User Guide For Parallel Secondo

Jiamin Lu, Ralf Hartmut Güting
Databases for New Applications, Fernuniversität Hagen, Germany

June 3, 2013

1 Parallel Secondo Infrastructure

Parallel SECONDO is constructed by coupling the Hadoop framework and discrete SECONDO databases on a set of computers connected with network, as shown in Figure 1. It can be deployed on either a single computer or a cluster containing tens or even hundreds of computers. Its two components Hadoop and discrete SECONDO databases coexist in the same system, and each can be used independently. In Hadoop, nodes communicate through HDFS (Hadoop Distributed File System), whereas each single-node SECONDO exchanges its data with the other SECONDO instances through the PSFS (Parallel SECONDO File System), which is a distributed file system perticularly prepared for Parallel SECONDO.

Whereas Hadoop is deployed by nodes, Parallel SECONDO is deployed by Data Servers. A Data Server is the minimum execution unit of the system, containing a compact SECONDO named Mini-SECONDO and its database, together with a PSFS node. It is possible that one cluster node contains several Data Servers. This is especially useful for nodes with multiple hard disks, since then the user can set a Data Server on each disk, in order to take full advantage of the cluster resources. When processing various parallel queries, a small amount of data is exchanged among nodes through the HDFS, in order to assign tasks to Data Servers. At the same time, most intermediate data are exchanged among Data Servers through the PSFS.

A particular Data Server set on the master node is called the *master Data Server*; its Mini-SECONDO is the only entrance to the system, called the *master database*. The other databases that are deployed on slave Data Servers are called *slave databases*. Through various PQC (*Parallel Query Converter*) operators, which are also called Hadoop operators, a parallel query submitted in the *master database* is converted to a Hadoop job, and then partitioned to tasks. These tasks are processed by Data Servers in parallel, as scheduled by the Hadoop framework. Different from the slave databases, the master database contains some global and meta data of the whole system; therefore it is also called the meta database as well.

An identical DS-Catalog (*Data Server Catalog*) is duplicated on every node of the cluster, describing access entries for all Data Servers. It is composed by two files, *master* and *slaves*, listing Data Servers by lines with three elements:

```
IP_Address:PSFS_Location:SecondoPort
```

For each line, its first element distinguishes nodes by their IP addresses, whereas the second and the third elements tell apart Data Servers within a same computer based on their PSFS node locations and Mini-SECONDO ports. The master file should contain only one Data Server. It is possible to use the master also as a slave Data Server. In a node with several Data Servers, the Hadoop applications and the DS-Catalog are only set in its first Data Server. The order of these Data Servers is decided by the DS-Catalog, and the master Data Server is always the first one on the master node.

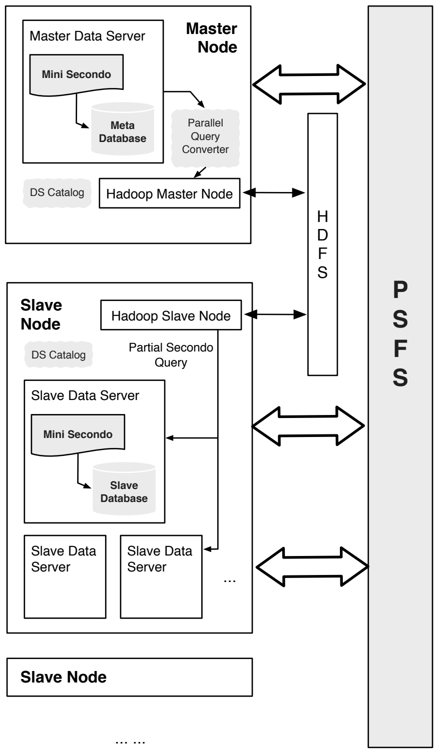

Figure 1: The Infrastructure of Parallel Secondo

2 Hadoop Algebra

SECONDO is composed by algebras, each containing a set of data types and relative operators. Two algebras are especially required by the Parallel SECONDO, Hadoop and HadoopParallel. Besides, some other components are also required:

1. Hadoop Archive. Hadoop archive with the version 0.20.2 is used as the underlying framework for Parallel SECONDO. It is not included in the SECONDO release by default, and has to be downloaded to the directory $SECONDO_BUILD_DIR/bin by the user. However, its installation is performed automatically later along with the deployment of the Parallel SECONDO.

2. Parallel SECONDO Auxiliary Utilities. A set of bash scripts is provided to manage the system. They are kept in the Hadoop algebra by default, in a folder named *clusterManagement*. These scripts include at least:

   ps-cluster-format: It is an automatic install script for initializing the Parallel SECONDO environment of the cluster, including the Hadoop framework.

   ps-cluster-uninstall: It performs the opposite function to the script above, removing the Parallel SECONDO from the cluster, and cleaning up the environment.

ps-secondo-buildMini: It extracts and distributes the Mini-SECONDO based on the local SECONDO system, to all Data Servers that are listed in the DS-Catalog.

ps-startMonitors: The SECONDO monitor is a database server process prepared to accept multiple remote clients visiting the same SECONDO database. In Parallel SECONDO, all SECONDO monitors have to be started up before processing any parallel queries. Considering there may exist several Data Servers in one cluster node, this script helps the user to start up monitors on the current node.

ps-start-AllMonitors: It starts up all Data Servers' SECONDO monitors in the cluster.

ps-stopMonitors: It shuts down Data Servers' SECONDO monitors on the current node.

ps-stop-AllMonitors: It shuts down all Data Servers' SECONDO monitors in the cluster.

ps-startTTY: It starts up a SECONDO text terminal interface of one Mini-SECONDO on the current node.

ps-startTTYCS: It opens a SECONDO Client-Server based text terminal interface, and connects to a started SECONDO monitors in the cluster.

ps-cluster-queryMonitorStatus: It checks the status of all SECONDO monitors in the cluster.

3. Parallel Configuration File. All Parallel SECONDO parameters are set in a file named ParallelSecondoConfigure.ini. Its example file is also kept in the Hadoop algebra, together with the above auxiliary utilities.

4. Template Hadoop Job. Several generic Hadoop jobs are prepared for the PQC operators. Their source files are kept in the Hadoop algebra too, and the jobs are generated when the algebra is being compiled.

3 Configuring Parallel Secondo

All Parallel SECONDO related parameters are set in a file named *ParallelSecondoConfig.ini*. Its example file is prepared to deploy the Parallel SECONDO on a single computer, and kept in the Hadoop algebra by default. After setting all required parameters according to the user's own environment, this file should be copied to the $SECONDO_BUILD_DIR/bin, and then read by the ps-cluster-format script.

This file follows the same format as the SecondoConfig.ini file, mainly divided into three sections: hadoop, cluster and options. The first **hadoop** section is prepared for setting up the Hadoop framework. All parameters listed in this section keep the Hadoop configuration on all involved nodes the same. Although it restricts the flexibility of the system, it helps new users to set up Parallel SECONDO quickly without learning many details about Hadoop. All parameters are listed by lines, composed of three elements including the file name, title and value. Each value is prepared for a parameter with the specific title in a particular file.

```
[fileName]:[title] = [value]
```

This section is further divided into four parts. The first contains all indispensable parameters prepared for Parallel SECONDO. Non-advanced users should keep this part unchanged, or else the Parallel SECONDO may not work correctly. The second part sets IP addresses and ports for different Hadoop daemons. They are also indispensable, although the user should change their values based on his own cluster, like the master node's IP address. Besides, daemons' port numbers can also be changed if the default values have already been taken by some other programs. The third part indicates the capability of the cluster, and the user should also set them based on his own cluster. For example, the option *mapred-site.xml:mapred.tasktracker.map.tasks.maximum* limits the number of map tasks running in parallel on one node. Usually we set this value by doubling the number of the processor cores contained in the computer, so does the other option *mapred.tasktracker.reduce.tasks.maximum*. There are also some other parameters like the *hdfs-site.xml:dfs.replication*, telling how many times each HDFS block is replicated on the cluster. If the Parallel SECONDO is deployed into a cluster composed of hundreds of elastic computers, then it is better to set this parameter with an integer more than 1. The last part is prepared for clusters shared by multiple users, where each user should set their daemons with different port numbers. In this case, the port numbers listed in the second part should also be adjusted.

The **cluster** section lists all involved Data Servers. Each Data Server is indicated with one line, and assigned as the master or a slave by the title. The value part contains three elements: IP Address, Data Server Path and Mini-SECONDO Port. The second element indicates a disk path where all Data Server components are kept. This path is created automatically if it does not exist before, but the user should guarantee that he/she has been granted the write access to this path. Take a cluster configuration for example:

```
Master  = 192.168.0.1:/Home/dataServer1:11234
Slaves += 192.168.0.1:/Home/dataServer1:11234
Slaves += 192.168.0.1:/Home/dataServer2:12234
```

Here a small cluster is simulated on one computer with the IP address 192.168.0.1. It contains two Data Servers, which are kept in the /Home/dataServer1 and the /Home/dataServer2, respectively. The first Data Server is used as the master and a slave at the same time. The master Mini-SECONDO can be accessed through the port 11234.

The last section **options** is prepared for some special settings in the cluster. At present, only one option named NS4Master (Normal Secondo For Master database) is provided here. If its value is set as true, then the master node's default SECONDO, i.e. where the $SECONDO_BUILD_DIR points to, is set to be the *master database*.

4 Mini-Secondo Management

In Parallel SECONDO, one node may contain multiple SECONDO databases, and a cluster may be composed by many computers. Therefore, some regular routine work like starting, stopping, and updating SECONDO systems are better processed with some auxiliary bash scripts. These auxiliary utilities are briefly introduced in Section 2, and their names are all started by "ps-". Most of the scripts support the usage explanation, which can be printed out with the "-h" argument. Here we only introduce several of them about managing the Mini-SECONDO of the system.

common operations in Parallel SECONDO accomplished with these scripts.

4.1 Update Mini-Secondo

The Mini-SECONDO is a compact SECONDO distribution, only containing essential components required to handle its database. All of them are built based on the SECONDO installed on the master node, keeping identical over the whole cluster. In case there will be any extension made for SECONDO, like creating new data types or operators, the user can update all Mini-Secondo in the cluster immediately.

```
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-stop-AllMonitors
$ cd $SECONDO_BUILD_DIR/
$ make
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-secondo-buildMini -c
```

The ps-secondo-buildMini provides two major arguments: c (cluster) and l (local). If the "-c" argument is set, then the update will be distributed to every Data Server of the cluster. In contrast, if the "-l" parameter is set, then the new SECONDO is only distributed to all Data Servers on the current node.

All essential components that a Mini-SECONDO needs are listed in a text file named miniSec_list, which is kept together with the script ps-secondo-buildMini, describing files and folders by lines. It is possible for the user to change this list according to his own requirement.

4.2 Start Up and Turn Off Mini Secondo Monitors

During parallel procedures, SECONDO databases are accessed through their monitors, which should be all started up before processing any queries. Considering there are tens or even hundreds of Mini-SECONDO systems inside a cluster, several utilities are proposed to start and stop these monitors without visiting them one after another.

The ps-startMonitors starts up all Mini-SECONDO monitors on the current computer, while ps-start-AllMonitors visits every node and runs the ps-startMonitors, so as to start up all monitors on the cluster. In contrast, the ps-stopMonitors turns off all Mini-SECONDO monitors on the current computer, and all monitors on the cluster are shut down by executing the ps-stop-AllMonitors.

4.3 Open Parallel Secondo Interface

In Parallel SECONDO, every Mini-SECONDO can be viewed as a normal SECONDO system and visited independently. Normally the user only needs to visit the master database, although scripts like ps-startTTYCS are capable to access any Mini-SECONDO in the cluster. One cluster node may contain several Mini-SECONDO databases, which are arranged according to the DS-Catalog. The commands for opening the Parallel SECONDO main text interface are:

```
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-startMonitors
$ ps-startTTYCS -s 1
```

All Mini-SECONDO monitors must be started up before running any parallel queries, hence the meta database can only be visited through its client-server interface. Since the master Mini-SECONDO is always the first database on the master node, we start up this specific interface by setting the argument *s* with the value 1. Besides, it is also possible to use the graphical user interface in Secondo to visit the meta database. The user can open the Javagui interface as usual, then connect to the indicated database by setting its host IP and port number in the menu "Server", and "Settings".

5 Secondo Parallel Query Expression

Here we introduce how to write parallel queries in SECONDO executable language. The SECONDO executable language describes query plans by composing database objects and operators in certain oder. It

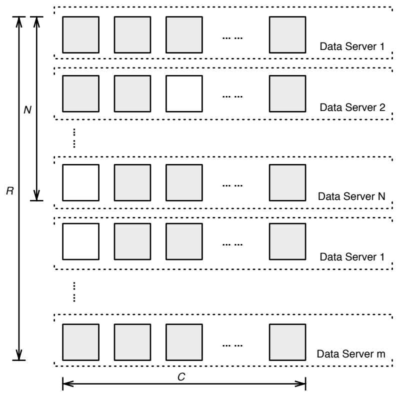

Figure 2: PS-Matrix

is of course more complicated than writing SQL queries, but far more easier than programming Hadoop programs.

5.1 PS-Matrix

In Parallel SECONDO, data is distributed over the cluster as a PS-Matrix, shown in Figure 2. A Secondo object is divided into pieces based on two functions, $d(x)$ and $d(y)$. The $d(x)$ divides the data into R rows; each row should be completely kept on one Data Server. It is possible for $d(x)$ to produce more rows than the cluster scale N, and keep multiple rows on one Data Server. Afterwards, $d(y)$ further divides every row to C columns. As a result, a PS-Matrix is composed by $R \times C$ pieces, but not all pieces in the matrix must contain data. Normally, a PS-Matrix is prepared for distributing large-sized data over the cluster, like relations containing millions of tuples, while the division functions are hash algorithms based on one of its attributes.

5.2 Data Type

In Parallel SECONDO, data type <u>*flist*</u> is especially introduced for representing the PS-Matrix. It is designed as a wrap structure, and able to encapsulate all available SECONDO objects, shown in Table 1. After a SECONDO object has been divided into a PS-Matrix, piece data are distributed and kept in slave Data Servers, while only their partition scheme is kept in the meta database as an <u>*flist*</u> object.

SPATIAL	$\underline{point}$, $\underline{line}$	$\rightarrow$ FLIST	$\underline{flist(point)}$, $\underline{flist(line)}$
RELATION	$\underline{rel(tuple(T))}$	$\rightarrow$ FLIST	$\underline{flist(rel(tuple(T)))}$
INDEX	$\underline{rtree(T)}$	$\rightarrow$ FLIST	$\underline{flist(rtree(T))}$

Table 1: Some Example <u>*flist*</u> Data Types

There are two kinds of methods keeping distributed data on slave data servers. The first keeps the partial data in slave Mini-SECONDO databases, saved as normal SECONDO objects. The second exports data to the PSFS nodes as disk files. Hereby, there are two kinds of <u>*flist*</u> objects in Parallel SECONDO.

1. Distributed Local Objects (DLO): A DLO divides a large-sized SECONDO object to a $N \times 1$ PS-

Matrix, each row is saved in a slave Mini-SECONDO database as SECONDO objects, called sub-objects. All sub-objects belonging to a same *flist* have the same name in different slave databases. Theoretically, DLO flist can wrap all available SECONDO data types.

2. Distributed Local Files (DLF): Data are divided into a $R \times C$ PS-Matrix, and each piece is exported as a disk file in PSFS, called sub-file. Sub-files can be exchanged among Data Servers during parallel procedures. At present, only relations can be exported and kept as sub-files, hence DLF flist is prepared for relations only.

Although it is possible to wrap any SECONDO data type with an *flist* object, there are some data too small to be distributed in this way. For example, a rectangle query window is used by every slave data server during a parallel query, but it is not advisable to divide it into smaller pieces which are then distributed over the cluster, since the rectangle itself is very small. In this case, this rectangle can be simply duplicated to every Data Server during the runtime. Apparently, not all SECONDO objects can be duplicated, since data are delivered as nested-lists, which require a relatively expensive transforming overhead. E.g, a relation containing one million tuples should not be delivered to every slave along with the query. Therefore, a new data kind called **DELIVERABLE** is introduced for Parallel SECONDO, and only data types associated with this kind can be used in parallel queries, and duplicated to slaves during the runtime. All **DELIVERABLE** data types are listed in Table 2.

5.3 Operators

Along with the creation of the *flist* data type, several operators are introduced to process distributed objects and parallel queries. Briefly, these operators are divided into four kinds: flow, assistant, Hadoop and PSFS. The PSFS operators are invisible to users, exchanging data between SECONDO instances and PSFS, therefore they are not specially introduced here.

5.3.1 Flow Operators

Flow operators connect sequential queries with parallel queries. At present, two operators named **spread** and **collect** are defined for this kind, and they can only process DLF *flist* objects. Sub-objects in one DLO *flist* are kept in slave databases and cannot be transferred over the nodes, hence they are not supported by the flow operators.

spread

```
stream(tuple(T)) x [fileName: string]
  x [filePath: text] x [dupTimes: int]
  x AI x [scaleN: int] x [KPAI: bool]
  x [AJ] x [scaleM: int] x [KPAJ: bool]
→ flist(stream(tuple(T')))
```

spread partitions a SECONDO relation into a PS-Matrix, distributing pieces into the cluster, and returning a DLF *flist*. The relation is first divided to rows in the PS-Matrix according to the indispensable partition attribute *AI*. If another partition attribute *AJ* is indicated, each rowcan be further partitioned into columns.

Each piece of the PS-Matrix is exported as a sub-file. Both the fileName and the filePath arguments are optional. If they are not indicated, sub-files are then kept in the default PSFS nodes, and their names are set by rules. The user is allowed to set the file name and path by himself, although it may create homonymic sub-files with different queries.

	Type	Algebra	NumOfFlobs	PersistencyMode		
1	**bool**	StandardAlgebra	0	Memoryblock-fix-core		
2	**cellgrid2d**	TemporalAlgebra	0	Memoryblock-fix-core		
3	**cluster**	TopRelAlgebra	0	Memoryblock-fix-core		
4	**duration**	DateTimeAlgebra	0	Serialize-fix-core		
5	**edge**	GraphAlgebra	0	Memoryblock-fix-core		
6	**geoid**	SpatialAlgebra	0	Memoryblock-fix-core		
7	**gpoint**	NetworkAlgebra	0	Memoryblock-fix-core		
8	**ibool**	TemporalAlgebra	0	Memoryblock-fix-core		
9	**iint**	TemporalAlgebra	0	Memoryblock-fix-core		
10	**instant**	DateTimeAlgebra	0	Serialize-fix-core		
11	**int**	StandardAlgebra	0	Serialize-fix-core		
12	**ipoint**	TemporalAlgebra	0	Memoryblock-fix-core		
13	**ireal**	TemporalAlgebra	0	Memoryblock-fix-core		
14	**istring**	TemporalExtAlgebra	0	Memoryblock-fix-core		
15	**point**	SpatialAlgebra	0	Memoryblock-fix-core		
16	**real**	StandardAlgebra	0	Serialize-fix-core		
17	**rect**	RectangleAlgebra	0	Memoryblock-fix-core		
18	**rect3**	RectangleAlgebra	0	Memoryblock-fix-core		
19	**rect4**	RectangleAlgebra	0	Memoryblock-fix-core		
20	**rect8**	RectangleAlgebra	0	Memoryblock-fix-core		
21	**string**	StandardAlgebra	0	Serialize-variable-extension		
22	**ubool**	TemporalAlgebra	0	Memoryblock-fix-core		
23	**uint**	TemporalAlgebra	0	Memoryblock-fix-core		
24	**upoint**	TemporalAlgebra	0	Memoryblock-fix-core		
25	**ureal**	TemporalAlgebra	0	Memoryblock-fix-core		
26	**ustring**	TemporalExtAlgebra	0	Memoryblock-fix-core		
27	**vertex**	GraphAlgebra	0	Memoryblock-fix-core		
28	**filepath**	BinaryFileAlgebra	1	Memoryblock-fix-core		
29	**text**	FTextAlgebra	1	Memoryblock-fix-core		

Table 2: DELIVERABLE data types

A sub-file consists of the type and data files. The type file describes the schema of the exported relation, being produced during the type mapping period and duplicated to every node inside the cluster. Data files contain tuples' binary blocks, and all data files kept in one Data Server share the same type file. The sub-files are readable with the **ffeed** operator.

Here the PS-Matrix has a scale of $scaleN \times scaleM$. They also both are optional arguments. The default $scaleN$ value is the cluster size, and the default $scaleM$ value is 1. Normally, the partition attribute *AI* is removed after the query, except the argument *keepAI* is set as true. Same for the other partition attribute *AJ*. The *AJ* must be different from *AI*.

Data files are named as fileName_row_column, row $\in [1, scaleN]$, and column $\in [1, scaleM]$. For the purpose of fault-tolerance, each partition file is duplicated on $dupTimes$ continuous slave nodes, and the default value of *dupTimes* is 1. All duplicated files are kept in PSFS nodes.

collect

```
flist(stream(T)) x [row: int] x [column: int] → stream(tuple(T))
```

The **collect** operator performs the opposite function to the **spread**. It accepts a DLF kind *flist* object, collects required sub-files over the cluster, and returns a tuple stream from sub-files. Both the row and column arguments are optional arguments, and their default values are 0, which means the complete row and column. If there is only one parameter given, then it is viewed as a row number. Only non-negative integer numbers are accepted as parameters.

By default, this operator reads all sub-files denoted in the given DLF flist. If the optional parameters are set, then it returns part of the PS-Matrix. For example:

- collect[1] and collect[1,0] read all sub-files in the first row.
- collect[0,2] reads all sub-files in the second column.
- collect[1,2] reads one piece sub-file in the PS-Matrix, located at the first row and the second column.
- collect[0,0] and collect[] read all sub-files inside the PS-Matrix.

If the required sub-files are located in a remote node, then they are copied to the current node before being read.

5.3.2 Assistant Operators

All assistant operators cannot be used alone, but have to work with the following hadoop operators.

para

```
flist(T) → T
```

The *flist* is designed to wrap all available SECONDO data types, and work with various SECONDO operators. However, it is impossible to let all operators recognize and process this new data type. Regarding this issue, we implement the **para** operator to unwrap *flist* objects and return their embedded data types, in order to pass the type checking of existing operators.

Note that there is no value mapping function provided for this operator, since it is only prepared for letting *flist* objects pass through different operators' type mapping functions. It is designed this way as there is no generic data able to express SECONDO objects with various types. Therefore, the **para** operator can only work with the hadoop operators that will be introduced later. It is set inside hadoop operators' interFunc arguments, which are not evaluated directly in the master database.

TPARA

```
flist(T) → T
```

This is a type operator, extracting the internal type from the input *flist* object, and delivering it to the internal function argument as its input parameter. It works similar as the above **para** operator, but it can only be used implicitly.

TPARA2

```
ANY x flist(T) → T
```

This is also a type operator , working similar like **TPARA**, but it gets the embedded type from the second *flist* input instead of the first one.

5.3.3 Hadoop Operators

hadoopMap

```
flist(T) x [subName: string] x [subPath: text]
  x [ kind: DLO | DLF ] x [ mapTaskNum: int ]
  x [ executed: bool ]
  x ( interFunc: map ( T → T1) )
→ flist(T1)
```

hadoopMap creates an *flist* object of either DLO or DLF kind, after processing its *interFunc* by slaves in parallel, during the map step of the template Hadoop job. Both subName and subPath are optional arguments, the default flist kind is DLO, and the mapTaskNum default value is the current cluster size. The interFunc is expressed as a function argument, and not evaluated in the master node, but delivered and processed in slaves. Take the creation of a distributed B-Tree as an example. The original sequential query is:

```
let dataSCcar_Licence_btree = dataSCcar createbtree[Licence];
```

This query creates a B-Tree index for a relation called dataSCcar, based on its Licence attribute. The parallel queries are:

```
let dataSCcarList = dataScar feed
  projectextend[Licence, Type, Model; Journey: .Trip]
  spread[;Licence, 10, TRUE;] hadoopMap[; . consume];

let carLicence_btreeList = dataSCcarList
  hadoopMap["dataSCcar_Licence_btree"; . createbtree[Licence]];
```

Here both the dataSCcarList and the carLicence_btreeList are *flist* objects. The first query distributes the tuple relation to slaves and returns a DLO flist. It first uses the **spread** operator to distribute data as sub-files, and returns a DLF *flist* object. All SECONDO indexes must be built based on stored relations, where each tuple has a unique tupleID that is indispensable for index structures. Therefore, the returned DLF flist is sent to the **hadoopMap** operator, letting each slave Data Server import the local sub-files to its Mini-SECONDO database, saved as a sub-object. At last, the second query reads the created dataSCcarList, and uses a **hadoopMap** to let slaves create their own index by executing the interFunc.

Limited by the current SECONDO query processor, the **hadoopMap** only describes the Map stage in the created Hadoop job. At the same time, the following **hadoopReduce** and **hadoopReduce2** operators describe only the Reduce stage in the job. In order to construct jobs including both Map and Reduce stages, i.e. both stages invoke Mini-SECONDO on slaves to process the interFunc queries, an additional boolean parameter named *executed* is provided in the **hadoopMap** operator. By default, this parameter is set to be true, then the operator creates a Hadoop job with only its Map stage containing the InterFunc query. Particularly, when **hadoopMap** concatenates with **hadoopReduce** or **hadoopReduce2**, and sets the *executed* parameter to be false, then it returns an unexecuted *flist* object, with the InterFunc being not processed. Such an unexecuted *flist* is delivered to the next reduce operator, in which a Hadoop job is created with Map tasks containing the InterFunc argument in the **hadoopMap** and the Reduce tasks processing the InterFunc described in the reduce operator. An example of this feature will be explained later.

hadoopReduce

```
flist(T) x partAttribute
  x [subName: string] x [subPath: text]
  x [ kind: DLO | DLF ] x [reduceTaskNum: int]
  x ( interFunc: map ( T → T1) )
→ flist(T1)
```

hadoopReduce also takes one flist as the input, delivering and processing its interFunc by slaves in parallel. However, the interFunc is processed in the reduce step instead of the map step of its template Hadoop job. In the map step, the input has to be redistributed based on the partitionAttribute, hence the input *flist* object must wrap a stream of tuples.

Compared with the **hadoopMap** operator, this operator needs two additional arguments, partAttribute and reduceTaskNum. Data are first re-distributed into reduceTaskNum columns based on the partAttribute in the map step. Then each reduce task collects one column data from the re-distributed PS-Matrix, using it as the input for the interFunc.

Take the third BerlinMOD query as an example, which can be processed with the following parallel query:

```
let OBACRres003 =
  QueryLicences_Top10_List
  hadoopMap[DLF, false
    ; . feed loopjoin[ para(dataSCcar_Licence_btree_List)
      para(dataSCcar_List) exactmatch[.Licence] {LL}
    projectextendstream[Licence_LL; UTrip: units(.Journey_LL)]
    extend[Box: scalerect(bbox(.UTrip), CAR_WORLD_X_SCALE,
      CAR_WORLD_Y_SCALE, CAR_WORLD_T_SCALE)]
    extendstream[Cell: cellnumber(.Box, CAR_WORLD_GRID)]]]
  hadoopReduce[Cell, "Q3_Result", DLF, REDUCE_SCALE
    ; . para(QueryInstants_Top10_Dup_List) feed {II} product
      projectextend[; Licence: .Licence_LL, Instant: .Instant_II,
        Pos: val(.UTrip atinstant .Instant_II)]
      filter[isdefined(.Pos)] ]
collect[]
sort rdup consume;
```

As shown in this example, the **hadoopMap** operation first selects trajectories by pruning the distributed B-Tree created before. Result trajectories are decomposed into units, which are then distributed into a global cell-grid, by adding a new attribute named Cell. Afterwards, reduce tasks fetch different intermediate results as their input based on the Cell attribute, and the interFunc is processed by slaves in parallel during the reduce step.

Note that in the above examples, interFunc arguments in both **hadoopMap** and **hadoopReduce** operations include several *flist* objects. For each operation, the first *flist* is set as the input, delivered to the interFunc by the implicit type operator **TPARA**. However, all the remaining *flist* objects have to be quoted by the **para** operator, since operators like **exactmatch** and **product** cannot accept any *flist* input.

The *executed* parameter in the **hadoopMap** is set to be false here, therefore only one Hadoop job is created in the **hadoopReduce** operation, which contains not only the Reduce stage described in **hadoopReduce**, but also the Map stage described in **hadoopMap**.

hadoopReduce2

```
flist(T1) x flist(T2)
  x partAttribute1 x partAttribute2
  x [subName: string] x [subPath: text]
  x [ kind: DLO | DLF ] x [reduceTaskNum: int]
  x [ isHDJ : bool ]
  x ( interFunc: map ( T1 x T2 → T3) )
→ flist(T3)
```

This operator is similar as the above **hadoopReduce**, except it accepts two flists as the arguments for the interFunc. Both inputs are redistributed based their partition attributes respectively, and make queries more flexible.

In general, intermediate data produced by Mini-SECONDO databases are shuffled in Parallel SECONDO through PSFS. At the same time, for the purpose of research, Parallel SECONDO also can shuffle intermediate data only by HDFS. This is named HDJ (Hadoop Distribute Join), since the data are distributed by the Hadoop system. This feature is only supported in the **hadoopReduce2** operator, by setting its optional parameter isHDJ to be true, which is false by default. More details about HDJ can be found in our technical report about Parallel SECONDO, ”Simple and Efficient Coupling of Hadoop With a Database Engine”.

6 A Tour in Parallel Secondo

In this section, a small example is prepared to demonstrate how to use Parallel SECONDO to process queries like building and searching a distributed index, or making a distributed hash-join query.

```
restore database opt from opt;
open database opt;
let Con_PLZ = 16928;
let PartFile = plz feed spread[;PLZ,6,TRUE;]
let PartRel = PartFile hadoopMap[;. consume];
close database;
```

In the above queries, the relation plz in database opt is distributed over the cluster. It is first distributed by **spread** into a 6×1 PS-Matrix, with the DLF *flist* PartFile as the result. Then the PartFile is sent to **hadoopMap**, and its sub-files are loaded into slave databases, and the DLO *flist* PartRel is returned. Assume this cluster contains two slave Data Servers, then the PartRel has a 2×1 PS-Matrix, while each sub-object contains half of the relation.

```
open database opt;
```

```
let SubBTree = PartRel hadoopMap[; . createbtree[PLZ]];
let ParaResult = SubBTree
  hadoopMap[ DLF; . para(PartRel) exactmatch[Con_PLZ] ] collect[] consume;
close database;
```

Here the **hadoopMap** is first used to create a distributed B-Tree over the cluster, returning the DLO *flist* SubBTree. Afterwards, another **hadoopMap** is used to prune the distributed B-Trees in every slave. The selected result is still distributed on slaves, and returned as a DLF *flist*. Here the Con_PLZ is delivered automatically to every slave Data Server, although it is created only in the master database, because its type *int* is associated with the DELIVERABLE kind. At last, the **collect** operator accumulates all distributed sub-files created by the former **hadoopMap** operation.

```
open database opt;
query PartRel hadoopMap[DLF
  ; . feed para(PartFile) {n} hashjoin[PLZ, PLZ_n] ] collect[] count;

# Wrong query, should be forbidden
query PartRel hadoopMap[DLF
  ; . feed para(PartRel) feed {n} hashjoin[PLZ, PLZ_n] ] collect[] count;

query PartFile hadoopReduce[ Ort , DLF, 5
  ; . para(PartFile) {n} hashjoin[PLZ, PLZ_n] ] collect[] count;

query PartRel PartFile hadoopReduce2[ Ort, Ort, DLF, 5
  ; . feed .. {n} hashjoin[Ort, Ort_n] ] collect[] count;

close database;
```

Above, we list four parallel queries for processing a same distributed self-hash-join operation, with **hadoopMap**, **hadoopReduce** and **hadoopReduce2** operators respectively. The first query finishes the operation in the map step only. It takes PartRel as the input flist, hence each map task reads one sub-object from its own Mini-database, and gets the other side from the PartFile quoted with the **para** operator. If a DLF flist is used in the interFunc of a hadoop operator, then all its sub-files will be collected to that task. Note the second query that uses PartRel in its interFunc cannot get the correct result, as each map task can only get its own sub-object.

The third query is a **hadoopReduce** operation, the input PartFile is repartitioned into 5 columns based on its Ort attribute in its map step. Averagely each reduce task finishes the interFunc with 20% data from the left side, and the whole data set from the right side.

The last query finishes with the **hadoopReduce2** operator, it reads SubRel and SubObj at the same time. Each map task reads the left side from its database, and reads the right side from the PSFS. Both side relations are repartitioned by their Ort attribute into 5 pieces. In its reduce step, each task gets 20% data from both sides.

A Setting Up Parallel Secondo On A Single Node

Nowadays, it is common that one computer has a powerful computing and storage capability, with several processors or cores, and several large hard disks. Therefore, it is possible to simulate a virtual cluster on one computer only, and set the Parallel SECONDO up on it. At present, Parallel SECONDO provides at least both Ubuntu and Mac OS X platforms. The deployment of Parallel SECONDO on one computer includes several steps.

Prerequisites

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

- Java^{TM} 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
  ```

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.[1] 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.

   - The *Cluster* [2] 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.

[1] The version must be 3.3.2 or higher.

[2] This parameter must be set before continue.

- 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 adjusted profile file 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=...
```

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
```

B Setting Up Parallel Secondo In 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.