Distributed Query Processing in Secondo

Using the Distributed Algebra 2

October 2015

Ralf Hartmut Güting and Thomas Behr

Faculty for Mathematics and Computer Sience

Database Systems for New Applications

58084 Hagen, Germany

Table of Contents

1 Introduction

In this document we describe how SECONDO can be used for distributed query processing on a cluster of computers. In contrast to earlier approaches which coupled SECONDO with either Hadoop (called Parallel SECONDO) or Cassandra (called Distributed SECONDO), here SECONDO is used on its own. This is possible with the facilities of the algebra called *Distributed2Algebra*, that is, the second version of the *Distributed Algebra*.

Distributed query processing uses one SECONDO instance called the *master* and a set of SECONDO instances called the *workers*. These may run on the same or different computers. Obviously each involved computer requires a SECONDO installation.

2 Passphrase-less Connection

The first thing is to set up an ssh connection from the master to the workers so that one can start SecondoMonitors on the computing nodes without the user having to enter a password. This is done as follows.

(1) In the master's home directory, enter

```
ssh-keygen -t dsa
```

This generates a public key. The system asks for a password, just type return for an empty password. Also type return for the default location to store the public key.

(2) Configure ssh to know the worker computers. Into a file .ssh/config, we put the following:

```
Host *ralf1
HostName 132.176.69.130
User ralf
Host *ralf2
HostName 132.176.69.78
User ralf
```

This makes the two worker computers 130 and 78 available with names *ralf1* and *ralf2* and also sets the user name and home directory on these computers.

(3) Transmit the public key to the target computers.

```
ssh-copy-id -i .ssh/id_dsa.pub <user>@server

ssh-copy-id -i .ssh/id_dsa.pub ralf@132.176.69.78
```

If the master system does not have the command ssh-copy-id, one can use instead:

```
cat ~/.ssh/*.pub | ssh <user>@<server> 'umask 077; cat >>.ssh/
authorized_keys'
```

At this point the system on *ralf2* asks once for a password. Subsequently it is possible to log in on *ralf2* by simply typing

```
ssh ralf2
```

One should log in once on each of the nodes because the system on the master needs to enter each node into its list of known hosts.

3 Setting Up a Cluster

3.1 Micro-Cluster

3.2 Mini-Cluster

3.3 Small Cluster

3.4 Starting and Stopping a Cluster

4 The Algebra

The *Distributed2Algebra* provides operations that allow one SECONDO system to control a set of SECONDO servers running on the same or remote computers. It acts as a client to these servers. One can start and stop the servers, provided SECONDO monitor processes are already running on the involved computers. One can send commands and queries in parallel and receive results from the servers.

The SECONDO system controlling the servers is called the *master* and the servers are called the *workers*.

This algebra actually provides two levels for interaction with the servers. The *lower level* provides operations

- to start, check and stop servers
- to send sets of commands in parallel and see the responses from all servers
- to execute queries on all servers
- to distribute objects and files

The *upper level* is implemented using operations of the lower level. It essentially provides an abstraction called *distributed arrays*. A distributed array has slots of some type *X* which are distributed over a given set of workers. Slots may be of any SECONDO type, including relations and indexes, for example. Each worker may store one ore more slots.

Query processing is formulated by applying SECONDO queries in parallel to all slots of distributed arrays which results in new distributed arrays.

Data can be distributed in various ways from the master into a distributed array. They can also be collected from a distributed array to be available on the master.

In the following, we describe the upper level of the *Distributed2Algebra* in terms of its data types and operations.

4.1 Types

The algebra provides two types of distributed arrays called

- <u>*darray*</u>(*X*) - *distributed array* - and
- <u>*dfarray*</u>(*Y*) - *distributed file array*.

Here *X* may be any Secondo type[1] and the repective values are stored in databases on the workers. In contrast, *Y* must be a relation type and the values are stored in binary files on the respective workers. In query processing, such binary files are transferred between workers, or between master and workers. Figure 1 illustrates both types of distributed arrays. Often slots are assigned in a cyclic manner to servers as shown, but there exist operations creating a different assignment. The implementation of a <u>*darray*</u> or <u>*dfarray*</u> stores explicitly how slots are mapped to servers. The type

[1] Except the distributed types themselves, so it is not possible to nest distributed arrays.

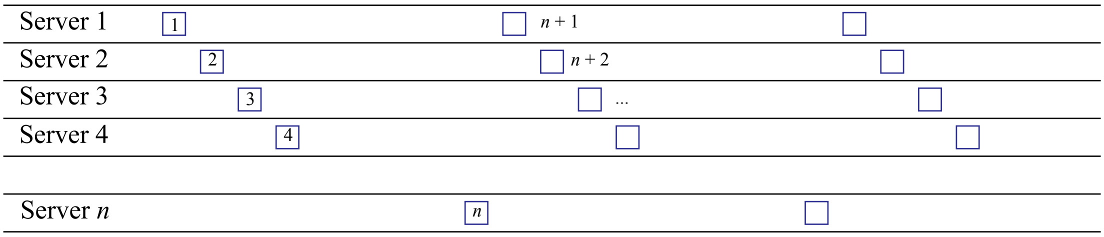

Figure 1: A distributed array. Each slot is represented by a square with its slot number.

information of a *darray* or *dfarray* consists of the set of workers, the type of slots, and the number of slots.

A distributed array is often constructed by partitioning data on the master into partitions P_1, ..., P_m and then moving partitions P_i into slots S_i. This is illustrated in Figure 2.

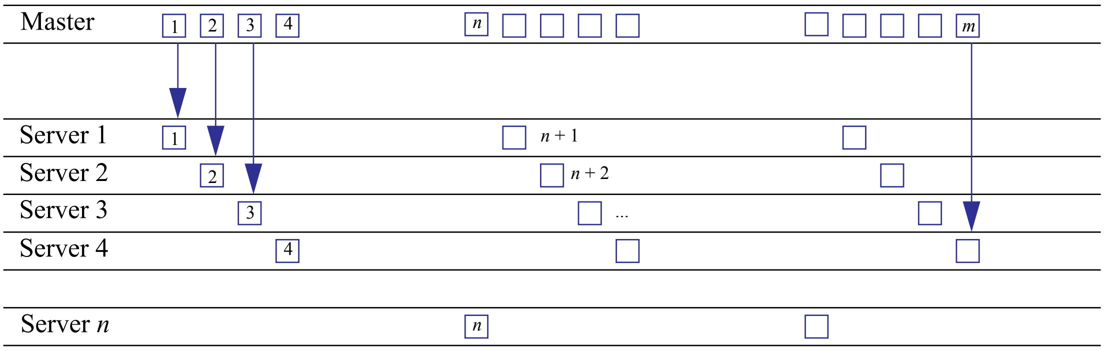

Figure 2: Creating a distributed array by partitioning data on the master.

A third type offered is

- *dfmatrix*(*Y*) - *distributed file matrix*.

Slots *Y* of the matrix must be relation-valued, as for *dfarray*. This type supports redistributing data which are partitioned in a certain way on workers already. It is illustrated in Figure 3.

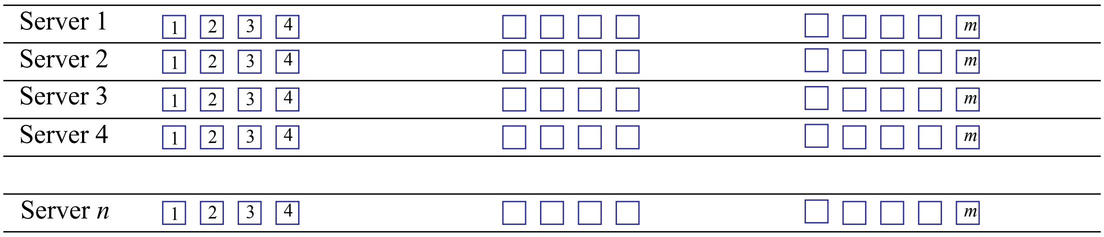

Figure 3: A distributed file matrix.

The matrix arises when all servers partition their data in parallel. In the next step, each partition, that is, each column of the matrix, is moved into one slot of a distributed file array as shown in Figure 4.

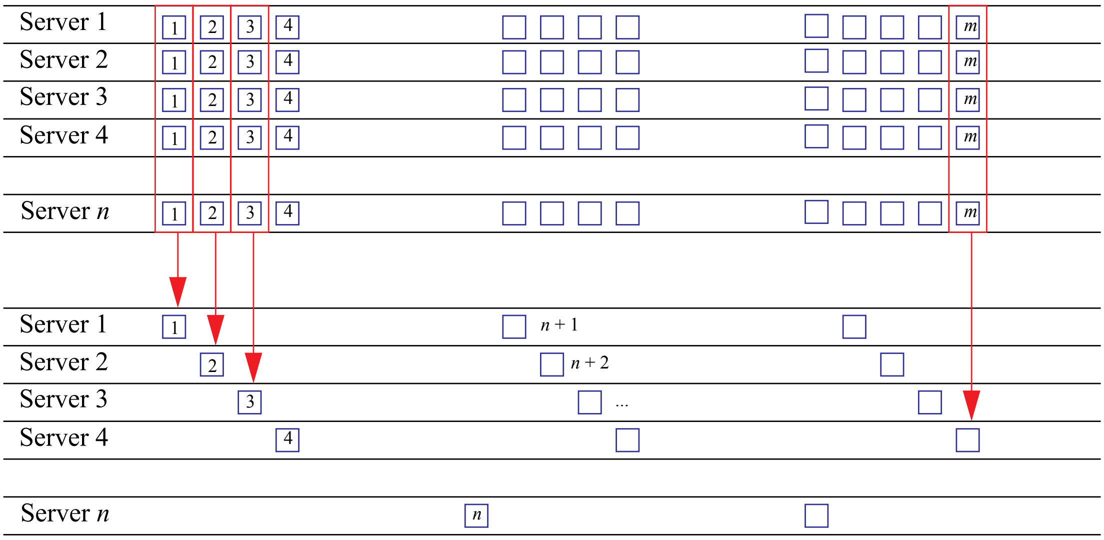

Figure 4: A distributed file matrix is collected into a distributed file array.

4.2 Operations

4.2.1 Distributing Data from the Master to Workers

4.2.2 Processing Distributed Data

4.2.3 Collecting Distributed Data from the Workers to the Master

5 Loading Data on the Master

6 Distributing Data to Workers

6.1 Random Partitioning

We distribute the Roads relation in a random way into distributed files on the mini-cluster:

```
let RoadsB1 = Roads feed dfdistribute3[50, TRUE, Workers14, "RoadsB1"]

2:19min
```

In this case, we distribute the Roads in round-robin fashion into a distributed file array with 50 slots.

6.2 Hash Partitioning

We distribute *Roads* by *Osm_id* into a distributed file array.

```
let RoadsB2 = Roads feed dfdistribute4[hashvalue(.Osm_id, 999997), 50,
Workers14, "RoadsB2"]

1:48min
```

6.3 Range Partitioning

We can efficiently partition a relation for an attribute *A* with a total order[2] in such a way that each slot receives all tuples within an interval of values and different slots have distinct value ranges. This is done by taking a sample and determining on the sample the partition boundary values.

We demonstrate this by distributing the subset of Roads that have a name into ranges of the road name. We first determine how many Roads have (non-blank) names.

```
query Roads feed filter[.Name starts "                 "] count

12.16 seconds, 13.48, Result: 941513
```

Hence there are 1505462 - 941513 = 563949 road tuples that have a name. Out of these we wish to take a sample of 50 * 100 = 5000 tuples. That is a fraction of 1/113 tuples.

```
let S = Roads feed filter[not(.Name starts "                 ")]
filter[(randint(999997) mod 113) = 0]
project[Name] sortby[Name] consume

41 seconds, 59, 15, 15, 20, 15; Result size: 5042
```

The 5042 tuples in the sample we wish to divide into about 50 partitions of size about 101. We now determine the attribute values that lie on partition boundaries.

```
let Boundaries = S feedproject[Name] addcounter[C, 0]
  filter[(.C mod 101) = 100] addcounter[D, 1]
```

[2] In fact, each attribute type technically does provide a total order as it is needed for sorting and duplicate removal.

```
  project[Name, D] consume

0.3 seconds
```

We need to add a pair containing an attribute value smaller than all occurring values and set this value for slot 0.

```
query Boundaries inserttuple["", 0] consume

0.2 seconds

query memload("Boundaries")

0.03 seconds

query mcreateAVLtree("Boundaries", "Name")

0.023, 0.011 seconds
```

We can now distribute Roads, using the main-memory-AVLtree "Boundaries_Name" to determine the slot number for each tuple.

```
let RoadsB3 = Roads feed filter[not(.Name starts "                  ")]
dfdistribute4["Boundaries_Name" "Boundaries" matchbelow[.Name]
extract[D], 50, Workers14, "RoadsB3"]

42.58 seconds, 59.03
```

A range-partitioned relation can easily be sorted on the same attribute:

```
let RoadsB3S = RoadsB3 dmap["RoadsB3S", . sortby[Name]]

5.58 seconds, 5.53, 6.9
```

6.4 Spatial Partitioning

7 Distributed Loading

8 Querying

8.1 Selection

8.1.1 By Scanning

8.1.2 Creating a Standard Index

8.1.3 Using a Standard Index

8.1.4 Creating a Spatial Index

8.1.5 Using a Spatial Index

8.2 Join

8.2.1 Standard Equijoin

8.2.2 Spatial Join

8.2.3 Index-Based Equijoin

8.2.4 Index-Based Spatial Join

8.3 Aggregation

8.3.1 Counting

8.3.2 Sum, Average

8.4 Sorting

By parallel sorting we mean the following problem. Given a partitioned relation $R = R_0, ..., R_{n\text{-}1}$ to be sorted by attribute A, the result should be relation S partitioned into $S_0, ..., S_{n\text{-}1}$. Here n can be the number of servers (or of slots of a distributed array where the number of slots is larger than that of servers). Let T_i denote the sequence of tuples of partition S_i. Then the concatenation of all sequences, $T_0 \circ ... \circ T_{n\text{-}1}$, is the relation R sorted by A.

The general strategy applied is the following [O08, TLX13].

1. By sampling, determine attribute values $a_0, ..., a_n$ such that the number of tuples of R with attribute value in the interval $[a_k, a_{k+1}]$ is roughly the same for all $k = 0, ..., n\text{-}1$.
2. Then redistribute R such that tuples with attribute value within $[a_k, a_{k+1}]$ are sent to server or slot k.
3. Let the server in charge of slot k sort its partition locally.

The following sequence of SECONDO commands sorts *Roads* by *Osm_id*:

```
10 query RoadsB1
11   dmap["", . extend[Randint: randint(999997)]
12     filter[(.Randint mod 61) = 0] project[Osm_id] ]
13   dsummarize
14 sortby[Osm_id]
15 addcounter[C, 0] filter[(.C mod 500) = 499]
16     addcounter[D, 1] project[Osm_id, D]
17     [const rel(tuple([Osm_id: string, D: int])) value ( ("" 0) )] feed
18     concat
19 intstream(0, 13) namedtransformstream[N] product
20 dfdistribute2[N, 14, Workers14, "X100"]
21 dmap["", . letmconsume["Boundaries"]]
22
23 query [const rel(tuple([Osm_id: string, D: int])) value ()] feed
24   letmconsume["Boundaries"]
25
26 query RoadsB1 dmap["", mcreateAVLtree("Boundaries", "Osm_id")]
27
28 query mcreateAVLtree("Boundaries", "Osm_id")
29
30 let RoadsSortedOsm_id = RoadsB1
31   partition["Boundaries_Osm_id" "Boundaries" matchbelow[.Osm_id]
32     extract[D], "X102", 0]
33   areduce["X103", . sortby[Osm_id], 1238]
```

This algorithm uses the following steps to sort a distributed Relation R by attribute A. Steps 1 through 7 (lines 10 - 28) serve to provide the partition boundaries on each server. Steps 8 to 9 (lines 30-33) perform the actual sorting.

1. From each relation R_i, $i = 0, ..., p$, p the number of partitions, take a random sample of size 500 of the attribute values and send it to the master. (Lines) [10 - 13].
2. On the master, sort the union of samples. Add a counter to the ordered tuple stream and select every 500th element. Number these elements with a counter D, starting from 1. These pairs (A, D) will form the boundaries between partitions, where D is the partition number. [14 - 16].

3. Add a pair with a minimal value for *A* and partition number 0. This will be the left boundary of the leftmost partition in the ordered sequence of partitions. [17 - 18].
4. Multiply the set of boundary pairs with a relation with attribute *N* containing the numbers {0, ..., *n* - 1}, where *n* is the number of servers. [19].
5. Distribute the resulting triples (*A*, *D*, *N*) by *N* to the servers. [20].
6. On each server, create a main memory relation called *Boundaries* for the triples it received. On the master, create such a relation as well (this is needed for type checking on the master). [21-24].
7. Create an AVLtree on *Boundaries* indexing by attribute *A*. Create the same index on the master. [26 - 28].
8. For each partition R_i, determine for each tuple with *A*-value a_x the value d_k of the tuple (a_k, d_k) indexed in the AVL-tree on Boundaries whose a_k value is the greatest one that is smaller than a_x; repartition *R* by *D*. Result is a matrix. [30 - 32].
9. Sort the tuples of each column R_j by *A*. [33].

Now the complete relation *R* is ordered by partition (slot) numbers 0, ..., *p* with the smallest slot having the first and the largest slot the last tuples of the sorted order. Within each slot, R_j is ordered by *A*.

A script containing lines 10 - 30 can be found in secondo/bin/Scripts/ParallelSort.sec. The running time for sorting Roads by Osm_id is about 53 seconds.

In contrast, sequential execution on the master using 6GB memory requires 12:27min.

```
let RoadsSOsm_id = Roads feed sortby[Osm_id] consume
```

References

[O08] Owen O’Malley, TeraByte Sort on Apache Hadoop. Technical Report, Yahoo, 2008.

[TLX13] Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal Mapreduce Algorithms. Proc. ACM SIGMOD 2013, 529-540.