td>It had been acknowledged that datasets contain foreign characters, which should be handled with care</td>
<td>Characters encoding is non-trivial problem in integration of datasets and must be handled with care</td>
</tr>
<tr>
<td>Dataset is used by some application (lets call it Awe-X-Movies) that Movie Rankings R Us maintains</td>
<td>Data is usually used in some way in real life, therefore it is reasonable to assume that there an application layer in the company that uses that data</td>
</tr>
<tr>
<td>Original Movie Rankings R Us dataset should remain the same</td>
<td>So that the assumed application layer will not require major changes</td>
</tr>
<tr>
<td>There are no two movies with the exactly the same name</td>
<td>Such requirement should exist in movies industry due to the copyrights. This assumption can be argued. This is also the reality to authors current knowledge.</td>
</tr>
<tr>
<td>It is desirable to make as little changes as possible</td>
<td>Less changes will have to be made in the application layer</td>
</tr>
<tr>
<td>Movie Rankings R Us uses MovieLens dataset as its official original dataset with the same tables outline as given by MovieLens</td>
<td>This had been inferred by the requirements, though, not explicitly stated</td>
</tr>
</tbody>
</table>

## 5 Organisation

### 5.1 Preparations

Both datasets had to be reviewed in order to find out what is relevant and what is not for the final data model. New data model is presented in this section later, which consists of the relevant information for the company. Since the company have been using relational databases prior the integration, it is assumed as well as verified that the overall dataset consists of many tables, where some particular key (i.e. movie_id/user_id/etc.) is used as a reference between different tables. In order to integrate wisely, only the most important data will be stored in the same table (to make design as simple as possible, as given in assumptions), while the rest will reside in different data stores in the same collection of tables.
5.2 Environment

Google Refine’ was used to perform source datasets cleaning as well as MongoDB for storing the new cleaned and integrated dataset with the new model. The choice fell to mentioned technologies/systems due to the existing working knowledge of the both prior the integration process started. This allows for immediate resource savings, such as time. It can be argued that the chosen technologies/systems are not effective for the task specified, therefore it is worth mentioning other existing technologies/systems in case if this project is to be performed once again using different organisation. Some of the alternatives will be enumerated in the Comparisons section. For now a short overview can be given:

- Key-Value Store: (Cassandra, redis, riak)
- Document Store: (MongoDB, Informix)
- Column Famiy Store: (HBase, Dynamo)
- Graph Store: (Neo4j)

5.3 Google Refine

A fast open source tool for cleaning dirty data sets, which allows to import various-delimiter separated values, which can then be transformed using powerful regular expressions as well as some commonly used transformations available as a menu choices. In addition to cleaning, Google Refine can make data analysis possible by presenting desired histograms/bar charts/statistics. Refined data can be exported to save delimited-value files for further integration/analysis.

5.4 MongoDB

MongoDB is a NoSQL system that allows data to be stored in document stores using dynamic flexible schema. The system uses BSON format, which is a flavour of JSON. MongoDB allows different document stores to be indexed, like in RDMS, but the underlying philosophy is made to avoid such relations, so that all the related data is stored in the same document. MongoDB allows to perform a powerful MapReduce procedures over aggregates. Data can be imported into the system in different formats (i.e. csv, tsv, etc.) and schema can be easily changed later if so needed. MongoDB provides consistency and partition-tolerance by default, while this property can be changed to make the whole system operate under availability and partition-tolerance property, so it can be seen that either consistency or availability must be sacrificed, giving raise to either eventual availability or eventual consistency.
## 5.5 New Data Model

<table>
<thead>
<tr>
<th>Preserved (from original Movie Rankings R Us dataset):</th>
</tr>
</thead>
<tbody>
<tr>
<td>• ID - there is a good chance application layer uses this information to analyse/present data (from assumptions)</td>
</tr>
<tr>
<td>• Title_Full</td>
</tr>
<tr>
<td>• Release date</td>
</tr>
<tr>
<td>• Video Release date</td>
</tr>
<tr>
<td>• IMDb URL</td>
</tr>
<tr>
<td>• Unknown</td>
</tr>
<tr>
<td>• Action</td>
</tr>
<tr>
<td>• Adventure</td>
</tr>
<tr>
<td>• Animation</td>
</tr>
<tr>
<td>• Childrens</td>
</tr>
<tr>
<td>• Comedy</td>
</tr>
<tr>
<td>• Crime</td>
</tr>
<tr>
<td>• Documentary</td>
</tr>
<tr>
<td>• Drama</td>
</tr>
<tr>
<td>• Fantasy</td>
</tr>
<tr>
<td>• Film-Noir</td>
</tr>
<tr>
<td>• Horror</td>
</tr>
<tr>
<td>• Musical</td>
</tr>
<tr>
<td>• Mystery</td>
</tr>
<tr>
<td>• Romance</td>
</tr>
<tr>
<td>• Sci-Fi</td>
</tr>
<tr>
<td>• Thriller</td>
</tr>
<tr>
<td>• War</td>
</tr>
<tr>
<td>• Western</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Merged (from MovieKill):</th>
</tr>
</thead>
<tbody>
<tr>
<td>• MovieKill ratings - it is tricky to assume that ratings of MovieKill users will be used in the data processing in the application layer, but this information can be stored in the new dataset as an additional feature</td>
</tr>
<tr>
<td>• MovieKill missing genre - due to the great subjectivity of this issue, Movie Rankings R Us might utilise the missing genres in the application layer.</td>
</tr>
<tr>
<td>• IMDb rating</td>
</tr>
<tr>
<td>• Length minutes</td>
</tr>
<tr>
<td>• Director</td>
</tr>
<tr>
<td>• MPAA</td>
</tr>
<tr>
<td>• Body counts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Additional:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Key - since movies IDs differ between the two original datasets, some way of looking up information between the two must exists. It had been decided to use trimmed movie title as a key for the lookup between the two datasets. It is assumed that there are no two movies with the exactly the same name. (from assumptions)</td>
</tr>
</tbody>
</table>
5.6 New Data Model Affects

Any decision in systems design/outline and basically anything else has consequences for the actions that can be associated with that system. In the case of the new data model, the important information had been fused together with the main (movies) table of the MovieLens dataset, given that the company uses the same tables outline as given by MovieLens dataset (from assumptions). Since no considerable changes were made to the original dataset, except for additional fields, it is expected that the application layer abstractions would still function normally (as long as field names are used instead of indexes to refer to various columns).

If all above is true, then the datasets integration would not have drastic effects on the performance of the application layer, nor the user experience of the possible services provided by the company.

Additional changes will be needed for the application layer to reflect the integration as well as configuration of the connection. Possible combinations will be given later, in the Results section, where possible queries over integrated data will be presented.

It is definitely the case that the new dataset is not ready for partitioning (in case if it had not been partition-friendly prior to the integration), since this feature was not part of the requirement, it had been overlooked during the integration. Wiser would have been to separate out the data that belongs to the Killer Movies, so that it would not interfere and blend together with the original dataset. On the other hand, in turn, it would violate the assumption that the integrated dataset should be as simple as possible.

Resulting dataset is expected to be relatively clean, since it had been referenced with the data that exists in the original dataset and all missing, corrupted, irrelevant data had been left behind (not copied over to the new dataset) The new dataset is representative of the both datasets.

The final dataset, loaded into MongoDB is not normalised, so some normalisation should be performed in order to link movies and their id to relevant information in different tables.

As for the access time of the new dataset, cross-references will be required in order to retrieve information that spans boundaries of one table (i.e. users data, ratings, occupations, info, movies). An example query could be: Show occupations of users that gave rating greater than 3 to the Batman (1989) movie, which will slow down the access time due to the rotations of the hard disk required. A number of normalizations can be performed to reconstruct the virtual schema of the dataset to make such queries (as mentioned previously) cheaper in the sense of time and power.

6 Transformations

This section will give a step-by-step guide to accomplishing the task specified in the beginning of the report.

Preconditions before starting the guide: data must be available and ready to be used, where Movie Rankings R Us dataset contains pipe separated values and
MovieKill dataset contains comma separated value.

### 6.1 Refine

Create new projects

1. Create Project → Choose Files → choose MovieKill dataset → Next → Project name: MovieKill → Create Project

2. Create Project → Choose Files → choose Movie Rankings R Us dataset: u.data → Next → Project name: Awe-X-Movies-Ratings → Deselect Parse next 1 line(s) as column header → Create Project

3. For each column: Edit column → Rename this column → (User ID, Item ID, Rating, Timestamp)

4. Create Project → Choose Files → choose Movie Rankings R Us dataset: u.item → Next → Project name: Awe-X-Movies → Deselect Parse next 1 line(s) as column header → Create Project

5. For each column: Edit column → Rename this column → (ID, Title_Full, Released, Released2, URL, Unknown, Action, Adventure, Animation, Children, Comedy, Crime, Documentary, Drama, Fantasy, Film-Noir, Horror, Musical, Mystery, Romance, Sci-Fi, Thriller, War, Western)

### Cleaning

1. Title_Full column → Edit column → Split into several columns.. → by separator, Separator: \\((.+\))\), RegEx, remove tick from Remove this column → rename Title_Full 1 to Key

2. Key column → Edit cells → Transform →

   ```
   fingerprint {
     replace {
       toLowercase(value)
     }
   }
   ``

   Note: fingerprint command handles non-standard characters with care (Guidry 2015)

3. In MovieKill dataset → Film column → Edit column → Add column based on this column.. → fingerprint(replace(toLowercase(value),','"")) → New column name: Key

4. In Awe-X-Movies dataset → Key field → Edit column → Add column based on this column.. →

   ```
   if (value != NULL, 
    if (cell.cross("MovieKill","Key").cells["Body_Count"] != NULL, 
     cell.cross("MovieKill","Key").cells["Body_Count"][value][0], 
     )'\'), 
   
   → New column name: Body counts
5. In Awe-X-Movies dataset → Key field → Edit column → Add column based on this column. → 

```javascript
if (value != NULL, 
    if (cell.cross("MovieKill", "Key") . cells["MPAA_Rating"] != NULL, 
        cell.cross("MovieKill", "Key") . cells["MPAA_Rating"] . value[0],
    ") ,
"" ) ,

→ New column name: MPAA
```

6. In Awe-X-Movies dataset → Key field → Edit column → Add column based on this column. → 

```javascript
if (value != NULL, 
    if (cell.cross("MovieKill", "Key") . cells["Director"] != NULL, 
        cell.cross("MovieKill", "Key") . cells["Director"] . value[0],
    ") ,
"" ) ,

→ New column name: Director
```

7. In Awe-X-Movies dataset → Key field → Edit column → Add column based on this column. → 

```javascript
if (value != NULL, 
    if (cell.cross("MovieKill", "Key") . cells["LengthMinutes"] != NULL, 
        cell.cross("MovieKill", "Key") . cells["LengthMinutes"] . value[0],
    ") ,
"" ) ,

→ New column name: Length minutes
```

8. In Awe-X-Movies dataset → Key field → Edit column → Add column based on this column. → 

```javascript
if (value != NULL, 
    if (cell.cross("MovieKill", "Key") . cells["IMDB_Rating"] != NULL, 
        cell.cross("MovieKill", "Key") . cells["IMDB_Rating"] . value[0],
    ") ,
"" ) ,

→ New column name: IMDb rating
```

9. In Awe-X-Movies dataset → Key field → Edit column → Add column based on this column. → 

```javascript
if (value != NULL, 
    if (cell.cross("MovieKill", "Key") . cells["Genre"] != NULL, 
        join(
            with (a, forEach(a, v, if (cell[v] != NULL, 
                if (cell[v] . value == '0',
                    v, 
                ") ,
"" ) ,
"" ) ,
                ") ,
"" ) ,

→ New column name: MovieKill missing genre
```

10. In Awe-X-Movies dataset → ID field → Edit column → Add column based on this column. → 

```javascript
sum(
    forEach(
        cell.cross("Awe-X-Movies-Ratings", "Item_ID") . cells["Rating"] . value, 
        n, 
        toNumber(n)
    ) / cell.cross("Awe-X-Movies-Ratings", "Item_ID") . cells["Rating"] . length()

```

7
New column name: Internal ratings average (integration time)

6.2 Export/Import

1. Export: The new dataset has to be exported from Google Refine using export function and export format must be of comma separated value (.csv)

2. Import:
   ```shell
   # mongoimport --db <...> -u <...> -p
   --host mongo-server-1
   --authenticationDatabase <...>
   --collection <...>
   --type csv
   --headerline
   --file <...>.csv
   ```
   If successful, the system will return how many objects were imported after executing above command. (imported N objects)

   The import command is performed on Info, Ratings, Occupations and Users in order to preserve the virtual schema that is somehow related to the original schema used by relational system used in the past.

7 Results

7.1 Analysis

Queries:

1. Get 5 movies with the most body count.
   Expected results: An array of 5 movies. Returned elements will contain ID and Title w/ year. Items will be sorted by # of bodies.

   Query:
   ```javascript
   db. < ... > .aggregate({
     $group: {
       id: {
         id: "$ID",
         name: "$Title_Full"
       },
       total: {
         $sum: "$Body counts"
       }
     },
     $sort: {
       total: -1
     },
     $limit: 5
   })
   ```

   Result:
   ```json
   [ {
     "id": { "id": 313, "name": "Titanic (1997)" }, "total": 307 }
   , { "id": { "id": 651, "name": "Glory (1989)" }, "total": 258 }
   , { "id": { "id": 271, "name": "Starship Troopers (1997)" }, "total": 256 }
   , { "id": { "id": 205, "name": "Patton (1970)" }, "total": 199 }
   , { "id": { "id": 22, "name": "Braveheart (1995)" }, "total": 184 }
   ]
   ```
2. Present MPAA categories and how many bodies are there per every category

**Expected results:** An array of items of size of MPAA categories available in the system together with the count of bodies for each category.

**Query:**
```javascript
db. < . . > . aggregate([{
  $group: {
    _id: "$MPAA",
    total: {
      $sum: "$Body counts"
    }
  }
}])
```

**Result:**
```
[{
  "id": "X",
  "total": 38
},
{
  "id": "Approved",
  "total": 67
},
{
  "id": "GP",
  "total": 199
},
{
  "id": "R",
  "total": 2941
},
{
  "id": "PG-13",
  "total": 1046
},
{
  "id": "PG",
  "total": 333
},
{
  "id": "",
  "total": 0
}]
```

3. Get body counts per director

**Expected results:** All available directors in the system together with the # of bodies in films they have directed

**Query:**
```javascript
db. < . . > . aggregate({
  $group: {
    _id: "$Director",
    total: {
      $sum: "$Body counts"
    }
  },
  $sort: {
    total: -1
  }
})
```

**Result:**
```
[{
  "id": "James Cameron",
  "total": 469
},
{
  "id": "Edward Zwick",
  "total": 258
},
{
  "id": "Oliver Stone",
  "total": 258
},
{
  "id": "Paul Verhoeven",
  "total": 256
},
{
  "id": "Steven Spielberg",
  "total": 230
},
{
  "id": "Robert Rodriguez",
  "total": 202
},
{
  "id": "Franklin J. Schaffner",
  "total": 199
},
{
  "id": "Mel Gibson",
  "total": 184
},
{
  "id": "Akira Kurosawa",
  "total": 182
},
{
  "id": "David Lean",
  "total": 180
},
{
  "id": "Katsuhiro Ohtomo",
  "total": 119
},
{
  "id": "John Milius",
  "total": 114
},
{
  "id": "Jim Abrahams",
  "total": 114
},
{
  "id": "Roland Emmerich",
  "total": 99
},
{
  "id": "Frank Marshall",
  "total": 92
},
{
  "id": "John McTiernan",
  "total": 91
},
{
  "id": "John Woo",
  "total": 81
},
{
  "id": "Tim Burton",
  "total": 74
},
{
  "id": "Peter Jackson",
  "total": 72
},
{
  "id": "Stanley Kubrick",
  "total": 67
}]
```

4. Get all the movies in the system sorted by their length (in minutes) Note: Change has to be made to the field that maps Length minutes to the value.

**Prior execution:**
```javascript
db. < . . > . update({}, {
  $rename: {
    "Length minutes": "Length_minutes"
  }
})
```

**Expected results:**
```javascript
[{
  "id": "Movie 1",
  "Length_minutes": 120
},
{
  "id": "Movie 2",
  "Length_minutes": 130
},
{
  "id": "Movie 3",
  "Length_minutes": 140
},
{
  "id": "Movie 4",
  "Length_minutes": 150
},
{
  "id": "Movie 5",
  "Length_minutes": 160
},
{
  "id": "Movie 6",
  "Length_minutes": 170
},
{
  "id": "Movie 7",
  "Length_minutes": 180
},
{
  "id": "Movie 8",
  "Length_minutes": 190
},
{
  "id": "Movie 9",
  "Length_minutes": 200
},
{
  "id": "Movie 10",
  "Length_minutes": 210
}]
```
(a) Field name Length minutes changed to Length_minutes
(b) Array of movies sorted by their time in minutes

Query:
```javascript
db. aggregate(

  { $match: { Length_minutes: { $gt: 0 } } },
  { $group: { _id: "$Title_Full", minQuantity: { $min: "$Length_minutes" } } },
  { $sort: { minQuantity: -1 } }
)
```

Result:
```javascript
```

Type "it" for more

5. Get 5 movies that are action, crime and drama and sort by their length

Expected results: Array of 5 movies sorted by their length that are Action, Crime and Drama (Expecting Batman..)

Query:
```javascript
db. aggregate(

  { $match: { $and: [{ "Action": 1 }, { "Crime": 1 }, { "Drama": 1 }] } },
  { $group: { _id: { id: "$Action", name: "$Title_Full" }, total: { $sum: "$Length_minutes" } } },
  { $sort: { total: -1 } },
  { $limit: 5 }
)
```
6. Remove duplicates

(a) Find duplicates
(b) Remove duplicates

Query:

```javascript
var duplicates = db. . . . aggregate([{
  $group: {
    _id: {
      release: "$Release",
      film_key: "$FilmKey",
      url: "$URL"
    },
    uniqueIds: {
      $addToSet: "$id"
    },
    count: {
      $sum: 1
    }
  },
  $match: {
    count: {
      $gt: 1
    }
  }
}])
duplicates.forEach(function(myDoc) {
  db. . . . remove({
    Key: myDoc._id.film_key
  });
});
```

Result:
No more duplicates in the system

8 Discussion

From Transformations section it can be seen that the task can be performed in relatively few simple steps. The downside of this is that the person doing the data refinements and datasets integrations should be familiar with the syntax and semantics of the particular technology he/she is using. That is the main rationale behind the decision to use the technologies/systems that were used to perform this task.

8.1 Comparisons

8.1.1 Data Model Affects

Described above technique for the integration of two datasets applies primarily to the NoSQL systems. If the similar steps to be performed in order to integrate different datasets in for example a relational database, more attention will have to be paid to the design of the overall schema, where such details as primary keys, normal form representations, foreign keys, and much more will have to be desided upon before the actual integration process starts and no later changes will
be permitted on the schema in place, because it would put unnecessary pressure on the underlying data. As for the NoSQL systems, schema is not set in stone and can be almost ad-hoc. From the integration steps described above, the reader may notice that there was a slight error made in the virtual schema, which was discovered later, when querying the data. This error had to do with the naming of the fields of a table, where space character in the MongoDB command to match a particular field could not be performed. A decision to change virtual schema had been made and the mapping between field name and the data in that column had been changed in a matter of milliseconds. This would not be feasible in the traditional relational database system.

As for the database normalisation, it seems that NoSQL greets all-in-one design patterns, which is a no-go case for the traditional database systems. It may be argued that normalization is a good design pattern since it makes the overall structure human-readable and relatively lightweight, whereas all-in-one pattern simply flushes everything (relevant) into one table, which definitely makes the process much faster at the beginning, but may become cumbersome after a while. In addition, all-in-one design would slow down the traditional relational model when performing queries on the underlying dataset because of the additional in-line processing that will be necessary. This can be viewed as a consequence of the philosophy of the NoSQL systems, which allow dirty datasets, while relational systems will explicitly require definition of all datatypes for every single field (does not apply to the SQLite because of the string representation and additional in-memory conversions).

### 8.1.2 NoSQL systems

Popular NoSQL systems today can be categorised into: Key-Value Stores, Document Stores, Column Family Stores and Graph-Based Databases. MongoDB, which is used for this project belongs to the Document Store category.

For the comparison reasons, it can be told that databases that belong to the Key-Value Stores category are designed to provide high distribution of data by the means of consistency trees and hash values to verify the consistency of data between nodes. Consistency is provided by some low-level implementation of a consensus protocol. In a traditional Key-Value databases all nodes are supposed to be equal and no master is being elected to impose any control over other nodes in the system. In Key-Value Stores consistency is sacrificed due to the high availability.

The reader is expected to get some perception of Document Stores used extensively in this report. As a side note, the 'document' word can be explained more here to stress on the importance of this word for the system. A document can be viewed as a bucket that contains all the context-relevant information. Data in a document can be nested, where nested pieces of information can either be placed directly in the same table of referenced by using some variation of a 'foreign key'.

Column Family Stores can be seen as a collection of buckets (families) of column within a huge table. each cell can have multiple versions of data based on the timestamp. Column Family Stores can be partitioned and distributed over different nodes. The model ensures consistency of data by using some external mechanism for the synchronisation, like Chubby (Google).
Graph-Based Databases uses graph (collection of vertices and edges) to work with the underlying data, where vertices represent data items and edges represent relationships between data items. Graph-Based Databases provide and efficient and native storage and processing opportunities.

From a short description it can be noted that different systems differ in their representation approaches and distribution guarantees that they provide due to the underlying philosophical models. It is therefore important to make a decision for which system to use based on the requirements rather than on the popularity.

9 Conclusion

Overall, it can be said that NoSQL datasets are more flexible with the properties that the BigData exhibits, such as Volume - partitioning can make huge volumes of data acceptable; Velocity - partitioning with eventual consistency allows for infinite streams of data to be stored on the fly; Variety - no definite schema allows mixed data types storage as well as mixed-context data; Veracity - a consequence of the variety of the data makes it not possible to predict whether stored data is relevant or of the same context.

This particular task exploited mostly the Variety of the data and none of the other properties. It would be interesting to see how other properties will come to the surface once, for example, dataset would increase considerable or the stream of data generation would be consistent.
References

URL: http://files.grouplens.org/datasets/movielens/ml-100k-README.txt

URL: https://github.com/OpenRefine/OpenRefine/wiki

URL: http://blog.ouseful.info/2011/05/06/merging-datasets-with-common-columns-in-google-refine/

Mocialov, B. (2015), ‘Personal space for the f21bd’.
URL: http://www2.macs.hw.ac.uk/ bm4/BDM/CW/1/

Yan, F. (2012), ‘How can i rename a field for all documents in mongodb?’.
URL: http://stackoverflow.com/a/9254495/1276374