SECONDO

Distributed Query Processing in Secondo

Using the Distributed Algebra 2

November 2015

Ralf Hartmut Güting and Thomas Behr

FernUniversität in Hagen

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 132.176.69.160
User ralf
Host *ralf2
HostName 132.176.69.98
User ralf
```

This makes the two worker computers 160 and 98 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@132.176.69.98
```

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

We use a mini-cluster consisting of two computers running Ubuntu 14.04. The two computers provide the following resources:

- ralf1: 4 disks, 6 cores, 16 GB main memory
- ralf2: 4 disks, 8 cores, 32 GB

On *ralf1*, we use one disk exclusively for the master, the remaining for workers. Regarding cores and main memory we let the master overlap with workers. The main case of master and workers working simultaneously is data distribution from the master to the workers and back. In this case, one worker is not very active compared to the master (who is quite busy) so that the overlapping core should not be a problem. Similarly there is not a lot of memory used in these transfers. We therefore configure the system as follows:

- Master (ralf1): 1 disk, one core, 4 GB
- Workers:
  - ralf1: 3 disks, 6 cores, 2000 MB per core (about 12 GB)
  - ralf2: 4 disks, 8 cores, 3600 MB per core (about 28 GB)

We will set up one database per disk and one worker per core.

3.1 Setting Up the Monitors

Per worker disk we need to start one SecondoMonitor. We can start the monitors using a script `remoteMonitors` from the `secondo/bin` directory. It takes the following parameters (see also `remoteMonitors.readme`):

```
remoteMonitors <description file> <action>
```

Here the `<description file>` contains entries describing the monitors to be started, one line per monitor. Such a line has the format

```
<Server> <Configuration file>  [<user>]
```

Here `<Server>` is the IP address or the name of the node on which the monitor is to be started, and the second parameter is the `SecondoConfig.ini` file version to be used. The third parameter is needed to specify the name of the user in case it is different on the remote computer to be used.

The `<action>` is one of `start`, `check`, or `stop` with the obvious meanings.

Here we provide a description file `ClusterRalf7` with the following content:

```
132.176.69.160 "SecondoConfig.ini".160.1471
132.176.69.160 "SecondoConfig.ini".160.1472
132.176.69.160 "SecondoConfig.ini".160.1473
132.176.69.98 "SecondoConfig.ini".98.1474
```

```
132.176.69.98 "SecondoConfig.ini".98.1475
132.176.69.98 "SecondoConfig.ini".98.1476
132.176.69.98 "SecondoConfig.ini".98.1477
```

The "SecondoConfig.ini" files need to lie in the secondo/bin directory of the respective computers (160 and 98). They are obtained from the standard `SecondoConfig.ini` by modifying

1. The database directory to be used.
   ```
   SecondoHome=/discA/secondo-databases
   ```
2. The host IP address
   ```
   SecondoHost=132.176.69.160
   ```
3. The port
   ```
   SecondoPort=1271
   ```
4. The global memory per server. We have:
   - 160: GlobalMemory=2000
   ```
   GlobalMemory=2000
   ```
   - 98: GlobalMemory=3600

The `secondo-databases` directories on the respective disc locations need to be created before using the `remoteMonitors` script.

3.2 Starting and Stopping the Monitors

We can now start the monitors:

```
remoteMonitors ClusterRalf7 start
```

As a result, we see:

```
ralf@ralf-ubuntu6:~/secondo/bin$ remoteMonitors ClusterRalf7 start
Try to start monitor on server 132.176.69.160
Monitor is running now at port 1471
Try to start monitor on server 132.176.69.160
Monitor is running now at port 1472
Try to start monitor on server 132.176.69.160
Monitor is running now at port 1473
Try to start monitor on server 132.176.69.98
Monitor is running now at port 1474
Try to start monitor on server 132.176.69.98
Monitor is running now at port 1475
Try to start monitor on server 132.176.69.98
Monitor is running now at port 1476
Try to start monitor on server 132.176.69.98
Monitor is running now at port 1477
```

Similar listings can be seen with the actions check and stop.

We can stop all monitors using the command

```
remoteMonitors ClusterRalf7 stop
```

3.3 Setting Up Workers

Workers are defined in a relation of a database to be used. We use one worker per core.

```
let Workers14 = [const rel(tuple([Host: string, Port: int, Config: 
string])) value
(
  ("132.176.69.160" 1471 "SecondoConfig.ini")
  ("132.176.69.160" 1472 "SecondoConfig.ini")
  ("132.176.69.160" 1473 "SecondoConfig.ini")
  ("132.176.69.98" 1474 "SecondoConfig.ini")
  ("132.176.69.98" 1475 "SecondoConfig.ini")
  ("132.176.69.98" 1476 "SecondoConfig.ini")
  ("132.176.69.98" 1477 "SecondoConfig.ini")
  ("132.176.69.160" 1471 "SecondoConfig.ini")
  ("132.176.69.160" 1472 "SecondoConfig.ini")
  ("132.176.69.160" 1473 "SecondoConfig.ini")
  ("132.176.69.98" 1474 "SecondoConfig.ini")
  ("132.176.69.98" 1475 "SecondoConfig.ini")
  ("132.176.69.98" 1476 "SecondoConfig.ini")
  ("132.176.69.98" 1477 "SecondoConfig.ini")

)]
```

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 provided 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

- *darray*(*X*) - *distributed array* - and
- *dfarray*(*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 *darray* or *dfarray* stores explicitly how slots are mapped to servers. The type information of a *darray* or *dfarray* consists of the set of workers, the type of slots, and the number of slots.

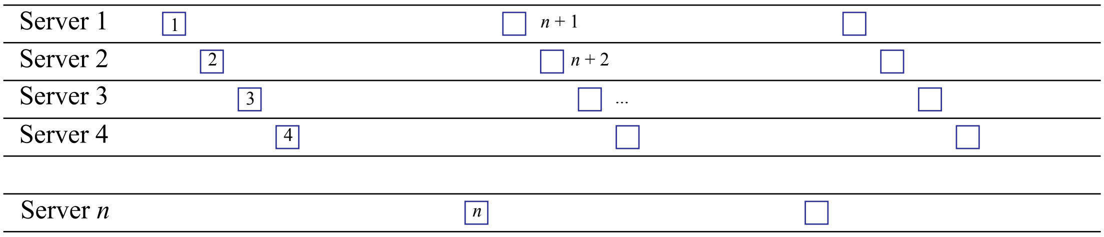

Figure 1: A distributed array. Each slot is represented by a square with its slot number.

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.

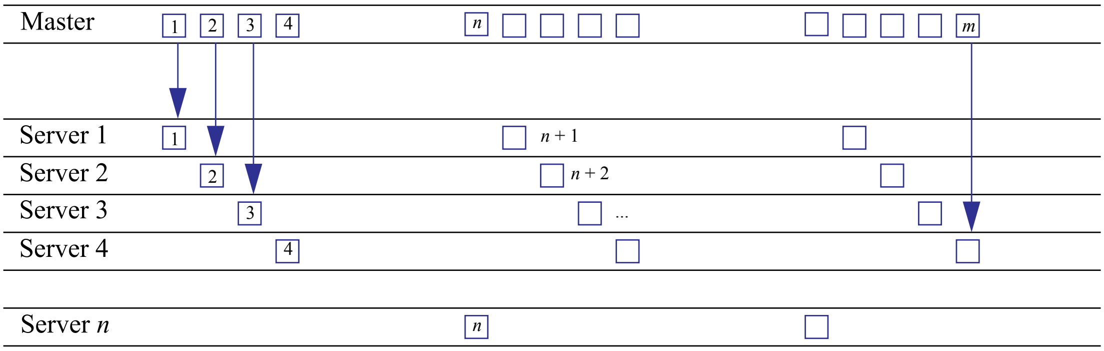

Figure 2: Creating a distributed array by partitioning data on the master.

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.

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

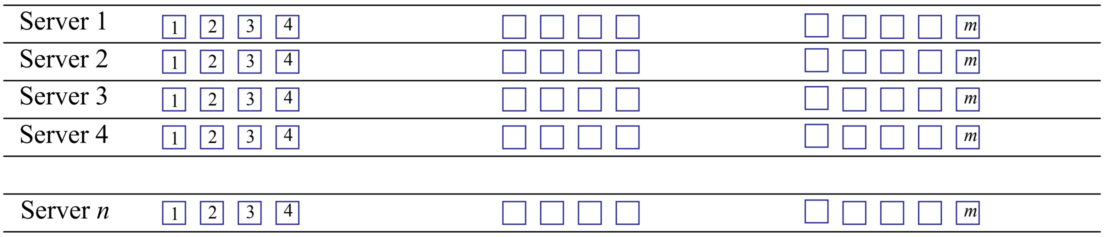

Figure 3: A distributed file matrix.

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.

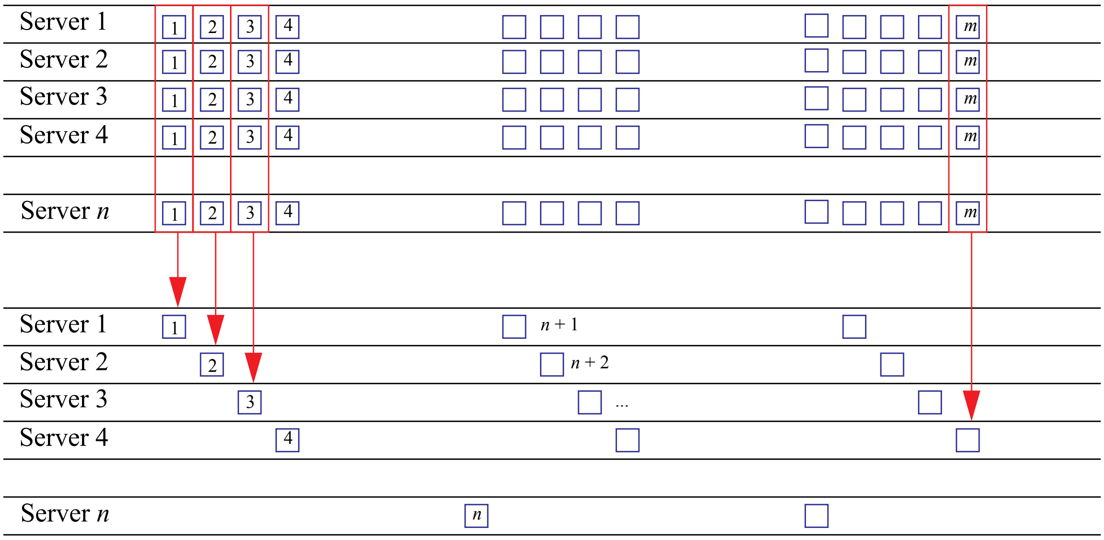

Figure 4: A distributed file matrix is collected into a distributed file array.

4.2 Operations

The following classes of operations are available:

- Distributing Data to the Workers
- Distributed Processing by the Workers
  - Applying a Function (Secondo Query) to Each Field of a Distributed Array
  - Applying a Function to Each Pair of Corresponding Fields of two Distributed Arrays (Supporting Join)
  - Redistributing Data on Workers
  - Adaptive Processing of Partitioned Data
- Collecting Data from the Workers

4.2.1 Distributing Data to the Workers

Operations:

Operation	Meaning
ddistribute2, dfdistribute2	Distributes a stream of tuples into a distributed array. The d-variant creates a *darray*, the df-variant a *dfarray*. Parameters are an integer attribute, the number of slots and a Workers relation in the format shown in Section 3.3. A tuple is inserted into the slot corresponding to its attribute value modulo the number of slots. See Figure 2.
ddistribute3, dfdistribute3	Distributes a stream of tuples into a distributed array. Parameters are an integer *i*, a Boolean *b*, and the Workers. Tuples are distributed round robin into *i* slots, if *b* is true. Otherwise slots are filled sequentially, each to capacity *i*, using as many slots as are needed.
ddistribute4, dfdistribute4	Distributes a stream of tuples into a distributed array. Here a function instead of an attribute decides where to put the tuple.
share	An object of the master database whose name is given as a string argument is distributed to all worker databases. Workers are given by a *darray* or *dfarray* argument.

4.2.2 Distributing Processing by the Workers

Operations:

Operation	Meaning
dloop, dmap	Evaluates a Secondo query on each field of a distributed array of type *darray* (**dloop**) or *dfarray* (**dmap**). Operator **dloop** returns a *darray*, **dmap** returns a *dfarray* if the result is representable as such (i.e. for a tuple stream or a relation result), otherwise a *darray*.
dloop2, dmap2	Binary variants of the previous operations mainly for processing joins. Always two files with the same index are arguments to the query.
partition, partitionF	Partitions the fields of a *darray* or *dfarray* by a function (similar to **ddistribute4** on the master). Result is a *dfmatrix*. An integer parameter decides whether the matrix will have the same number of slots as the argument array or a different one. Variant **partitionF** allows one to manipulate the input relation of a field e.g. by filtering tuples or by adding attributes, before the distribution function is applied. See Figure 3.

Operation	Meaning
collect2	Collects the columns of a *dfmatrix* into a *dfarray*. See Figure 4.
areduce	Applies a function (Secondo query) to all tuples of a partition (column) of a *dfmatrix*. In contrast to all previous operations it is not predetermined which worker will read the column and evaluate it. Instead, when the number of slots *s* is larger than the number of workers *m*, then each worker *i* gets assigned slot *i*, for *i* = 0, ..., *m*-1. From then on, the next worker who finishes its job, will process the next slot. This is very useful to compensate for speed differences of machines or size differences in assigned jobs.
areduce2	Binary variant of **areduce**, mainly for processing joins.

4.2.3 Collecting Data From the Workers

Operations:

Operation	Meaning
dsummarize	Collect all tuples from a *darray* or *dfarray* into a tuple stream on the master.
getValue	Convert a distributed array over atomic field values into a local array.
tie	Apply aggregation to a local array, e.g., to determine the sum of field values.

5 Getting Spatial Data to the Master

We use OpenStreetMap data about the German state of North-Rhine-Westphalia obtained in the form of shapefiles from GeoFabrik. On `http://download.geofabrik.de/` one can navigate a bit selecting Europe, then Germany. Then from the table Sub Regions in the row for Nordrhein-Westfalen, download

```
http://download.geofabrik.de/europe/germany/nordrhein-westfalen-lat-
est.shp.zip
```

You can also use this link to download directly, if it is still available. Unpack the zip file into a directory `nrw` within `secondo/bin`.

The database has eight kinds of objects, for example, Roads, Buildings, Waterways.

In directory `secondo/bin` start a Secondo system without transactions (for loading data) using the command

```
SecondoTTYNT
```

At the prompt, enter

```
Secondo => @Scripts/nrwImportShape.SEC
```

The contents of the file `nrwImportShape` are shown here:

```
# Importing NRW Data
close database

create database nrw

open database nrw

let Roads = dbimport2('../bin/nrw/roads.dbf') addcounter[No, 1]
shpimport2('../bin/nrw/roads.shp') namedtransformstream[GeoData]
addcounter[No2, 1] mergejoin[No, No2] remove[No, No2]
filter[isdefined(bbox(.GeoData))] validateAttr consume

let Waterways = dbimport2('../bin/nrw/waterways.dbf') addcounter[No, 1]
shpimport2('../bin/nrw/waterways.shp') namedtransformstream[GeoData]
addcounter[No2, 1] mergejoin[No, No2] remove[No, No2]
filter[isdefined(bbox(.GeoData))] validateAttr consume

...
```

This is repeated to load all eight relations. The script creates relations *Roads*, *Waterways*, etc. within a new database *nrw*. Recently, this database is fairly large, for example, has 5.8 mio buildings. It takes roughly one hour to run this script.

The easiest way to get information about a database is to let the optimizer collect it. Opening the database in the query optimizer (SecondoPL) collects information about the existing database objects which can be displayed with the predicate *showDatabase*.

```
2 ?- showDatabase.

Collected information for database 'nrw':

Relation Natural(Auxiliary objects:)
  AttributeName               Type          Memory  DiskCore    DiskLOB
  GeoData                     region        152.0      152.0    4500.864
  Type                        string        64.0       22.0     0
  Name                        string        64.0       54.0     0
  Osm_id                      string        64.0       17.0     0

  Indices:

  Ordering:  []

  Cardinality:   129317
  Avg.TupleSize: 4745.864 = sizeTerm(480.0,245.0,4500.864)
  (Tuple size in memory is 136 + sum of attribute sizes.)

Relation Places(Auxiliary objects:)
  AttributeName               Type          Memory  DiskCore    DiskLOB
  GeoData                     point         32.0       32.0     0
  Population                  int           16.0       5.0      0
  Type                        string        64.0       22.0     0
  Name                        string        64.0       54.0     0
  Osm_id                      string        64.0       17.0     0

  Indices:

  Ordering:  []

  Cardinality:   15908
  Avg.TupleSize: 130.0 = sizeTerm(376.0,130.0,0)
  (Tuple size in memory is 136 + sum of attribute sizes.)

...

(Type 'showDatabaseSchema.' to view the complete database schema.)
true.

3 ?-
```

6 Distributing Data to Workers

The following ways of data distribution are of interest:

- random partitioning
- partitioning by standard attribute: hash partitioning
- partitioning by standard attribute: range partitioning
- partitioning by spatial attribute: spatial partitioning
- replication

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["RoadsB1", 50, TRUE, Workers14]

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 array.

```
let RoadsB2 = Roads feed ddistribute4["RoadsB2", hashvalue(.Osm_id,
999997), 50, Workers14]
```

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]
  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 "                 ")]
ddistribute4["RoadsB3", "Boundaries_Name" "Boundaries" matchbelow[.Name]
extract[D], 50, Workers14]

42.58 seconds, 59.03
```

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

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

Spatial partitioning is based on an attribute of a spatial data type such as *point*, *line*, or region. The idea is to use a regular grid where cells are numbered, covering all spatial attribute values. For each tuple, the bounding box of its spatial attribute is determined, that is, the smallest axis-parallel rectangle enclosing the geometry. The bounding box *b* is placed into the grid and the cell numbers of cells overlapping *b* are computed. For each cell number returned, one copy of the tuple is put into a partition corresponding to this cell number (modulo the number of partitions). This is illustrated in Figure 5.

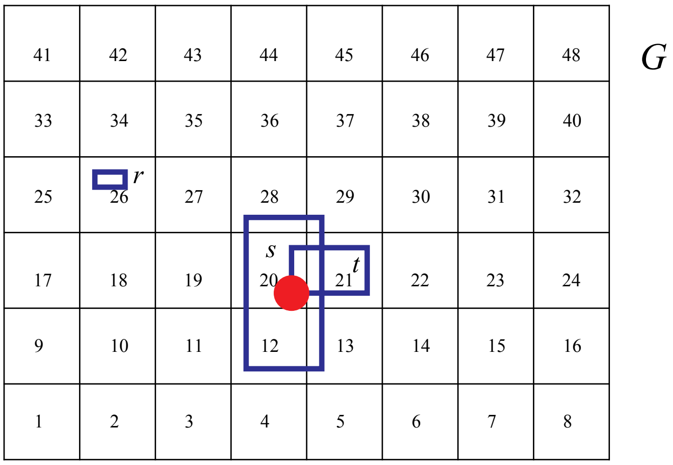

Figure 5: Rectangles *r*, *s*, and *t* are mapped to cell numbers using grid *G*.

The first step for constructing the grid is to determine a rectangle enclosing all geometries.[3] For the database NRW, this can be done by a query:

```
query
  Buildings feed projectextend[; Box: bbox(.GeoData)]
  Landuse feed projectextend[; Box: bbox(.GeoData)] concat
  Natural feed projectextend[; Box: bbox(.GeoData)] concat
  Places feed projectextend[; Box: bbox(.GeoData)] concat
  Points feed projectextend[; Box: bbox(.GeoData)] concat
  Railways feed projectextend[; Box: bbox(.GeoData)] concat
  Roads feed projectextend[; Box: bbox(.GeoData)] concat
  Waterways feed projectextend[; Box: bbox(.GeoData)] concat
  aggregateB[Box; fun(r1: rect, r2: rect) r1 union r2;
    [const rect value undef]]
```

The result is:

```
[const rect value (7.29786 7.75405 51.3778 51.6712)]
```

[3] This can also be done in other ways, e.g., reading coordinates from a map.

Next we create a 20 x 20 grid covering this area. It is no problem if the grid is a bit larger than the enclosing rectangle.

```
let grid = [const cellgrid2d value (7.29 51.37 0.025 0.025 20)]
```

Here the five parameters for the grid are the left bottom corner (7.29 51.37), the cellwidth and cellheight (each 0.025) and the number of cells in a row (20).

With an operator called **cellnumber** we can now assign to each spatial object in the area of the grid the numbers of the grid cells which are covered by its bounding box. The **cellnumber** operator takes a rectangle and a grid definition and returns a stream of numbers of the cells covered by the rectangle.

```
let RoadsB4 = Roads feed
  extendstream[Cell: cellnumber(bbox(.GeoData), grid)]
  dfdistribute2["RoadsB4", Cell, 50, Workers14]
```

Here the **extendstream** operator produces as many copies of each argument tuple as numbers are returned by **cellnumber**.

Spatial distribution obviously introduces duplicates of the original tuples. If later we want to apply an operation exactly to the original tuples, excluding the duplicates, for example, counting them, it is a good idea to designate one of the copies as the original one. Hence we add a Boolean attribute which is true for the first or only copy and false for further copies.

```
let BuildingsB4a = Buildings feed
  extendstream[Cell: cellnumber(bbox(.GeoData), grid)]
  extend[Original: .Cell = cellnumber(bbox(.GeoData), grid)
    transformstream extract[Elem]
ddistribute2["BuildingsB4a", Cell, 50, Workers14]
```

Furthermore, for computing distance-based spatial joins, it is necessary to perform the spatial join operation on bounding boxes which have been enlarged by the distance. In the distributed case we may want to prepare for such joins by enlarging rectangles already in the distribution. Suppose we want to support distance based joins for buildings within a distance of 500 meters. In geographical coordinates in NRW, a difference of 0.01 in x-direction corresponds to about 700 meters, in y-direction 0.01 correspond to about 1.1 km. Hence if we enlarge the bounding boxes of buildings by 0.01 in each direction we should be sure to overlap with the bounding boxes of other objects within 500 meters distance. Hence we distribute the *Buildings* relation as follows:

```
let BuildingsB4 = Buildings feed
  extend[EnlargedBox: enlargeRect(bbox(.GeoData), 0.01, 0.01)]
  extendstream[Cell: cellnumber(.EnlargedBox, grid)]
  extend[Original: .Cell = cellnumber(.EnlargedBox, grid)
    transformstream extract[Elem]
ddistribute2["BuildingsB4", Cell, 50, Workers14]
```

6.5 Replication

An object in the master database, atomic or relation, can be moved in a simple way into the worker databases by the share operation. Suppose we want to *replicate* the Roads relation.

```
query share("Roads", FALSE, RoadsB1)
```

The object name is supplied as a string parameter. The second, Boolean parameter decides whether an existing object in the worker database should be replaced. Any existing *darray* or *dfarray* can be used to identify the workers who should do the job.

7 Querying

7.1 Selection

We consider selection by a standard attribute and by a spatial attribute.

```
select count(*) from Roads_d where Name = "Universitätsstraße"
```

```
select * from Buildings_d where GeoData intersects eichlinghofen
```

Here *eichlinghofen* is an object of type *region* designating the area of a suburb of Dortmund, Eichlinghofen.

7.1.1 By Scanning

Selection by standard attribute:

```
query RoadsB1 dmap["", . filter[.Name = "Universitätsstraße"] count]
  getValue tie[. + ..]
```

For the spatial selection, we need to distribute the object *eichlinghofen*, if it is not yet distributed.

```
query share("eichlinghofen", TRUE, BuildingsB4)
```

Usually all distributed arrays use the same set of workers and we can use any of them to refer to the active workers.

```
query BuildingsB4 dmap["", . feed filter[.Original]
  filter[.GeoData intersects eichlinghofen] ]
  dsummarize consume
```

BuildingsB4 is spatially distributed and has duplicates. Therefore we need to insert the filtering on the *Original* attribute.

7.1.2 Creating a Standard Index

We can create an index on RoadsB2 which is a *darray* (not a *dfarray*).

```
let RoadsB2_Name = RoadsB2 dloop["RoadsB2_Name", . createbtree[Name] ]
```

Result is a distributed B-tree.

7.1.3 Using a Standard Index

```
query RoadsB2_Name RoadsB2
  dloop2["", . .. exactmatch["Universitätsstraße"] ] getValue tie[. + ..]
```

The **dloop2** operator allows one to mention the two arguments required by the **exactmatch** operator.

7.1.4 Creating a Spatial Index

BuildingsB4 is a darray, hence suitable for adding an R-tree index. We construct the index by bulkloading.

```
let BuildingsB4_GeoData = BuildingsB4 dloop["",
  . feed addid extend[Box: scalerect(.EnlargedBox, 1000000.0, 1000000.0)]
  sortby[Box] remove[Box] bulkloadrtree[EnlargedBox] ]
```

Spatial data are sorted into z-order. For this to be effective on geographical coordinates, we need to scale up the bounding boxes before sorting. We use the enlarged boxes constructed for distribution to be able to use the index also for distance-based queries.

7.1.5 Using a Spatial Index

Note that one needs to check whether *eichlinghofen* is distributed before executing this query. The *grid* also needs to be distributed.

```
query share("grid", TRUE, BuildingsB4)

query BuildingsB4_GeoData BuildingsB4
  dmap2["", . .. windowintersects[eichlinghofen]
  filter[gridintersects(grid, .EnlargedBox, bbox(eichlinghofen), .Cell)]
  filter[.GeoData intersects eichlinghofen] ]
  dsummarize consume
```

The **gridintersects** operator is needed to avoid duplicates. This is explained below for spatial join.

7.2 Join

We can distinguish three kinds of joins:

- equijoin on a standard attribute
- spatial join
- arbitrary join

The first two may also be supported by distributed indices.

7.2.1 Equijoin

Find pairs of distinct objects of class Natural with the same name.

```
select * from [Natural_d as n1, Natural_d as n2]
where [n1:Name = n2:Name, n1:Osm_id < n2:Osm_id]
```

We have to distinguish the two cases:

1. A copy of *Natural* is available which is distributed by attribute *Name*.

2. This is not the case.

(1) Distributed by Join Attribute

Let us assume *NaturalB1* is *Natural* distributed by attribute *Name* as a *darray*.

```
query NaturalB1 dmap["",
  . feed {n1} . feed {n2} itHashJoin[Name_n1, Name_n2]
  filter[Osm_id_n1 < Osm_id_n2]]
  dsummarize consume
```

Because this is a self-join, we need only one distributed version of *Natural* and can use **dmap** insteead of **dmap2**.

(2) Arbitrary Distribution

We assume *NaturalB2* is a *dfarray* distributed not by attribute *Name* without duplicates. Hence we first need to redistribute.

```
query NaturalB2 partition["", hashvalue(.Name, 999997), 0]
  areduce["", . {n1} . {n2} itHashJoin[Name_n1, Name_n2]
  filter[Osm_id_n1 < Osm_id_n2], 1238]
  dsummarize consume
```

Because this is a self-join, we need to repartition only once and can use **areduce** instead of **areduce2** for the join. The last argument in **areduce**, 1238, is just a port number that can be used for data transfer. The same port number can be used in different operations of a query.

7.2.2 Spatial Join

```
select count(*) from [Roads_d as r, Waterways_d as w]
where r:GeoData intersects w:GeoData
```

We need to distinguish whether the two arguments are distributed spatially or not.

(1) Both arguments are distributed by spatial attributes

Let *RoadsB4* and *WaterwaysB4* be the two spatially distributed relations (as *dfarrays*).

```
query RoadsB4 WaterwaysB4 dmap2["",
  . {r} .. {w} itSpatialJoin[GeoData_r, GeoData_w]
  filter[.Cell_r = .Cell_w]
  filter[gridintersects(grid, bbox(.GeoData_r),
    bbox(.GeoData_w), .Cell_r)]
  filter[.GeoData_r intersects .GeoData_w] count ]
  getValue tie[. + ..]
```

Each partition covers a set of grid cells. If the bounding box b_r of a road intersects the bounding box b_w of a waterway, then there must exist one or more cells (and hence partitions) where they both appear. Hence the spatial join executed on each partition will connect the two tuples.

However, this may happen more than once. The boxes may overlap several cells which then appear in different partitions. To avoid duplicates in the result, the operator **gridintersects** allows one to select only one of the results. It takes as arguments the grid definition, the two rectangles, and a cell number. If the two rectangles intersect, it computes their intersection. The bottom left point of the intersection can lie in only one cell *c* (cell boundaries belong to only one cell). If the cell number argument *d* is equal to *c* then the operator returns true, otherwise false. This is illustrated in Figure 5 where the bottom left point of the intersection of rectangles *s* and *t* lies in cell 20. Hence the result is reported only once, for cell 20.

Two further tests are needed: (i) It is possible that the two rectangles overlap in different cells but in the same partition. Therefore we check that the cell numbers are equal (`.Cell_r = .Cell_w`). (ii) The spatial join operation generally is only a filter as it checks that bounding boxes overlap. It is still necessary to evaluate the given predicate on the exact geometries (`.GeoData_r intersects .GeoData_w`).

(2) Not distributed by spatial attributes

In this case, it is necessary to repartition those arguments that are not spatially partitioned. Let us assume this is the case for both arguments, given as *RoadsB1* and *WaterwaysB1*, both of type darray. Also, the *grid* is already distributed. After repartitioning, the query is the same as in the previous case.

```
query
  RoadsB1 partitionF["",
    . extendstream[Cell: cellnumber(bbox(.GeoData), grid)], .Cell, 0]
  WaterwaysB1 partitionF["",
    . extendstream[Cell: cellnumber(bbox(.GeoData), grid)], .Cell, 0]
  areduce2["",
    . {r} .. {w} itSpatialJoin[GeoData_r, GeoData_w]
    filter[.Cell_r = .Cell_w]
    filter[gridintersects(grid, bbox(.GeoData_r),
      bbox(.GeoData_w), .Cell_r)]
    filter[.GeoData_r intersects .GeoData_w] count ]
  getValue tie[. + ..]
```

Expressions in the Select Clause

We consider a further variant of this join query which not only finds pairs of intersecting roads and waterways, but also returns their intersection.

```
select [r:Osm_id, r:Name, w:Osm_id, w:Name,
  intersection(r:GeoData, w:GeoData) as BridgePosition]
from [Roads_d as r, Waterways_d as w]
where r:GeoData intersects w:GeoData
```

There are two ways to handle this:

- the projection and derivation of new attributes is handled on the master
- it is done by the workers

In the first case, we extend the previous query as follows:

```
query RoadsB4 WaterwaysB4 dmap2["",
  . {r} .. {w} itSpatialJoin[GeoData_r, GeoData_w]
  filter[.Cell_r = .Cell_w]
  filter[gridintersects(grid, bbox(.GeoData_r),
    bbox(.GeoData_w), .Cell_r)]
  filter[.GeoData_r intersects .GeoData_w] ]
dsummarize
projectextend[Osm_id_r, Name_r, Osm_id_w, Name_w; BridgePosition:
  intersection(GeoData_r, GeoData_w]
consume
```

If the result is large, the computation of the intersections is actually the most expensive part. Hence it highly desirable to let it be executed in parallel by the workers. This is done in the following version of the query:

```
query RoadsB4 WaterwaysB4 dmap2["",
  . {r} .. {w} itSpatialJoin[GeoData_r, GeoData_w]
  filter[.Cell_r = .Cell_w]
  filter[gridintersects(grid, bbox(.GeoData_r),
    bbox(.GeoData_w), .Cell_r)]
  filter[.GeoData_r intersects .GeoData_w]
  projectextend[Osm_id_r, Name_r, Osm_id_w, Name_w; BridgePosition:
    intersection(GeoData_r, GeoData_w]
]
dsummarize
consume
```

7.2.3 General Join

A join with an arbitrary join condition can only be evaluated by checking all pairs of tuples in the Cartesian product of the two relations.

```
select * from [Roads_d as r, Waterways_d as w]
where r:Name contains w:Name
```

This can be evaluated only if one of the two relations is distributed, the other replicated. Let *RoadsB1* be distributed (as a *dfarray*, without duplicates), *Waterways* replicated. The replicated relation has the same name on all workers as on the master.

```
query RoadsB1 dmap["", . {r} Waterways feed {w}
  symmjoin[.Name_r contains ..Name_w] ]
  dsummarize
  consume
```

7.2.4 Index-Based Equijoin

We consider the following query: Find all pairs of distinct roads with the same name.[4]

```
select * from [Roads_d as r1, Roads_d]
where [r1:Name = Name, r1:Osm_id < Osm_id]
```

A problem in the NRW database is that there is a large fraction of roads whose name field is empty. Including all combinations with empty fields would let the result size (and query time) explode. We therefore want to execute the join only on tuples with non-empty name fields.

[4] The distributed implementation of the query currently only works if the first argument relation is renamed but not the second. A slight change in the implementation of the gettuples operator is needed to correct this.

In Section 6.3, we have constructed a distributed array *RoadsB3* for Roads, range partitioned on *Name*, where those with empty names are omitted. Like hash partitioning, range partitioning is suitable for equijoin.

We now create an index on this partition:

```
let RoadsB3_Name = RoadsB3 dloop["RoadsB3_Name", . createbtree[Name]]
```

Then the query can be expressed as follows:

```
query RoadsB3 RoadsB3_Name dmap2["",
  . feed {r1} loopjoin[.. exactmatchS[.Name_r1] ]
  RoadsB3
  dmap2["", . .. gettuples filter[.Osm_id_r1 < .Osm_id] ]
  dsummarize
  consume
```

7.2.5 Index-Based Spatial Join

We consider the query: Find all pairs of roads and buildings such that the building is within 500 meters distance from the road.[5]

```
select count(*) from [Roads_d as r, Buildings_d]
where distance(gk(r:GeoData), gk(GeoData)) < 500
```

We use the spatially partitioned relations *RoadsB4* and *BuildingsB4* constructed in Section 6.4. Buildings have been distributed with an enlarged bounding box so that distances of 500 meters are covered. We also have an R-tree index *BuildingsB4_GeoData* available as constructed in Section 7.1.4. The *grid* needs to be distributed as well.

```
query RoadsB4 BuildingsB4_GeoData dmap2["",
  . feed {r} loopjoin[.. windowintersectsS[.GeoData_r] ]
  BuildingsB4
  dmap2["", . .. gettuples filter[.Cell_r = .Cell]
    filter[gridintersects(grid, bbox(.GeoData_r), .EnlargedBox, .Cell_r)]
    filter[distance(gk(.GeoData_r), gk(.GeoData)) < 500] ]
  getValue tie[. + ..]
```

7.3 Aggregation

7.3.1 Counting

How many roads are there for each type?

```
select [Type, count(*) as Cnt]
from Roads_d
groupby Type
```

We use *RoadsB1*, distributed randomly, a *dfarray*.

```
query RoadsB1 dmap["", . sortby[Type] groupby[Type; Cnt: group count] ]
  dsummarize sortby[Type] groupby[Type; Cnt: group feed sum[Cnt]]
  consume
```

[5] As in the previous section, currently we can rename only the first argument.

7.3.2 Sum, Average

For each type of waterway, what is the average width?

```
select [Type, avg(Width) as AWidth]
from Waterways
groupby[Type]
```

We use *WaterwaysD1*, distributed randomly into a *dfarray*.

```
query WaterwaysD1 dmap["", . sortby[Type]
  groupby[Type; Cnt: group count, SWidth: group feed sum[Width]] ]
  dsummarize sortby[Type]
    groupby[Type; AWidth: group feed sum[SWidth] / group feed sum[Cnt]]
  consume
```

The **groupby** operator relies on a preceding sorting step. There is a more efficient implementation of grouping with the **groupby2** operator which groups by hashing. For this, the GroupbyAlgebra needs to be activated. In this case, the query can be expressed as:

```
query WaterwaysD1 dmap["", .
  groupby2[Type;
    Cnt: fun(t: TUPLE, agg:int) agg + 1::0),
    SumWidth: fun(t: TUPLE, agg:int) agg + attr(t, Width)::0)]
]
  dsummarize sortby[Type]
    groupby[Type; AWidth: group feed sum[Width] / group feed sum[Cnt]]
  consume
```

The **groupby2** operator requires for each aggregate to be computed a function, which combines a previous aggregate with a new tuple, and a value to initialize the aggregate. These two are denoted in the form `<function>::<value>`. This operator is more efficient for grouping on large sets of tuples as it avoids the sorting.

7.4 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, $T_0 \circ \ldots \circ T_{n\text{-}1}$, 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*:

```
query RoadsB1
  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["X100", N, 14, Workers14]
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["X102", "Boundaries_Osm_id" "Boundaries" matchbelow[.Osm_id]
32     extract[D], 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.