

# SECONDO

## Example: How to Write Parallel Queries in Parallel SECONDO

Jiamin Lu, Oct 2012

SECONDO was built by the SECONDO team

### 1 Overview of Parallel Secondo

Parallel SECONDO extends the scalability of SECONDO database system, in order to process moving objects data in parallel on a computer cluster. It is composed by Hadoop and a set of single-computer SECONDO databases. Here Hadoop is used as a task coordinator and communication level, while most database queries are processed by SECONDO databases simultaneously. The functions of Parallel SECONDO are provided by two algebras, Hadoop and HadoopParallel. Both have been published since SECONDO 3.3.2.

The basic infrastructure of Parallel SECONDO is shown in Figure 1. HDFS is a distributed file system prepared by Hadoop to shuffle its intermediate data during parallel procedures. PSFS (Parallel SECONDO File System) is a similar system provided in Parallel SECONDO, directly exchanging data among single-computer SECONDO databases without passing through HDFS, in order to improve the data transfer efficiency in the parallel system. The minimum execution unit in Parallel SECONDO is called Data Server (DS). Each DS contains a compact SECONDO named Mini-SECONDO, its database storage and a PSFS node. At present, it is quite common that a low-end PC has several hard disks, large memory and several processors with multiple cores. Therefore, it is possible for the user to set several DSs on one computer, in order to fully use the computing resource of the underlying cluster. Hadoop nodes are set independently with DSs, each is set inside the first DS of a computer. More details about Parallel SECONDO can be found in the document “User Guide For Parallel SECONDO”, which is also openly published on our website.

At the current stage, all parallel queries in Parallel SECONDO are expressed in SECONDO executable language. In this document, the way of changing a sequential query into a parallel query is demonstrated, in order to help the user quickly getting familiar with this system.

### 2 Install and Uninstall Parallel Secondo

Parallel SECONDO can be installed on either a single computer or a cluster composed by tens and even hundreds of computers. In this demonstration, Parallel SECONDO is installed on a computer having one AMD Phenom(tm) II X6 1055T processor with six cores, 8 GB memory and one 500 GB hard disk. Here the Ubuntu 10.04.2-LTS-Desktop-64bit is used as the operating system, while other platforms including Ubuntu 12.04 and MacOSX 10.5 have also been tested.

At the same time, a virtual machine (VM) image of a single-computer Parallel SECONDO has also been provided on our website. The user can experience Parallel SECONDO with the VM image without installing it on his own machine.

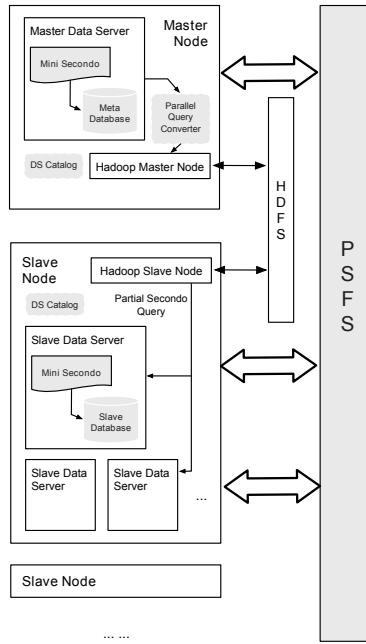


Figure 1: The Infrastructure of Parallel Secondo

## Prerequisites

Few utilities must be prepared before installing Parallel SECONDO. They are all ordinary linux commands working on many Linux platforms and MacOSX.

- Java™ 1.6.x or above. The openjdk-6 is automatically prepared along with the installation of SECONDO.
- SSH connection. Both Data Servers and the underlying Hadoop platform rely on secure shell as the basic communication level. For example on Ubuntu, the SSH server is not installed by default, and can be installed with the command:

```
$ sudo apt-get install ssh
```

- **screen** is also requested by Parallel SECONDO scripts. In Ubuntu, it can be installed like:

```
$ sudo apt-get install screen
```

- Particularly in Hadoop, a passphraseless SSH connection is required. The user can check and set it up through the following commands. First, carry out the “ssh” command to see wether a passphrase is required.

```
$ ssh <IP>
```

Here the “IP” is the IP address of the current computer, and usually can be found out through the “ifconfig” command. If a password is asked for this connection, then an authentication key-pair should be created with the commands:

```
$ cd $HOME
$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

Afterward, try to ssh to the local computer again, this time it may asks to add its current IP address to the known\_hosts list, like:

```

$ ssh <IP>
The authenticity of host '... (...)' can't be established.
RSA key fingerprint is .....
Are you sure you want to continue connecting (yes/no)?

```

This step happens only once when the ssh connection is built at the first time. The user can simply confirm the authentication by typing “yes”. If the user prefers to avoid this step, the following three lines can be added into the file \$HOME/.ssh/config.

```

Host *
StrictHostKeyChecking    no
UserKnownHostsfile       /dev/null

```

## 2.1 Installation Steps

A set of auxiliary tools, which practically are bash scripts, are provided to help the user to install and use Parallel SECONDO easily. These tools are kept in the Hadoop algebra of SECONDO 3.3.2, in a folder named clusterManagement. The installation of Parallel SECONDO on a single computer includes the following steps:

1. Install SECONDO. The installation guide of SECONDO for different platforms can be found on our website, and the user can install it as usual.<sup>1</sup> Afterward, the user can verify the correctness of the installation and compile SECONDO.

```

$ env | grep '^SECONDO'
SECONDO_CONFIG=... /secondo/bin/SecondoConfig.ini
SECONDO_BUILD_DIR=... /secondo
SECONDO_JAVA=... /java
SECONDO_PLATFORM=...

$ cd $SECONDO_BUILD_DIR
$ make

```

Particularly, in Ubuntu, the following line in the profile file \$HOME/.bashrc

```
source $HOME/.secondorc $HOME/secondo
```

should not be set at the end of the file. Instead, it should be set above the line:

```
[ -z "$PS1" ] && return
```

2. Download Hadoop. Go to the official website of Hadoop, and download the Hadoop distribution with the version of 0.20.2. The downloaded package should be put into the \$SECONDO\_BUILD\_DIR/bin directory without changing the name.

```

$ cd $SECONDO_BUILD_DIR/bin
$ wget http://archive.apache.org/dist/hadoop/core/
    hadoop-0.20.2/hadoop-0.20.2.tar.gz

```

3. A profile file named ParallelSecondoConfig.ini is prepared for setting all parameters in Parallel SECONDO. Its example file is kept in the clusterManagement folder of the Hadoop Algebra, which is basically made for a single computer with Ubuntu. However, few parameters still need to be set, according to the user’s computer.

---

<sup>1</sup>The version must be 3.3.2 or higher.

- The *Cluster*<sup>2</sup> parameter indicates the DSs in Parallel SECONDO. In a single computer, it can be set like:

```
Master = <IP>:<DS_Path>:<Port>
Slaves += <IP>:<DS_Path>:<Port>
```

The “IP” is the IP address of the current computer. The “DS\_Path” indicates the partition for a Data Server, for example /tmp. Note that the user must have the read and write access to the DS\_Path. The “Port” is also a port number prepared for a DS daemon, like 11234. Different DS can be set on the same computer, but their DS\_Paths and Ports must be different.

- Set *hadoop-env.sh:JAVA\_HOME* to the location where the JAVA SDK is installed,
- The user might already have installed SECONDO before, and created some private data in the database. If so, the *NS4Master* parameter can be set true, in order to let Parallel SECONDO visit the existing databases.

```
NS4Master = true
```

Note here if there is no SECONDO databases created before, and the *NS4Master* parameter is set to be true, then the below installation will fail.

- At last, the transaction feature is normally turned off in Parallel SECONDO, in order to improve the efficiency of exchanging data among DSs. For this purpose, the following RTFlag parameter in the SECONDO configuration file should be uncommented.

```
RTFlags += SMI:NoTransactions
```

The file is named *SecondoConfig.ini*, being kept in the *\$SECONDO\_BUILD\_DIR/bin*.

After setting all required parameters, copy the file *ParallelSecondoConfig.ini* to *\$SECONDO\_BUILD\_DIR/bin*, and start the installation with the auxiliary tool *ps-cluster-format*.

```
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ cp ParallelSecondoConfig.ini $SECONDO_BUILD_DIR/bin
$ ps-cluster-format
```

During this installation, all Data Servers will be created and the Hadoop is installed. Besides, the Namenode in Hadoop is formatted at last.

4. Close the shell and start a new one. Verify the correctness of the initialization with the following command:

```
$ cd $HOME
$ env | grep '^PARALLEL_SECONDO'
PARALLEL_SECONDO_MASTER=.../conf/master
PARALLEL_SECONDO_CONF=.../conf
PARALLEL_SECONDO_BUILD_DIR=.../secondo
PARALLEL_SECONDO_MINIDB_NAME=msec-databases
PARALLEL_SECONDO_MINI_NAME=msec
PARALLEL_SECONDO_PSFSNAME=PSFS
PARALLEL_SECONDO_DBCONFIG=.../SecondoConfig.ini....
PARALLEL_SECONDO_SLAVES=.../conf/slaves
PARALLEL_SECONDO_MAINDS=.../dataServer1/...
PARALLEL_SECONDO=.../dataServer1
PARALLEL_SECONDO_DATASERVER_NAME=...
```

---

<sup>2</sup>This parameter must be set before continue.

5. The third step initializes the DS in the computer, but the Mini-SECONDO system is not distributed into those Data Servers yet. This is because the Hadoop algebra cannot be compiled before setting up the environment of Parallel SECONDO. Now both Hadoop and HadoopParallel algebras should be activated, and SECONDO should be recompiled. The new algebras are activated by adding the following lines to the algebra list file `$$SECONDO_BUILD_DIR/makefile.algebras`.

```
ALGEBRA_DIRS += Hadoop
ALGEBRAS      += HadoopAlgebra

ALGEBRA_DIRS += HadoopParallel
ALGEBRAS      += HadoopParallelAlgebra
```

After re-compiling the SECONDO system, the user can distribute Mini-SECONDO to all local DSs with the auxiliary tool `ps-secondo-buildMini`.

```
$ cd $$SECONDO_BUILD_DIR
$ make
$ cd $$SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-secondo-buildMini -lo
```

So far, Parallel SECONDO has been installed. If the user wants to completely remove it from the computer or the cluster, an easy-to-use tool `ps-cluster-uninstall` is also provided.

```
$ cd $$SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-cluster-uninstall
```

Besides installing Parallel SECONDO on the user's own computer, a VMware image containing a single-computer Parallel SECONDO is also provided, with which the user can easily prepare the system by starting a virtual machine.

### 3 Start and Shutdown Parallel Secondo

In Parallel SECONDO, Hadoop and DSs are deployed independently, therefore they also are started and closed with separate steps. Hadoop is started by its own script `start-all.sh`, while the DSs are started by `ps-startMonitors`. This script starts all monitors, i.e. daemons of Mini-SECONDO databases distributed in DSs. Besides, the user can apply the `ps-cluster-queryMonitorStatus` to check the status of all involved Mini-SECONDO.

```
$ start-all.sh
$ cd $$SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-startMonitors
$ ps-cluster-queryMonitorStatus
$ ps-startTTYCS -s 1
```

At last, the user opens a text interface of Parallel SECONDO with the script `ps-startTTYCS`. This interface connects to the Mini-SECONDO database in the master DS, which is the first DS in the master node of the cluster. Here the Parallel SECONDO is installed on a single computer with one DS only, thus the `s` argument is set as one.

Besides opening the text interface, the user can also start up the graphical interface provided by SECONDO, and connect with the master DS by setting the IP address and the access port that we set in the `ParallelSecondoConfig.ini` file.

Parallel SECONDO can be shut down by separately stopping all Mini-SECONDO monitors and the Hadoop framework.

```
$ stop-all.sh
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-stopMonitors
```

## 4 A Simple Parallel Query Example

In order to make the user familiar with Parallel SECONDO as soon as possible, a simple example is offered first. Start the text interface of Parallel SECONDO, and then load an example database named opt:

```
restore database opt from opt;
```

In this example database, there prepared two simple relations: plz and Orte. The plz has two attributes: Ort (string) and PLZ (int), recording zip codes of some German cities. Orte describes some cities' further information, except the post code. Both relations use the string attribute Ort for city names. The following query performs a hash join operation on these two relations:

```
query plz feed {p} Orte feed {o} hashjoin[Ort_p, Ort_o] count;
```

```
10052
```

In the below of this section, the parallel data model in Parallel SECONDO and the method of changing a sequential query to a parallel query are introduced.

### 4.1 Parallel Data Model

In order to describe the parallel procedure of the above example, the Parallel data model is provided. It helps the user to write parallel queries like ordinary sequential queries. The data type *flist* encapsulates all SECONDO objects and distributes their content on the computer cluster. Besides, a set of parallel operators are implemented in Parallel SECONDO, making the system compatible with the single-computer database. These operators can be roughly divided into four kinds: flow operators, Hadoop operators, PSFS operators, and other assistant operators, as shown in Table 1.

Flow operators connect parallel operators with other existing SECONDO operators. Relations can be distributed over the cluster with flow operations, and then be processed by all DSs in parallel. Later the distributed result, which is represented as an *flist* object, can be collected from the cluster in order to be processed with single-computer SECONDO. Hadoop operators accept queries written in SECONDO executable language and create Hadoop jobs with their respective template Hadoop programs. Thereby the user can write parallel queries like common sequential queries. Assistant operators work together with Hadoop operators, helping to denote *flist* objects in parameter functions. At last, PSFS operators are particularly prepared for accessing data exchanged in PSFS, being internally used in Hadoop operations. Details about these operators can be found in the User Guide of Parallel SECONDO.

### 4.2 The Parallel Solution

With the Parallel data model, the example join query can be processed in parallel with following queries.

```
let CLUSTER_SIZE = 1;
let PS_SCALE = 6;

let plz_p = plz feed spread[; Ort, CLUSTER_SIZE, TRUE];
```

Type	Operator	Signature
Flow	spread	$\text{stream}(T) \rightarrow \underline{\text{flist}}(T)$
	collect	$\underline{\text{flist}}(T) \rightarrow \text{stream}(T)$
Hadoop	hadoopMap	$\underline{\text{flist}} \times \text{fun}(\text{mapQuery}) \times \text{bool} \rightarrow \underline{\text{flist}}$
	hadoopReduce	$\underline{\text{flist}} \times \text{fun}(\text{reduceQuery}) \rightarrow \underline{\text{flist}}$
	hadoopReduce2	$\underline{\text{flist}} \times \underline{\text{flist}} \times \text{fun}(\text{reduceQuery}) \rightarrow \underline{\text{flist}}$
Assistant	para	$\underline{\text{flist}}(T) \rightarrow T$
PSFS	fconsume	$\text{stream}(T) \times \text{fileMeta} \rightarrow \text{bool}$
	fdistribute	$\text{stream}(T) \times \text{fileMeta} \times \text{Key} \rightarrow \text{stream}(\text{key}, \text{tupNum})$
	ffeed	$\text{fileMeta} \rightarrow \text{stream}(T)$

Table 1: Operators Provided as Extensions In Parallel Secondo

```

let Orte_p = Orte feed spread[; Ort, CLUSTER_SIZE, TRUE];

query plz_p Orte_p hadoopReduce2[ Ort, Ort, DLF, PS_SCALE
; . {p} .. {o} hashjoin[Ort_p, Ort_o] ]
collect[] count;

10052

```

At the beginning the queries, two constants `CLUSTER_SIZE` and `PS_SCALE` are defined. The `CLUSTER_SIZE` tells the number of slave DS, hence here it is set as one. The constant `PS_SCALE` indicates the number of tasks running in parallel during the Hadoop procedure. It is set to six as there are six processor cores in the current computer.

Next, both *flist* objects `plz_p` and `Orte_p` are created with the `spread` operator. The `spread` distributes a relation into pieces on the cluster, based on an indicated attribute. Here the first argument in both queries is `Ort`, hence both relations are divided by the `Ort` attribute. The second argument indicates the scale of the partition. Usually it is set according to the number of the slave DSs, i.e. the `CLUSTER_SIZE`. The third argument is true, in order to keep the partition attribute in distributed data pieces.

At last, the hadoop operator `hadoopReduce2` is used to process the parallel query. Both relations are re-partitioned based on the `Ort` attribute, with the scale up to `PS_SCALE`. The result is set as a DLF *flist*, in order to be collected into the master DS with the `collect` operator. The argument function set in the `hadoopReduce2` defines the database query being executed in every reduce task. As you can see, it is the same as the sequential query.

The result of the parallel query is correct, but it takes a much longer time than the sequential query. This is normal since there exist constant overhead for carrying out a Hadoop job. Usually parallel procedures are prepared for i/o- and cpu- intensive queries dealing with a large-scale data set, like the one we will introduce in the next section.

## 5 A Spatial Query Example

In this section, a more complex spatial query is introduced and adapted for parallel processing. It studies the problem of dividing a massive amount of large regions into small pieces, based on a set of polylines that overlap them. For example, a forest is split into pieces by roads. Regarding this issue, an operator `findCycles` is provided in `SECONDO`, and used in the following query:

```

query Roads feed {r}
Natural feed filter[.Type contains 'forest']
itSpatialJoin[GeoData_r, GeoData, 4, 8]

```

```

    sortby[Osm_id]
    groupby[Osm_id; InterGraph:
      intersection( group feed projecttransformstream[GeoData_r]
        collect_line[TRUE], group feed extract[GeoData])
      union boundary(group feed extract[GeoData])]
    projectextendstream[Osm_id; Curves: findCycles(.InterGraph)]
count;

Total runtime ... Times (elapsed / cpu): 22:53min (1372.55sec)
/899.94sec = 1.52516

195327

```

Here two relations *Natural* and *Roads* are used. Both are public data sets fetched from the OpenStreetMap project, describing the geographic information of the German state North Rhine Westphalia (NRW). They can be downloaded from the website and then imported into the SECONDO database as relations with queries that will be shown at the end of this section.

The *Natural* relation contains many region shapes. They are classified into four kinds according to their actual usages, including forest, park, water and riverbank, indicated by the attribute **Type**. Shapes in the *Roads* relation are polylines, consisting of the street network in that area. Both relations use the geometric attribute **GeoData** to record their precise shape descriptions, and each shape is identified by the key attribute **Osm\_id**. When this document is written, the *Natural* relation contains 92,875 regions, from which 71,415 (77%) are forests. Its size is about 421 MB. At the same time, there are in total 1,153,587 roads listed in *Roads*, which is as large as 1.5 GB. Note that the data sets on OpenStreetMap are updated aperiodically, therefore the scale of these data sets and also the above query result may change along with the upgrade of the data sets.

In the example query, the forest regions in the relation *Natural* are split into pieces by the road network from the relation *Roads*. The query is written in SECONDO executable language, consisting of SECONDO objects and operators. It describes the data flow of the query, and the user can use different methods and operators for the same query.

This example query is mainly divided into three steps. At first, forest regions in the relation *Natural* are joined with the relation *Roads*, by using the **itSpatialJoin** operator. The **itSpatialJoin** is the most efficient sequential spatial join operation in the current SECONDO. It builds up a memory R-tree on one side relation, and probes this R-tree with the tuples from the other relation. Pair-tuples which have overlapped bounding boxes are output as the join result, which then be grouped based on forest regions' identifiers. For each group, a graph is built up consisting of all boundary edges of the region and all its intersecting roads. For example, as shown in Figure 2, the region *R* is divided into five pieces by three intersected lines *a*, *b* and *c*, and the created graph is shown in Figure 2b.

First, all roads intersecting the region are put together into one *line* value by the **collect\_line** operator. The **intersection** operator then computes all parts of this line (road lines) lying within the forest region, like P1P2, P3P4, P7P8, etc. The **boundary** operator produces all boundary edges of the region, like P5P6. Boundary edges and roads inside the region are put together by the **union** operation.

At last, the **findCycles** operator traverses each graph (represented as a *line* value) and returns split regions, like the red cycle shown in the Figure 2b. It is used as an argument function in the operator **projectextendstream**, thereby each forest region's identifier is duplicated for all its split regions. This sequential query produces 195,327 split regions and costs about 1373 seconds.

## 5.1 Import Spatial Relations

Both the *Natural* and *Roads* relations are imported from the OpenStreetMap project with the following steps.



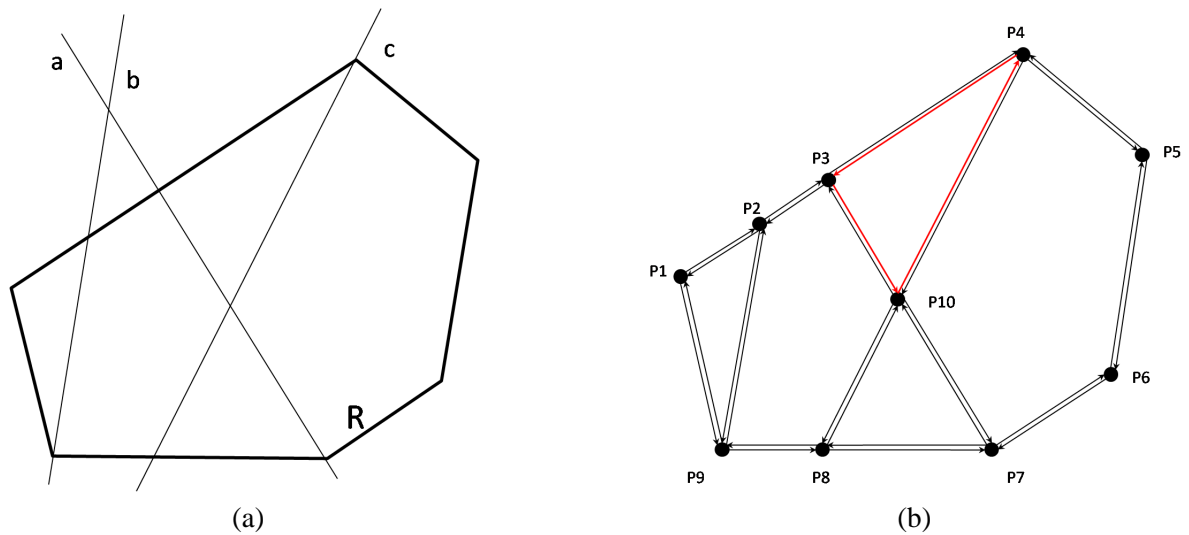


Figure 2: findCycles Operation

```
$ cd $SECONDO_BUILD_DIR/bin
$ wget http://download.geofabrik.de/europe/germany/
  nordrhein-westfalen-latest.shp.zip
$ mkdir nrw
$ unzip -d nrw nordrhein-westfalen.shp.zip
```

By now, all shape files about streets and regions in NRW are kept in the directory named *nrw* in the *\$SECONDO\_BUILD\_DIR/bin*. The user can then use the following *SECONDO* queries to load them into the database, and the elapsed time of partial queries are listed below.

```
create database nrw;
open database nrw;
```

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

```
Total runtime ... Times (elapsed / cpu): 2:08min (128.28sec)
/90.49sec = 1.41761
```

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

```
Total runtime ... Times (elapsed / cpu): 39.1086sec / 30.38sec = 1.28731
```

```
close database;
```

Here we use the relative paths to indicate the data and shape files that the **dbimport2** and **shpimport2** operators need. In case they cannot find these files for various reasons, the user can replace these relative paths with their absolute paths.

## 5.2 The Parallel Solution

The above sequential spatial query can be processed in parallel by adopting the idea of semi-join, consisting of the queries shown in the following:

```

open database nrw;

let CLUSTER_SIZE = 1;
let PS_SCALE = 6;

let JoinedRoads_List = Roads feed
  Natural feed filter[.Type contains 'forest'] {r}
  itSpatialJoin[GeoData, GeoData_r, 4, 8]
  projectextend[Osm_id, GeoData; OverRegion: .Osm_id_r]
  spread[; OverRegion, CLUSTER_SIZE, TRUE;]

Total runtime ... Times (elapsed / cpu): 9:03min (542.65sec)
/82.81sec = 6.55295

let Forest_List = Natural feed filter[.Type contains 'forest']
  spread[; Osm_id, CLUSTER_SIZE, TRUE;]

Total runtime ... Times (elapsed / cpu): 10.8835sec
/ 3.62sec = 3.00649

query JoinedRoads_List Forest_List
  hadoopReduce2[OverRegion, Osm_id, DLF, PS_SCALE
  ; . {r} .. hashjoin[OverRegion_r, Osm_id]
  sortby[Osm_id] groupby[Osm_id
  ; InterGraph: intersection( group feed projecttransformstream[GeoData_r]
  collect_line[TRUE], group feed extract[GeoData])
  union boundary(group feed extract[GeoData])]
  projectextendstream[Osm_id; Curves: findCycles(.InterGraph)]
collect[] count;

Total runtime ... Times (elapsed / cpu): 7:01min (420.704sec)
/4.31sec = 97.6111

195327

close database;

```

At first, the constants `CLUSTER_SIZE` and `PS_SCALE` should be set in the `nrw` database. The first sequential query generates all roads that intersect with forest regions, with the `itSpatialJoin` operator. In the mean time, if a road passes several forests, it is duplicated for every intersection. At last, each road tuple is only extended with the identifiers of the forest regions that it overlaps, by using the `projectextend` operator.

The parallel query is mainly built up with the `hadoopReduce2` operator. It first distributes forest regions and all their joined roads based on forest identifiers, hence in each reduce task, forest-road pairs can be created with a `hashjoin` operation. Thereafter, forest regions are grouped with all roads that it intersects by the `groupby`

operator, and the following query is just the same as the sequential solution. In the end, the distributed results are collected back to the standard `SECONDO` by using the flow operator `collect`.

This complete solution costs 975 seconds in total, and gains about 1.4 times speed-up in a single computer.

## A Alternate Parallel Approaches

Till now, queries in Parallel `SECONDO` are represented in `SECONDO` executable query language, with which the user can choose different combinations of operators to process the same query.

At the same time, in total there are three Hadoop operators provided in Parallel `SECONDO`, as shown in the Table 1, describing parallel queries processed in the Map and Reduce stages of Hadoop jobs, respectively. The `hadoopReduce2` operator demonstrated above processes the query, which is described as an argument function, in the Reduce stage. The `hadoopReduce` accepts only one input *flist* object, but also process the query in the Reduce stage. On the contrary, the `hadoopMap` processes its argument function by Map tasks only. With these Hadoop operators, there are two alternate parallel methods also can process the example spatial query that we presented above. Although they are not as efficient as the approach that we introduced, it indicates that Parallel `SECONDO` can flexibly process the user's various custom problems with different parallel solutions.

Since the two alternate solutions are not efficient enough being processed on a single computer, we would like to introduce and evaluate them in a computer cluster. This cluster contains six computers like the one that we used before, but each computer has two hard drives, thus we can set up two Data Servers on every computer. Hereby, the `CLUSTER_SIZE` is set as twelve, and the `PS_SCALE` is thirty six since we have so many cores in the whole cluster. In the mean time, we evaluate the former parallel solution on the cluster as well.

### A.1 Install Parallel Secondo on a Cluster

Installing Parallel `SECONDO` on a cluster is almost the same as the installation on a single computer. The auxiliary tools help the user to set up and use the system easily, with only the configure file need to be adjusted.

#### Prerequisites

Before the installation, there are some basic environment that the cluster should prepare. First, all computers in the cluster should have a same operating system, or at least all of them should be Linux or Mac OSX. Second, a same account needs to be created on all machines. This can be easily done with services like NIS. Thereafter, the SSH and screen services should be prepared on all computers, and the computers can visit each other through SSH without using the passphrase. Third, one or several disk partitions should be prepared on every machine, and the account prepared before should have read and write access on them. At last, all required libraries of `SECONDO` and the `.secondorc` file should be installed on every computer. Likewise, if the operating system is Ubuntu, then the line

```
source $HOME/.secondorc $HOME/secondo
```

should be set at the top of the profile file `$HOME/.bashrc`, before

```
[ -z "$PS1" ] && return
```

#### Installation Steps

One computer of the cluster must be indicated as the master node, and the complete installation is done on this machine only, with the following steps:

1. Enter the master node, download the latest SECONDO distribution, which must be newer than 3.3.2. Then extract the SECONDO archive to the master's \$SECONDO\_BUILD\_DIR, and compile it. Afterward, download the required Hadoop archive to \$SECONDO\_BUILD\_DIR/bin.

```
$ tar -xzf secondo-v33*.tar.gz
$ cd $SECONDO_BUILD_DIR
$ make
$ cd $SECONDO_BUILD_DIR/bin
$ wget http://archive.apache.org/dist/hadoop/core/
      hadoop-0.20.2/hadoop-0.20.2.tar.gz
```

2. Prepare the ParallelSecondoConfig.ini file, according to the environment of the cluster. For example, the *cluster* can be set as:

```
Master = 192.168.1.1:/disk1/dataServer1:11234
Slaves += 192.168.1.1:/disk1/dataServer1:11234
Slaves += 192.168.1.2:/disk1/dataServer1:11234
Slaves += 192.168.1.1:/disk2/dataServer2:14321
Slaves += 192.168.1.2:/disk2/dataServer2:14321
```

Here a cluster with two computers and four DSs are described. The two computers are set with the IP address of 192.168.1.1 and 192.168.1.2, respectively. Since every computer has two disks, each is set with two DSs. The computer 192.168.1.1 is set to be the master node of the cluster, and its first DS with the port of 11234 is set to be the master and the slave DS at the same time. Besides, the NS4Master can also be set as true, if the user want to visit the existing SECONDO database on the master node.

Particularly, the following five parameters in the parallel configure file should be changed, by replacing the localhost with the master node's IP address.

```
core-site.xml:fs.default.name = hdfs://192.168.1.1:49000
hdfs-site.xml:dfs.http.address = 192.168.1.1:50070
hdfs-site.xml:dfs.secondary.http.address = 192.168.1.1:50090
mapred-site.xml:mapred.job.tracker = 192.168.1.1:49001
mapred-site.xml:mapred.job.tracker.http.address = 192.168.1.1:50030
```

Additionally, the transaction feature should also be disabled by uncommenting the line in the file \$SECONDO\_BUILD\_DIR/bin/SecondoConfig.ini on the master node.

```
RTFlags += SMI:NoTransactions
```

At last, run the ps-cluster-format script.

```
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ cp ParallelSecondoConfig.ini $SECONDO_BUILD_DIR/bin
$ ps-cluster-format
```

3. Activate the Hadoop and HadoopParallel algebras into the file makefile.algebras, recompile SECONDO, and distribute Mini-SECONDO to all DSs at last.

```
$ cd $SECONDO_BUILD_DIR
$ make
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-secondo-buildMini -co
```

The parameter *c* in the ps-secondo-buildMini indicates the whole cluster.

Till now, Parallel SECONDO is completely installed. Accordingly, the user can also easily remove the system with the `ps-cluster-uninstall` script. The user can start all SECONDO monitors with the script `ps-start-AllMonitors`, and close them with `ps-stop-AllMonitors`. The text interface of the whole Parallel SECONDO is still opened by connecting to the master DS. All these steps should, and only need to be done on the master node.

```
$ start-all.sh
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-start-AllMonitors
$ ps-cluster-queryMonitorStatus
$ ps-startTTYCS -s 1
```

Sometimes, it is quite difficult for the user to build up a physical cluster. Regarding this issue, a Amazon AMI of Parallel SECONDO is provided. With this image, the user can quickly set up a virtual cluster on the Amazon Web Services (AWS), and the Parallel SECONDO is already built inside it, hence he/she can test the system directly.

## A.2 The Former Method

The former method introduced in Section 5.2 can of course be evaluated again in the cluster. However, on a single computer, its first query is done sequentially, taking a considerable overhead. Since here the method is carried out on the cluster, this spatial join should also be processed in a parallel way. The parallel spatial join adopts the method named PBSM (Partition-Based Spatial Merge). It first partitions both roads and forest regions into a coarsely divided cell-grid, and then processes the join operation independently within each cell. At the same time, the operator `gridintersects` removes duplicate join results produced in different cells. The generation of the cell-grid is demonstrated in the Section B, and must be carried out before the following queries.

```
open database nrw;

let Roads_BCell_List = Roads feed extend[Box: bbox(.GeoData)]
  projectextendstream[Osm_id, GeoData, Box
    ; Cell: cellnumber(.Box, Big_RNCellGrid)]
  spread[; Cell, CLUSTER_SIZE, TRUE;];

Total runtime ... Times (elapsed / cpu): 1:24min (83.5428sec)
/32.63sec = 2.56031

let Forest_BCell_List = Natural feed filter[.Type contains 'forest']
  extend[Box: bbox(.GeoData)]
  projectextendstream[Osm_id, GeoData, Box
    ; Cell: cellnumber(.Box, Big_RNCellGrid)]
  spread[;Cell, CLUSTER_SIZE, TRUE;];

Total runtime ... Times (elapsed / cpu): 18.1649sec
/ 4.03sec = 4.50742

let JoinedRoads_List = Roads_BCell_List Forest_BCell_List
  hadoopReduce2[Cell, Cell, DLF, PS_SCALE
    ; . . {r} itSpatialJoin[GeoData, GeoData_r, 4, 8]
    filter[ (.Cell = .Cell_r) and
      gridintersects(Big_RNCellGrid, .Box, .Box_r, .Cell)]
    projectextend[Osm_id, GeoData; OverRegion: .Osm_id_r ]];

Total runtime ... Times (elapsed / cpu): 1:14min (73.7165sec)
/0.1sec = 737.165
```

```

let Forest_List = Natural feed filter[.Type contains 'forest']
  spread[; Osm_id, CLUSTER_SIZE, TRUE;]

Total runtime ... Times (elapsed / cpu): 20.6345sec
/ 4.47sec = 4.61623

query JoinedRoads_List Forest_List
  hadoopReduce2[OverRegion, Osm_id, DLF, PS_SCALE
    ; . {r} .. hashjoin[OverRegion_r, Osm_id]
    sortby[Osm_id] groupby[Osm_id; InterGraph:
      intersection( group feed projecttransformstream[GeoData_r]
        collect_line[TRUE], group feed extract[GeoData])
      union boundary(group feed extract[GeoData])]
    projectextendstream[Osm_id; Curves: findCycles(.InterGraph)]
  collect[] count;

Total runtime ... Times (elapsed / cpu): 1:45min (105.169sec)
/13sec = 8.0899

195327

```

The top three queries achieve the parallel spatial join with PBSM, and generate all roads that intersect with forest regions. The other two queries are the same as we used before. This time, the whole processing costs 302 seconds in total, earning a speed-up of nearly factor five. It turns out that this solution fits the example spatial problem well, both on a single computer and on a cluster.

### A.3 Method 1

This solution divides the example query into two stages, Map and Reduce, following the programming paradigm of Hadoop. The spatial join operation is first finished in the Map stage, again with the PBSM method, then the join results are distributed over the cluster based on forest identifiers, and the remaining steps are processed on all slaves simultaneously in the Reduce stage. The queries are listed below.

```

let Roads_BCell_List = Roads feed extend[Box: bbox(.GeoData)]
  projectextendstream[Osm_id, GeoData, Box
    ; Cell: cellnumber(.Box, Big_RNCellGrid)]
  spread[; Cell, CLUSTER_SIZE, TRUE;];

Total runtime ... Times (elapsed / cpu): 1:24min (83.5428sec)
/32.63sec = 2.56031

let Forest_BCell_List = Natural feed filter[.Type contains 'forest']
  extend[Box: bbox(.GeoData)]
  projectextendstream[Osm_id, GeoData, Box
    ; Cell: cellnumber(.Box, Big_RNCellGrid)]
  spread[;Cell, CLUSTER_SIZE, TRUE;];

Total runtime ... Times (elapsed / cpu): 18.1649sec
/ 4.03sec = 4.50742

query Roads_BCell_List
  hadoopMap[DLF, FALSE
    ; . {r} para(Forest_BCell_List)
    itSpatialJoin[GeoData_r, GeoData, 4, 8]

```

```

    filter[ (.Cell = .Cell_r) and
            gridintersects(Big_RNCellGrid, .Box_r, .Box, .Cell_r)]
hadoopReduce[Osm_id, DLF, PS_SCALE
; . sortby[Osm_id]
  groupby[Osm_id; InterGraph:
    intersection( group feed projecttransformstream[GeoData_r]
                  collect_line[TRUE], group feed extract[GeoData])
    union boundary(group feed extract[GeoData])]
  projectextendstream[Osm_id; Curves: findCycles(.InterGraph)]
collect[] count;

```

```

Total runtime ... Times (elapsed / cpu): 12:56min (775.609sec)
/15.86sec = 48.9034

```

```
193527
```

Here the **hadoopMap** and **hadoopReduce** operators are used together in one parallel query, and each one describes one stage of the Hadoop job. In principle, one Hadoop operator starts a Hadoop job, and processes its argument function in one of the two stages. Nevertheless, an argument named *executed* is provided in the **hadoopMap** operator. If this argument is set false, then the Hadoop job does not start, and the argument function will be put off to the next Hadoop job represented with a **hadoopReduce** or **hadoopReduce2** operator. Therefore here both operations are actually processed with one Hadoop job in the end.

The **hadoopMap** is a unary operation, but the **itSpatialJoin** operator requires two inputs. Regarding this issue, the assistant operator **para** is used to identify the distributed relation *Forest\_BCell\_List*. The spatial join results produced in the Map stage are then shuffled based on the forest regions' identifiers. We can see that the argument function embedded in the **hadoopReduce** query is exactly the same as in the sequential query.

This approach is not efficient for the example problem. It costs in total 878 seconds, and the speed-up factor is about 1.6, although it is processed with six machines. This is mainly caused by the spatial join result, which are generated and accessed mainly in the memory during the sequential procedure. However in the parallel query, they are generated between the Map and Reduce stage, and have to be materialized and shuffled as disk files. Since the join result is extremely large, the overhead of exporting and loading them degrades the performance.

#### A.4 Method 2

This solution intends to finish both the spatial join and the **findCycles** operation in the Map stage, in order to avoid the shuffle procedure. The parallel spatial join is again processed with the PBSM method.

However, during the experiment, a new problem arises. When a region overlaps several cells, not all its intersected lines overlap the same cells. For example, as shown in Figure 3, the region is partitioned into two cells C1 and C2. According to PBSM, it is duplicated in both cells. The lines P3P7 and P4P8 are also partitioned to C1 and C2. However, the line P2P9 is only assigned to C1. Therefore, in the cell C2, because of the absence of P2P9, the wrong region P1-P3-P10-P8-P9 is produced. This problem is called Over-Cell Problem.

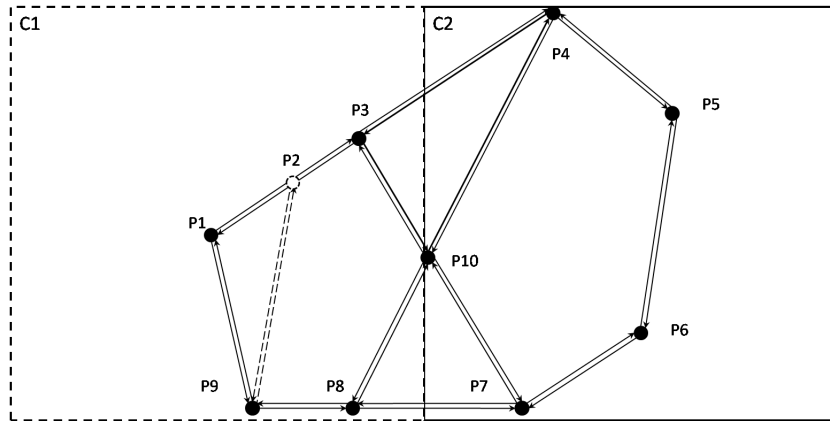


Figure 3: Over-Cell Problem

```
let Roads_BCell_List = Roads feed extend[Box: bbox(.GeoData)]
  projectextendstream[Osm_id, GeoData, Box
    ; Cell: cellnumber(.Box, Big_RNCellGrid)]
  spread[; Cell, CLUSTER_SIZE, TRUE;];
```

Total runtime ... Times (elapsed / cpu): 1:24min (83.5428sec)  
/32.63sec = 2.56031

```
let OneCellForest_BCell_OBJ_List =
  Natural feed filter[.Type contains 'forest']
  extend[Box: bbox(.GeoData)]
  filter[cellnumber( .Box, Big_RNCellGrid) count = 1]
  projectextendstream[Osm_id, GeoData, Box
    ; Cell: cellnumber(.Box, Big_RNCellGrid)]
  spread[; Cell, CLUSTER_SIZE, TRUE;]
  hadoopMap[; . consume];
```

Total runtime ... Times (elapsed / cpu): 1:16min (76.4691sec)  
/5.61sec = 13.6308

```
query Roads_BCell_List
  hadoopMap[DLF ; . {r} para(OneCellForest_BCell_OBJ_List) feed
    itSpatialJoin[GeoData_r, GeoData, 4, 8]
    filter[gridintersects(Big_RNCellGrid, .Box_r, .Box, .Cell_r)]
    sortby[Osm_id]
    groupby[Osm_id; InterGraph:
      intersection( group feed projecttransformstream[GeoData_r]
        collect_line[TRUE], group feed extract[GeoData])
      union boundary(group feed extract[GeoData])]
    projectextendstream[Osm_id; Curves: findCycles(.InterGraph)]
    collect[] count;
```

Total runtime ... Times (elapsed / cpu): 2:44min (163.782sec)  
/7.16sec = 22.8746

169874

```
let MulCellForest = Natural feed filter[.Type contains 'forest']
  filter[cellnumber( bbox(.GeoData), Big_RNCellGrid) count > 1]
  consume;
```



```
Total runtime ... Times (elapsed / cpu): 3.26407sec
/ 1.47sec = 2.22046
```

```
query Roads feed {r} MulCellForest feed
  itSpatialJoin[GeoData_r, GeoData, 4, 8]
  sortBy[Osm_id] groupby[Osm_id; InterGraph:
    intersection( group feed projecttransformstream[GeoData_r]
      collect_line[TRUE], group feed extract[GeoData])
    union boundary(group feed extract[GeoData])]
  projectextendstream[Osm_id; Curves: findCycles(.InterGraph)]
count
```

```
Total runtime ... Times (elapsed / cpu): 5:42min (341.884sec)
/170.5sec = 2.00519
```

```
25453
```

Regarding the Over-Cell problem, the forest regions are divided into two types, those covering only one cell of the grid and those covering multiple cells. For the first type, we process them in Parallel `SECONDO`, while the second type is still processed in the sequential query.

In particular, a `hadoopMap` operation is used to load all forest regions that cover only one cell into distributed `SECONDO` databases, creating a DLO type *flist* object at last. This is because such an object is used by the assistant operator `para` in the parallel query, each task can use part of its value only when it is a DLO *flist*.

This solution costs in total 668 seconds, achieving a speed-up factor of two. Although the result is correct, the efficiency is not satisfying. At the same time, because of the Over-Cell problem, the solution has to be divided to two steps, making the user not so easy to understand.

## B Assistant Objects

Here the cell-grid used in the above PBSM method is created, named `Big_RNCellGrid`. The cell number is set to the number of DSs, i.e., `CLUSTER_SIZE`. This grid is coarsely divided, since the efficient `itSpatialJoin` operation is executed in every cell.

At present, the two relations have to be completely scanned so as to obtain the bounding box of the grid. The grid is created by an operator named `createCellGrid2D`, requiring five arguments. The first two arguments indicate the bottom-left point of the grid, while the other three define the size of cell edges and the cell number on the X-axis.

```
let CLUSTER_SIZE = 12;

let PS_SCALE = 36;

let RoadsMBR = Roads feed
  extend[Box: bbox(.GeoData)]
  aggregateB[Box; fun(r:rect, s:rect) r union s
    ; [const rect value undef]];

let NaturalMBR = Natural feed
  extend[Box: bbox(.GeoData)]
  aggregateB[Box; fun(r:rect, s:rect) r union s
    ; [const rect value undef]];
```

```
let RN_MBR = intersection(RoadsMBR, NaturalMBR);
let BCellNum = CLUSTER_SIZE;
let BCellSize = (maxD(RN_MBR,1) - minD(RN_MBR,1)) / BCellNum;

let Big_RNCellGrid =
  createCellGrid2D(minD(RN_MBR,1), minD(RN_MBR,2),
    BCellSize, BCellSize, BCellNum);
```