Exercise 3

Additional Jaql Core Operators and SQL Support

Contents

Lab 1 Using Jaql Core Operators

In this exercises you will use Jaql core operators to manipulate csv data as well as using Jaql's SQL support to access your data.

After completing this hands-on lab, you'll be able to:

- Use the Jaql core operators to manipulated data
    - Filter arrays
    - Sort arrays
    - Split arrays
    - Join data from two arrays
- Use the Jaql SQL to manipulate data
- Generate explains of Jaql scripts

Allow 20 minutes to complete this lab.

This version of the lab was designed using the InfoSphere BigInsights 2.1 Quick Start Edition, but has been tested on the 2.1.2 image. Throughout this lab you will be using the following account login information. If your passwords are different, please note the difference.

	Username	**Password**
VM image setup screen	root	password
Linux	biadmin	biadmin

1.1 Core operators

Section 1: *Core operators*

__ 1. Located on the lab image is a file of JSON records that contain book titles, the author, the year published and some reviews. First read this file so that you can repeatedly access the data in the file. Then display the contents of the file.

```
books=read(file("/home/labfiles/SampleData/bookreviews.json"));
books;
```

__ 2. What if you were going to be reading several JSON files from the *SampleData* directory? Since none of you gets paid by the keystroke, you can lessen the amount of future typing by incorporating your *read* into a function. Normally the file name is passed to the *read()* function in quotation marks. Here you must concatenate the directory string and the filename parameter. (The following is really a single line.)

```
readjson=fn(filename) read(file("/home/labfiles/SampleData/"+filename));
```

Next test your function by re-reading the *bookreviews.json* file.

```
readjson("bookreviews.json");
```

__ 3. The *bookreviews* file contains books from two authors, J. K. Rowlings and David Baldacci. List only those books that were written by *David Baldacci.*

```
books -> filter $.author == 'David Baldacci';
```

__ 4. List all books that were published before 2001.

```
books -> filter $.published < 2001;
```

__ 5. Total the number of books by each author and display the name of the author and the book total.

```
books-> group by author = $.author into {author, num_books: count($)};
```

__ 6. Next, for those books published in 2000, write a record that only contains the author and the title.

```
books->filter $.published == 2000 ->transform {$.author, $.title};
```

__ 7. What if you wanted to split your data into multiple files based upon the author? You want only the data for *David Baldacci* to be in one file and all other authors in another.

```
books -> tee( -> filter $.author == 'David Baldacci' ->
write(jsonTextfile("file:///home/biadmin/bd.json"))) -> filter $.author !=
'David Baldacci' -> write(jsonTextFile("file:///home/biadmin/other.json"));
```

__ 8. And if you wanted to sort your data on *published?*

```
books -> sort by [$.published asc];
```

__ 9. There is a second file, */home/labfiles/SampleData/books.csv,* that only contains the information about each book. This file is in a csv format. Each record has a book number, the author, the title, and the year published. Read this file but be aware that the *del()* I/O adapter defaults to reading from a Hadoop file system.

```
booksonly=read(del("file:///home/labfiles/SampleData/books.csv"));
booksonly;
```

__ 10. What if for some reason (and this is a lab exercise so there does not have to be a logical reason to do anything), you wanted the individual arrays in *booksonly* to be put into a single array? (This takes longer to run.)

```
booksonly -> expand $;
```

__ 11. Because all Jaql had to read was that delimited data, there were no fields assigned and if you notice, the years are read as strings. Reread the file but this time specify a schema to assign fields and to make sure that the data is read correctly.

```
booksonly=read(del("file:///home/labfiles/SampleData/books.csv",
schema=schema{booknum:long,author:string,title:string,published:long}));
booksonly;
```

__ 12. There is another delimited file, called *reviews.csv,* in the same directory. Each record contains a book number, name of the reviewer, and the number of stars. Read it.

```
reviews=read(del("file:///home/labfiles/SampleData/reviews.csv",
schema=schema{booknum:long,name:string,stars:long}));
reviews;
```

__ 13. Join the *booksonly* and *reviews* arrays on the *booknum* fields resulting in just the title of the book and the stars associated. This will take some time.

```
booksandstars=join b in booksonly, r in reviews where b.booknum == r.booknum
into {b.title, r.stars};
booksandstars;
```

__ 14. Now calculate the average number of stars for each book.

```
booksandstars->group by book = $.title into {book, avg_stars: avg($.stars)};
```

__ 15. You can accomplish the same thing by using co-grouping as you did with the join and the group by statements.

```
group booksonly by g = $.booknum as bs, reviews by g = $.booknum as rs into
{title:bs[0].title, avg_stars: avg(rs.stars)};
```

__ 16. While you are at it, look and see whether the previous Jaql statement will be rewritten to use map / reduce.

```
explain group booksonly by g = $.booknum as bs, reviews by g = $.booknum as
rs into {title:bs[0].title, avg_stars: avg(rs.stars)};
```

1.2 Use Jaql SQL

__ 1. Earlier you read `/home/labfiles/SampleData/bookreviews.json` into books. This contained both book and review information. Use a select statement to access `author, title,` and `published` from this file.

```
select author, title, published from books;
```

What happened? There are some limitations when using a select statement in Jaql. The record format needs to be consistent.

__ 2. Use the *transform* operator to get a consistent set of records.

```
sqlbooks = books -> transform {$.author, $.title, $.published};
```

__ 3. Now do your select again.

```
select author, title, published from sqlbooks;
```

__ 4. Read in the data from `/home/labfiles/SampleData/books.csv` as follows:

```
csvbooks=read(del("file:///home/labfiles/SampleData/books.csv"));
```

__ 5. Select `author, title,` and `published` from this file.

```
select author, title, published from csvbooks;
```

Once again, not a lot of success. But why? There is no schema information when just reading a csv file.

__ 6. Earlier you read this file with a schema into *booksonly.* Select `author, title,` and `published` from this file.

```
select author, title, published from booksonly;
```

__ 7. Next add a condition to your select. Get only those books published since 2003.

```
select author, title, published from booksonly where published > 2003;
```

__ 8. Earlier you did a join of two arrays and then a group by to come up with the average number of stars for each book. Then you used co-grouping to accomplish the same task. Now use Jaql SQL in order to do the averaging a third way.

```
select b.title, avg(r.stars) as avg_stars from booksonly b, reviews r group
by b.title;
```

__ 9. Do an explain on the above *select* and see the code that gets generated. What is interesting is that the explain for the co-grouping distills to a map / reduce whereas the SQL distills to an mrAggregate. If you were to do an explain of your previous join being streamed into your previous group by, it would distill to an mrAggregate as well. Go ahead and try that.

End of exercise

NOTES

NOTES