Tutorial: Distributed Query Processing in SECONDO

Using the Distributed Algebra 2

Version 4
October 29, 2019

Ralf Hartmut Güting and Thomas Behr

FernUniversität in Hagen

Faculty for Mathematics and Computer Sience

Database Systems for New Applications

58084 Hagen, Germany

Abstract

SECONDO is an open source DBMS prototype with a focus on extensible architecture and on support of spatial and spatio-temporal data, also known as *moving objects* or *trajectories*. Extensibility allows one to add collections of data types and related operations in the form of *algebra modules*. An algebra may provide any kind of data type from atomic types such as <u>*int*</u> or <u>*region*</u> to more complex structures such as relations or indexes. For example, SECONDO provides algebras for nested relations or raster data or indexes such as B-tree, R-tree, M-tree, X-tree, TB-tree. Many specialized algebras exist e.g. for clustering (DBScan, OPTICS), handling OpenStreetMap data, not to speak of the management of trajectories.

One can formulate queries to the Secondo kernel by writing algebraic expressions using all operations of the implemented (and activated) algebras. This is called the *executable language*. One can also formulate queries in an SQL dialect, using the optimizer.

Starting with SECONDO 4.0, an algebra is included to support distributed query processing, using many Secondo system instances on one or on different computers, usually one instance per available core. The purpose of this document is to explain the setup of a distributed SECONDO system and the use of this algebra, the *Distributed2Algebra*.

Table of Contents

1 Introduction

SECONDO is an open source DBMS prototype with a focus on extensible architecture and on support of spatial and spatio-temporal data, also known as *moving objects* or *trajectories*. Extensibility allows one to add collections of data types and related operations in the form of algebra modules. An algebra may provide any kind of data type from atomic types such as *int* or *region* to more complex structures such as relations or indexes. For example, Secondo provides algebras for nested relations or raster data or indexes such as B-tree, R-tree, M-tree, X-tree, TB-tree. Many specialized algebras exist e.g. for clustering (DBScan, OPTICS), handling OpenStreetMap data, not to speak of the management of trajectories.

One can formulate queries to the SECONDO kernel by writing algebraic expressions using all operations of the implemented (and activated) algebras. This is called the *executable language*. One can also formulate queries in an SQL dialect, using the optimizer.

Starting with SECONDO 4.0, an algebra is included to support distributed query processing, using many SECONDO system instances on one or on different computers, usually one instance per available core. The purpose of this document is to explain the setup of a distributed SECONDO system and the use of this algebra, the *Distributed2Algebra* (just called Distributed Algebra in the sequel).

Distributed query processing uses one SECONDO instance called the *master* and a set of SECONDO instances called the *workers*. One writes SECONDO executable queries at the master using operations of the Distributed Algebra; the implementation of such operations calls workers to execute standard SECONDO executable queries. A worker is in fact not aware that it is working in a distributed system.

The fundamental abstraction for distributed data management and query processing used is a *distributed array*. It is an array with fields of any SECONDO type; the values of fields are stored as SECONDO objects in different worker databases. The most important case is that each field (also called *slot* in this document) contains a relation, which is a subrelation of a global relation represented by the distributed array as a whole.

Operations are provided by the Distributed Algebra to apply a SECONDO function to field values. Result is a distributed array with fields of the result type of the function. Functions are evaluated by workers; all workers work in parallel; each worker may be in charge of several slots which it processes sequentially.

Here is a simple example: Let *R* be a relation, so we can apply an operator **count** to get its cardinality, writing

```
query R count
```

The **count** operator maps a type *rel*(*tuple*(*Attrs*)) into type *int*.

Then if *D* is a distributed array of relations, hence of type *darray*(*rel*(*tuple*(*Attrs*))), we can use a **dmap** operator to apply a function to each field, writing

```
query D dmap["", . count]
```

Result is a distributed array of integers, hence of type *darray*(*int*). The values of the fields are still located in the worker databases. The queries

```
query D dmap["", . count] getValue
query D dmap["", . count] getValue tie[. + ..]
```

move the distributed integer array to the master into a local array by operation **getValue**. The **tie operator** applies an aggregate function to all fields of a local array where the parameter function specifies how to combine two adjacent field values. (referred to by "." and "..", respectively). Hence the last query computes the total cardinality of the global relation represented by *D*.

A crucial feature of the approach is that operations of the Distributed Algebra such as **dmap** are completely generic and work for any SECONDO data type and operation. Everything that has been implemented so far in SECONDO is available for large scale distributed query processing. Any extension that is programmed is immediately available for distributed processing as well.

This document is intended as a hands-on tutorial to demonstrate the use and the capabilities of distributed query processing in SECONDO. It is therefore structured as follows. Section 2 starts with preliminaries on how to set up a passphrase-less connection on different computers to make life easier for the following steps. Section 3 shows how to set up clusters of SECONDO instances to serve as workers. Section 4 explains in more detail the concepts of the Distributed Algebra. Query processing needs also shuffling of data between workers which is inspired by data transfer between map and reduce steps in Hadoop/MapReduce. Section 5 sets up an example database on the master, using spatial data from OpenStreetMap. Section 6 discusses various strategies for partitioning data on the master and moving them into distributed arrays. This includes, for example, random partitioning, hash partitioning, range partitioning, and spatial partitioning. Section 7 shows how distributed queries can be written, handling in a systematic manner selections and joins with and without index support and aggregate computation. It also gives a procedure for parallel sorting. Section 8 provides some tools to analyse how workers executed a query in parallel; this allows one to graphically visualize balance between workers and their idle times.

Section 9 is meant to illustrate how many capabilities in SECONDO work together to solve a somewhat complicated task that would be very hard to program in simpler frameworks such as Hadoop, for example. The task is to construct from a given OpenStreetMap database a road network suitable for tasks like shortest path computation or map matching of trajectories.

We provide the sizes of the data sets used and show for most examples the actual running times to give an indication of effort required and efficiency. For parallel sorting and the example of Section 9, the speedup is shown.

All examples should work on a SECONDO system of version 4.1.3 which is available on the web site;[1] almost all work also on SECONDO 4.0.

[1] SECONDO web site: http://dna.fernuni-hagen.de/Secondo.html

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.[2] This is done as follows.

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

```
ssh-keygen -t rsa
```

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 (from now on we name them by the end of their IP addresses) 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_rsa.pub <user>@server
```

For example:

```
ssh-copy-id -i .ssh/id_rsa.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 needs to log in once on each of the nodes because the system on the master must enter each node into its list of known hosts.

[2] This is just to make life easier; one could also start the monitors manually with entering passwords. The *Distributed2Algebra* itself can be used without this.

3 Setting Up a Cluster

3.1 Example 1: Mini-Cluster

We use a mini-cluster consisting of two computers running Ubuntu 18.04. 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 three 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 (which 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)

In this example we set up one database per disk and one worker per core. This means that two workers use the same database and it is necessary to run the worker systems with transaction management. It is in fact faster to let each worker run on its own separate database; then transaction management can be switched off. We will demonstrate this in the second example.

3.1.1 Setting Up the Monitors

Per worker disk we need to start one SECONDO Monitor. 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> [ <bin> [ <home> [ <port> [ <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. Further parameters are optional:

- `<bin>` is the path to the `secondo/bin` directory from which SECONDO is to be started and this is also the location of the configuration file (default is the user's home directory `secondo/bin`),
- `<home>` is the path to the database directory,
- `<port>` is the port number,

- `<user>` the user name (by default, the name of the user running the `remoteMonitors` command)

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 /home/ralf/secondo/bin /discA/secondo-databases 1471
132.176.69.160 SecondoConfig.ini.160 /home/ralf/secondo/bin /discB/secondo-databases 1472
132.176.69.160 SecondoConfig.ini.160 /home/ralf/secondo/bin /discC/secondo-databases 1473
132.176.69.98 SecondoConfig.ini.98 /home/ralf/secondo/bin /home/ralf/distributed2/secondo-databases 1474
132.176.69.98 SecondoConfig.ini.98 /home/ralf/secondo/bin /disk2/secondo-databases 1475
132.176.69.98 SecondoConfig.ini.98 /home/ralf/secondo/bin /disk3/secondo-databases 1476
132.176.69.98 SecondoConfig.ini.98 /home/ralf/secondo/bin /disk4/secondo-databases 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 just the global memory per server. Note that we do not need to change IP addresses, ports or database directories as these are overridden by the `ClusterRalf7` file. We have:

- 160: `GlobalMemory=2000`
- 98: `GlobalMemory=3600`

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

3.1.2 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
```

3.1.3 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")
)]
```

3.2 Example 2: A Larger Cluster

In this example, we use a cluster consisting of five computers, called *newton1*, ..., *newton5*, each with the following resources:

- 8-core processor
- 32 GB memory
- 4 disks

We let the master run on *newton1*, with an extra database directory `/home/<user>/secondo-databases2`.

Each computer has 8 cores, so we use 40 workers. In this example, we run each worker on its own database directory. This means that no concurrency control is needed and transactions can be switched off. However, we need 40 database directories, two on each disk. Each worker can get 3600 MB of memory.

In this section, we also show some techniques to handle larger clusters conveniently.

3.2.1 Prerequisites

The following has been prepared by the system administrator:

- all computers run Ubuntu 18.04
- the SECONDO-SDK has been installed on all computers

- for user *ralf*, the following directories with write permission have been created on each computer:
  - /home/ralf
  - /diskb/ralf
  - /diskc/ralf
  - /diskd/ralf
- a range of free ports is provided exclusively for user *ralf*.

3.2.2 SECONDO Installation

The new user *ralf* needs to do the following:

- Get pass-phrase-less access from the master to all involved worker computers.
- On each computer, install SECONDO in the user's home directory (`/home/ralf`). Use the option `-onlyrc` with the installation script, because the software for SECONDO has already been installed on these computers.

Since all machines should run the same SECONDO version, it is a good idea to handle updates by just updating the version on one machine, performing the make process there, and then copying the compiled system to all other machines. To this end, we can extend the `makefile` (in directory `secondo`) on, say, *newton1*, as follows:

- insert these lines before `.PHONY: help`

```
.PHONY: remoteServers
remoteServers:
  scp bin/SecondoBDB 132.176.69.194:/home/ralf/secondo/bin/
  scp bin/tmp/*.examples 132.176.69.194:/home/ralf/secondo/bin/tmp/
  scp bin/SecondoMonitor 132.176.69.194:/home/ralf/secondo/bin/
  scp bin/SecondoListener 132.176.69.194:/home/ralf/secondo/bin/
  scp bin/SecondoRegistrar 132.176.69.194:/home/ralf/secondo/bin/
  scp bin/SecondoCheckpoint 132.176.69.194:/home/ralf/secondo/bin/

  scp bin/SecondoBDB 132.176.69.195:/home/ralf/secondo/bin/
  scp bin/tmp/*.examples 132.176.69.195:/home/ralf/secondo/bin/tmp/
  scp bin/SecondoMonitor 132.176.69.195:/home/ralf/secondo/bin/
  scp bin/SecondoListener 132.176.69.195:/home/ralf/secondo/bin/
  scp bin/SecondoRegistrar 132.176.69.195:/home/ralf/secondo/bin/
  scp bin/SecondoCheckpoint 132.176.69.195:/home/ralf/secondo/bin/
  ...
```

  The 6 lines need to be repeated for all used computers.

- extend the target `ALL_TARGETS` by `remoteServers`

```
ALL_TARGETS = makedirs \
  buildlibs \
  buildAlgebras \
  buildapps \
  $(OPTIMIZER_SERVER) \
  java2 \
  tests \
        examples \
  update-config  \
  API \
remoteServers
```

From now on, the make process on *newton1* will automatically synchronize all involved Secondo servers.

3.2.3 Creating Database Directories

This can be done as follows. Create a file *newton* on some computer *X* containing all IP addresses of *newton1*, ... *newton5*, one per line. Hence it has contents:

```
132.176.69.193
132.176.69.194
132.176.69.195
132.176.69.196
132.176.69.197
```

Make sure that computer *X* can login to *newton1*, ..., *newton5* without pass-phrase. Then in a bash enter commands:

```
for s in $(cat newton) ; do ssh ralf@$s mkdir /home/ralf/secondo-databases
; done
for s in $(cat newton) ; do ssh ralf@$s mkdir /diskb/ralf/secondo-data-
bases ; done
for s in $(cat newton) ; do ssh ralf@$s mkdir /diskc/ralf/secondo-data-
bases ; done
for s in $(cat newton) ; do ssh ralf@$s mkdir /diskd/ralf/secondo-data-
bases ; done
for s in $(cat newton) ; do ssh ralf@$s mkdir /home/ralf/secondo-databas-
esB ; done
for s in $(cat newton) ; do ssh ralf@$s mkdir /diskb/ralf/secondo-databas-
esB ; done
for s in $(cat newton) ; do ssh ralf@$s mkdir /diskc/ralf/secondo-databas-
esB ; done
for s in $(cat newton) ; do ssh ralf@$s mkdir /diskd/ralf/secondo-databas-
esB ; done
```

Each of the 8 commands must be in a single line.

3.2.4 Prepare a Cluster Description File

This file, called `ClusterRalfNewton`, is to be used by the `remoteMonitors` script. The format is:

```
<IP address> <config file> <bin directory> <db directory> <port>
```

For the *newton* cluster, it looks as follows:

```
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/sec-
ondo-databases  63414
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/sec-
ondo-databases 63415
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/sec-
ondo-databases 63416
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/sec-
ondo-databases 63417
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/sec-
ondo-databases  63414
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/sec-
ondo-databases 63415
```

```
...
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databases 63417
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databasesB  63410
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databasesB 63411
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databasesB 63412
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databasesB 63413
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databasesB  63410
...
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databasesB 63412
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databasesB 63413
```

Hence there are 40 monitors to be started, each on its own database. For clarity, the complete file is shown in the appendix.

3.2.5 Prepare a Workers Relation File

We describe workers in a file `WorkersNewton` in the `secondo/bin` directory with the format of a saved relation.

```
(OBJECT WorkersNewton
    ()
    (rel
        (tuple
            (
                (Host string)
                (Port int)
                (Config string))))
    (
        ("132.176.69.193" 63414 "SecondoConfig.ini")
        ("132.176.69.193" 63415 "SecondoConfig.ini")
        ("132.176.69.193" 63416 "SecondoConfig.ini")
        ("132.176.69.193" 63417 "SecondoConfig.ini")

        ("132.176.69.194" 63414 "SecondoConfig.ini")
...
        ("132.176.69.197" 63412 "SecondoConfig.ini")
        ("132.176.69.197" 63413 "SecondoConfig.ini")

))
```

The relation defines 40 workers, one for each monitor. For clarity, the complete file is shown in the appendix. In each new database on the master, the workers can be created simply by the command:

```
restore WorkersNewton from WorkersNewton
```

3.2.6 Adapt and Distribute File SecondoConfig.ini

In `SecondoConfig.ini` we change the following entries:

```
# Switch off the transaction and logging subsystem of Berkeley-DB
RTFlags += SMI:NoTransactions

# Global memory available for all operators in MB
# default is 512
GlobalMemory=3600
```

This means that workers do not use transactions and the memory available for each worker is 3600 MB. We then need to move the configuration files (e.g. from computer *X*) to all computers.

```
for s in $(cat newton) ; do scp SecondoConfig.ini ralf@$s:/home/ralf/sec-
ondo/bin ; done
```

To keep the configuration of the master independent of that of the workers, we may introduce a copy of `SecondoConfig.ini`, say `SecondoConfig.ini.M` and change a line in the file `.secondorc` (in the user's home direcctory, defining variables for SECONDO) to

```
export SECONDO_CONFIG=$SECONDO_BUILD_DIR/bin/SecondoConfig.ini.M
```

This allows us to run the master on a cluster computer as well and start it in the standard way. For example, the master may use more memory, run with transactions. It must use a database directory different from that of workers.

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[3] to all slots of distributed arrays which results in new distributed arrays.

[3] To be precise, all workers work in parallel, but each worker processes its assigned slots sequentially.

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.*

There exist also variants of these types called *pdarray* and *pdfarray*, respectively, where only some of the fields are defined.

Here *X* may be any SECONDO type[4] 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 and the type of slots, the number of slots is part of the value.

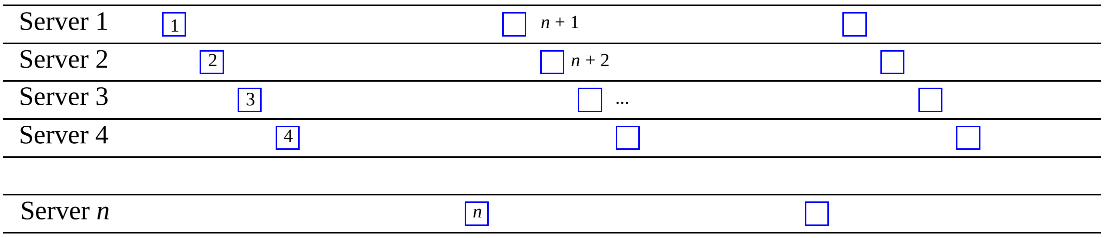


Figure 1: A distributed array. Each slot is represented by a square with its slot number.

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.

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.

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.

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

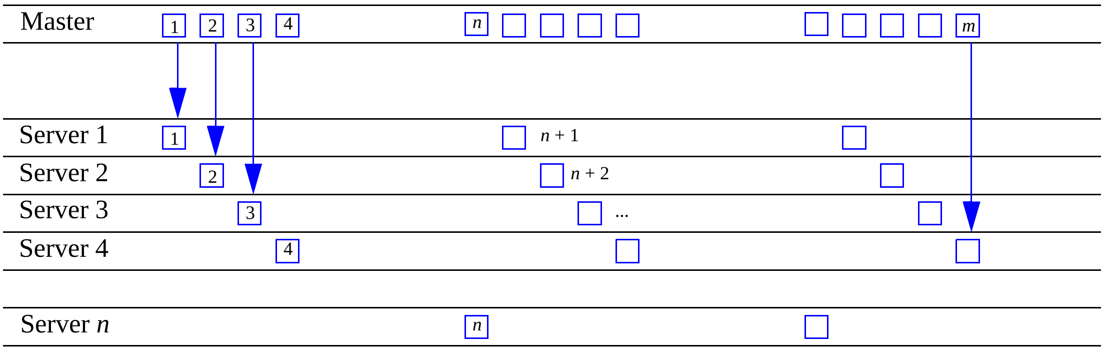


Figure 2: Creating a distributed array by partitioning data on the master.

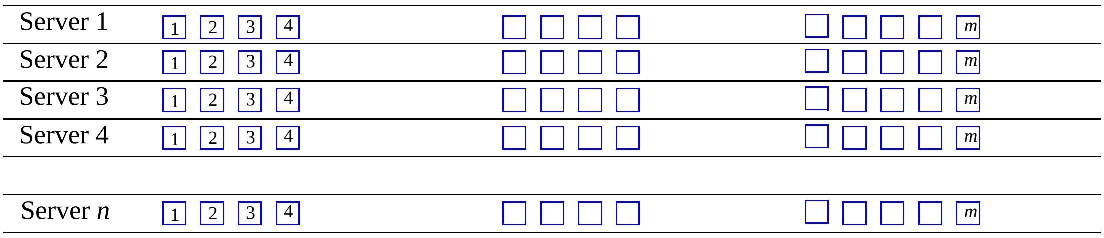


Figure 3: A distributed file matrix.

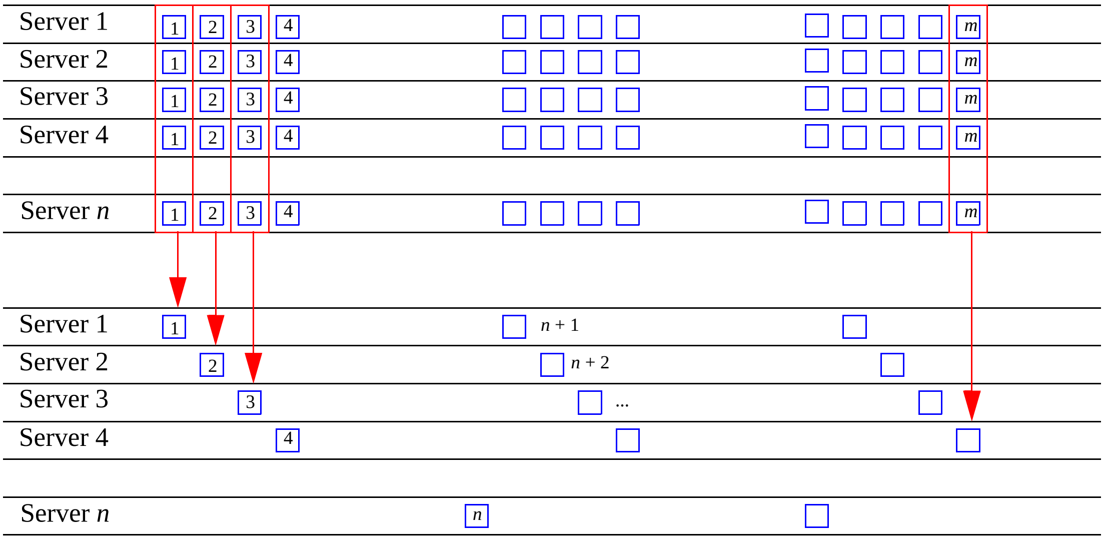


Figure 4: A distributed file matrix is collected into a distributed file array.

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

The following operations come in a d-variant and a df-variant (prefix). The d-variant creates a *darray*, the df-variant a *dfarray*.

Operations:

Operation	Meaning
ddistribute2, dfdistribute2	Distributes a stream of tuples into a distributed array. Parameters are an integer attribute, the number of slots and a Workers relation in the format shown in Section 3.1.3. 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.

4.2.2 Distributed Processing by the Workers

Operations:

Operation	Meaning
dloop, dmap	Evaluates a SECONDO query on each field of a distributed array of type *darray* or *dfarray*. Operator **dloop** returns a *darray*, **dmap** returns a *dfarray* if the result is a tuple stream, otherwise a *darray*. In a parameter query, one refers to the field argument by "."
dloop2, dmap2	Binary variants of the previous operations mainly for processing joins. Always two fields with the same index are arguments to the query. One refers to field arguments by "." and "..", respectively.

Operation	Meaning
dmap3, ..., dmap8	Variants of **dmap** for up to 8 argument arrays. One can refer to fields by ".", "..", or by $1, ... $8.
pdmap, ..., pdmap8	Variants of **dmap** which take as an additional first argument a stream of slot numbers and evaluate parameter queries only on those slot numbers. They return a partial darray or dfarray (*pdarray* or *pdfarray*) where unevaluated fields are undefined.
dproduct	Arguments are two *darray*s or *dfarray*s with relation fields. Each field of the first argument is combined with the union of *all* fields of the second argument. Can be used to evaluate a Cartesian product or a generic join with arbitrary condition. No specific partitioning is needed for a join. But the operation is expensive, as all fields of the second argument are moved to the worker storing the field of the first argument.
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.
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.
dcommand	Executes an arbitrary command on each worker associated with its argument array.
dlet	Executes a *let* command on each worker associated with its argument array; it further executes the same command on the master. This is needed so that the master can do type checking on the query expressions to be executed by workers in following **dmap** operations.

4.2.3 Collecting Data From the Workers

Operations:

Operation	Meaning
dsummarize	Collect all tuples (or values) from a *darray* or *dfarray* into a tuple stream (or value stream) on the master. Works also for *pdarray* and *pdfarray*.
getValue	Convert a distributed array into a local array. Recommended only for atomic field values; may otherwise be expensive.
getValueP	Variant of **getValue** applicable to *pdarray* or *pdfarray*. Provides a parameter to replace undefined values in order to return a complete local array on the master.
tie	Apply aggregation to a local array, e.g., to determine the sum of field values. (An operation not of the *Distributed2Algebra* but of the *ArrayAlgebra* in SECONDO).

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-latest-
free.shp.zip
```

You can also use this link to download directly, if it is still available. Unpack the zip file.

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

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

```
SecondoPLTTYNT
```

At the prompt, enter

```
Secondo => @../bin/Scripts/nrwImportShapeNew.osec
```

The contents of the file `nrwImportShapeNew.osec` are shown here. Adapt the path in `DIR`.

```
# Importing NRW Data
close database

create database nrw

open database nrw
```

```
let DIR = '/home/ralf/Daten/nordrhein-westfalen-latest-free.shp/'

let Roads = dbimport2(DIR + 'gis.osm_roads_free_1.dbf') shpimport2(DIR +
'gis.osm_roads_free_1.shp') namedtransformstream[GeoData] obojoin fil-
ter[isdefined(bbox(.GeoData))] validateAttr trimAllUndef consume
```

This is repeated to load all relations. The script creates relations *Roads*, *Waterways*, etc. within a new database *nrw*. It ensures that geometries are defined, attribute names are capitalized, and empty strings are represented as undefined values. Recently, this database is fairly large, for example, has 7.5 mio buildings. It takes 2-3 hours to run this script.

The query optimizer collects information about the existing database objects which can be displayed with the predicate *showDatabase*.

```
SecondoPLTTY => showDatabase

Relation Roads(Auxiliary objects: SelSample(8144)  JoinSample(8144) )
  AttributeName               Type          Memory  DiskCore    DiskLOB
  GeoData                     line          220.224    220.224  943.488
  Tunnel                      string        64.0       7.0      0
  Bridge                      string        64.0       7.0      0
  Layer                       int           16.0       5.0      0
  Maxspeed                    int           16.0       5.0      0
  Oneway                      string        64.0       7.0      0
  Ref                         string        64.0       6.336    0
  Name                        text          77.9       77.9     0
  Fclass                      string        64.0       14.434   0
  Code                        int           16.0       5.0      0
  Osm_id                      string        64.0       14.524   0

  Indices:

  Ordering:  []

  Cardinality:   1628963
  Avg.TupleSize: 1312.906 = size-
Term(866.124,369.4179992675781,943.4880007324218)
  (Tuple size in memory is 136 + sum of attribute sizes.)
...
```

The relations in our example database have the following cardinalities and spatial data types:

Relation	Cardinality	Geometry	Relation	Cardinality	Geometry
Buildings	7464304	region			
Landuse	548688	region			
Natural	159468	point	NaturalA	556	region
Places	18222	point	PlacesA	948	region
Pofw	2693	point	PofwA	9136	region
Points	284798	point	PointsA	115471	region
Railways	47104	line			
Roads	1628963	line			

Relation	Cardinality	Geometry	Relation	Cardinality	Geometry
Traffic	190162	point	TrafficA	65019	region
Transport	88401	point	TransportA	233	region
Water	37825	region			
Waterways	91302	line			

Some classes of objects are represented in two relations with different spatial data types. *Points* means points of interest, *Pofw* places of worship.

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

After starting monitors and master we create the workers relation in the database (Section 3.1). Further, we define a port number to be used in some data exchanges which on a shared cluster should be used exclusively by us.

```
let Workers14 = ...;
let myPort = ...;
```

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]

# 2:27 min
```

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]

# 2:38 min
```

6.3 Range Partitioning

We can efficiently partition a relation for an attribute *A* with a total order[5] 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 defined names.

```
query Roads feed filter[isdefined(.Name)] count

# Result: 594494
```

Hence there are 594494 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 1/119 tuples.

```
let S = Roads feed filter[isdefined(.Name)] nth[119, FALSE] project[Name]
sortby[Name] consume

# 52.9 seconds, Result size: 4995
```

The 4995 tuples in the sample we wish to divide into about 50 partitions of size about 100. We now determine the attribute values that lie on partition boundaries.

```
let Boundaries = S feedproject[Name] nth[100, TRUE]
  addcounter[D, 1] project[Name, D] consume

# 1.37 seconds
```

We put the relation Boundaries into memory and create an AVL-tree index over it.

```
let BoundariesM = Boundaries feed mconsume;
let BoundariesM_Name = BoundariesM mcreateAVLtree[Name]

# 0.03 seconds
```

We can now distribute Roads, using the main-memory-AVLtree *Boundaries_Name* to determine the slot number for each tuple. Names smaller than the smallest entry receive slot number 0.

```
let RoadsB3 = Roads feed filter[isdefined(.Name)]
  ddistribute4["RoadsB3", BoundariesM_Name BoundariesM
    matchbelow2[.Name, D, 0], 50, Workers14]

# 2:10 min
```

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

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

# 6.33 sec
```

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

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.

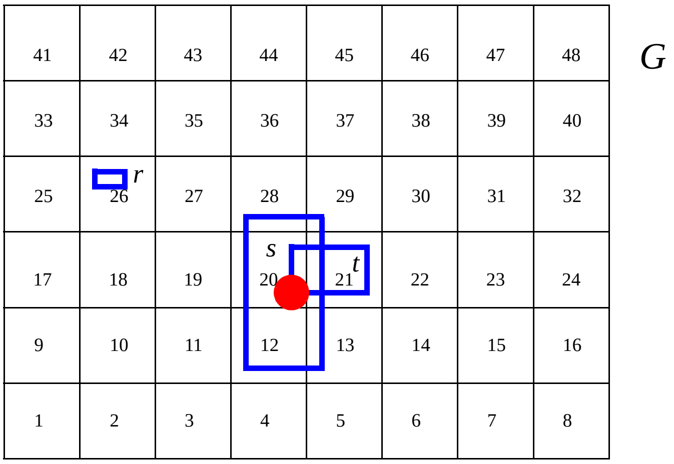


Figure 5: Rectangles *r*, *s*, and *t* are mapped to cell numbers using grid *G*.

The first step for constructing the grid is to determine a rectangle enclosing all geometries.[6] 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
  transformstream collect_box[TRUE]
```

The result is:

```
[const rect value (5.83191 9.64789 50.1346 52.5661)]
```

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. Moreover it is not a problem if some objects lie outside the grid boundary. They are still mapped to indices.

```
let grid = [const cellgrid2d value (5.8 50.1 0.2 0.2 20)]
```

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

Here the five parameters for the grid are the left bottom corner (5.8, 50.1), the cellwidth and cellheight (each 0.2) 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]

# 1:56 min
```

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.

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 corresponds to about 1.1 km (although this varies with the y-coordinate). 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]

# 52:36 min
```

This way of distributing takes a lot of time as many computations have to be done on the master. A faster way is to first distribute data quickly in a random way and then to create the spatial distribution by repartitioning. However, as now the workers distribute data spatially, we first need to make the grid available to them.

```
let BuildingsB1 = Buildings feed dfdistribute3["BuildingsB1", 50, TRUE,
Workers14]

# 24:11 min

query share("grid", TRUE, Workers14)
```

```
let BuildingsB4a = BuildingsB1 partitionF["",
  . feed extend[EnlargedBox: enlargeRect(bbox(.GeoData), 0.01, 0.01)]
  extendstream[Cell: cellnumber(.EnlargedBox, grid)]
  extend[Original: .Cell = cellnumber(.EnlargedBox, grid)
    transformstream extract[Elem] ],
  ..Cell, 0]
  collect2["", myPort]
  dmap["BuildingsB4a", . feed consume]

# 13:00 min
```

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, Workers14)

# 21:55 min
```

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.

7 Querying

In the following queries, in SQL we distinguish distributed from local relations by a suffix `"_d"`. Hence *Roads* refers to the local relation on the master, *Roads_d* to the distributed version.

7.1 Selection

We consider selection by a standard attribute and by a spatial attribute, supported or not by an index.

```
select count(*) from Roads_d where Name = "Universitätsstraße"
select count(*) from Buildings_d where Type = "school"
select * from Roads_d where GeoData intersects eichlinghofen
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, created as follows:

```
let eichlinghofen = [const region value (
    (
        (
            (7.419515247680575 51.47332155746125)
            (7.394967670776298 51.47332155746125)
            (7.394967670776298 51.48716614802665)
            (7.419515247680575 51.48716614802665)))) ]
```

7.1.1 By Scanning

Selection by standard attribute:

```
query BuildingsB1 dmap["", . feed filter[.Type = "school"] count]
  getValue tie[. + ..]

# 9.05 seconds, result 6037
```

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

```
query share("eichlinghofen", TRUE, Workers14)


query RoadsB1 dmap["", . feed filter[.GeoData intersects eichlinghofen]]
  dsummarize consume

# 6.6 seconds, result size 697
```

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] ]

# 9.06 seconds
```

Result is a distributed B-tree.

7.1.3 Using a Standard Index

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

# 2.8 seconds, result 98
```

7.1.4 Creating a Spatial Index

BuildingsB4a is a *darray*, hence suitable for adding an R-tree index. We construct the index by bulkloading.

```
let BuildingsB4a_GeoData = BuildingsB4a dmap["BuildingsB4a_GeoData",
  . feed addid extend[Box: scalerect(.EnlargedBox, 1000000.0, 1000000.0)]
  sortby[Box] remove[Box] bulkloadrtree[EnlargedBox] ]

# 50.7 seconds
```

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.

```
query BuildingsB4a_GeoData BuildingsB4a
  dmap2["", . .. windowintersects[eichlinghofen]
    filter[.Original]
    filter[.GeoData intersects eichlinghofen], myPort
  ]
  dsummarize consume

# 12.5 seconds, result size 2272
```

We avoid duplicates by restricting to the *Original* field being *true*.

7.1.6 Selective Evaluation

For some queries we may know that relevant data can only be found in certain slots of a distributed array. This is the case when a predicate is correlated with the partitioning method of the given distributed array. For example, in the previous query, relation *BuildingsB4a* is partitioned spatially according to *grid*. Hence query results can only arise in the grid cells intersected by *eichlinghofen* and the slots they are mapped to. We can determine these slots on the master by the query

```
query cellnumber(bbox(eichlinghofen), grid) use[. mod size(BuildingsB4a)]
```

returning a stream of slot numbers. Here the **use** operator maps stream elements by a parameter function and **size** returns the number of slots of a distributed array. The region *eichlinghofen* intersects only two cells of the grid and hence is mapped to only two slots of the distributed array.

The **pdmap** operators are variants of the respective **dmap** operators which take additionally as a first argument a stream of slot numbers. They evaluate the parameter function only on those slots. They return a *partial* darray or dfarray (of type *pdarray* or *pdfarray*, respectively) where all fields are undefined that have not been evaluated. To such a partial array of relations, operation **dsummarize** is applicable, returning only the contents of defined fields. Instead of operation **getValue** a variant **getValueP** can be used which has an additional parameter to specify a value replacing all undefined fields. We illustrate selective evaluation with the following queries:

```
query cellnumber(bbox(eichlinghofen), grid) use[. mod size(BuildingsB4a)]
  BuildingsB4a_GeoData BuildingsB4a
  pdmap2["", . .. windowintersects[eichlinghofen]
    filter[.Original]
    filter[.GeoData intersects eichlinghofen], myPort
  ]
  dsummarize consume

# 5.03 seconds, result size 2272
```

Counting variants of this query are the following:

```
query BuildingsB4a_GeoData BuildingsB4a
  dmap2["", . .. windowintersects[eichlinghofen]
    filter[.Original]
    filter[.GeoData intersects eichlinghofen] count, myPort
  ]
  getValue tie[. + ..]

# 4.33 seconds, result 2272
```

With selective evaluation:

```
query cellnumber(bbox(eichlinghofen), grid) use[. mod size(BuildingsB4a)]
  BuildingsB4a_GeoData BuildingsB4a
  pdmap2["", . .. windowintersects[eichlinghofen]
    filter[.Original]
    filter[.GeoData intersects eichlinghofen] count, myPort
  ]
  getValueP[0] tie[. + ..]

# 1.11 seconds, result 2272
```

Once can oberve that in some cases a considerable speedup can be achieved in comparison to evaluating all slots. This technique is applicable also for hash partitioning or range partitioning for suitable predicates, obviously not for random partitioning.

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 *NaturalB2* be *Natural* distributed by attribute *Name* as a darray. We reduce to names that are defined.

```
let NaturalB2 = Natural feed filter[isdefined(.Name)]
  ddistribute4["NaturalB2", hashvalue(.Name, 999997), 50, Workers14]

# 16.5 seconds
```

```
query NaturalB2 dmap["",
  . feed {n1} . feed {n2} itHashJoin[Name_n1, Name_n2]
  filter[.Osm_id_n1 < .Osm_id_n2]]
  dsummarize consume

# 22.97 seconds, result size 3131
```

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

(2) Arbitrary Distribution

Let *NaturalB1* be a *dfarray* distributed not by attribute *Name* without duplicates.

```
let NaturalB1 = Natural feed
  dfdistribute3["NaturalB1", 50, TRUE, Workers14]

# 4.6 seconds
```

Hence we first need to redistribute.

```
query NaturalB1 partitionF["",  . feed filter[isdefined(.Name)],
  hashvalue(..Name, 999997), 0]
  collect2["", myPort]
  dmap["",
  . feed {n1} . feed {n2} itHashJoin[Name_n1, Name_n2]
  filter[.Osm_id_n1 < .Osm_id_n2]]
  dsummarize consume

# 47.76 seconds, result size 3131
```

Because this is a self-join, we need to repartition only once and can use **dmap** instead of **dmap2** for the join.

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

For the following query, the *grid* needs to be available on the workers. It was already distributed in Section 6.4.

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

```
let WaterwaysB4 = Waterways feed
  extendstream[Cell: cellnumber(bbox(.GeoData), grid)]
  dfdistribute2["WaterwaysB4", Cell, 50, Workers14]

# 6.4 seconds

query RoadsB4 WaterwaysB4 dmap2["",
  . feed {r} .. feed {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, myPort]
  getValue tie[. + ..]

# 5:50 min, result 66054
```

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.

```
let WaterwaysB1 = Waterways feed dfdistribute3["WaterwaysB1", 50, TRUE,
  Workers14]

# 3.28 seconds

query
  RoadsB1 partitionF["",
    . feed extendstream[Cell: cellnumber(bbox(.GeoData), grid)],
    ..Cell, 0]
  WaterwaysB1 partitionF["",
    . feed extendstream[Cell: cellnumber(bbox(.GeoData), grid)],
    ..Cell, 0]
  areduce2["",
    . feed {r} .. feed {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, myPort]
  getValue tie[. + ..]

# 5:34 min, result 66054
```

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["",
  . feed {r} .. feed {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], myPort ]
  dsummarize
  projectextend[Osm_id_r, Name_r, GeoData_r, Osm_id_w, Name_w, GeoData_w;
    BridgePosition: crossings(.GeoData_r, .GeoData_w)]
  consume

# 14:03 min
```

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["",
  . feed {r} .. feed {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:
    crossings(.GeoData_r, .GeoData_w)], myPort
  ]
  dsummarize
  consume

# 6:02 min
```

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, isdefined(r:Name),
  r:Fclass = "pedestrian", w:Fclass = "river"]
```

This can be evaluated efficiently, 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 share("Waterways", TRUE, Workers14)

# 1:14 min

query RoadsB1 dmap["", . feed filter[isdefined(.Name)]
  filter[.Fclass = "pedestrian"] {r}
  Waterways feed filter[.Fclass = "river"] {w}
  symmjoin[.Name_r contains ..Name_w] ]
  dsummarize
  consume

# 28.18 seconds, result size 2435
```

If both arguments are distributed only (none is replicated) one can still evaluate it using the **dproduct** operator. Let us assume *WaterwaysB1* is distributed as well.

```
query RoadsB1 WaterwaysB1 dproduct["", . feed
  filter[isdefined(.Name)] filter[.Fclass = "pedestrian"] {r}
  .. feed filter[.Fclass = "river"] {w}
  symmjoin[.Name_r contains ..Name_w], myPort]
  dsummarize
  consume

# 19.76 seconds, result size 2435
```

7.2.4 Index-Based Equijoin

We consider the following query: Find all pairs of distinct roads with the same name. We add a further condition to let the query not be too expensive.

```
select * from [Roads_d as r1, Roads_d]
where [r1:Name = Name, r1:Osm_id < Osm_id, r1:Fclass = "pedestrian",
  r2:Fclass = "pedestrian", isdefined(r1:Name)]
```

In Section 6.3, we have constructed a distributed array *RoadsB3* for Roads, range partitioned on *Name*, where roads 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]]

# 9.19 seconds
```

Then the query can be expressed as follows:

```
query RoadsB3 RoadsB3_Name RoadsB3 dmap3["", $1 feed
  filter[.Fclass = "pedestrian"] {r1}
  loopjoin[$2 $3 exactmatch[.Name_r1] filter[.Fclass = "pedestrian"]]
  filter[.Osm_id_r1 < .Osm_id], myPort]
  dsummarize
  consume

# 1:31min (cold), 19.11 seconds (warm), result size 33824
```

In a **dmapX** operator one can refer to argument *i* by `$i`; for more than two arguments this is necessary (instead of the `.` and `..` notation). Here it makes a big difference whether the query is executed for the first time (cold) or the second time (warm), when B-tree pages are cached.

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. Again we add another condition to let the query be not too expensive.

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

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 BuildingsB4 dmap3["",
  $1 feed filter[.Fclass = "bridleway"] {r}
    loopjoin[$2 $3 windowintersects[.GeoData_r] {b}]
    filter[.Cell_r = .Cell_b]
    filter[gridintersects(grid, bbox(.GeoData_r), .EnlargedBox_b,
      .Cell_r)]
    filter[distance(gk(.GeoData_r), gk(.GeoData_b)) < 500] count, myPort]
    getValue tie[. + ..]

# 10:54min, result 478646
```

Even though there are only 2737 roads of class bridleway, the query takes a long time. This is because (1) lots of buildings are retrieved due to their enlarged bounding boxes of size about 1 km^2, and (2) the exact evaluation of the distance requires conversion into Gauss-Krüger coordinates and is expensive itself. The query without the final filter condition yields the following results:

```
query RoadsB4 BuildingsB4_GeoData BuildingsB4 dmap3["",
  $1 feed filter[.Fclass = "bridleway"] {r}
    loopjoin[$2 $3 windowintersects[.GeoData_r] {b}]
    filter[.Cell_r = .Cell_b]
    filter[gridintersects(grid, bbox(.GeoData_r), .EnlargedBox_b,
      .Cell_r)] count, myPort]
    getValue tie[. + ..]

# 18.25 seconds, result 2474541
```

7.3 Aggregation

7.3.1 Counting

How many roads are there for each class?

```
select [Fclass, count(*) as Cnt]
```

```
from Roads_d
groupby Fclass
```

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

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

# 14.71 seconds
```

7.3.2 Sum, Average

For each class of waterway, what is the average width? Some objects have huge or negative width values which are omitted.

```
select [Fclass, avg(Width) as AWidth]
from Waterways
where between(Width, 0, 10000)
groupby[Fclass]
```

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

```
query WaterwaysB1 dmap["", . feed filter[.Width between[0, 10000]]
  sortby[Fclass]
    groupby[Fclass; Cnt: group count, SWidth: group feed sum[Width]]
  ]
  dsummarize sortby[Fclass]
  groupby[Fclass; SumWidth: group feed sum[SWidth],
    SumCnt: group feed sum[Cnt]]
  extend[AvgWidth: .SumWidth / .SumCnt]
  project[Fclass, AvgWidth]
  consume

# 18.1 seconds
```

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 WaterwaysB1 dmap["", . feed filter[.Width between[0, 10000]]
  groupby2[Fclass; Cnt: fun(t: TUPLE, agg:int) agg + 1::0,
    SWidth: fun(t2: TUPLE, agg2:int) agg2 + attr(t2, Width)::0]
  ]
  dsummarize sortby[Fclass]
  groupby[Fclass; AWidth: group feed sum[SWidth] / group feed sum[Cnt]]
  consume

# 24.2 seconds
```

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 than **groupby** 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, $T_0 \circ ... \circ T_{n\text{-}1}$, 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 first two steps correspond to range partitioning as described in Section 6.3. The only difference is that we need to get the sample from the distributed array.

This algorithm uses the following steps to sort a distributed Relation *R* by attribute *A*. Steps 1 through 4 serve to provide the partition boundaries on each server. Steps 5 to 6 perform the actual sorting. As an example, we consider sorting *BuildingsB1* by *Osm_id* where *BuildingsB1* is a distributed file array with random distribution.

The following example queries require at least SECONDO version 4.1.3.

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.

2. On the master, sort the union of samples. From the ordered tuple stream 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.

```
# (Steps 1 - 2)

let Size = BuildingsB1 dmap["", . feed count] getValue tie[. + ..];
let NSlots = size(BuildingsB1);
let Fraction = (Size div NSlots) div 500;
query share("Fraction", TRUE, Workers14);

let Boundaries = BuildingsB1 dmap["", . feed nth[Fraction, FALSE]
  project[Osm_id]] dsummarize sort nth[500, TRUE] addcounter[D, 1]
  consume
```

3. Share the boundary pairs with the workers.

```
query share("Boundaries", TRUE, Workers14)
```

4. On each worker, set the size of available main memory and create a main memory relation called *BoundariesM* and an AVLtree index on its attribute *Osm_id* called *Boundaries_Osm_id*.

   The operator **dcommand** executes an arbitrary command on each worker associated with its argument array, here *BuildingsB1*. The operator **dlet** executes a *let* command on each worker associated with its argument array; it further executes the same command on the master. This is needed so that the master can do type checking on the query expressions to be executed by workers in following **dmap** operations.

5. 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 *BoundariesM* whose a_k value is the greatest one that is smaller than a_x; for a value $a_x < a_1$ return 0. Repartition R by D. Result is a *dfmatrix*.

   ```
   let BuildingsSortedOsm_idA = BuildingsB1
     partition["",
       Boundaries_Osm_id BoundariesM matchbelow2[.Osm_id, D, 0],
       0]
   ```

   The **matchbelow2** operator searches on the given AVL-tree for the largest entry smaller than the given *Osm_id* attribute value; it returns the associated *D* value. If no such entry can be found, it returns its last argument, here 0.

6. Sort the tuples of each column R_j by A.

   ```
   let BuildingsSortedOsm_id = BuildingsSortedOsm_idA
     areduce["", . feed sortby[Osm_id], myPort]
   ```

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.

The running times for the steps are shown in Table 1. The last column reports on using the same algorithm on the relation `CityNodesB0` (sorting by `NodeId`) decribed in Section 9.4, with about 250 million tuples, from the database `germany`. This experiment was done on cluster newton (as in Section 9.4).

Steps	Action	Time [seconds] BuildingsB1 Cardinality 7.464.304	Time [seconds] CityNodesB0 Cardinality 246.682.839
1 - 4	Determining and distributing boundaries	36.86	2.46
5	Repartitioning	29.38	54.92
6	Sorting partitions	129.18	98.02
Total:		**195.24**	**155.39**

Table 1: Running times for parallel sorting

A difference between the two examples is that `BuildingsB1` is a *dfarray* and `CityNodesB0` a *darray*. On a relation, counting and taking a sample is much faster than on a file. This is why the steps 1 through 4 on the second example are so much faster. In that case, the costly steps are only the redistribution and the sorting of partitions.

8 Observing Balance and Utilization of Workers

A well-known general problem in distributed query processing is that a query is finished only when the last worker finishes its job. Necessarily the other workers are idle just before that time. It is desirable to keep the idle times of workers minimal. In other words, it would be optimal if all workers finished at the same time.

Workers may use different amounts of time because their tasks may be not of quite the same size. This is called skew in the distribution of data. If different types of machines are used, the problem is aggravated because some workers now may need more time than others even for tasks of the same size.

In our distributed algebra, operators **areduce** and **areduce2** adapt to different amounts of time needed per worker and job. They assign more tasks to fast workers than to slow ones.

In this section we describe some tools to record and visualize the distribution of tasks to workers and their idle times.

To prepare such measurements, one needs to define a distributed array *ControlWorkers* containing the worker numbers, that is, the numbers 0, ..., *n*-1 if there are *n* workers. We continue the example of Section 7.4 on parallel sorting. There we have 14 workers.

```
let ControlWorkers = intstream(0, 14 - 1) transformstream
  ddistribute3["ControlWorkers", 14, TRUE, Workers14]
  dloop["", . feed extract[Elem]]
```

We then run a script (from `secondo/bin/Scripts`)

```
@%Scripts/DistCost.sec
```

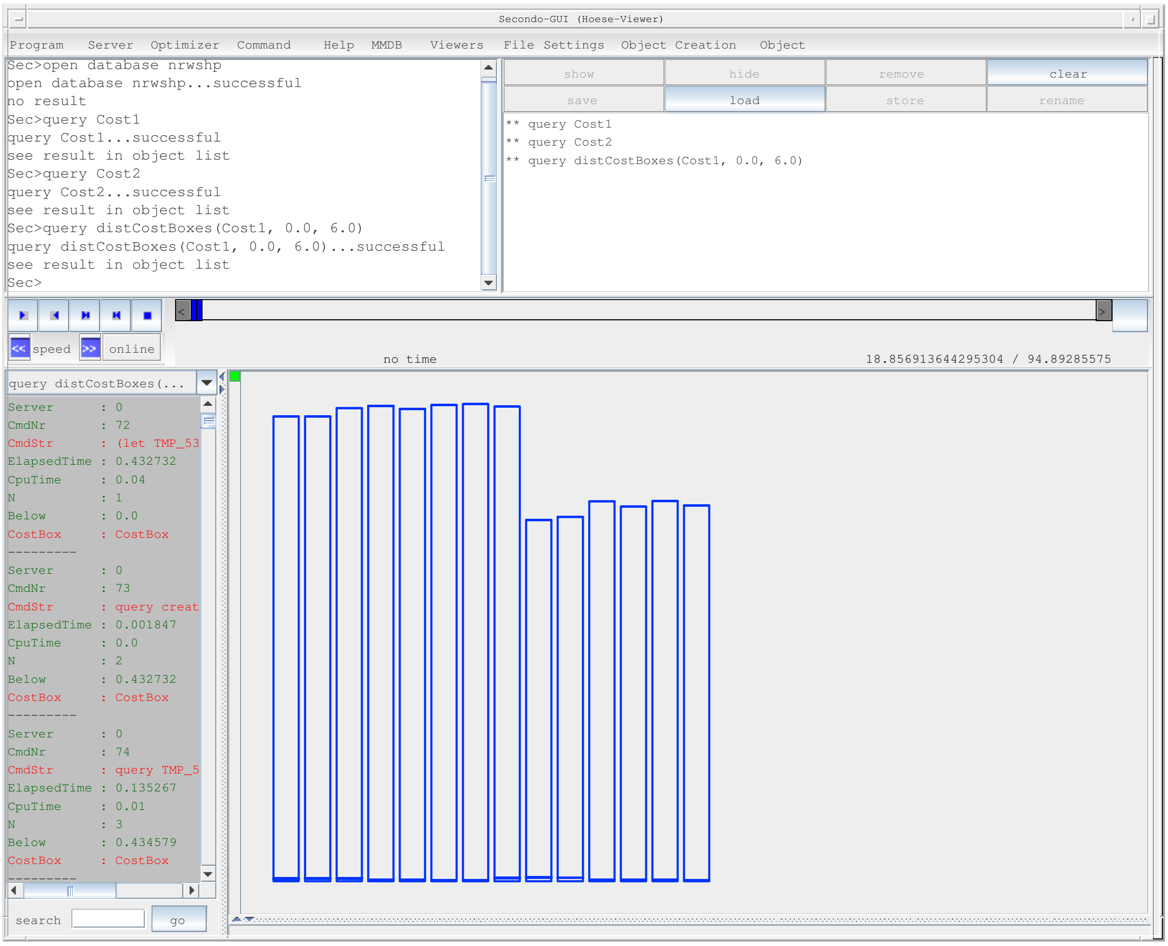
which defines some SECONDO objects and functions in the database:

- *LastCommand*: a relation recording for each worker the number of the last command of this session executed there;
- *distCostReset*: a function to update *LastCommand*;
- *distCostSave*: a function saving into a relation the running times of all commands of all workers executed since the last command;
- *distCostBoxes*: a function constructing from the saved cost relation a relation with rectangles, one for each command executed on each worker. The position of the x-interval of the rectangle corresponds to the worker number (width is the same for all). The y-interval corresponds to the amount of time spent on this command. Boxes are stacked vertically per worker in the order of execution. Hence we get a diagram with subdivided vertical bars per worker and at the top we can see the idle times.
- *distCostUtil*: a function computing from the saved cost relation a number (percentage) called the *utilization* of workers. This is simply the sum of times spent by all workers relative to the total time if all workers had worked to the end of the query.

We measure the last two steps of the distributed sorting algorithm of Section 7.4.

```
update LastCommand := distCostReset(ControlWorkers)
```

```
let BuildingsSortedOsm_idA = BuildingsB1
```



```
Sec>open database nrwshp
open database nrwshp...successful
no result
Sec>query Cost1
query Cost1...successful
see result in object list
Sec>query Cost2
query Cost2...successful
see result in object list
Sec>query distCostBoxes(Cost1, 0.0, 6.0)
query distCostBoxes(Cost1, 0.0, 6.0)...successful
see result in object list
Sec>
```

```
** query Cost1
** query Cost2
** query distCostBoxes(Cost1, 0.0, 6.0)
```

```
Server      : 0
CmdNr       : 72
CmdStr      : (let TMP_53
ElapsedTime : 0.432732
CpuTime     : 0.04
N           : 1
Below       : 0.0
CostBox     : CostBox
---------
Server      : 0
CmdNr       : 73
CmdStr      : query creat
ElapsedTime : 0.001847
CpuTime     : 0.0
N           : 2
Below       : 0.432732
CostBox     : CostBox
---------
Server      : 0
CmdNr       : 74
CmdStr      : query TMP_5
ElapsedTime : 0.135267
CpuTime     : 0.01
N           : 3
Below       : 0.434579
CostBox     : CostBox
---------
```

Figure 6: Workers’ running time diagram derived from *Cost1*.
From left to right there are bars for workers 0, ..., 13

```
    partition["", "Boundaries_Osm_id" "Boundaries" matchbelow[.Osm_id]
    extract[D],  0]

  let Cost1 = distCostSave(ControlWorkers);
  update LastCommand := distCostReset(ControlWorkers)

  let BuildingsSortedOsm_id = BuildingsSortedOsm_idA
    areduce["", . feed sortby[Osm_id], myPort]

  let Cost2 = distCostSave(ControlWorkers)
```

We can now open the database from a Javagui interface and issue the query:

```
  query distCostBoxes(Cost1, 0.0, 6.0)
```

The second parameter is a filter to return only the boxes whose elapsed time is larger than this threshold. Here we select all boxes. The third parameter allows one to adjust the width of a vertical bar. The resulting display is shown in Figure 6.

Remember that *Cost1* is the cost of the query that partitions *BuildingsB1* by *Boundaries*. There are 50 fields to be processed, distributed in a cyclic manner over workers. Hence each of the first 8 workers needs to distribute the data of 4 fields (= 32 fields), the last 6 can only distribute the data of 3 fields each (= 18). This explains why the first 8 bars in Figure 6 are larger than the last 6.

The result of the same query for *Cost2* is shown in Figure 7. Here we have used the *Show only viewer* feature of the Javagui. In this query, the **areduce** operator performs two actions for each field:

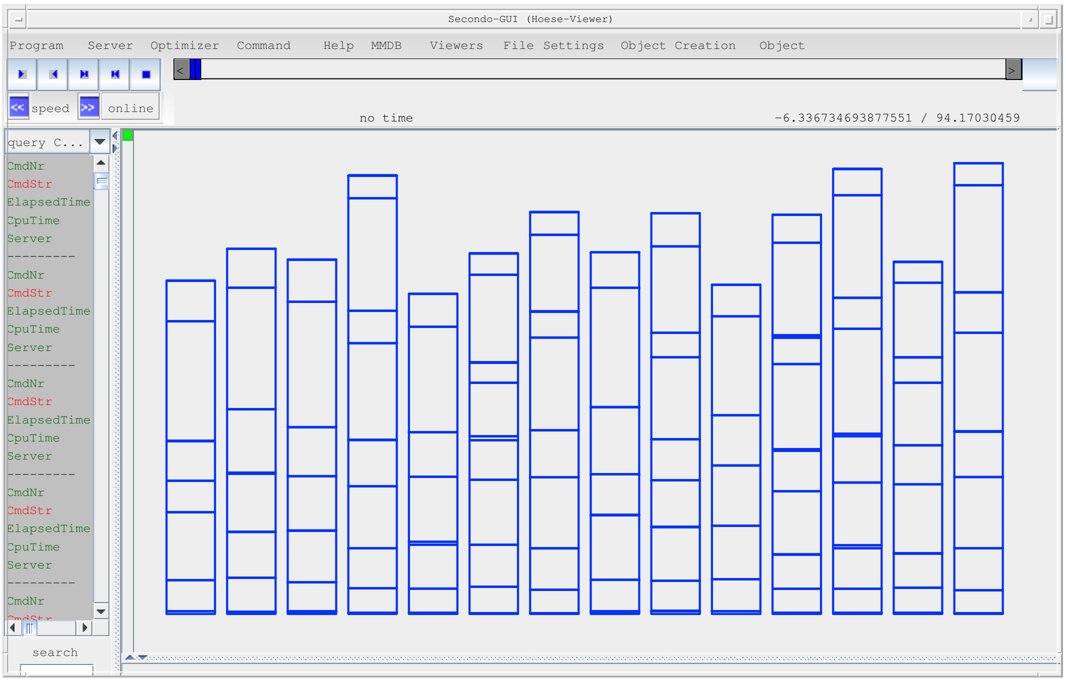


Figure 7: Workers' running time diagram for *Cost2*.

1. It collects the data for this partition from all servers (workers);
2. it sorts the set of tuples for this partition.

This is why we see for each worker an even number of larger boxes (there are also other commands performed by the workers taking only short time). Here, too, some workers process 3 and some 4 fields (6 and 8 boxes, respectively), but which workers process more is distributed dynamically.

We can also measure the worker utilization.

```
    query distCostUtil(Cost1);
    query distCostUtil(Cost2);
```

The results are 90 % and 84 %, respectively.

9 Example: Constructing a Road Network Graph From OSM Data

In this section we explain the process of constructing a road network as a directed graph from OpenStreetMap data. There are two scripts in SECONDO providing a sequential/local and a parallel/distributed implementation, both to be found in `secondo/bin/Scripts`.

- Sequential: script `ORGfromOSM.sec`.
- Parallel: script `importGermanyOsmShort.sec`

In the sequel, we first explain the necessary background, then the implementation on a single machine, and finally a distributed implementation.

9.1 Input Data and Their Structure

Data can be downloaded from GeoFabrik at `http://download.geofabrik.de/` in the format `.osm.bz2`. Place a file such as `germany-latest.osm.bz2` into a directory, e.g. *Daten* in your home directory and unpack it with the command

```
bunzip2 germany-latest.osm.bz2
```

resulting in the file `germany-latest.osm`.

The input file is an XML file which contains as a nested structure three kinds of objects:

- nodes
- ways
- relations

A *node* defines a location in geographic space in terms of latitude and longitude. It also has a unique node identifier. A node may have an associated list of node tags. A *node tag* is a (key, value) pair.

A *way* describes a linear geometry, has its own way identifier, and is defined as a sequence of node identifiers. A way may describe a road or the boundary of a park or building, for example. Note that geographic locations are present only in nodes. Hence two roads meeting in a junction would share a node with a unique location. Ways may have associated *way tags* (= (key, value) pairs) as well.

A *relation* has its own identifier and is defined as a list of references to nodes, ways, or relations. Hence a relation can describe a set of anything, even in a nested manner. Relations may also have relation tags.

9.2 Overview

The construction proceeds in the following major steps which are shown and explained below:

1. Process the input file creating a relational representation, consisting of six relations for nodes, node tags, ways, way tags, relations, and relation tags.
2. Create a copy of the node relation such that latitude and longitude are translated into *point* attributes. Further, assign new numeric identifiers that are clustered spatially (similar numbers will lie in similar locations).
3. From the six relations, reconstruct *Ways* as a nested relation. Each way tuple contains a subrelation containing the sequence of nodes defining the geometry of the way. The geometry is also contained as an attribute of type *line*. The way tuple further contains a subrelation for any tags, i.e., (key, value) pairs.
4. Based on tag information, select *Roads* as a subset of Ways.

5. Construct the *Nodes* of the road network graph. Nodes are either *junctions* of two distinct roads, or first or last locations of the way geometry (called *terminal nodes* here).
6. Construct the directed edges of the road network graph. Edges are pieces of roads between two nodes, e.g. between two junction nodes, or between a junction node and a terminal node. This is done in three steps:
   a. Construct the directed edges in the direction the original way is defined over nodes. If a road is a one-way road, then the allowed direction of traversing it corresponds to this order of nodes.
   b. Construct the directed edges in the opposite direction if the road is not a one-way.
   c. Form the union of the two sets of edges constructed in (a) and (b).

9.3 Local Implementation

In this section we describe the implementation of this procedure on a single SECONDO system.

9.3.1 Data Import: Creating Six Relations

This is done by the following query:

```
query fullosmimport('/home/ralf/Daten/germany-latest.osm', "City")
```

The **fullosmimport** operator performs a single scan of the (possibly huge) input file creating six relations simultaneously with schemas:

```
CityNodes(NodeId: longint, Lat: real, Lon: real)
CityNodeTags(NodeIdInTag: longint, NodeTagKey: text, NodeTagValue: text)
CityWays(WayId: longint, NodeCounter: int, NodeRef: longint)
CityWayTags(WayIdInTag: longint, WayTagKey: text, WayTagValue: text)
CityRelations(RelId: longint, RefCounter: int, MemberType: text,
  MemberRef: longint, MemberRole: text)
CityRelationTags(RelIdInTag: longint, RelTagKey: text, RelTagValue: text)
```

Several tuples in `CityNodeTags` may refer to the same `NodeId` via `NodeIdInTag`. A way is split into tuples such that each tuple contains one node (`NodeRef`) as well as the order number of this node in the sequence (`NodeCounter`). Again, any number of `CityWayTags` may refer to the same `WayId`. Similar strategies apply to `CityRelations` and their tags. For the construction of the road network we only need `CityNodes`, `CityWays`, and `CityWayTags`.

9.3.2 Create Spatially Clustered NodesNew

The road network we construct later consists of *Nodes* and *Edges*. Nodes are given by node identifiers and edges are tuples with two attributes *Source* and *Target*, describing a directed edge from node *Source* to node *Target*. We will store edges clustered by *Source* node identifier to enable efficient graph traversal. Nodes of the graph are based on nodes of OpenStreetMap.

At the same time, an interesting query is to retrieve edges in a certain spatial area. If we use the original node identifiers from OpenStreetMap for *Nodes* of the graph, then these numbers are arbitrarily scattered over the entire geographical area. This means that *Edges* from a small area are stored scattered over the entire set of edges. If we retrieve edges from a small area, we need one page access per edge and retrieval is slow.

This is the reason why we assign new node identifiers which are clustered spatially. Hence nodes whose numbers are similar are likely to be close in the geographical space. The following query constructs the new node identifiers in attribute NodeIdNew which is added to node tuples.

```
let CityNodesNew = CityNodes feed
  extend[Easting: .Lon * 1000000, Northing: .Lat * 1000000]
  extend[Box: rectangle2(.Easting, .Easting, .Northing, .Northing)]
  sortby[Box]
  projectextend[NodeId;  Pos: makepoint(.Lon, .Lat)]
  addcounter[NodeIdNew, 1]
  consume
```

The query first adds to each tuple of *CityNodes* attributes *Easting* and *Northing* which are scaled-up from *Lon* and *Lat*, respectively. It then adds a bounding box *Box* for Easting and Northing (a degenerated rectangle). The stream of tuples is then sorted by this *Box* attribute.

What does it mean to sort a stream of tuples by a rectangle attribute? Rectangles are mapped back into points (e.g. the center) and point values are sorted into *z*-order (see e.g. [Or90]). With the **addcounter** operator we add a numbering attribute to the ordered stream of tuples and use these values as new node identifiers; these are clustered spatially.

Note that sorting coordinates into z-order only works on the integer part of the coordinates. This is why scaling up the numbers initially is necessary.

9.3.3 Create Ways

In the next step we reconstruct the linear features, called *Ways*.

```
let Ways =
  CityNodesNew feed
  CityWays feed itHashJoin[NodeId, NodeRef] sortby[WayId, NodeCounter]
    nest[WayId; NodeList]
    extend[Curve  : .NodeList afeed projecttransformstream[Pos]
      collect_line[TRUE]]
  CityWayTags feed nest[WayIdInTag; WayInfo] itHashJoin[WayId,
    WayIdInTag]
  extend[Box: bbox(.Curve scale[1000000.0])]
  sortby[Box] remove[Box]
  consume
```

CityWays tuples contain references to nodes and they are now joined with the nodes containing the geographic points. They are ordered by WayId and sequence number of the node. This means, all tuples for a given way now appear consecutively in the stream of tuples and in the correct order of nodes. They are now **nest**-ed by *WayId* which means that for a given *WayId* only one tuple with the *WayId* attribute is kept at the top level and all the remaining attributes (the sequence of nodes, in this case) go into a subrelation called *NodeList*. From the *NodeList* a *line* value is computed as attribute *Curve*; the line is simply given by the sequence of node positions.

From the *CityWayTags* in a second step a nested relation is computed which has for each *WayId* a subrelation for the tags. This nested relation is joined with the one resulting from the first step via *WayId*.

Finally, the resulting tuples (one per *Wayid*) are sorted into z-order similarly to the previous subsection.

9.3.4 Select Roads

The next step is simple: from the *Ways* we select *Roads* by the property that a *highway* tag exists.

```
let Roads = Ways feed
  filter[.WayInfo afeed filter[.WayTagKey = "highway"] count > 0]
  consume
```

9.3.5 Construct Nodes

Remember that the *nodes* making up a way describe its complete geometry. In contrast, we now want to determine a subset of the nodes as *Nodes* of the graph, namely junction nodes of two roads, start nodes of a road, and end nodes of a road.

```
let Nodes =
  CityWays feed
  CityWays feed {h2}
  itHashJoin[NodeRef, NodeRef_h2]
  filter[.WayId # .WayId_h2]
  CityNodesNew feed
  itHashJoin[NodeRef, NodeId]
  Roads feed project[WayId] {r1} itHashJoin[WayId, WayId_r1]
  Roads feed project[WayId] {r2} itHashJoin[WayId_h2, WayId_r2]
  project[WayId, NodeCounter, NodeIdNew, Pos]
Roads feed
  projectextend[WayId; Node: .NodeList afeed filter[.NodeCounter = 0]
    aconsume]
  unnest[Node]
  project[WayId, NodeCounter, NodeIdNew, Pos]
  concat
Roads feed
  extend[HighNodeNo: (.NodeList afeed count) - 1]
  projectextend[WayId; Node:  fun(t: TUPLE)
    attr(t, NodeList) afeed filter[.NodeCounter = attr(t, HighNodeNo)]
    aconsume]
  unnest[Node]
  project[WayId, NodeCounter, NodeIdNew, Pos]
  concat
  sortby[WayId, NodeCounter]
  rdup
  consume
```

This query consists of three parts, namely

```
1. CityWays feed ...
2. Roads feed ...
3. Roads feed ...
```

computing these three sets. The first part finds pairs of *CityWays* tuples with the same node identifier but different way identifiers. Such pairs are joined with the node (new version). They are further joined with Roads on each involved way identifier to make sure that it is actually a junction of roads, not of arbitrary ways. The second part finds start nodes of roads and the third part end nodes, respectively. All found *Nodes* are brought into the same format, sorted by *WayId* and *NodeCounter*, and duplicates are eliminated.

Note that junction nodes are created twice, once for each involved *Wayid*. Hence for each *Wayid* we now have a sequence *start node*, *junction node* 1, ..., *junction node n*, *end node* (where junction nodes may be missing or coincide with end nodes)

9.3.6 Construct Edges

The task is now to construct edges of the road network, that is, pieces of road between start/end points and junctions. For a given way (*WayId*), we have on the one hand the sequence of *nodes* (coming from *CityNodes*) and the sequence of *Nodes* derived in the previous step (see Figure 8).

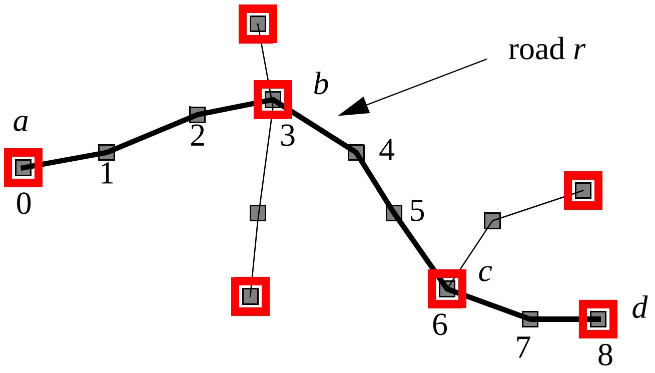


Figure 8: A way/road (drawn fat), its nodes (small, black) and its Nodes (large, red). Labels *a*, *b*, *c*, *d* represent node identifiers.

For the road *r* shown in Figure 8, we need to construct three directed edges in *Up* direction (the direction of increasing node numbers), namely *a-b*, *b-c*, and *c-d* as well as the reverse edges in *Down* direction, if this road does not happen to be a one-way.

Up Edges

```
let EdgesUp =
  Nodes feed nest[WayId; SectionNodes]
    projectextend[WayId; Sections: .SectionNodes afeed
      extend_last[Source: ..NodeIdNew::0, Target: .NodeIdNew::0,
        SourcePos: ..Pos::[const point value undef],
        TargetPos: .Pos::[const point value undef],
        SourceNodeCounter: ..NodeCounter::0,
        TargetNodeCounter: .NodeCounter::0]
      filter[.Source # 0]
      project[Source, Target, SourcePos, TargetPos,
        SourceNodeCounter, TargetNodeCounter]
      aconsume]
```

The query processes the stream of *Node* tuples which is ordered by *Wayid* and *NodeCounter* from the previous step. It is nested by *WayId* so that we have one tuple for the road with subrelation *SectionNodes*. For the road *r* of Figure 8, the subrelation would have four tuples. In the **projectextend** operator, the subrelation *SectionNodes* is processed. The **extend_last** operator allows one to construct new attributes based on attributes of the current tuple (accessed by `.<attr>`) as well as attributes of the preceding tuple (accessed by `..<attr>`) within a stream of tuples. For example, for the second tuple of the stream for road *r*, *Source* is set to *a*, *Target* to *b*, *SourcePos* to the position (<u>*point*</u>) of *a*, *TargetPos* to the position of *b*, *SourceNodeCounter* to 0, and *TargetNodeCounter* to 3. The first tuple is a special case, as there is no preceding tuple in the stream; in this case the new attribute values are taken from the constant specified behind the :: notation. The first tuple is later eliminated by the filter condition. Hence as a result, three tuples are constructed and stored in a new subrelation *Sections*, each describing an edge.

```
Roads feed {r}
itHashJoin[WayId, WayId_r]
projectextend[WayId; Sections: fun(t:TUPLE)
  attr(t, Sections) afeed
  extend[
    Curve: fun(u: TUPLE)
    attr(t, NodeList_r) afeed
      filter[.NodeCounter_r between[attr(u, SourceNodeCounter),
        attr(u, TargetNodeCounter)] ]
      projecttransformstream[Pos_r]
      collect_sline[TRUE],
    RoadName: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "name"]
      extract [WayTagValue_r],
    RoadType: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "highway"]
      extract [WayTagValue_r]
  ]
  aconsume ]
unnest[Sections]
consume
```

Continuing the query, the *Roads* are joined by *WayId* so that now we have a tuple with the complete road information as well as the sequence of edges for this road. In the **projectextend** operator, the *Sections* subrelation is transformed: for each edge (input *Section* tuple), the geometric <u>*line*</u> value for this piece of road is constructed in attribute *Curve*; this is done by scanning the node list of the road between the *SourceNodeCounter* and the *TargetNodeCounter*, feeding node positions into the **collect_line** operator. Moreover, the edge is extended by the name of the road and its type. Finally, the stream of tuples, each with a subrelation *Sections*, is **unnest**-ed so that the output are flat tuples that can be collected into a regular relation.

Down Edges

```
let EdgesDown =
  Nodes feed nest[WayId; SectionNodes]
    projectextend[WayId; Sections: .SectionNodes afeed
      sortby[NodeCounter desc]
      extend_last[Source: ..NodeIdNew::0, Target: .NodeIdNew::0,
        SourcePos: ..Pos::[const point value undef],
        TargetPos: .Pos::[const point value undef],
        SourceNodeCounter: ..NodeCounter::0,
        TargetNodeCounter: .NodeCounter::0]
      filter[.Source # 0]
      project[Source, Target, SourcePos, TargetPos,
        SourceNodeCounter, TargetNodeCounter]
      aconsume]
  Roads feed
    filter[.WayInfo afeed filter[.WayTagKey = "oneway"]
      filter[(.WayTagValue = "yes")] count = 0] {r}
  itHashJoin[WayId, WayId_r]
  projectextend[WayId; Sections: fun(t:TUPLE)
    attr(t, Sections) afeed extend[Curve: fun(u: TUPLE)
      attr(t, NodeList_r) afeed sortby[NodeCounter_r desc]
        filter[.NodeCounter_r between[attr(u, TargetNodeCounter),
          attr(u, SourceNodeCounter)] ]
        projecttransformstream[Pos_r]
        collect_sline[TRUE],
      RoadName: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "name"]
        extract [WayTagValue_r],
      RoadType: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "highway"]
        extract [WayTagValue_r]
    ]
    aconsume ]
  unnest[Sections]
  consume
```

The construction of reverse (*Down*) edges is almost the same. The difference is that *Nodes* are now processed in the opposite order (via `sortby[NodeCounter desc]`) and *nodes* as well. Furthermore, *Roads* are checked to not contain a (key, value) pair (*oneway*, *yes*) in which case *Down* edges must not be constructed.

Edges

```
  let Edges = EdgesUp feed EdgesDown feed concat
    projectextend[Source, Target, SourcePos, TargetPos, SourceNodeCounter,
      TargetNodeCounter, Curve, RoadName,
      RoadType; WayId: .WayId]
    consume
```

Finally, the set of *Edges* is constructed as the union of *Up* and *Down* edges, putting the *WayId* attribute to the end of tuples, in order to have *Source* and *Target* as the first two attributes.

9.4 Distributed Implementation

9.4.1 Prerequisites

We assume that a set of computers with SECONDO installations is available. A saved relation file with worker definitions is present as `WorkersNewton` in `secondo/bin`. The monitors controlling these workers are defined in file `ClusterRalfNewton` in `secondo/bin`. This corresponds to the setup of cluster *newton* in Section 3.2.

9.4.2 Distributing OSM Data Fast

Providing the OSM data file for local data import requires two steps:

- download to *newton1*
- unpacking on *newton1*

It is crucial not to loose much time by providing data for the distributed import. We therefore proceed as follows:

- download to *newton1*
- copy the compressed file *in parallel* from *newton1* to the other machines (*newton2*, ..., *newton5*)
- unpack *in parallel* on all machines *newton1*, ..., *newton5*.

Workers will later read portions of the complete input file, now available on all computers, independently in parallel.

For the file `germany-latest.osm`, the file sizes are:

- compressed: `germany-latest.osm.bz2:` 4.4 GB
- decompressed: `germany-latest.osm:` 52.6 GB

The running times for the steps for this file are as follows:

Table 2:

Local Implementation		Distributed Implementation	
download to newton1	06:35 min	download to newton1	06:35 min
		copy compressed to others	02:42 min
unpack on newton 1	20:54 min	unpack on all	20:54 min
Total Time	**27:29 min**		**30:11 min**

Hence there is very little overhead for making the data available for parallel import. A shell script `importGermanyOsmPrepare.sh` performing the parallel copying and unpacking is shown in Appendix.

As a result, the data file to be processed has been distributed to all computers providing workers and is available at the same path, say

```
/home/ralf/Daten/germany-latest.osm
```

9.4.3 Preparations

A bash is started and we navigate to `secondo/bin`. Monitors are started with the shell command:

```
remoteMonitors ClusterRalfNewton start
```

The master SECONDO system is started:

```
SecondoTTYBDB
```

Any other SECONDO user interface, e.g. the `Javagui` or `SecondoTTYCS` may also be used for the master. The following commands are entered at the master.

```
create database germanyosm
```

```
open database germanyosm
```

We create a new database and open it.

```
restore WorkersNewton from WorkersNewton
```

```
let Workers = WorkersNewton

let NWorkers = Workers count

let NSlots = 160

query share("NSlots", TRUE, Workers)

let File = '/home/ralf/Daten/germany-latest.osm'

query share("File", TRUE, Workers)
```

Workers are restored and a few constants defined and shared with workers (that is, objects *NSlots* and *File* are defined in each worker database and can therefore be used in queries on workers).

Note that the numbers of workers and slots can be chosen independently although there should be more slots than workers (otherwise some workers will be idle). In our example implementation we use a cluster with 40 workers; hence each worker normally needs to process 4 slots sequentially.

```
let ControlSlots = intstream(0, NSlots - 1) transformstream
  ddistribute3["ControlSlots", NSlots, TRUE, Workers]
  dloop["", . feed extract[Elem]]
```

A distributed array is defined whose fields contain the integers 0, ..., 159.

9.4.4 Data Import: Creating Six Relations

```
let Division = ControlSlots
  dloop["Division", divide_osm3(File, NSlots, .)]
```

Workers create in parallel portions of the input file. The **divide_osm3** operator assumes that the input file *File* is to be split into *NSlots* parts of about equal size and extracts the *n*-th part if *n* is its third parameter. The part is stored with name *File_n*. The **dloop** operator lets workers process all the slots of array *ControlSlots* in parallel and sequentially per worker, referring to the slot value by "." so that for each slot the corresponding file is made available. The **divide_osm3** operator returns a Boolean indicating success; hence *Division* is created as a distributed array of <u>*bool*</u>.

```
let Import = ControlSlots
  dloop["Import", fullosmimport(File + "_" + num2string(.), "City", .)]
```

For each slot, the corrresponding input file is processed by the **fullosmimport** operator. It constructs the six relations as seen before. Here it has a third parameter (the slot number) which is used as a suffix for the created relation, e.g. `CityNodes_27`. *Import* is returned as a distributed array of <u>*bool*</u>.

Remember that the big OSM-file has node and node tag definitions at the beginning, ways and way tags in the middle, and relations and relation tags at the end. Therefore, for the first slots only relations *CityNodes* and *CityNodeTags* will have entries; the others are empty. This will change for later slots.

We now need to create distributed arrays on the master controlling the fields whose values (relations in this case) have already been constructed by the workers.

```
let CityNodes_type = [const rel(tuple([NodeId: longint, Lat real,
  Lon: real])) value ()];
let CityNodeTags_type = [const rel(tuple([NodeIdInTag: longint,
  NodeTagKey: text, NodeTagValue: text])) value ()];
let CityWays_type = [const rel(tuple([WayId: longint, NodeCounter: int,
  NodeRef: longint])) value ()];
let CityWayTags_type = [const rel(tuple([WayIdInTag: longint,
  WayTagKey: text, WayTagValue: text])) value ()];
let CityRelations_type = [const rel(tuple([RelId: longint,
  RefCounter: int, MemberType: text, MemberRef: longint,
  MemberRole: text])) value ()];
let CityRelationTags_type = [const rel(tuple([RelIdInTag: longint,
  RelTagKey: text, RelTagValue: text])) value ()];
```

First, for each of the six relations empty templates are created whose types can be used in creating the distributed arrays.

```
let CityNodesB0 = Workers feed
  createDArray["CityNodes", NSlots, CityNodes_type, TRUE]

let CityNodeTagsB0 = Workers feed
  createDArray["CityNodeTags", NSlots, CityNodeTags_type, TRUE]

let CityWaysB0 = Workers feed
  createDArray["CityWays", NSlots, CityWays_type, TRUE]

let CityWayTagsB0 = Workers feed
  createDArray["CityWayTags", NSlots, CityWayTags_type, TRUE]

let CityRelationsB0 = Workers feed
  createDArray["CityRelations", NSlots, CityRelations_type, TRUE]

let CityRelationTagsB0 = Workers feed
  createDArray["CityRelationTags", NSlots, CityRelationTags_type, TRUE]
```

Then, the **createDArray** operator processes each available worker and asks it which slots of the distributed array to be constructed are available in the worker’s database.

In the following, we only need relations *CityNodes*, *CityWays*, and *CityWayTags* for the construction of the road network. Moreover, we will next join *CityNodes* with *CityWays* via attributes *NodeId* and *NodeRef* and later *CityWayTags* with a previous result via attribute *WayIdInTag*. We therefore now redistribute (shuffle) each of these distributed relations by the respective attributes to prepare for the joins.

```
let CityNodesB1_NodeId = CityNodesB0 partitionF["", . feed,
  hashvalue(..NodeId, 999997), NSlots]
  collect2["CityNodesB1", myPort]

let CityWaysB1_NodeRef = CityWaysB0 partitionF["", . feed,
    hashvalue(..NodeRef, 999997), NSlots]
  collect2["CityWaysB1", myPort]

let CityWayTagsB1_WayIdInTag = CityWayTagsB0 partitionF["", . feed,
  hashvalue(..WayIdInTag, 999997), NSlots]
  collect2["CityWayTagsB1", myPort]
```

The **partitionF** operator, applied to a distributed array whose fields contain relations, lets workers in parallel (and sequentially per worker) partition the relation of a field into *NSlot* relations. The second argument of **partitionF** (here `""`) is a string to name resulting data on the workers; if empty, naming is done automatically. The third parameter (here `. feed`) is a function yielding a stream of tuples; it can be used to manipulate tuples of the relation before redistributing them (e.g. filter them or add attributes). The fourth argument (here `hashvalue(..NodeId, 999997)`) is a function for distribution where "`..`" refers to the tuples returned by the third argument function. The last argument provides the number of slots over which to distribute data. The overall result of **partitionF** is a *distributed file matrix*, here with *NSlot* rows and *NSlot* columns.

The **collect2** operator aggregates all the files in a column of the matrix into a single file, for each column. Result is a distributed file array whose fields contain relations.

The resulting distributed (file) arrays contain *CityNodes* partitioned by *NodeId*, *CityWays* by *NodeRef*, and *CityWayTags* by *WayIdInTag* and they are named accordingly.

9.4.5 Create Spatially Clustered NodesNew

```
let NCityNodes = CityNodesB1_NodeId dloop["", . feed count] getValue
  tie[. + ..]

let MaxNodesPerSlot = (NCityNodes div NSlots) * 3

query share("MaxNodesPerSlot", TRUE, Workers)

let CityNodesNewB1_NodeId =
  CityNodesB1_NodeId ControlSlots dmap2["CityNodesNewB1",
  . feed
  extend[Easting: .Lon * 1000000, Northing: .Lat * 1000000]
  extend[Box: rectangle2(.Easting, .Easting, .Northing, .Northing)]
  sortby[Box]
  projectextend[NodeId;  Pos: makepoint(.Lon, .Lat)]
  addcounter[NodeIdNew, (.. * MaxNodesPerSlot) + 1] , myPort]
```

The relation *CityNodesNew* is constructed to contain node positions as points and new, spatially clustered identifiers. Here this is done in parallel, for each field of *CityNodesB1_NodeId* and the corresponding field number. The latter is taken from array *ControlSlots*. The **dmap2** operator processes in parallel pairs of fields of its two input arrays, namely the two fields with index 0, two fields with index 1, and so forth. The parameter query refers to the two argument field values by "`.`" and "`..`", respectively. The second argument (the field number) is just needed in the **addcounter** operation to make the sets of identifiers of distinct fields disjoint.

Note that within each field, node identifiers are spatially clustered, but the nodes of each field are scattered over the entire geographic area. This is not a problem if range queries are applied to fields in parallel. Even globally, the clustering should still be fairly good.

9.4.6 Create Ways

```
let WaysB1_WayId = CityNodesNewB1_NodeId CityWaysB1_NodeRef dmap2["",
  . feed
  .. feed itHashJoin[NodeId, NodeRef], myPort]
  partitionF["", . feed, hashvalue(..WayId, 999997), 0]
  collect2["WaysB1", myPort]
```

We join *CityNodes* with *CityWays* via *NodeId* and *NodeRef* and redistribute resulting join tuples by *WayId*. This implies that all tuples of one *Way* arrive in exactly one partition.

```
let WaysB3_WayId = WaysB1_WayId CityWayTagsB1_WayIdInTag dmap2["",
  . feed sortby[WayId, NodeCounter] nest[WayId; NodeList]
    extend[Curve  : .NodeList afeed projecttransformstream[Pos]
      collect_line[TRUE]]
  .. feed sortby[WayIdInTag] nest[WayIdInTag; WayInfo]
  itHashJoin[WayId, WayIdInTag]
  extend[Box: bbox(.Curve scale[1000000.0])]
  sortby[Box]
  remove[Box]
  consume, myPort]
```

The rest of the query to construct *Ways* can be performed partition-wise.

9.4.7 Select Roads

```
let RoadsB1_WayId = WaysB3_WayId
  dloop["RoadsB1_WayId", . feed
    filter[.WayInfo afeed filter[.WayTagKey = "highway"] count > 0]
    consume]
```

For each partition (field), we select roads.

9.4.8 Construct Nodes

```
let NodesB1 =
  CityWaysB1_NodeRef CityNodesNewB1_NodeId dmap2["",
    . feed
    . feed {h2}
    itHashJoin[NodeRef, NodeRef_h2]
    filter[.WayId # .WayId_h2]
    .. feed
    itHashJoin[NodeRef, NodeId]
    project[WayId, NodeCounter, NodeIdNew, Pos, WayId_h2], myPort
  ]
  partitionF["", . feed, hashvalue(..WayId, 999997), NSlots]
  collect2["", myPort]
  RoadsB1_WayId dmap2["", . feed .. feed project[WayId] {r1}
    itHashJoin[WayId, WayId_r1], myPort]
  partitionF["", . feed, hashvalue(..WayId_h2, 999997), NSlots]
  collect2["", myPort]
  RoadsB1_WayId dmap2["", . feed .. feed project[WayId] {r2}
    itHashJoin[WayId_h2, WayId_r2]
    project[WayId, NodeCounter, NodeIdNew, Pos], myPort]
```

The first query constructs junction nodes. *CityWays* and *CityNodesNew* can be joined field-wise as they are already distributed on the join attributes *NodeRef* and *NodeId*, respectively. Note that the resulting join tuples contain two way identifiers *WayId* and *WayId_h2* because they represent junctions of two ways. These tuples need to be shuffled by *WayId* to join with the *Roads* for the first way; the resulting join tuples need to be reshuffled by *WayId_h2* a second time to be joined with the *Roads* for the second way.

```
let NodesB2 =
  RoadsB1_WayId dloop["", fun(r: DARRAYELEM) r feed
```

```
    projectextend[WayId; Node: .NodeList afeed filter[.NodeCounter = 0]
      aconsume]
    unnest[Node]
    project[WayId, NodeCounter, NodeIdNew, Pos]
  r feed
    extend[HighNodeNo: (.NodeList afeed count) - 1]
    projectextend[WayId; Node:  fun(t: TUPLE)
      attr(t, NodeList) afeed filter[.NodeCounter = attr(t, HighNodeNo)]
      aconsume]
    unnest[Node]
    project[WayId, NodeCounter, NodeIdNew, Pos]
    concat consume
  ]
```

The second query computes start nodes and end nodes within partitions of *Roads*.

```
let NodesB3_WayId = NodesB1 NodesB2 dmap2["",
  . feed .. feed concat, myPort]
  partitionF["", . feed, hashvalue(..WayId, 999997), NSlots]
  collect2["", myPort]
  dmap["", . feed sortby[WayId, NodeCounter] rdup consume]
```

The third query forms on each partition the union of all nodes constructed and then repartitions them all by *WayId*. The resulting partitions contain for each way all its *Nodes*, sorted by *Wayid* and *NodeCounter*, as needed in the next step.

9.4.9 Construct Edges

Up Edges

```
let NodesB4_WayId = NodesB3_WayId dloop["",
    . feed nest[WayId; SectionNodes]
    projectextend[WayId; Sections: .SectionNodes afeed
      extend_last[Source: ..NodeIdNew::0, Target: .NodeIdNew::0,
        SourcePos: ..Pos::[const point value undef],
        TargetPos: .Pos::[const point value undef],
        SourceNodeCounter: ..NodeCounter::0,
        TargetNodeCounter: .NodeCounter::0]
      filter[.Source # 0]
      project[Source, Target, SourcePos, TargetPos,
        SourceNodeCounter, TargetNodeCounter]
      aconsume]
    consume]
```

This implements the first part of the query of Section 9.3.6, constructing *Sections* subrelations. The *Nodes* relation is already partitioned by *Wayid* from the previous step; hence this query can be computed partition-wise. Within each partition, tuples are already sorted by *WayId* and *NodeCounter*; hence the sequential query can simply be copied.

```
let EdgesUpB1_WayId = NodesB4_WayId RoadsB1_WayId dloop2["",
  . feed .. feed {r}
  itHashJoin[WayId, WayId_r]
  projectextend[WayId; Sections: fun(t:TUPLE)
    attr(t, Sections) afeed
    extend[
      Curve: fun(u: TUPLE)
      attr(t, NodeList_r) afeed
```

```
        filter[.NodeCounter_r between[attr(u, SourceNodeCounter),
          attr(u, TargetNodeCounter)] ]
        projecttransformstream[Pos_r]
        collect_sline[TRUE],
      RoadName: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "name"]
        extract[WayTagValue_r],
      RoadType: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "highway"]
        extract[WayTagValue_r]
    ]
    aconsume ]
  unnest[Sections] consume]
```

Second part of the query of Section 9.3.6 which includes a join with *Roads*. Partitioning is correct, so the join can be applied directly.

Down Edges

```
let NodesB5_WayId = NodesB3_WayId dloop["",
  . feed nest[WayId; SectionNodes]
    projectextend[WayId; Sections: .SectionNodes afeed
      sortby[NodeCounter desc]
      extend_last[Source: ..NodeIdNew::0, Target: .NodeIdNew::0,
        SourcePos: ..Pos::[const point value undef],
        TargetPos: .Pos::[const point value undef],
        SourceNodeCounter: ..NodeCounter::0,
        TargetNodeCounter: .NodeCounter::0]
      filter[.Source # 0]
      project[Source, Target, SourcePos, TargetPos,
        SourceNodeCounter, TargetNodeCounter]
      aconsume]
    consume]

let EdgesDownB1_WayId = NodesB5_WayId RoadsB1_WayId dloop2["",
  . feed
  .. feed filter[.WayInfo afeed filter[.WayTagKey = "oneway"]
    filter[(.WayTagValue = "yes")] count = 0] {r}
  itHashJoin[WayId, WayId_r]
  projectextend[WayId; Sections: fun(t:TUPLE)
    attr(t, Sections) afeed extend[Curve: fun(u: TUPLE)
      attr(t, NodeList_r) afeed sortby[NodeCounter_r desc]
        filter[.NodeCounter_r between[attr(u, TargetNodeCounter),
          attr(u, SourceNodeCounter)] ]
        projecttransformstream[Pos_r]
        collect_sline[TRUE],
      RoadName: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "name"]
        extract[WayTagValue_r],
      RoadType: attr(t, WayInfo_r) afeed filter[.WayTagKey_r = "highway"]
        extract[WayTagValue_r]
    ]
    aconsume ]
  unnest[Sections]
  consume]
```

Similar to the previous subsection.

9.4.10 Edges

```
let EdgesB1_WayId = EdgesUpB1_WayId EdgesDownB1_WayId dloop2["",
  . feed .. feed concat
  projectextend[Source, Target, SourcePos, TargetPos, SourceNodeCounter,
    TargetNodeCounter, Curve, RoadName,
    RoadType; WayId: .WayId]
  consume ]
```

The union of the two sets of edges is computed partition-wise.

9.5 Experimental Comparison

We compare the local and the distributed implementation for three data sets, namely the administrative region of Arnsberg, the state of North Rhine-Westfalia, and the whole country of Germany. Sizes of data are shown in Table 3.

	Arnsberg	NRW	Germany
Decompressed file size [GB]	2.77	12.93	52.61
# CityNodes	13080110	59596159	246682839
# CityWays	17247983	81996292	326748302
# CityWayTags	6269695	30359606	110938532
# Ways	2016456	10028723	39201644
# Roads	371624	1621132	9481614
# Nodes	1068932	4728257	27583243
# Edges	1359481	6022426	35359838

Table 3: File Sizes and Cardinalities of Relations

The running times of the single node and the distributed implementation, the latter using cluster newton with 40 workers, are shown in Table 4.

	Arnsberg	NRW	Germany
Single node implementation (28800 MB)	3914	37548	358659
Distributed implementation (40 workers)	339	1229	6343
Speedup	**11.5**	**30.5**	**56.5**

Table 4: Running times of single node and distributed implementation [seconds]

Transaction management was disabled in both versions. The single node implementation used roughly all the available main memory to be as fast as possible. For larger problems, the speedup gets larger because the single node version cannot process all data in memory any more.

References

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

[Or90] Jack A. Orenstein: A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces. SIGMOD Conference 1990: 343-352.

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

Appendix

A File ClusterRalfNewton

```
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databases  63414
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databases 63415
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databases 63416
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databases 63417
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databases  63414
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databases 63415
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databases 63416
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databases 63417
132.176.69.195 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databases  63414
132.176.69.195 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databases 63415
132.176.69.195 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databases 63416
132.176.69.195 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databases 63417
132.176.69.196 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databases  63414
132.176.69.196 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databases 63415
132.176.69.196 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databases 63416
132.176.69.196 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databases 63417
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databases  63414
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databases 63415
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databases 63416
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databases 63417
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databasesB  63410
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databasesB 63411
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databasesB 63412
132.176.69.193 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databasesB 63413
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databasesB  63410
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databasesB 63411
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databasesB 63412
132.176.69.194 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databasesB 63413
132.176.69.195 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databasesB  63410
132.176.69.195 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databasesB 63411
132.176.69.195 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databasesB 63412
132.176.69.195 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databasesB 63413
132.176.69.196 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databasesB  63410
132.176.69.196 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databasesB 63411
132.176.69.196 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databasesB 63412
132.176.69.196 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databasesB 63413
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /home/ralf/secondo-databasesB  63410
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /diskb/ralf/secondo-databasesB 63411
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /diskc/ralf/secondo-databasesB 63412
132.176.69.197 SecondoConfig.ini /home/ralf/secondo/bin /diskd/ralf/secondo-databasesB 63413
```

B File WorkersNewton

```
(OBJECT WorkersNewton
    ()
    (rel
        (tuple
            (
                (Host string)
                (Port int)
                (Config string))))
    (
        ("132.176.69.193" 63414 "SecondoConfig.ini")
        ("132.176.69.193" 63415 "SecondoConfig.ini")
        ("132.176.69.193" 63416 "SecondoConfig.ini")
        ("132.176.69.193" 63417 "SecondoConfig.ini")

        ("132.176.69.194" 63414 "SecondoConfig.ini")
        ("132.176.69.194" 63415 "SecondoConfig.ini")
        ("132.176.69.194" 63416 "SecondoConfig.ini")
        ("132.176.69.194" 63417 "SecondoConfig.ini")
        ("132.176.69.195" 63414 "SecondoConfig.ini")
        ("132.176.69.195" 63415 "SecondoConfig.ini")
        ("132.176.69.195" 63416 "SecondoConfig.ini")
        ("132.176.69.195" 63417 "SecondoConfig.ini")

        ("132.176.69.196" 63414 "SecondoConfig.ini")
        ("132.176.69.196" 63415 "SecondoConfig.ini")
        ("132.176.69.196" 63416 "SecondoConfig.ini")
        ("132.176.69.196" 63417 "SecondoConfig.ini")

        ("132.176.69.197" 63414 "SecondoConfig.ini")
        ("132.176.69.197" 63415 "SecondoConfig.ini")
        ("132.176.69.197" 63416 "SecondoConfig.ini")
        ("132.176.69.197" 63417 "SecondoConfig.ini")

        ("132.176.69.193" 63410 "SecondoConfig.ini")
        ("132.176.69.193" 63411 "SecondoConfig.ini")
        ("132.176.69.193" 63412 "SecondoConfig.ini")
        ("132.176.69.193" 63413 "SecondoConfig.ini")

        ("132.176.69.194" 63410 "SecondoConfig.ini")
        ("132.176.69.194" 63411 "SecondoConfig.ini")
        ("132.176.69.194" 63412 "SecondoConfig.ini")
        ("132.176.69.194" 63413 "SecondoConfig.ini")

        ("132.176.69.195" 63410 "SecondoConfig.ini")
        ("132.176.69.195" 63411 "SecondoConfig.ini")
        ("132.176.69.195" 63412 "SecondoConfig.ini")
        ("132.176.69.195" 63413 "SecondoConfig.ini")

        ("132.176.69.196" 63410 "SecondoConfig.ini")
        ("132.176.69.196" 63411 "SecondoConfig.ini")
        ("132.176.69.196" 63412 "SecondoConfig.ini")
        ("132.176.69.196" 63413 "SecondoConfig.ini")

        ("132.176.69.197" 63410 "SecondoConfig.ini")
        ("132.176.69.197" 63411 "SecondoConfig.ini")
        ("132.176.69.197" 63412 "SecondoConfig.ini")
        ("132.176.69.197" 63413 "SecondoConfig.ini")
))
```

C Script importGermanyOsmPrepare.sh

```
#!/bin/bash

datafile="germany-latest.osm.bz2"

date

scp ralf@newton1:/home/ralf/Daten/$datafile ralf@newton2:/home/ralf/Daten/ &
pid2=$!
scp ralf@newton1:/home/ralf/Daten/$datafile ralf@newton3:/home/ralf/Daten/ &
pid3=$!
scp ralf@newton1:/home/ralf/Daten/$datafile ralf@newton4:/home/ralf/Daten/ &
pid4=$!
scp ralf@newton1:/home/ralf/Daten/$datafile ralf@newton5:/home/ralf/Daten/ &
pid5=$!

echo "wait for copying ..."

wait $pid2 $pid3 $pid4 $pid5

echo "finished"

date
```

```
# 2:42 min

ssh ralf@newton1 bunzip2 /home/ralf/Daten/$datafile &
pid11=$!
ssh ralf@newton2 bunzip2 /home/ralf/Daten/$datafile &
pid12=$!
ssh ralf@newton3 bunzip2 /home/ralf/Daten/$datafile &
pid13=$!
ssh ralf@newton4 bunzip2 /home/ralf/Daten/$datafile &
pid14=$!
ssh ralf@newton5 bunzip2 /home/ralf/Daten/$datafile &
pid15=$!

echo "wait for unpacking ..."

wait $pid11 $pid12 $pid13 $pid14 $pid15

echo "finished"

date

# 20:54 min
```