Map-Reduce Paradigm

Francesco Di Giacomo

1 Introduction

In these notes we will present the `Map-Reduce` paradigm, typical of No-SQL systems such as MongoDB or Hadoop. In Section 2 we will provide some background knowledge necessary to understand the topics that will be presented later. In Section 3 we will present the `Map` function and show that it is equivalent to the SQL `Select` statement. We will also show how to implement the `Where` statement with a function similar to `Map`. In Section 4 we will implement the `Reduce` function and show that with it we can implement `Where` and `Join`.

2 Background

This section provides some background knowledge necessary to understand the topic of map-reduce. We mainly introduce features of C# that are necessary for the implementation.

2.1 Properties

C# allows to define properties in classes. Properties allow to use the syntax of variable assignments while, at the same time, restricting the direct access to fields of a class. Properties allows to define a field and at the same time its getter and setter methods. An automatic property is defined in the following way

```
public class Foo
{
  public T FooProperty<T> { get; set; }
}
```

The compiler will automatically create a private field associated to `FooPropert` and generate the getter and setter for it. The property can appear directly as right argument of a variable assignment, like

```
T f = myFoo.FooProperty;
```

In this case the getter of the property is called. We can also set a property by using variable assignment, like

```
myFoo.FooProperty = x;
```

This will call the setter of `FooProperty` by passing the input `x`.

It is also possible to define custom getters and setters that are less trivial. For example the following getter returns the value of the property and adds 1 to it.

```
public clas Bar
{
  private int barField;

  public int BarProperty
  {
    get
    {
      int old = barField;
      barField++;
      return old;
    }
  }
}
```

It is possible also to define custom setters in the same way. The setter uses a special identifier `value` which contains the value to set. The following setter adds an offset of 10 to the value.

```
public clas Bar
{
  private int barField;

  public int BarProperty
  {
    get
    {
      int old = barField;
      barField++;
      return old;
    }
    set
    {
      barField = value + 10;
    }
  }
}
```

It is not mandatory to specify both getters and setters for a property. It is also possible to alter the access to either the getter or the setter by putting an access modifier different than the one used by the property just before the keyword `get` or `set`.

```
public class Bar
{
  public int BarProperty
  {
    get { ... }
    private set { ... }
  }
}
```

2.2 Anonymous types

In C# it is possible to create anonymous types to instantiate a class without explicitly declare it in the program. For example, the following creates an anonymous type containing two strings:

```
var foo =
  new
  {
    Name = "Jack",
    Surname = "Sparrow"
  }
```

Note that the keyword `var` is used in C# to let the compiler infer the type of the variable. If it is necessary to provide the type of a generic data structure that contains instances of an anonymous type, then we use the keyword `dynamic` to let the compiler detect the type at run-time, rather than at compile-time. This is necessary because we cannot know the type name of an anonymous type (it is indeed anonymous).

```
var foos = new List<dynamic>();
foos.Add(new { Name = "Jack", Surname = "Sparrow" });
foos.Add(new { Name = "Edward", Surname = "Teach" });
foos.Add(new { Name = "Edward", Surname = "Kenway" });
foos.Add(new { Name = "Kathryn", Surname = "Janeway" });
```

2.3 Lambdas

In C# it is possible to treat functions as values and use them as arguments of other functions. In order to define an anonymous function (or lambda), it is possible to use the built-in type `Func`. `Func` is a type that takes `N` generic arguments. The first `N - 1` generic arguments contains the type of the function arguments, while the last one contains the return type. For example the following function computes the integer logarithm of two numbers:

```
Func<int, int, int> integerLog =
  (n,_base) =>
  {
    int i = 0;
```

```
    while (n > 0)
    {
      n = n / _base;
      i = i + 1;
    }
    return i;
  };
```

You can also write a lambda whose body is just an expression in the following way:

```
Func<int, int, int> add = (x,y) => x + y
```

3 The Map Function

In order to understand the idea behind the map function, let us proceed with some examples: assume we want to implement a simple function that squares the elements of an array of integer numbers without changing the input array. We must create a result array with the same size as the input one. Then we must scan the whole input array, multiply each element by itself, and copy the result into the corresponding element in the output array.

```
static int Square(int[] l)
{
  int[] squares = new int[l.Length];
  for (int i = 0; i < l.Length; i++)
  {
    squares[i] = l[i] * l[i];
  }
  return squares;
}
```

Now let us make a function that converts all the elements of an array into their string representation. This time the output will be an array of strings and the input an array with generic type `T`.

```
static string[] Convert<T>(T[] numbers)
{
  string[] result = new string[numbers.Length];
  for (int i = 0; i < numbers.Length; i++)
  {
    result[i] = numbers.ToString();
  }
  return result;
}
```

Now let us make a more complex example. Let us assume we have the representation of an employee of a company database through the following class:

```
class EmployeeTuple
{
  //id, name, surname, sex, salary
  public Tuple<int, string, string, char, double> Tuple
    { get; set; }

  public EmployeeTuple(Tuple<int, string, string, char, double> tuple)
  {
    Tuple = tuple;
  }
}
```

We use a tuple because, by definition, the rows of a table in a database are ordered sequences of values (indeed they are often called tuples as well). Let us now assume that the company wants to raise the monthly salary of all employees by 10%. The table can be represented as a list of employees. The class definition is the following:

```
class EmployeeTuple
{
  public Tuple<int, string, string, char, double> Tuple
    { get; set; }

  public EmployeeTuple(int id, string name, string surname, char sex, double salary)
```

```
  {
    Tuple = new Tuple<int, string, string, char, double>(id, name, surname, sex, salary);
  }
}
```

Now the function that raises the salary is analogous to the other two, except that we insert a new tuple with the modified salary.

```
static List<EmployeeTuple> RaiseSalary(List<EmployeeTuple> employees)
{
  List<EmployeeTuple> result = new List<EmployeeTuple>();
  for (int i = 0; i < employees.Count; i++)
  {
    result[i] = new EmployeeTuple
    (
      employees[i].Tuple.Item1,
      employees[i].Tuple.Item2,
      employees[i].Tuple.Item3,
      employees[i].Tuple.Item4,
      employees[i].Tuple.Item5 + employees[i].Tuple.Item5 * 0.1);
  }
  return result;
}
```

Finally, let us assume that we want to change the representation of an employee to have a more convenient structure, i.e. an object with some fields. The class definition will then be:

```
class Employee
{
  public int Id { get; set; }
  public string Name { get; set; }
  public string Surname { get; set; }
  public char Sex { get; set; }
  public double Salary { get; set; }

  public Employee(int id, string name, string surname, char sex, double salary)
  {
    Id = id;
    Name = name;
    Surname = surname;
    Sex = sex;
    Salary = salary;
  }
}
```

If we want to implement the raise function we have to make a new overload for the method

```
static List<Employee> RaiseSalary(List<Employee> employees)
{
  List<Employee> result = new List<Employee>();
  for (int i = 0; i < employees.Count; i++)
  {
    result.Add(new Employee
    (
    employees[i].Id,
    employees[i].Name,
    employees[i].Surname,
    employees[i].Sex,
    employees[i].Salary + employees[i].Salary * 0.1));
  }
  return result;
}
```

At this point it should appear clear that we are basically doing the same thing every time except for minor differences. All these functions exhibit the same pattern:

1. They take as input a collection of elements.
2. They initialize a collection for the result.
3. They iterate the whole collection and apply a transformation to each element.
4. They assign the result of the transformation to the corresponding element in the result collection.
5. They return the result collection containing the transformed elements.

Software engineering principles suggest that, whenever a pattern in the code is detected, this should be captured into a single computational unit and not repeated in different versions (as we did above). What we will try to do now, is to write a single function that captures the pattern of all these functions. Let us now compare two of the functions above:

Listing 1: Code to raise the salary

```
1   static List<Employee> RaiseSalary(List<
        Employee> employees)
2   {
3     List<Employee> result = new List<
          Employee>();
4       for (int i = 0; i < employees.Count;
            i++)
5       {
6         result.Add(new Employee
7         (
8         employees[i].Id,
9         employees[i].Name,
10        employees[i].Surname,
11        employees[i].Sex,
12        employees[i].Salary + employees[i
              ].Salary * 0.1));
13      }
14    return result;
15  }
```

Listing 2: Code to convert numbers into strings

```
1   static string[] Convert<T>(T[] numbers)
2   {
3     string[] result = new string[numbers.
          Length];
4       for (int i = 0; i < numbers.Length;
            i++)
5       {
6         result[i] = numbers.ToString();
7       }
8     return result;
9   }
```

The declaration of the method (line 1) differs only in the return type and the type of the input argument. In Listing 1 the method returns a `List<Employee>` while in Listing 2 it returns a `string[]`. This might suggest that the generalization, which we conveniently call `Map`, is capable of returning a generic collection that is iteratable. Moreover, the first function takes as input a `List<Employee>` while the second takes `int[]`. This suggests that `Map` takes as argument another generic collection whose generic type is different from that of the result. The loop is always identical in all the different versions, so it can be kept as it is. The transformation is, however, completely different in all the versions. But what if the `Map` took as input the transformation to apply to each element? In this way we could pass the specific transformation to use when we call `Map` and the code in the method body would simply use it. We follow this idea and give the following definition for the function:

```
static class MapReduce
{
  public static IEnumerable<T2> Map<T1, T2>(IEnumerable<T1> collection, Func<T1, T2>
      transformation)
  {
    T2[] result = new T2[collection.Count()];
    for (int i = 0; i < collection.Count(); i++)
    {
      result[i] = transformation(collection.ElementAt(i));
    }
    return result;
  }
}
```

How do we use this function? We can create two different collections for our test, one containing employees and one containing integer numbers. The function passed as argument to the `Map` will contain the details of the transformation. For the employee it creates a new `Employee` instance with the same values of the input except the `Salary`, which is increased by 10% as required. The function for the conversion of numbers into strings will simply execute the conversion.

```
int[] numbers = { 3, -1, 4, -20, 6 };
List<Employee> employeeTable =
  new List<Employee>(new Employee[]
  {
    new Employee(3952, "Frank", "Moses", 'M', 2500.50),
    new Employee(1403, "John", "Ford", 'M', 1200.50),
    new Employee(3433, "Michelle", "Brown", 'F', 3250.25),
    new Employee(3540, "Daniel", "Smith", 'M', 2500.50)
  });
IEnumerable<Employee> raised = MapReduce.Map(employeeTable,
  employee =>
  {
```

```
    return
      new Employee(
      employee.Id,
      employee.Name,
      employee.Surname,
      employee.Sex,
      employee.Salary + employee.Salary * 0.1);
  });
IEnumerable<string> converted = MapReduce.Map(numbers, x => x.ToString());
```

In order to be absolutely sure that this works not because of the doing of an evil spirit inhabiting our computer who will ask our soul in exchange of the correctness of the execution, but rather thanks to our ingenuity, we will try to check step-by-step what happens when calling `Map` with the conversion function.

First of all, let us check that the types actually make sense: the first argument passed to the function is the collection of elements to convert, which has type `int[]`, so the generic type `T1` will be replaced by `int`. The lambda that we pass to the function takes an element of type `int` and converts it to a string, so its type is `Func<int,string>`. Thus the generic `T2` will be replaced by `string`. The function thus returns a `string[]`, which by polymorphism is equivalent to `IEnumerable<string>`.

The function is called by passing the array `{ 3, -1, 4, -20, 6 }` as argument and the lambda `x => x.ToString()`. At line 3 the function creates an array of type `string[]` (remember the replacement of generics explained above). At each iteration of the loop the lambda is applied to each element of the input, with the following results:

```
1. x => x.ToString → 3.ToString() → "3"
2. x => x.ToString → -1.ToString() → "-1"
3. x => x.ToString → 4.ToString() → "4"
4. x => x.ToString → -20.ToString() → "-20"
5. x => x.ToString → -6.ToString() → "-6"
```

These results are stored into the output array, which is then returned as result of `Map`. The reader can verify, as an exercise, the typing and the steps necessary to evaluate the call of `Map` with `List<Employee>`.

3.1 Map is Select

In this section we will show that the `Map` function behaves like the `Select` statement in SQL-like languages. Let us consider the salary raise operation performed earlier: the equivalent query in SQL would be

```
SELECT id, name, surname, sex, salary + salary * 0.1
FROM employee
```

Now someone could say that this is not enough because we are not able to select a subset of attributes using `Map`, as the real select does, because we are outputting all the attributes of each row (even if their value was changed). For example let us consider the SQL query

```
SELECT name, surname
FROM employee
```

how can we use `Map` to get the same result, so a collection of objects containing only `Name` and `Surname`? The answer is using a lambda that transforms every `Employee` instance into an instance of a different type containing only `Name` and `Surname`. In order to do this, we use anonymous types.

```
var data = MapReduce.Map(employeeTable,
  employee =>
    {
      return new
      {
        Name = employee.Name,
        Surname = employee.Surname
      };
  });
```

In this way every element of the input list will be mapped to a different element that contains only the attributes that we want to keep as result of a query.

3.2 Filtering the result

At this point, we have a function that is equivalent to the SELECT statement of a query, but we have no way of filtering the rows that appear in the result. It is not possible to implement `WHERE` with `Map`. Just think about

the fact that, by definition, the result of `Map` always contains the same amount of elements as the input, while the result of `WHERE` contains an amount of elements that is smaller or equal than those in the input collection.

By following a method analogous to what done for the `Map` we can define a function `Where`. This function takes a collection with generic type `T` as input and checks a condition for all elements of the collection. If the condition is met the element is put in the result, otherwise it is discarded.

One common mistake is to think that the condition can be expressed with a boolean expression. For example, let us consider the query

Listing 3: Query with filtering

```
SELECT name, surname
FROM employee
WHERE salary > 1500
```

If `Where` were to be called with a boolean expression we would have something like

```
IEnumerable<Employee> filtered = Where(employeeTable, salary > 1500)
```

but at this point the boolean expression would be evaluated, resulting into either true or false, and then this value would be passed to the function. Thus, the condition would immediately be always true or false for all the elements of the collection. The condition is thus a lambda with type `Func<T, bool>`, so it is a function that takes an element of the same type of those in the input collection and returns either true or false. This allows to dynamically evaluate the condition for all the elements in the collection. The definition of `Where` is thus the following:

```
public static IEnumerable<T> Where<T>(IEnumerable<T> collection,  Func<T, bool> condition)
{
  List<T> result = new List<T>();
  for (int i = 0; i < collection.Count(); i++)
  {
    if (condition(collection.ElementAt(i)))
      result.Add(collection.ElementAt(i));
  }
  return result;
}
```

The query above becomes then

```
IEnumerable<Employee> filtered = MapReduce.Where(employeeTable, e => e.Salary > 1500);
data = MapReduce.Map(filtered,
  employee =>
  {
    return new
    {
      Name = employee.Name,
      Surname = employee.Surname
    };
  });
```

What we miss now is the possibility of running aggregation functions, such as in

```
SELECT SUM(salary)
FROM employee
```

4 The Reduce Function

Let us consider the query

Listing 4: Example query for Reduce

```
SELECT SUM(salary)
FROM employee
WHERE salary > 1500
```

with what we have now we are able to evaluate the `WHERE` part and to select some of the attributes but not to compute the sum. We cannot compute the sum with `Map` because the return type of `Map` is a collection while the sum returns a single value. We need something that is able to apply an operation to each element of the collection and accumulate (or aggregate) the result. At this purpose, let us consider first the specific code for

two examples, one that computes the concatenation of the string conversion of a sequence of numbers, and the other that sums the salaries of a collection of employees:

Listing 5: Code to concatenate the string representations of numbers

```
1 static string Concat<T>(IEnumerable<T> l)
2 {
3   string c = "";
4   for (int i = 0; i < l.Count(); i++)
5   {
6     c = c + l.ElementAt(i).ToString();
7   }
8   return c;
9 }
```

Listing 6: Code to compute the sum of the salaries of the employees

```
1 static double SumSalary(IEnumerable<
      Employee> employeeTable)
2 {
3   double sum = 0;
4   for (int i = 0; i < employeeTable.Count
        () ; i++)
5   {
6     sum = sum + employeeTable.ElementAt(i)
          .Salary;
7   }
8   return sum;
9 }
```

Let us try to pinpoint the differences between the two functions and to recycle their pattern (which is the same). both functions share the following behaviour

1. They initialize a variable containing the result (accumulator).
2. For all elements of the collection they update the accumulator by applying an operation involving the accumulator and an element of the collection.
3. They return the final value of the accumulator.

First of all, let us look at the types of the method declaration. Listing 5 returns a `string` while Listing 6 returns `double`. The type of the argument of Listing 5 is `IEnumerable<T>` while the one of Listing 6 is `IEnumerable<Employee>`. Thus we can say that our `Reduce` function returns a generic type `T2` and takes as input a `IEnumerable<T1>`. At line 3 both functions initialize an accumulator containing the result with a different value. This value must be passed as input to `Reduce` because it is specific of the operation we want to execute. This is not enough: at line 6 both functions run a completely different operation that updates the accumulator. Thus, analogously to what we did for `Map`, we pass a function as argument that is able to perform the calculation necessary to update the accumulator. So `Reduce` takes two extra arguments, the initial value of the accumulator, whose type is `T2`, and the operation to execute. This is a lambda taking as input the accumulator itself and an element of the input collection, and returns the updated accumulator, thus its type is `Func<T2, T1, T2>`. The definition of the `Reduce` becomes then

```
public static T2 Reduce<T1, T2>(IEnumerable<T1> collection, T2 init, Func<T2, T1, T2>
    operation)
{
  T2 result = init;
  for (int i = 0; i < collection.Count(); i++)
  {
    result = operation(result, collection.ElementAt(i));
  }
  return result;
}
```

Let us now implement the two functions above with `Reduce`. The first one takes as input a collection of numbers and concatenates their string representations. Thus the input will contain a collection of numbers, the accumulator will be set to `""`, and the lambda will take the accumulator and a number, and add the string representation of the number to the accumulator.

```
string concatenation = MapReduce.Reduce(numbers, "", (accumulator, x) => accumulator + x.
    ToString());
```

In order to compute the sum of the salaries we take as input the collection of employees, we initialize the accumulator to 0.0, and we pass a function that adds to the accumulator each salary.

```
double salarySum = MapReduce.Reduce(employeeTable, 0.0, (accumulator,e) => accumulator + e.
    Salary);
```

At this point we are capable of implementing the query in Listing 4. We first filter the collection with `Where` and then we compute the sum with `Reduce`.

```
IEnumerable<Employee> filtered = MapReduce.Where(employeeTable, e => e.Salary > 1500);
double filteredSum = MapReduce.Reduce(filtered, 0.0, (accumulator, e) => accumulator + e.
    Salary);
```

At this point we have `Map`, `Reduce`, and `Where` and we are able to implement a full SQL queries. Moreover we have clearly estabilished that everything works as result of black magic rituals. But at this point a careful reader would have noted that the title of this document is `Map-Reduce`, and not `Map-Reduce-Where`. In other words, we have an "intruder": the function `Where`. But would it be possible to implement `Where` in terms of `Reduce`? Keep reading and you will find out.

4.1 Where is Reduce

In the previous section we showed that `Map-Reduce-Where` is equivalent to SQL. We have also left the open question about whether it is possible to implement `Where` in terms of `Reduce`, because the `Where` function is something extra. We will answer this question right now.

To express the `Where` in terms of `Reduce` we must define the data structure for the accumulator, its initial value, and the update function for the accumulator. `Where` returns a new collection containing values filtered from the input according to a condition. Thus the accumulator will be a collection. Its initial value is an empty collection: the filter might remove, as an extreme case, all the elements from the original collection if none satisfies the condition. The lambda takes as input the result collection and each element of the input collection and adds an element if it satisfies the predicate. Thus let us consider again the query in Listing 3, we implement the query as follows:

```
IEnumerable<Employee> filteredWithReduce =
  MapReduce.Reduce(
    employeeTable,
    new List<Employee>(),
    (queryResult, e) =>
      {
        if (e.Salary > 1500)
          queryResult.Add(e);
        return queryResult;
      });
var data = MapReduce.Map(filteredWithReduce,
  employee =>
    {
      return new
      {
        Name = employee.Name,
        Surname = employee.Surname
      };
    });
```

Note that the lambda passed to `Reduce` checks the condition as well, and adds the result only if the condition is met. This shows that only with `Map-Reduce` we can build a SQL query. After `Reduce` is run we use `Map` to select only the attributes that we want in the result. At this point you would think it is over, but there is more: in the next section we will discover that `Reduce` is even more powerful.

4.2 Map is Reduce

In the previous section we explained how to implement `Where` using `Reduce`. But we can do more. We can implement `Map` with `Reduce` as well.

The result of `Map` is a collection, so again the accumulator used in `Reduce` will be a collection. The initial value of the collection is again an empty collection. The lambda takes each element of the input collection, applies the transformation on each element, and adds it to the accumulator. For instance, the query

```
SELECT name, surname
FROM employee
```

becomes

```
var dataWithReduce =
  MapReduce.Reduce(employeeTable,
    new List<dynamic>(),
    (queryResult, e) =>
      {
```

```
      queryResult.Add(
        new
        {
          Name = e.Name,
          Surname = e.Surname
        });
    return queryResult;
  });
```

Note that we have to assign `dynamic` to the generic argument because we are using an anonymous type for the result, so we cannot know its type at compile time.

The query

```
SELECT name, surname
FROM employee
WHERE salary > 1500
```

becomes instead

```
var filterAndProjectionWithReduce =
  MapReduce.Reduce(employeeTable,
  new List<dynamic>(),
  (queryResult, e) =>
  {
    if (e.Salary > 1500)
      queryResult.Add(
        new
        {
          Name = e.Name,
          Surname = e.Surname
        });
    return queryResult;
  });
```

For convenience, we keep using two separate functions, `Map` and `Reduce`, but only `Reduce` will be enough to build a language equivalent to SQL.

4.3 Join is Reduce

Up to this point, we are able to use `Map-Reduce` to implement any SQL query on one table. But what if we want to retrieve information frm multiple tables?

Let us consider the example of the same employee table as above, and an additional data structure representing company cars. Each car can be assigned to at most one employee, but a single employee can use more than one company car. The car is represented by a plate number, a model, and a reference to the employee it is assigned to (his ID). Imagine that we want to retrieve the information about the name, surname, and model of car assigned to each employee. The corresponding SQL query would be

```
SELECT e.name, e.surname, c.model
FROM Employee e, Car c
WHERE e.id = c.id
```

We will now show that only with Map-Reduce it is also possible to implement a join. Recalling the definition of join, this is simply the Cartesian Product of the two tables followed by a filter applied to the result. Remember that the Cartesian Product takes each row from the first table and combines it with all the rows from the second table. Our `Join` function will thus take two tables and a condition defined as a lambda in the same fashion of what we did for `Where`. It will then combine each row of the first table with all the rows of the second table and put the combination in the result only if the condition is met.

Let us first give the definition of the class representing the car:

```
class CompanyCar
{
  public string Plate { get; set; }
  public string Model { get; set; }
  public int EmployeeId { get; set; }
  public CompanyCar(string plate, string model, int employee)
  {
    Plate = plate;
    Model = model;
    EmployeeId = employee;
```

```
  }
}
```

Now let us think about the signature of the function for `Join`: the function takes as input two different tables (or collections) with different element types. Thus the function will have two generic type arguments, `T1` and `T2`. The function returns a collection containing the combinations of each row of the first table with all the rows of the second table. Each of these combinations can be represented by a pair containing an element from the first list and an element from the second list. Thus the returned type will be `IEnumerable<Tuple<T1,T2>>`. The `Join` also takes a condition to filter out some of these combinations. The condition is a lambda that takes a combination and returns a boolean, thus its type is `Func<Tuple<T1, T2>, bool>`.

The accumulator to use in the reduce will thus be a list containing the combinations created by `Join` and its initial value is an empty list. In the lambda passed to `Join` we take each element from the first table and create its combinations with all the rows of the second table.

To do this, let us focus first on how to create the combination of one row of the first table with the rows of the second table. This can be done with `Reduce` itself by taking as input the second table, initializing the accumulator to en empty list, and adding the pair containing the elements from both the first table and the second if the condition is met. Assuming that `x` is the variable containing the row from the first table, the following code creates the combination between `x` and all the rows of the second table, filtered according to the condition:

```
List<Tuple<T1, T2>> combination =
  Reduce(table2, new List<Tuple<T1, T2>>(),
    (c, y) =>
    {
      Tuple<T1, T2> row = new Tuple<T1, T2>(x, y);
      if (condition(row))
        c.Add(row);
      return c;
    });
```

Now this call can be used in the lambda of the `Reduce` we use to join the whole table, obtaining:

Listing 7: Join with Reduce

```
1  public static IEnumerable<Tuple<T1, T2>> Join<T1, T2>(IEnumerable<T1> table1, IEnumerable<T2>
       table2, Func<Tuple<T1, T2>, bool> condition)
2  {
3    return
4      Reduce(table1, new List<Tuple<T1, T2>>(),
5      (queryResult, x) =>
6        {
7          List<Tuple<T1, T2>> combination =
8            Reduce(table2, new List<Tuple<T1, T2>>(),
9              (c, y) =>
10             {
11               Tuple<T1, T2> row = new Tuple<T1, T2>(x, y);
12               if (condition(row))
13                 c.Add(row);
14               return c;
15             });
16         queryResult.AddRange(combination);
17         return queryResult;
18       });
19 }
```

In Listing 7 line 4 calls `Reduce` to perform the whole join, that is the combination of each row from `table1` with all the rows of `table2`. At line 8 we call the `Reduce` to create the combination of a single row from `table1` with all the rows of `table2`, filtered according the `condition` of `Join`. At line 11 we create the pair containing a row from `table1` and a row from `table2`. At line 12 we test the condition and add the combination only if the condition is true. At line 16 we add the combinations created at line 8 to the accumulator.

5 Conclusion

In these notes we have presented the `Map-Reduce` paradigm and shown how you can implement SQL statements just by using the `Map` and `Reduce` functions. Moreover we have shown that `Joins` are also possible in `Map-Reduce`. What is left is to implement the grouping functions and aggregate functions on group. This can be done with

`Reduce` as well, but for brevity we do not present it here. The `Map-Reduce` paradigm is widely used in distributed databases, where the `Map` function can be run at the same time on multiple servers, and then their result is collected through the `Reduce` function.