Referential integrity and dependencies between documents in a document oriented database

Kalin Georgiev

***Abstract*—Reliability of foreign keys, which is natural in relational databases, requires additional efforts when working with non-relational databases, as non-relational database management systems generally don't support foreign key constraints due to their distributed nature. Referential integrity is an important property whenever documents need to refer to each other, which is the common case. This work discusses an implementation of a verification approach which makes use of the MapReduce programming model, in order to detect incorrect references in document oriented databases that may be caused by errors in the program code or incomplete transactions. Furthermore, the method can be applied for the verification of more complex dependencies between documents, such that bind aggregated values from certain sets of documents with the values of documents referred by them.**

***Index Terms*—Referential integrity, noSQL, Non-relational databases, MapReduce, MongoDB.**

I. Introduction

NON-RELATIONAL database management systems (NRDBMS), unlike their counterpart - the relational database management systems (RDBMS), commonly lack built in support for foreign key constraints. Therefore, ensuring referential integrity of data, which is an important property whenever records in different collections refer to each other, requires additional efforts, which are commonly implemented on the application level.

Referential integrity as a database property requires that records, which refer to each other, use correct references. These "references" are usually values of the referred document's identifying (primary) key. The problem of ensuring and verifying referential integrity is relevant to any data structure, regardless of whether it's represented in a relational database, a non-relational database, or in memory, for example.

Relational database management systems (RDMS) ensure by default that every update transaction maintains the referential integrity of the data. Non-relational DBMS, however, reply on flat data structures and distributed architecture. Under a distributed architecture, verifying the correctness of a foreign key can not be achieved without access to other nodes or some other data shared across the nodes, which would introduce a performance drawback.

This work addresses several problems related to the use of foreign keys and some semantic relationships between documents that may be in the same collection or spread across different collections: (1) duplicate primary keys, (2) references to non-existent primary keys, and (3) various properties of aggregated values. The discussed implementation makes use of the MapReduce [1] programming model with its specific implementation built in the NRDBMS MongoDB [2]. However, the approach behind the implementation is fairly general an can be applied to other MapReduce frameworks with little change.

Manuscript received December 20, 2012; This work was supported by the European Social Fund through the Human Resource Development Operational Programme under contract BG051PO001-3.3.06-0052 (2012/2014).

K. Georgiev is with the Faculty of Mathematics and Informatics, Sofia university, Sofia, Bulgaria, e-mail: kalin@fmi.uni-sofia.bg

Used Notations

All schema-less databases admit heterogeneous collections. MnogoDB, as a non-relational DBMS, is not an exception. Documents in MnogoDB are represented in the JSON (Java Script Object Notation) format. The format is based on the JavaScript object literal, which allows for arbitrary fields and nested structures. MongoDB does not enforce strict typing of the field values, not even does it ensure that all documents have the same fields. However, we will assume for simplicity that all documents which need to be verified do contain the fields that participate in the correctness property and that the values of these fields have the required types. We will assume that a mechanism is available for the filtering or the specific handling of documents which don't satisfy this requirement.

Therefore, we will adopt a classical notation to denote database documents and their fields. Essentially, we will enforce a partial schema on the collections of interest by requiring that each document contains at least a set of predefined fields and that their values have proper types. If `d` is a document and `l` is field label, by `d[l]` we will refer to the value of the filed `l` in the document `d`.

II. Brief Introduction to MapReduce

MapReduce [1] is a programming model named after the classical map and reduce functions known from functional programming, although the semantics of the map and reduce functions in MapReduce have deviated from their original forms. The model employes a functional style for the implementation of embarrassingly parallel problems [4] over large data sets.

There are different variants and implementations of the MapReduce model found in various databases and data frameworks in general. In this section we introduce a basic, limited model of the process, which will allow us to lay down our application of MapReduce, without dealing with intricate technical details and implementation differences.

Let us have a collection `A` of documents. Let $\mathcal{K}$ be a set of keys and $\mathcal{P}$ be some arbitrary value type. Then a map function is any function of the form $map : \mathcal{A} \to 2^{\mathcal{K}\times\mathcal{P}}$.

DOI: 10.5176/ 2251-3043_2.4.205

The map function is an isolated part of the computational process focused at processing individual documents from the collection, without any information about the rest of the documents. The map function is free to do anything with the document's data and produce a number of values based on it. The process is called *emitting* and the map function can emit any number of values, including 0. A simple example for a map function will be the projection (in relational-algebraic terms) of a field or fields from a record or the calculation of some simple formula based on one or more record fields: for example, emit the `age` field from all documents in the `people` collection.

The outputs of all map processes $R = \bigcup\{map(d)|d \in A\}$ are grouped by the key emitted by map to form a number of classes $[d] = \{data|(k, data) \in R\}$ for every different key emitted by map.

Each such class is then passed to a reduce process - $reduce : \mathcal{K} \times 2^{\mathcal{P}} \rightarrow \mathcal{F}$, where $\mathcal{F}$ if the type of the "final" result. The reduce process is what "aggregates" the results from the individual map processes, which are first grouped by the key chosen by the map function.

As a very simple example, imagine that we have a collection of `people` of both sexes. Let the sex of the individual person be identified by the `sex` field and their age - by the `age` field. Image that we need the average age of all males and all females in our database.

We will use a map function which emits the age of each person and groups ages by their sex, and then a reduce function will just calculate the average value of the sets of values produced by map:

```
function map ()
  {emit(this.sex,this.age);}

function reduce (key,emits)
  {return average(emits);}
```

where `average(emits)` is the average of the numbers in the `ages` array. Note that MongoDB provides a reference to the document as the variable `this` in the `map` function, instead as a function parameter. In our example, the map function will output a number of tuples `sex:age`, one for each person. The MapReduce framework will then group the age elements in two classes `[male]` and `[female]` containing the ages of all males and all females respectively. Finally, the reduce function will be executed twice with the arguments `reduce(male,[male])` and `reduce(female,[female])`. The final result consists of two tuples `(male,am)` and `(female,af)`, where `am` is the average age of all males, and `af` is the average age of all females in our collection.

Re-reduce

Because the sets which the reduce function has to process may become large and because of other technical considerations, there is another (optional) step in the MapReduce process - re-reduce. The classes `[k]` may be broken down into a number of subsets. Different reduce processes then process these subsets and the lists of results of these processes are passed to a re-reduce function. In other words, reduce performs a "partial" aggregation which is completed by re-reduce. However, considering this stage is not essential for our needs as we will predominantly use commutative and associative operations in our reduce functions, so that they can serve as re-reduce functions as well.

III. Verification of One-to-many Relations

Let us have two collections `A` and `B` and their corresponding fields `kA` and `kB`, which we will consider "keys". Similarly to the foreign keys in relational databases, we can imagine that `kA` is the "primary" key for the records in `A` and `kB` contains references to that primary key. Then the goal for verifying referential integrity between the two collections is to check whether `dB[kB]` for each $dB \in B$ can be found in exactly one $dA \in A$ as the value of $dA[kA]$. In other words, that $\forall dB \in B : \exists! dA \in A$ such that $dA[kA] = dB[kB]$. This expresses the "one-to-many" relation between the records in `A` and `B`, where the records in `B` refer to records in `A`.

The approach that we will use to verify the integrity of a one-to-many relation is similar to an approach used to implement (or "simulate") join operations in non-relational databases using MapReduce [3]. The essence of the approach is to use the map function to emit "counters" for every key in `A` and `B`, grouping all counters by the value of the foreign key and then letting the reduce function sum up all counters. In this way, the resulting data set can be used to detect mismatches of the number of records in `A` and `B` that refer to each other.

Let

```
function mapBase(){
  emit (this["kA"],{"sums":[1,0]});
}
```

be the map function applied to the records in `A`. Similarly

```
function mapRef(){
  emit (this["kB"],{"sums":[0,1]});
}
```

is applied to the records in `B`. And finally

```
function reduce (key,emits){
  var sums = [0,0];
  for (var i in emits){
    sums[0] += emits[i].sums[0];
    sums[1] += emits[i].sums[1];
  }
  return {"sums":sums};
}
```

will sum the components of all tuples that are emitted under the same key. Invoking the MapReduce process in MongoDB involves the following expressions:

```
db[A].
   mapReduce (mapBase,
              reduce,
              {out: {reduce: "result"}});
db[B].
   mapReduce (mapRef,
```

```
                reduce,
                {out: {reduce: "result"}});
```

As a result, there will be a tuple $t_{key} = (key, count_A, count_B)$ in the collection `result` for every unique key seen in `A` or `B`. The tuples contain the number of occurrences of key as values of `kA` and `kB` respectively. The referential correctness with respect to each key can be verified by examining the values of $count_A$.

- $count_A = 0$ will suggest that key is present in B but there are no occurrences in A
- $count_A > 1$ will suggest that key is not unique in A

Therefore, by filtering all items in the result set for which $count_A \neq 1$ a list can be obtained of all incorrect references in `B`.

Example

For example, let't have the following data in `A` and `B` (using the JSON notation):

```
A:{ {"key":"ka"},
    {"key":"ka"},
    {"key":"kb"}}
B:{ {"key_a":"ka"},
    {"key_a":"kb"},
    {"key_a":"kc"}}
```

The outputs of `mapBase` and `mapRef` will be

```
{{"ka":{"sums":[1,0]},
 {"ka":{"sums":[1,0]},
 {"kb":{"sums":[1,0]}}
{{"ka":{"sums":[0,1]},
 {"kb":{"sums":[0,1]},
 {"kc":{"sums":[0,1]}}
```

The reduce processes will be executed with the following parameters:

```
reduce ("ka",{ {"sums":[1,0]},
               {"sums":[1,0]},
               {"sums":[0,1]}})
    = [2,1]

reduce ("kb",{ {"sums":[1,0]},
               {"sums":[0,1]}})
    = [1,1]

reduce ("kc",{ {"sums":[0,1]}})
    = [0,1]
```

Which reveals issues with `ka` and `kc`.

IV. Many-to-many Relations

While relational database management systems only support one-to-many relations, most non-relational database engines do not suffer from this limitation thanks to the support of list properties. In our setup, a single record in `B` can easily refer to a number of records in `A` and vice versa.

A common use case for such a model is the Books and Authors example, where each book can have multiple authors and a single author may have written multiple books. A classical relational implementation of this model would be to introduce a junction table (BooksAuthors) and use join queries to extract data.

If this example is to be implemented in a document database, the documents in the Authors collection could contain lists of book identifiers and in the same time, the documents in the Books collection could contain lists of author identifiers. This structure introduces data redundancy, however, this denormalization approach is commonly used for query optimizations in databases [5].

Let `dA[kA]` and `dB[kB]` be unique record identifiers and there are two fields `refB` in each `dA` and `refA` in each `dB` containing list of keys in `B` and `A` respectively. Then, in this case, the goal of verifying referential integrity is to check whether every member of every `dA[refB]` is a valid key in `B` and every member of every `dB[refA]` is a valid key in `A`.

Our approach in this case is similar: we use map to count all keys that refer to records in `A` and `B`, but we emit a number of values this time – one for each element of the corresponding reference lists.

```
function mapBase(){
  emit (this["kA"],{"sums":[1,0]});
  for (var i in this["refB"]{
    emit (this["refB"][i],
          {"sums":[0,1]});
  }

}
```

Similarly

```
function mapRef(){
  emit (this["kB"],{"sums":[1,0]});
  for (var i in this["refA"]{
    emit (this["refA"][i],
          {"sums":[0,1]});
  }

}
```

Note that we have reserved the first member of the tuples for the primary key counter, while the second member counts the keys used as references. The reduce function does not need changes for this setup.

```
function reduce (key,emits){
  var sums = [0,0];
  for (var i in emits){
    sums[0] += emits[i].sums[0];
    sums[1] += emits[i].sums[1];
  }
  return {"sums":sums};
}
```

Again, the output of the process is a set of tuples $t_{key} = (key, count_{key}, count_{ref})$ and examining the values of the tuples will reveal the following inconsistencies:

- $count_{key} = 0$ will suggest that key is present as a reference in some record, but it does not exist as a primary key in the referenced collection
- $count_{key} > 1$ will suggest that key is not an unique key in the corresponding collection

Finally, please note that similar verifications can be performed over two types of records in the same collection in the schema-less databases. However, the map function needs to be extended to differentiate the type of records by using a distinction rule in order to emit the proper key field (for example "has the value `person` in the field `type`").

V. Binding Aggregates and Fields

The resulting set produced by the map and reduce functions of the previous section contains tuples, the elements of which count the occurrences of every primary key and every key used as a reference. The reduce function is very simple: it sums a number of 1s. This is a very limited use of the capabilities of the reduce function, which is capable of accumulating the data emitted by the map function in a variety of ways, the most obvious of which is to use some commutative and associative binary operator.

If, together with the key counters, the map function emits the values of some other field, the reduce function could afterwards aggregate these values using a commutative and associative operator. The values of these aggregates can then become parameters for various predicates expressing custom correctness properties which are specific for the use case. For example, verify that "total amount" in the records of the "expenses" collection is equal to the the sum of all "amount" fields in the records of the "invoices" collection which refer to the expense record.

Let there be two more fields - `sum` in `A` and `item` in `B` in our one-to-many setup. The goal is to verify that for each $dA \in A$

$$dA[sum] = \sum\{dB[item] | \quad dB \in B \wedge dB[keyB] = dA[keyA]\}$$

where $\sum$ is any commutative and associative binary operator defined for the type of the items in B (not necessarily a sum). In other words, verify that the value of every `dA[sum]` holds the correct sum of all `dB[item]` for the documents in `B` which refer to `dA`.

The map function is extended to emit the values of the elements of the sum in order to allow the reduce function to accumulate those values and produce the total sum, again for every unique key in `A`:

```
function mapBase(){
  emit (this["kA"],
        {"sums":[1,
                 0,
                 this["sum"],
                 0]});
}

function mapRef(){
  emit (this["kB"],
        {"sums":[0,
                 1,
                 0,
                 this["item"]]});
}
```

Again, we have reserved the first member of the tuple for the primary key counter, while the second member counts the keys used as references. The third member holds the expected value for the total sum and the fourth - the individual values of each item, which are summed by the reduce function. Note that 0 (zero) in the third and fourth element is not necessarily the number 0, it's the zero element of the corresponding type with respect to the operator $\sum$.

```
function reduce (key,emits){
  var sums = [0,0];
  for (var i in emits){
    for (var j = 0; j < 4; j++){
      sums[j] += emits[i].sums[j];
    }
  }
  return {"sums":sums};
}
```

Examining the resulting tuples $t_{key} = (key, count_A, count_B, val_A, sum_B)$ is analogous to the one-to-many setup with the additional step of verifying that $val_A = sum_B$.

Note that in this way the method can be applied to verify a number of different aggregates by letting map emit additional members of its tuples and summing all of them in reduce.

VI. Conclusion

When the database layer of a software system doesn't provide mechanisms for ensuring the referential integrity of the data, issues with the program code and incomplete transactions can cause significant issues for the system. In theory, when the correctness of the program code is fully verified and tested, cases of inconsistency should be reduced to a minimum. However, every complex system would benefit from a mechanism for the verification of its data, in order to handle undetected issues with the program logic or failures of other kinds, which cause update operations to create inconsistencies in the data.

The approach proposed in this work allows for a post factum detection of the following classes of issues:

- duplicate primary keys;
- references to non-existent primary keys;
- various properties binding values from one collection with aggregated values from another collection.

The proposed mechanism would only report the existence of such issues, without any suggestions about the cause for the generation of inconsistent records. However, if the verification is run periodically, it can be an important instrument for the early detection of issues with the code and their timely resolution, before the issues are exhibited broadly across the system.

Experiments and performance

The implementation discussed in this work has been applied for the post-factum detection of foreign key inconsistencies in the SocialBook web application (www.livemargin.com). SocialBook is a book publishing platform with a variety of multimedia and social features. There are a number of interrelated collections in the application's database that reference each other – books, authors, readers, comments, replies, and others. This method has been applied for the detection and removal of "leaking" data, such as orphan comments, replies, and books, which have been produced due to incorrect application logic or failed transactions.

The experimental results hint towards a liner complexity of the method, which is supported by the theory behind MapReduce. However, a precise analysis and a performance evaluation of the method is still to be completed. This is a subject of the author's continuing research in the area.

ACKNOWLEDGMENT

This work was supported by the European Social Fund through the Human Resource Development Operational Programme under contract BG051PO001-3.3.06-0052 (2012/2014).

REFERENCES

[1] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters.*, CACM, 2008, 51(1), pp. 107-113
[2] *MongoDB online documentation*, Available: http://www.mongodb.org/display/DOCS
[3] F. N. Afrati and J. D. Ullman, *Optimizing Joins in a Map-Reduce Environment. Technical Report.*, EDBT '10 Proceedings of the 13th International Conference on Extending Database Technology (March 22 - 26, 2010, Lausanne, Switzerland), pp. 99-110
[4] I. Foster, *Designing and Building Parallel Programs. Section 1.4.4.*, Addison-Wesley, Boston, 1995
[5] S. K. Shin, G. L. Sanders, *Denormalization strategies for data retrieval from data warehouses.*, DECIS SUPPORT SYST, 2006, 42(1), pp. 267-282

Kalin Georgiev works at the Faculty of Mathematics and Informatics of Sofia university as an assistant professor where he is involved in various computer science courses (C++ programming, functional programming, logical programming, computability theory, project management, and others) and a number of technological courses. He is a part of several academic research projects while with Sofia University and The Bulgarian Academy of Sceinces. Presently, Kalin participates in research activities in the areas of mathematical psychology and formal verification methods. In parallel, Kalin is involved in the management of several commercial projects applying moder mobile, web, and cloud development technologies.