MapReduce & SQL in Javascript

The MapReduce paradigm is at the basis of parallel computing frameworks like *Apache Hadoop* that process huge amounts of data.

It's a kind of *functional programming*, where functions can be passed as arguments to other functions, which in turn return functions, and so on.

In this way, we can write very compact and elegant algorithms, although a little cryptic.

In Javascript we can combine the powerful of functional programming with the easygoing management of arrays and objects in a very useful and fun way.

For my examples I was inspired by typical queries in SQL, imagining as database tables some arrays of javascript objects taken from the well-known sample database "Northwind".

Please refer to very good articles, i.e this, this, this or this, for a basic explanation of the functions map(), reduce(), filter(), sort(), the Set() object, and the Spread operator [...].

You can download all files from GitHub

BASIC OPERATIONS ON TABLES

PROJECTION

The *projection* is a filtering on the columns

For example, from the CUSTOMERS table we want to display only some fields

- SQL

  ```
  SELECT id, companyName, country
  FROM customers
  ```

- Javascript

  ```
   let projection = customers.map(c=>new Object({"id":c.id,"companyName": c.companyName, "cou
  ```

SELECTION

the *selection* is a filtering on the rows

From table CUSTOMERS we only want customers who come from UK

- SQL

  ```
  SELECT *
  FROM customers
  WHERE country = "UK"
  ```

- Javascript

  ```
  let selez = customers.filter(cc=>cc.country=="UK")
  ```

CARTESIAN PRODUCT

The Cartesian Product between two sets is given by all the elements of A combined with each element of B

- JS

  ```
  let a = ['a', 'b', 'c', 'd']
  let b = [1, 2, 3]

  let prod = [].concat(...a.map(p => b.map(b => p + b)))

  /* [
      'a1', 'a2', 'a3',
      'b1', 'b2', 'b3',
      'c1', 'c2', 'c3',
      'd1', 'd2', 'd3'
  */  ]
  ```

DISTINCT CLAUSE

Retrieves only distinct (unique) values in a specified list of columns: in Javascript we can use the Set() object

Example: List the countries in the customers table

- SQL

  ```
  SELECT DISTINCT country
  FROM customers
  ORDER BY country
  ```

- Javascript

  ```
  const distinct = Array.from(new Set(customers.map(cc => cc.country)))
  ```

SORTING

To sort the records we use the sort() function

- SQL

  ```
  SELECT *
  FROM customers
  ORDER BY companyName
  ```

- JS

  ```
  let sorted = customers.sort((a,b)=> ( b.companyName < a.companyName));
  ```

AGGREGATION FUNCTONS

COUNT

Example: ***Total customers amount***

- SQL

  ```
  SELECT COUNT(*)
  FROM customers
  ```

- JS

  ```
  let customers_amount = customers.reduce((count, val) => count + 1, 0);
  ```

SUM

Example: ***Total products in stock***

- SQL

  ```
  SELECT SUM(unitsInStock)
  FROM products
  ```

- JS

  ```
  let totp = products.map(prod => prod.unitsInStock).reduce((sum, cc) => sum + cc, 0);
  
   //or
  
   let totp_alt = products.reduce((sum, prod) => sum + prod.unitsInStock, 0);
  ```

MAX

Example: ***Highest price of all products***

- SQL

```
SELECT MAX(unitPrice)
FROM products
```

- JS

```
const maxPrice = products.map(cc => cc.unitPrice).reduce((max, d) => d > max ? d : max);
```

MIN

Example: *Lowest price*

- SQL

```
SELECT MIN(unitPrice)
FROM products
```

- JS

```
const minPrice = products.map(cc => cc.unitPrice).reduce((min, d) => d < min ? d : min);
```

AVG

Example: *Average price*

- SQL

```
SELECT AVG(unitPrice)
FROM products
```

- JS

```
const avgPrice = products.map(cc => cc.unitPrice).reduce( (r, p) =>{ r.sum += p; ++r.count
```

TOP N

Example: *The ten most expensive products*

- SQL

```
SELECT TOP 10(unitPrice)
FROM products
```

- JS

```
const topN = products.sort( (a, b) =>  a.unitPrice - b.unitPrice ).reverse().slice(0, 10)
```

GROUP BY

Example 1:*Total products for each category*

- SQL

  ```
  SELECT categoryId, COUNT(*)
  FROM products
  GROUP BY(categoryId)
  ```

- JS

  ```
  const groupByCategory = products.reduce((groups, cc) => {
  groups[cc.categoryId] = (groups[cc.categoryId] || 0) + 1;
  return groups;
  }, {})
  ```

Example 2:*Total customers for each country*

- SQL

  ```
  SELECT country, COUNT(*)
  FROM customers
  GROUP BY(country)
  ```

- JS

  ```
  const groupByCustomersCountry = customers.reduce((gruppi, cc) => {groups[cc.country] = (g
  ```

  then sort by total customers

  ```
  let filtra = Object.entries(groupByCustomersCountry).map(kk => new Object({ c: kk[0], n:
  ```

Example 3: *Total orders for each customer (minimum 5)*

- SQL

  ```
  SELECT customerId, count(*)
  FROM orders
  GROUP BY customerId
  HAVING count(*) >= 5
  ORDER BY count(*) DESC
  ```

- JS

```
const OrdersgroupByCustomer = Object.entries(orders.reduce((groups, cc) => {
groups[cc.customerId] = (groups[cc.customerId] || 0) + 1;
return gruppi;
}, {}))
.map(kk => new Object({ c: kk[0], n: kk[1] })).sort((a, b) => b.n - a.n)
.filter(o => o.n >= 5)
```

FURTHER TYPICAL HADOOP FUNCTIONS

BINNING

Splits records based on a criterion

Example: split products into 4 categories based on price

- JS

```
const categ = products.reduce((cate, cc) => {
let pr = parseInt(cc.unitPrice)
if (pr < 4) cate["very_cheap"].push(cc)
else if (pr < 10) cate["cheap"].push(cc)
else if (pr < 50) cate["expensive"].push(cc)
else cate["luxury"].push(cc)
return cate;
}, { very_cheap: [], cheap: [], expensive: [], luxury: [] })
```

INVERTED INDEX

Used to create dictionary-like structures

Example 1: for each month, provide the list of the customers who placed at least one order

- JS

```
const invIndex = orders.reduce((months, ord) => {
let year=new Date(ord.orderDate).getFullYear()
let month=new Date(ord.orderDate).getMonth()
let key=month+"-"+year
months[key] = (months[key] || new Set()).add(ord.customerId);
return months;
}, {})
```

Example 2: for each customer provide all the orders dates

- JS

```
const invIndex2 = orders.reduce((order_dates, cc) => {
order_dates[cc.customerId] = (order_dates[cc.customerId] || new Set()).add(cc.orderDate.su
return order_dates;
}, {})
```

JOINS

Used to combine two or more tables that relate through one or more columns

LEFT OUTER JOIN: *filters all the values in the first table even if they don't match in the second table*

- SQL

```
SELECT *
FROM products
LEFT JOIN categories
ON products.categoryId = categories.id;
```

- JS

```
const left_join = products.reduce((loj, pp) => {
let lookup_cat = categories.filter(cc => pp.categoryId == cc.idCat);
let union_rec
if (lookup_cat.length) union_rec = { ...pp, ...lookup_cat[0] }; else union_rec = pp;
loj.push(union_rec)
return loj;
}
, [])
```

RIGHT OUTER JOIN: *retains all the values from the second table even if they don't match the first*

- SQL

```
SELECT *
FROM categories
LEFT OUTER JOIN products
ON categories.id= products.categoryId;
```

- JS

```
const right_join = categories.reduce((roj, cc) => {
let lookup_prod = products.filter(pp => pp.categoryId == cc.idCat);
if (lookup_prod.length == 0) roj.push(cc); else {
lookup_prod.map(v => {
let union_rec = { ...cc, ...v }
roj.push(union_rec)
})
}
return roj;
}, [])
```

INNER JOIN (NATURAL JOIN)

Inner Join and Natural Join are very similar - only the number of columns returned changes

In this solution we first make the Cartesian product between the two tables and then filter only the records with the same "hinge" column values

- SQL

```
SELECT *
FROM products, categories
WHERE products.categoryId = categorie.id;
```

- JS

```
let prod_cart = [].concat(...products.map(pp => categories.map(cc => Object.assign({}, {

let natjoin = prod_cart.filter(f => (f.categoryId == f.idCat))
```

SUBQUERY

Example 1: list all customers who have placed at least one order

- SQL

```
SELECT *
FROM customers
WHERE id IN ( SELECT customerId FROM orders)
```

)

- JS

```
const customer_list=customers.filter(cl=>orders.filter(oo=>oo.customerId=cl.idCli))
```

Example 2: list all products that have been ordered at least once in quantities greater than 100 units

- SQL

  ```
  SELECT DISTINCT idProd as id, name as descr
  FROM products
  WHERE productId IN (SELECT productId from details WHERE quantity>100)
  ```

-%--

Example 3: list of orders that include products of at least two different categories

- SQL

  ```
  SELECT Details.OrderId, (SELECT COUNT( DISTINCT categoryId)) as ncat
  FROM Details, Products
  WHERE Details.ProductId=Products.ProductId
  GROUP BY Details.OrderId
  HAVING ncat>1
  ```

- JS

  ```
   const groupByCat = (details.reduce((groups, cc) => {
   var prod=[]
   cc.products.forEach(pp=>prod.push(rispc[pp.productId]))
   let diversi=new Set(prod).size
   if(diversi>1) groups.push(new Object({"ord":cc.orderId, "det":prod.length, "div":diversi}
   return groups;
   }, []))
  ```