The Implementation of Relational Algebra Operators via Map and Reduce Functions

by

Trever Anderson
Department of Computer Science
College of Information Science and Technology
University of Nebraska at Omaha
Omaha, NE 68182-0500
402-210-9298
treveranderson@unomaha.edu

For

CSCI 8390, Advanced Topics in Data Base Management

Fall, 2019

Contents

1 Introduction

The objective of this project is to implement the basic relational algebra (RA) operators with map and reduce functions. By implementing RA operators in terms of map and reduce, the querying power of RA can be extended to map and reduce systems. This objective requires a simulated map and reduce system, a schema, and good examples that show the compositional power of the map and reduce equivalent operators. The simulated map and reduce system is capable of invoking a map function, performing grouping on the intermediate keys, and invoking a reduce function on those grouped keys to obtain the final return value. This return value can be sent back through the simulated system to support operator composition. A simple schema, which was introduced in report 5, and six example queries were devised to show off the capabilities of the operators.

2 Methodology

Each RA operator's equivalent function returns a generalized map and reduce function as output. This is necessary because RA operators are general with respect to their arguments. For example, the select operator accepts a condition θ and a set of tuples and returns the tuples that pass θ. Therefore, the map function equivalent to select should also accept a condition θ. To fully implement select, the map function should contain the logic to filter the tuples with respect to θ and the reduce function should be an identity function. This full implementation that contains map and reduce functions as output is the semantic equivalent to select for a given θ. The other operators were implemented with the same conceptual underpinning. To discuss this idea in detail, we will walk through the implementation of each RA operator. We will also walk through the implementation of the map and reduce system along with an example query. The language of implementation is Clojure.

3 Records

Records are read in from CSV files where each CSV file represents a table. The schema was given in report 5. Tuples are represented in memory as Clojure maps. Function names and variable names appear in *italics*.

4 Implementations

4.1 Select

```
1.   (defn reduce-identity [key vals]
2.       [key vals])
```

```
3.    (defn select [cond-func]
4.       {:map (fn [record]
5.           (if (cond-func record)
6.                ["passed" record]
7.                ["failed" [(keyword (record :table))
8.                       (record :offset)]]))
9.         :reduce reduce-identity})
```

All of the operators return a map with two keys. The keys are named map and reduce and the values are map and reduce functions respectively. Lines 4 and 9 show the declaration of these key names respectively. Tuples or records are maps and each map function accepts a record. Select is implemented in the map function as the reduce side contains the identity function which simply returns its arguments. The map side implementation accepts some condition function on line 3 which is passed in from the query, which is also written in Clojure (see Figure 1). This condition function is called on line 5 and passed the record. If the record passes the condition, then the return value on line 6 is a vector with two entries, "passed" and record respectively. If the record does not pass the condition then the return value is on line 7 which is a vector with two entries, "failed" and another vector with two entries. The records will be aggregated on the "passed" or "failed" values as those are the first entries in the return result.

4.2 Project

```
1.    (defn project [attr-keys gk-key-func]
2.       {:map (fn [record]
3.               [(gk-key-func record)
4.                (select-keys record attr-keys)])
5.        :reduce reduce-identity})
```

The project operator accepts a set of attribute keys and a function that takes a record and returns a group by structure. The first entry in the resultant vector is returned from a function on line 3. Thus, the caller of *project* can control grouping with a supplied function. The map function performs the projection logic on Line 4 where the attribute keys are selected from the record, and a subset of the record map is returned as the second entry in the resultant vector.

4.3 Cartesian Product

```
1.    (defn cartesian-product [group-keys reduce-gk]
2.       {:map (fn [record]
3.               [(if (some? group-keys)
4.               (mapv record (sort group-keys))
5.               "default_key") record])
6.        :reduce (fn [key vals]
7.               (let [groups (group-by (keyword reduce-gk) vals)
```

```
8.                    [_ first-table] (first groups)
9.                    [_ second-table] (second groups)]
10.             [key (map (fn [r]
11.             (combine-record r)) (relate-all first-table
12.             second-table))]))})
```

The implementation for *cartesian-product* is more involved and requires the implementation of a function that combines records and a function that relates tuples in cartesian product fashion. *combine-record* accepts two records and combines them into one such that intersecting attributes are renamed with a .1 and .2 appended to the name while non intersecting attribute names are preserved. *relate-all* is a function that accepts two collections of tuples and performs cartesian product logic. I decided to omit these functions for brevity as their implementations are somewhat long and english descriptions can serve in place of their behavior.
cartesian-product accepts a set of grouping keys and a key to use when grouping on the reduce side. Line 2 shows the start of the map implementation and line 3 checks if a set of group keys were given. If group keys were given, then the grouping key is a vector of values obtained from the record on line 4. Otherwise, the grouping key is the default key. The second entry in the resultant vector is the record on line 5.
The reduce function starts on line 6 and accepts an intermediate key and the associated value collection as inputs. The values are grouped on line 7 by passing the reduce group by key to the group-by Clojure function. The groups result is then split out into two relations on lines 8 and 9 respectively. The cartesian product is then computed on these groups, rather than the entire value list. This separation needs to be done because computing the cartesian product on the entire value collection of tuples would result in (1).

$$(R1XR1)\bigcup(R2XR2)\bigcup(R1XR2)\bigcup(R2XR1) = (R1\bigcup R2)X(R1\bigcup R2) \tag{1}$$

(1) is wrong because it contains duplicated records which cannot be filtered out by only including records from different tables. Thus, the value collection must be grouped on some attribute key which can separate the collection into two relations. The output on line 10 is then a vector with the given aggregation key as the first entry and a vector of records as the second entry.

4.4 Aggregation with grouping

```
1.    (defn agg-group [group-keys agg-func-map]
2.       {:map (fn [record]
3.           [(if (some? group-keys)
4.           (mapv record (sort group-keys))
5.           (record :table)) record])
6.        :reduce (fn [key vals]
```

```
7.          [key (into (array-map) (map (fn [[k af]]
8.                                [k (af (map k vals))])
9.                           agg-func-map))])})
```

Aggregation with grouping takes a set of grouping keys and a map of aggregate functions as input on line 1. Line 2 is the start of the map function. Line 3 checks if group keys were given to *agg-group*. If true, line 4 shows the first entry in the resultant vector is a vector of values from the input record. If false, line 5 shows that the first entry will be the table from the record. The second entry in the resultant vector is the record. Thus, the grouping logic is controlled by the set of grouping keys passed in as *group-keys*.
The reduce function starts on line 6 and accepts an intermediate key and value collection as inputs. The resultant vector on line 7 has two entries, with the first being the given intermediate key and the second is the result of the aggregation functions applied over the records in the value list.

4.5 Join

```
1.(defn join [sep-key join-cond-func rel1 rel2 out-func]
2.(let [{cp-mp :map cp-rd :reduce} (ra/cartesian-product nil sep-key)
3.      {s-mp :map s-rd :reduce} (ra/select (fn [r](join-cond-func
4.                                            r)))]
5.     (out-func(model/map-reduce(model/grab-records(model/map-reduce
6.                                      [rel1 rel2]
7.                                      cp-mp cp-rd))
8.                                      s-mp s-rd))))
```

The code for this implementation was difficult to format, and the author apologizes for this inconvenience. The implementation of join relies on the implementations of *select* and *cartesian product*, which have been discussed. On line 1, *join* accepts a key to separate the relations on the reduce side in *cartesian-product*, a join condition function to pass to *select*, two data relations, and a function to control the output shape of the data. *cartesian-product* is invoked on line 2 and the map and reduce functions are destructured in *cp-mp* and *cp-rd*. The select function is invoked on line 3 and the map and reduce functions are destructured into *s-mp* and *s-rd*. The two sets of map and reduce functions are then passed to map-reduce between lines 5 and 8. The first call to map-reduce is passed the *cp-mp*, *cp-rd*, and the two relations. map-reduce is invoked again with *s-mp*, *s-rd*, and the output of the cartesian-product. The *grab-records* and *out-func* functions are necessary for composition.

4.6 Set operations

```
1.    (defn set-operation [op s1 s2]
2.    (vec (op (set s1) (set s2))))
```

```
3.   (defn mr-union [s1 s2]
4.   (set-operation clojure.set/union s1 s2))

5.   (defn mr-difference [s1 s2]
6.   (set-operation clojure.set/difference s1 s2))

7.   (defn mr-intersection [s1 s2]
8.   (set-operation clojure.set/intersection s1 s2))
```

Union, set difference, and intersection are implemented via Clojure's set operations. These operators accept two outputs of the map-reduce system and perform the corresponding set logic. set-operation is the base function that performs the necessary manipulation on the output so that it can be composed with other operations.

4.7 MapReduce System

```
1.   (defn map-reduce [data map-func reduce-func]
2.   (let [flat-lis (flatten data)
3.     map-stage (map (fn [mr](map-func mr)) flat-lis)
4.     group-stage (into (sorted-map)(map (fn [[k v]][k (mapv
5.                                 #(second %) v)])
6.                   (group-by (fn [[k _]]k) map-stage)))
7.     reduce-stage (map (fn [[k v]](reduce-func k v)) group-stage)]
8.       reduce-stage))
```

The code for this implementation was also difficult to format, and the author apologizes for this inconvenience. A mock map and reduce framework is necessary to invoke the map and reduce functions returned by the aforementioned implementations. This function represents the simulated map and reduce system. Input data, a map function, and reduce function are accepted as input on line 1. Line 2 shows the flattening of the input data. This is done to ensure that the output of *map-reduce* can be fed back into itself as the *data* argument. Line 3 is the application of the map function on the input collection. Line 4 performs the intermediate grouping key logic, where the outputs of the map stage are grouped on the first value of each resultant vector. Line 7 performs the reduce stage by invoking the given reduce function on the output of the grouping stage. The output of the reduce stage is returned on line 8.

4.8 Query: sum the home team points and attendance for each home game

```
1.   (defn home-team-points-attendance [gd]
2.   (query/agg-group #{:home_team} {:home_points (fn [vs]
3.                                    (reduce + vs))
```

```
4.                      :attendance (fn [vs](reduce + vs))}
5.                      gd model/raw-mr))
```

This implementation shows how to perform a query in this system. This query finds the total number of home team points and attendance for each home team in each game. It accepts data on line 1, which comes from the game table. Line 2 shows the call to the aggregation grouping function where the first argument is a set of keys to perform grouping on, the second is a map of attribute name to aggregation function, the third is the data, and the fourth is to control the shape of the output. Line 2 is a concrete example of why the RA operators in this system are semantically equivalent to their theoretical counterparts. In this case, aggregation with grouping can accept any set of keys to create groups, and a set of functions to perform aggregation on corresponding attributes. This report will omit the explanations of the implementations of the other example queries as the explanation becomes redundant. The implementations for the other 5 example queries can be seen in the codebase.

5 Platform

I implemented all of the code with Clojure, Leiningen, Emacs, Ubuntu 18.04.3, git and github. The code has been tested on loki and can be executed by passing the artifact jar to the java executable.

6 Artifact Execution

Once the jar is executed, seven examples will be printed to standard out and the program will end. Each of the example outputs has a description section at the top, an equivalent SQL query, and the output of the map reduce system. The program output also has an example 0 that describes the output.

7 What I Learned

I learned a great deal about map reduce as I have never worked with it prior to this project. I also learned how to build an artifact jar with Leiningen and extended my Clojure programming abilities. I also had to reason about semantic equality between RA operators and their map and reduce counterparts, and this reasoning manifested itself in both the implementations and map reduce system.

8 Conclusion

I have presented a simulated map and reduce system that can accept map and reduce functions as input, invoke the map and reduce functions on an input collection of data, and return the appropriate output. This system and its

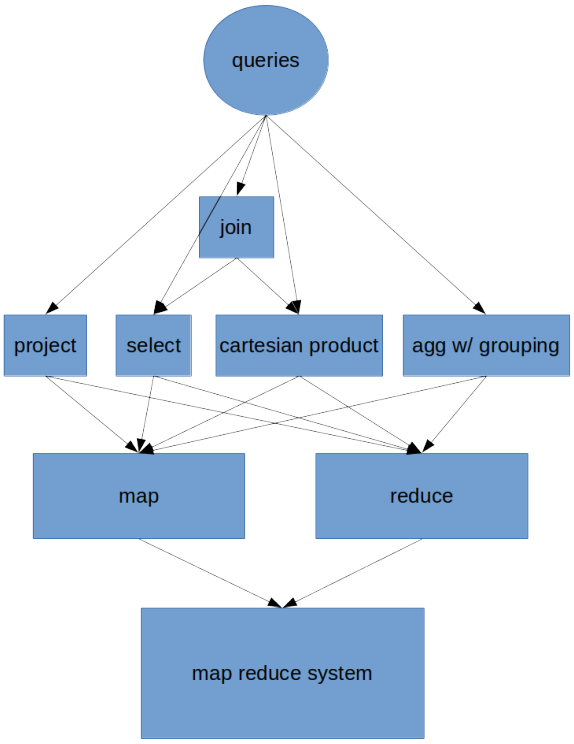

Figure 1: Diagram of the simulated map and reduce system. Everything is written in Clojure. Each box is a function. The RA equivalent map and reduce functions produce map and reduce functions as output. Thus, each box above the map and reduce boxes has two outputs. User queries sit atop the diagram and use the RA operators to answer questions. Queries are written in Clojure.

operators are capable of arbitrary composition, similar to MapReduce and RA, and each operator is semantically equivalent to its theoretical counterpart.