User Guide For Parallel Secondo

May 30, 2012

1 Parallel Secondo Infrastructure

Parallel SECONDO is constructed by coupling simply the Hadoop framework and discrete SECONDO databases on a computer cluster, as shown in Figure 1. It can be deployed on either a single computer or a cluster containing tens or even hundreds of computers. Briefly, its two components Hadoop and discrete SECONDO databases coexist in the same system, and each can be used independently. In Hadoop, nodes are communicated through its HDFS (Hadoop Distributed File System), while each single-node SECONDO exchanges its data with the others through the PSFS (Parallel SECONDO File System). Unlike Hadoop that 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 may contain several data servers, especially nodes with multiple hard disks, in which the user can set a data server on each disk, hence can take the full advantage of the cluster resources. During various parallel procedures, a few data are exchanged among nodes through the HDFS, in order to assign tasks to data servers. At the same time, the bulk of intermediate query results are exchanged among data servers through the PSFS.

A particular data server set on the master node is denoted as the master data server, its Mini-SECONDO is set to be the only entrance to the system, called the master database. Through various PQC (Parallel Query Converter) operators provided in the master database, a custom parallel query is converted to a Hadoop job, and then partitioned to tasks. These tasks are processed by data servers in parallel, complying the scheduling of the Hadoop framework. Different from the other databases in slave data servers, the master database contains some global and meta data of the whole system, therefore it is also called the meta

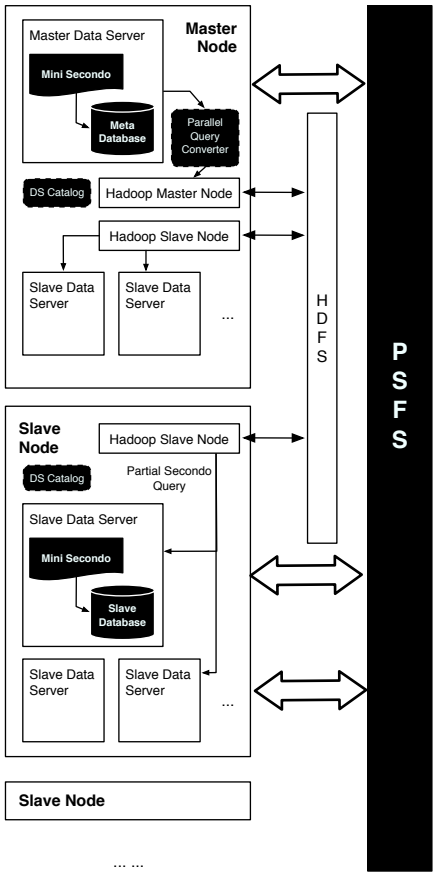

Figure 1: The Infrastructure of Parallel Secondo

database.

An identical DS-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, while the second and the third elements tell apart data servers within a same node based on their PSFS locations and Mini-SECONDO ports. The master file should only contain 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.

2 Hadoop Algebra

SECONDO is composed by algebras, each containing a set of data types and operators. Two algebras are especially required by the Parallel SECONDO, Hadoop and HadoopParallel. They must be activated before deploying the system to the cluster. Besides, some other components are also required:

1. Hadoop Package. Hadoop 0.20.2 is used as the underlying framework for Parallel SECONDO, although itself is not included in SECONDO by default. It has to be downloaded by the user, and put into the $SECONDO_BUILD_DIR/bin. Nonetheless, its installation is not required, and can be performed automatically along with the deployment of the Parallel SECONDO.

2. Parallel Secondo Auxiliary Utilities. Parallel SECONDO includes a set of bash scripts to manage the system. They are kept in the Hadoop algebra by default. These scripts include at least:

   ps-cluster-format: An automatic deployment tool for initializing the Parallel SECONDO environment of the cluster, including the Hadoop framework.

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

   ps-secondo-buildMini: It extracts and distributes the Mini-SECONDO base on the current single-node 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 severs in one cluster node, this script helps 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 starts up a SECONDO Client-Server based text terminal interface, and connects any SECONDO monitors that has been started up in the cluster.

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

3. Parallel Configuration File. All Parallel SECONDO parameters are set in the file named ParallelSecondoConfigure.ini, and its example file can also be found in the Hadoop algebra.

4. Precast Hadoop Job. Several precast generic Hadoop jobs are prepared for several parallel operators in SECONDO. Their source files are kept with the Hadoop algebra together, and the jobs are generated when the algebra is compiled.

3 Configuring Parallel Secondo

Parallel SECONDO configurations are all set in the 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 set uniform Hadoop configurations for all involved nodes. In this way, the flexibility of the system is restricted, but the new user can set up the Parallel SECONDO quickly without learning too many details about Hadoop. All parameters are listed by lines, composed of three elements including the file name, title and value. Each parameter is inserted into the specific file with its corresponding value.

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

This section is further divided into 4 parts. The first part contains all indispensable parameters prepared for Parallel SECONDO. The non-advanced user 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 it with double the number of the processor cores, 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 a value 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, including some services set in the second part.

The second **cluster** section lists all involved data servers. Each data server is indicated by 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. 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
```

In the above example, 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 one is used as the master, and also as a slave data server. The master Mini-SECONDO can be accessed through the port 11234.

The last section **options** is prepared for some special cases 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 automatically to be the master Mini-SECONDO.

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-". Each script has a usage explanation, and can be printed with the "-h" argument. Here we only introduce several common operations in Parallel SECONDO accomplished with these scripts.

4.1 Update Mini-Secondo

The Mini-SECONDO is a compact SECONDO distribution, it only contains essential components required to manipulate the slave databases. All of them are built based on the user's accustomed version, and keep identical over the whole cluster. In case there is any extension made for SECONDO , like creating new data types or operators, users can reconstruct the system once, and distribute the update to the whole cluster immediately.

```
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-stop-AllMonitors
$ 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 the contrast, if the "-l" parameter is set, then the new SECONDO is only distributed to all data servers on the current node.

The Mini-SECONDO contains all essential components a SECONDO database needs. All of them are listed in a text file named miniSec_list, which is also kept in the Hadoop algebra. Each file or folder in the single-node SECONDO is listed there with one line. 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 have to 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 first script ps-startMonitors starts up all Mini-SECONDO monitors on the current computer, while the second script ps-start-AllMonitors visits every node and runs the ps-startMonitors, so as to start up all monitors on the cluster. In the contrast, the script ps-stopMonitors turns off all Mini-SECONDO monitors on the current computer, and all monitors on the cluster are shut down with 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 proposed to access any Mini-SECONDO database inside the system. One cluster node may contain several Mini-SECONDO databases, which are arranged in order, starting from 1, according to the DS-Catalog. The commands for opening the Parallel SECONDO main 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 with the value of 1. Besides, it is also possible to use the graphic Java interface in Secondo to visit the meta database. The user can open the javagui interface as usual, then connect the indicated database by setting its host IP and port number in the menu “Server”, and “Setting”.

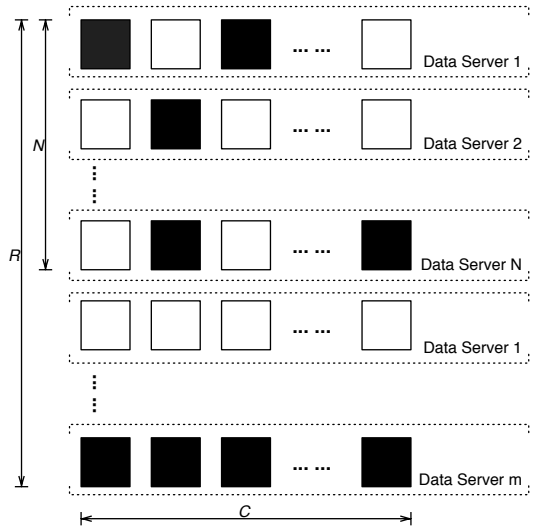

Figure 2: PS-Matrix

5 Secondo Parallel Query Expression

Here we introduce how to write queries in Parallel SECONDO in executive level language.

5.1 PS-Matrix

In parallel SECONDO, data is distributed over the cluster as 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 can only be kept on one data server. It is possible for $d(x)$ to produce more rows than the cluster scale N, and to let data servers contain multiple rows. Afterwards, $d(y)$ divides each row to C columns. As a result, a PS-Matrix is composed by $R \times C$ pieces, but not all pieces contain data. Normally, 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 or several of its attributes.

5.2 Data Type

In Parallel SECONDO, data type *flist* is especially proposed for expressing 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 is being divided into a PS-Matrix, piece data are distributed and kept in slave data

servers, while only their entry information are kept in the meta database, expressed as a *flist* object.

SPATIAL	*points line*	→ FLIST	*flist(point) flist(segment)*
RELATION	*rel(tuple(T))*	→ FLIST	*flist(rel(tuple(T)))*
INDEX	*rtree*	→ FLIST	*flist(rtree)*
TEMPORAL	*moving*	→ FLIST	*flist(unit)*

Table 1: Flist Data Types

There are two of kinds methods keeping distributed data on slave data servers. The first keeps the partial data into slave Mini-SECONDO databases, saved as normal SECONDO objects. The second exports data to the PSFS nodes as disk files. Hereby, there also will be two kinds of *flist* 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, like common SECONDO objects, called sub-objects. All sub-objects belonging to a same flist share 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 from SECONDO databases as disk files, saved in PSFS nodes, called sub-files. Sub-files can be exchanged among data servers during parallel procedures. At present, only tuple relations can be exported and kept as sub-files, hence DLF flist is prepared for tuple relations only.

Although it is possible to wrap any SECONDO data type with a *flist* object, there are some data are too small to be distributed 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 together. Therefore, a new data kind called **DELIVERABLE** is proposed for parallel SECONDO, and only data types associated with this kind can be encapsulated into parallel queries, and duplicated to slaves during the runtime. All **DELIVERABLE** data types are listed in Table 2.

	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

5.3 Operators

Along with the creation of the *flist* data type, several operators are proposed to process distributed objects and parallel queries. Briefly, these operators are divided into three kinds: pipe operators, assistant operators and hadoop operators.

5.3.1 Pipe Operators

Pipe operators connect single-node SECONDO objects and *flist* objects. At present, two operators named **spread** and **collect** are proposed for this kind, and they can only process DLF *flist* objects. Sub-objects are kept in slave databases, and cannot be transferred over nodes, hence they are not supported by this kind of 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 pieces by rows according to the indispensable partition attribute *AI*. If another partition attribute *AJ* is indicated, each piece can be further partitioned, still by rows.

Each piece of the PS-Matrix is exported as sub-files. 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 cause homonymic sub-files with different queries.

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.

The PS-Matrix that **spread** creates 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 $dupTime$ continuous slave nodes, and the default value of *dupTime* 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 examples:

- 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 locate 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
```

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 match them with existing sequential operators.

Note there is no value mapping function provided for this operator yet, 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, **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 ( interFunc: map ( T → T1) )
→ flist(T1)
```

hadoopMap creates a *flist* object of either DLO or DLF kind, after processing its interFunc by slaves in parallel, during the map step of the precast 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 distribute 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 existing 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, and lets 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.

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)

T1 and T2 are either rel(tuple) or stream(tuple)
```

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 precast 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 query:

```
let OBACRres003 =
  QueryLicences_Top10_List
  hadoopMap[DLF; . 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 extending 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 in 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 left *flist* objects have to be quoted by the **para** operator, since operators like **exactmatch** and **product** cannot accept any *flist* input.

hadoopReduce2

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

T1 and T2 are either rel(tuple) or stream(tuple)
```

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

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 distributed index, or making a distributed hash-join query.

```
restore database opt from opt;
open database opt;
let Con_PLZ = 16928;
let SubFile = plz feed spread[;PLZ,6,TRUE;]
let SubRel = SubFile 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* SubFile as the result. Then the SubFile is sent to **hadoopMap**, and its sub-files are loaded into slave databases, and the DLO *flist* SubRel is returned. Assume this cluster contains two slave data servers, then the SubRel has a 2×1 PS-Matrix, while each sub-object contains half of the relation.

```
open database opt;
let SubBTree = SubRel hadoopMap[; . createbtree[PLZ]];
let ParaResult = SubBTree
  hadoopMap[ DLF; . para(SubRel) 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 SubRel hadoopMap[DLF
```

```
  ; . feed para(SubFile) {n} hashjoin[PLZ, PLZ_n] ] collect[] count;

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

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

query SubRel SubFile 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 SubRel as the input flist, hence each map task reads one sub-object from its own Mini-database, and gets the other side from the SubFile 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 SubRel 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 SubFile 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 commercial computer also 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 following steps:

1. Prepare the authentication key-pair. Both data servers and the underlying Hadoop platform rely on security shell as the basic communication level, and it is better to connect shells without password, even on a single computer. For this purpose, the user should create and set up the authentication key-pair on the computer with commands:

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

2. Install SECONDO. The install guide can be found on our website, and the user can install SECONDO database as usual. After the installation is finished, the user can verify the correctness of the installation, and then compile the SECONDO system. Note both Hadoop and HadoopParallel algebras must be activated.

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

   $ cd $SECONDO_BUILD_DIR
   $ make
   ```

3. 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.

4. Set up the ParallelSecondoConfig.ini according to the current computer. The example file is kept in the Hadoop algebra's clusterManagement folder, prepared for setting up Parallel SECONDO on a single computer with Ubuntu operating system. Detailed explanations about this file can be found in Section 3. After all required parameters are set, copy the file also to the $SECONDO_BUILD_DIR/bin, and initialize the environment with ps-cluster-format.

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

   Since this step defines a set of environment variables, the user should start a new shell after the format is finished. The correctness of the initialization can be checked with the following command:

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

5. The above steps set up both data servers and Hadoop on the computer. Afterwards, the Namenode in the Hadoop framework should be formated before starting it.

```
$ hadoop namenode -format
$ start-all.sh
```

6. The fourth step only initializes the data servers in the computer, but the Mini-SECONDO systems are not distributed yet. They are distributed by script ps-secondo-buildMini:

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

7. Start up mini SECONDO Monitors.

```
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-startMonitors
$ ps-cluster-queryMonitorStatus
```

   The second script is used to check whether all Mini-SECONDO monitors are started.

8. Open the text interface of the master database.

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

9. Parallel SECONDO can be closed off by stopping all Mini-SECONDO monitors and the Hadoop framework.

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

10. In case the user wants to delete Parallel SECONDO from the computer, script ps-cluster-uninstall can clean the system completely with one command.

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

B Setting Up Parallel Secondo In A Cluster

Parallel SECONDO can also be deployed on a cluster containing tens or even hundreds of computers with following 10 steps.

1. Create the entrance account for the user on all computers of the cluster, each account should have at least the same user name, here it is assumed as Jeffrey. This step usually is done by the cluster manager. Using services like NIS can make this step easier, if the user wants to create the same account on hundreds of computers.

   ```
   $ su -
   # useradd  jeffrey
   ```

2. Like the setup on the single computer, the user also needs to visit all computers without password. For this purpose, the authentication key-pair should be created and deployed on all computers. Here it is better to use services like NFS to let all computers share a same $HOME space, or else the key-pair must be uploaded to every involved computer.

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

3. Install all libraries needed by SECONDO on every cluster node, as each data server runs its Mini-SECONDO independently. Details of installing the environment are explained on our website, regarding different operating systems. The installation of SECONDO can be checked by the following commands. Only the master node should have the SECONDO source code, and construct the system before continuing.

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

   $ cd $SECONDO_BUILD_DIR
   $ make
   ```

If all four basic Secondo environment variables SECONDO_CONFIG, SECONDO_BUILD_DIR, SECONDO_JAVA and SECONDO_PLATFORM have already been set like above, and all their values are available on the computer, then this computer is able to process any SECONDO application. Particularly, if the operating system is Ubuntu, then by default the line

```
source .secondorc
```

is appended at the end of the file $HOME/.bashrc. However, in Parallel SECONDO, this line must be set above the line

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

or else, the parallel Secondo cannot work correctly.

4. Download the hadoop distribution with 0.20.2 version, put its package into $SECONDO_BUILD_DIR/bin, together with the file ParallelSecondoConfig.ini after it has been set properly according to the practical cluster, as explained in Section 3. Afterwards, use the format script to initialize the Parallel SECONDO on the complete cluster.

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

5. Start a new shell and check whether all following variables are properly set before continuing.

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

6. Initialize and start up Hadoop with two steps:

```
$ hadoop namenode -format
$ start-all.sh
```

   All these tools are provided by Hadoop, and the user can check whether Hadoop works properly on the cluster by visiting its web monitors. Usually it takes some time for Hadoop to be started completely.

7. Distribute the Mini-SECONDO to the whole cluster. Ensure that both the Hadoop and HadoopParallel algebras have been activated in the user's accustomed SECONDO system before constructing the system. Afterwards, the Hadoop job ParallelSecondo.jar should have been generated and put in the path $SECONDO_BUILD_DIR/bin, and then the user can distribute Mini-SECONDO to the whole cluster with the ps-secondo-buildMini script.

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

8. All involved Secondo Monitors must be started up before running Parallel Secondo, with steps:

```
$ cd $SECONDO_BUILD_DIR/Algebras/Hadoop/clusterManagement
$ ps-start-AllMonitors
$ ps-cluster-queryMonitorStatus
```

   The second script is used to make sure that all data servers' Secondo monitors are started.

9. Start up the text interface:

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

10. Parallel SECONDO can be stopped by turning off all mini Secondo monitors and the Hadoop with two steps respectively:

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