SmartMet Server

GRID ENGINE

Finnish Meteorological Institute

2021-12-27

TABLE OF CONTENTS

1 GRID ENGINE 3
1.1 Introduction 3
1.2 Configuration file 3
1.2.1 External configuration parameters 3
1.2.2 Environment variables 4
1.2.3 Parameter namespaces 4
1.2.4 Directory paths 5
1.2.5 Dividing definitions into multiple files 5
1.2.6 Conditional declarations 6
2 MODULES 7
2.1 Introduction 7
2.2 Content Server 8
2.2.1 Content Storage 8
2.2.2 Connection 9
2.2.3 Content Cache 10
Cache swapping 10
Non-swapping cache 10
2.2.4 Logs 10
2.3 Data Server 11
2.3.1 Grid storage 12
2.3.2 Virtual grids 12
2.3.3 Logs 13
2.4 Query Server 14
2.4.1 Data querying 14
1. Producer and/or geometry information missing 14
2. Generation information missing 15
3. Level type or/and level information missing 15
4. Forecast type or/and forecast number 16
2.4.2 Query cache 16
2.4.3 LUA functions 16
2.4.4 Parameter aliases 17
2.4.5 Logs 18
3 PARAMETER MAPPINGS 19
3.1 Introduction 19
3.2 Mapping files 20
3.3 Automated mapping file generation 20
4 NEWBASE MAPPINGS 22
4.1 Introduction 22
4.2 Producer and parameter mapping 22
4.3 Configuration 23

1 GRID ENGINE

1.1 Introduction

The grid engine is a component that allows SmartMet Server plugins to access and query available grid information and grid data. When the grid engine is used in the SmartMet server, the path to its main configuration file is added into the SmartMet Server's main configuration file.

```
engines:
{
    grid:
    {
        configfile    = "smartmet-engine-grid/cfg/grid-engine.conf";
    };
};
```

The grid engine's main configuration file is usually named as "grid-engine.conf". This file is used in order to configure all essential features related to the grid engine or to the libraries that it uses.

1.2 Configuration file

The grid engine’s main configuration file is a simple text file that uses JSON syntax. There are some special features related to this file that we want to describe first, because these features are used in quite many examples in this document.

1.2.1 External configuration parameters

A typical configuration of a system requires a lot of sensitive or environment specific information (passwords, directory and file names, connection parameters, etc.). This information is not usually public in spite of that we might publish the source code and the common configuration files.

For this reason, the grid-engine's main configuration file can import parameters from other configuration files. The idea is that we can import all sensitive information from our private file, meanwhile the actual configuration file can be public.

For example, we might have a private file named "production.conf" that contains all sensitive information (like passwords, directories, connection parameters, etc.) related to the production setup. The file might look like this:

```
REDIS_CONTENT_SERVER_PRIMARY_ADDRESS = "127.0.0.1"
REDIS_CONTENT_SERVER_PRIMARY_PORT = 6379
REDIS_CONTENT_SERVER_TABLE_PREFIX = "a."
```

We can import all these definitions into the grid engine's main configuration file by using "**@include**" command. After that we can reference to these parameter definitions like this.

```
@include "production.conf"

redis :
{
    address     = "$(REDIS_CONTENT_SERVER_PRIMARY_ADDRESS)"
    port        = $(REDIS_CONTENT_SERVER_PRIMARY_PORT)
    tablePrefix = $(REDIS_CONTENT_SERVER_TABLE_PREFIX)
}
```

In this way it does not matter if the main configuration file is publicly available (for example, in github), because it does not contain any sensitive information. All the sensitive information can be stored locally.

1.2.2 Environment variables

In the previous example we showed how all sensitive information can be imported from an external file. On the other hand, we can easily change the running environment from the production to the testing / development by just changing the imported file. We can do this be editing the current configuration file.

However, there is also an easier way to do this. In practice, all environment variables can be used in configuration parameters, which means that we can change the name of the imported file by changing value of single environment variable. For example, if we define the include clause in the previous example like this:

```
@include "$(SERVER_ENV)"
```

Then we can change the imported file by changing the environment variable "**SERVER_ENV**" on the command line like this:

```
export SERVER_ENV="/myserver/conf/testing.conf"
```

The point is that in this way we can change the whole runtime environment with just one "export" command. It also means that we can easily run multiple different installations in the same server.

1.2.3 Parameter namespaces

The SmartMet server environment contains tens of different configuration files which have (for historical reasons) quite variable style of defining configuration parameter names. All configuration parameters related to the grid support follows the same naming style. The idea is that each configuration parameter belongs to a name space that identifies the functional module where the current parameter is used. For example, all configuration parameters used by the grid engine are using the namespace "**smartmet:engine:grid**".

```
smartmet :
{
    engine :
    {
        grid :
        {
            # Grid engine's configuration parameters
        }
    }
}
```

The grid engine uses also some external libraries that needs to be configured. That's why the grid engine's configuration file might contain references to other namespaces. For example, all functionality related to extraction of GRIB files comes from the library "smartmet-library-grid-files". The current library is responsible for identification and extraction of GRIB, NetCDF and Newbase parameters, which requires a lot of configuration. Luckily most of this configuration is already done and we just need to define the location of the current module's main configuration file.

```
smartmet :
{
    library :
    {
        grid-files :
        {
          configFile = "%(DIR)/../../libraries/grid-files/grid-files.conf"
        }
    }
}
```

The purpose of the namespace definitions is to make sure that each configuration parameter in the SmartMet server can be uniquely identified. For example, the "configFile" parameter above can be uniquely identified as "smartmet.library.grid-files.configFile".

Actually, we could write the above configuration information also like this:

```
smartmet.library.grid-files.configFile = "%(DIR)/../../libraries/grid-files/grid-files.conf"
```

1.2.4 Directory paths

Technically it is possible to include other configuration files that also include or refer some other files and so on. In this kind of situation, it might be difficult to follow the actual location of these files. For example, if a parameter value that contains a relative path is used in configuration file A, then this relative path probably refers to a wrong directory in the configuration file B if the current configuration file is located in a different directory. In other words, the relative path is valid only in the configuration file where it was originally defined. If this relative path is imported into another configuration, it does not work, because it is not relative in this configuration file.

For that reason, it is not recommended to use relative paths in parameters that might be imported into another configuration file. This does not mean that we should write absolute paths into the configuration files. There is a special definition "**%(DIR)**" that tells the absolute path of the current configuration file. When a parameter value with the "**%(DIR)**" definition is imported into another configuration file, then this definition is replaced with the absolute path value of the current source configuration file.

1.2.5 Dividing definitions into multiple files

The grid-engine uses quite many different configuration files (like mappings files, alias files, LUA files, etc.). The point is that the grid engine's main configuration file does not contain mapping definitions or alias definitions. It just defines the files from where these definitions can be found. For example, parameter mapping files are defined in the grid-engine's main configuration file like this:

```
mappingFiles =
[
  "%(DIR)/mapping_netCdf.csv",
  "%(DIR)/mapping_newbase.csv"
];
```

Different definitions can be divided into multiple files in spite of that they technically could be in the same file. We might want to separate definitions for several reasons. For example, we might want to separate ate "standard definitions" (= definitions that come with the installation package) from the definitions that the user had made by itself. The idea is that when a user installs a new version of the package, it does not overwrite his own definitions.

The example above shows how NetCDF and Newbase parameter mappings are separated in different files in spite of that they could be technically in the same file. In the other words, the grid engine does not know which parameter names are NetCDF parameter names and which parameter names are Newbase parameter names. It just maps some parameter names to FMI parameter names.

This same idea is used in the "smartmet-library-grid-files" module, where different parameters and parameter name mappings are configured. We mentioned it here, because the reader should understand that almost all features are configurable. For example, we are talking a lot of FMI parameter names that every parameter (in GRIB1, GRIB2, NetCDF and Querydata files) should be mapped to. On the other hands, almost everything is configurable, even the FMI parameter names. Let's say that we have a GRIB parameter named "myParameterX" and there are no matching FMI name for it. The simplest way to solve the problem it to just create FMI parameter name "myParameterX" and make mappings from "GRIB:myParameterX" => "FMI:myParameterX".

1.2.6 Conditional declarations

Configuration files can support conditional declarations, which are checked when the configuration file is read the first time. For example, it possible to check if a parameter or an environment value is defined, and do something according to the result. For example, it is possible to print warning that something is missing. If there is some critical configuration information missing then it is possible to thrown an exception, which usually makes the SmartMet server to stop.

```
@ifdef SMARTMET_ENV_FILE
    @include "$(SMARTMET_ENV_FILE)"
@else
    @print "WARNING: The environment variable 'SMARTMET_ENV_FILE' not defined!" @location
    @throw "The environment variable 'SMARTMET_ENV_FILE' not defined!"
@endif
```

2 MODULES

2.1 Introduction

The grid engine can be divided into the following modular parts:

1. Content Server
2. Data Server
3. Query Server

The names of these parts might be a little bit confusing. Why we are callings some parts as servers? The main reason for this that they all contain well defined service interfaces that can be used locally or remotely.

When these servers are used locally, the server objects are embedded into the grid engine and the grid engine just calls methods of these server objects. When these servers are used remotely then the client objects are embedded into the grid engine and the grid engine calls methods of these client objects, which are exactly the same as the methods in server objects. In this case the client objects forward service calls to the actual servers that might be running in different computers. Both the server objects and the client objects have the same parent class, which means that the grid engine does not know whether it calls services locally (=> the server objects) or over the network (=> the client objects).

The usage of these objects is not just limited into the grid engine. Technically they are implemented in the "smartmet-library-grid-content" module, which are used also for other purposes. For example, the Content Server client objects are used in applications (radon2smartmet, filesys2smartmet) that update information in the Content Storage.

Strong modularity is one of the key principles related to the software development. This means in practice, that different functionalities should always be used through well-defined service interfaces (= APIs or higher-level protocols). On the other hand, it means that the module users should not be able to know how the actual implementation is done. For example, in our case the grid engine does not know whether its modules are embedded servers or embedded clients. It also does know what kind of databases these modules are using (Redis, Memory, PostgreSql, Oracle, etc.), because all these details are hide behind these service interfaces.

Technically this means that we can replace any of these modules with other modules as long as they have the same service interface. For example, we might replace the Content Server module implementation that uses the Redis database with the Content Server implementation that uses PostgreSql database, and other components do not even notice this, because they are still using the same service interface as earlier.

The original idea behind the distributed system was that we wanted to enable resource sharing between different SmartMet Server installation. In this way we could use SmartMet Servers also in workstations that do not necessary have so much resources. For example, the current SmartMet Server installation assumes that the server can directly access all grid files (=> network directory mounts) and that the server has so much memory that memory mapping works smoothly.

In the distributed system we can locate these kinds of services into a shared server and let workstations to access them as they were installed locally. On the other hands, when the SmartMet Servers are used in workstations, users can much easier to modify its features for they own purposes. For example, they can define their own LUA functions that they can be used in queries. They can define their own alias names for parameters or for queries. They can create their own products (wms, wfs) and tune the colors and other details in grid-gui. The point is that some researchers and users might have some special needs, which they can implement faster by themselves than asking the development or the maintenance team to implement these features for them.

2.2 Content Server

The Content Server is one of the grid engine's main modules or the service interfaces. The Content Server can be seen as a directory service that other component are using in order to find information about available producers, generations, files and grids. Technically the Contents Server offers a well-defined service interface that other components (= plugins, etc.) are using in order to access information in the Content Storage.

2.2.1 Content Storage

The Content Storage is a database that contains information about the available producers, generations, files and grids. This information is organized in the hierarchical way. The idea is that we have a list of producers, which all contain a list of generations. Each generation contains a list of files and each file contains a list of content (= grid information).

```
Producer 1
    Generation 1
        File 1
        File 2
            Content 1
            Content 2
    Generation 2
        File 3
        File 4
            Content 3
            Content 4
```

From the other components point of view the most essential information is found from the content records. These records define the location (file-id + message-index) of the grids. In addition, they define what kind of information the grids contain:

NAME	DESCRIPTION
producer-id	Grid belongs to this producer
generation-id	Grid belongs to this generation
forecast-time	This is the time of the grid (forecast) data
fmi-parameter-id	FMI parameter identifier (= number)
fmi-parameter-name	FMI parameter name (= string)
fmi-level-id	FMI level type identifier
level	Level value
geometry-id	Geometry identifier
forecast-type	Forecast type (=> ensemble forecasts)
forecast-number	Forecast number (=> ensemble forecasts)

Other components use this information when they are searching correct grids according to some criteria (for example, with the parameter name and the time interval). After that they request the actual grid data from the Data Server. In this phase they ask grid data from the Data Server according to the file-id, message-index and some coordinates.

At the moment we have two different Content Server implementation that can be used for storing information. The Redis implementation stores and searches information by using the Redis-database. The memory implementation storages information into file system and searches information from memory structures. In future, we might have an implementation that stores this information into the PostgreSql database.

At this point it is important to understand that the grid engine is not responsible for storing information into the Content Storage - it only uses this information. In the other words, there must be another process that takes care of that the Content Storage contains information that is up to date.

At the moment there are two applications that can be used for updating information in the Content Storage: 1) radon2smartmet and 2) filesys2smartmet

radon2smartmet

Finnish Meteorological Institute uses "**radon2smartmet**" application in order to keep the Content Storage up to date. This application just updates the Content Storage according to information that it gets from the other FMI's internal database (Radon). Unfortunately, this application is useless for the users who do not have the Radon database.

filesys2smartmet

The "**filesys2smartmet**" application scans filesystem directories and registers grid files into the Content Storage. This is probably the most viable option for most of the users. However, this might also require a lot of configuration. That's because the current application needs to recognize parameters in the grid files and register them into the Content Storage with such parameter identifiers (= FMI identifiers) that can be used for querying data.

These applications belong to the "smartmet-tools-grid" package. The usage and the configuration of these applications are explained in the documentation of the current package.

2.2.2 Connection

The grid engine can connect to the Content Storage in different ways. When the Content Storage is the Redis database then the simplest way to is use to the Redis implementation of the Content Server. In this case the Content Server is installed as an embedded server object that uses Redis server via TCP connection. It is important to understand that the Redis server is separate server that is not embedded into the grid engine.

In the grid engine's main configuration file, the Content Storage is references as "content-source", because it is just a content source for the Content Server.

The configuration of the grid engine's content source is quite easy (especially if the current Content Storage is already up and running). First, we need to define the type of the content source and after that we just fill the configuration parameters related to the current type.

For example, if we select the type to be "redis" then we just fill the rest of the "redis" related parameters and ignore parameters related to other types (corba, http, file).

```
content-server :
{
    content-source :
    {
        type = "redis"

        redis :
        {
            address          = "192.168.0.170"
            port             = 6379
            tablePrefix      = "a."
            secondaryAddress = "192.168.0.171"
            secondaryPort    = 6379
            reloadRequired   = false;
        }
    }
}
```

When we are using Redis implementation we can define separate connections for two different Redis servers. The idea is that if the primary Redis server goes down then the grid engine can automatically start to use the secondary Redis server.

If the primary and the secondary Redis server has identical content (= they have master-slave relationship) then servers can be switched without reloading the content information into the grid engine. However, if they have different content (i.e. they are using different feeding applications) then the grid engine should clear all its caches and reload the content information.

2.2.3 Content Cache

From the information searching point of view the Content Storage is usually quite slow, especially when the Redis server is used. That's why the grid engine usually caches all the content information into its memory. In order to do that, this caching should be enabled in the grid engine's main configuration file.

```
smartmet.engine.grid.content-server.content-cache.enabled = true
```

Cache swapping

If there is a heavy load in the server and if the information in the Content Storage changes all the time, then continuous cache updates might slow down the actual request processing. The main reason for this is that when storage structures in the memory are updated, other threads are not allowed to access the same structures at the same time. It these update operations last long or if they are done very often, it usually slows down threads that are querying information from these structures.

That's why we need to use the cache swapping in the case of heavy load. This means in practice that there are two different caches where the one is actively used for information queries, meanwhile the other is updated on the background. The updated cache is swapped to the active cache in the given interval. This swapping operation is very fast and it does not significantly disturb the information queries. The drawback is that the cached information might be few minutes old (which is not usually any problem with the weather information).

In the grid engine's main configuration file, we can enable the cache swapping and define the cache swapping interval (in seconds).

```
smartmet.engine.grid.content-server.content-cache.contentSwapEnabled     = true
smartmet.engine.grid.content-server.content-cache.contentUpdateInterval  = 180
```

Non-swapping cache

Non-swapping cache is used when the Content Storage information does not change continuously or if the query response times are not so critical. This kind of cache consumes less memory than the swapping cache. If we want to use non-swapping cache then the cache swapping should be disabled in the grid engine's main configuration file.

```
smartmet.engine.grid.content-server.content-cache.contentSwapEnabled = false
```

2.2.4 Logs

The Content Server contains two different logs: 1) the processing log and 2) the debug log

The processing log is used in order to log information about all the service requests made to it. This log contains times and durations of the service calls made to the Content Server. So, this is the easiest way to find out if some service calls are too slow. The processing log looks like this:

```
[2021-11-30/10:23:54][160][getProducerInfoList(0,ProducerInfo[46]);result 0;time 0.000016;]
[2021-11-30/10:23:54][161][getGenerationInfoList(0,GenerationInfo[1392]);result 0;time 0.000353;]
[2021-11-30/10:23:55][162][getLevelInfoList(0,LevelInfo[14175]);result 0;time 1.360386;]
[2021-11-30/10:24:35][163][getProducerInfoList(0,ProducerInfo[46]);result 0;time 0.000065;]
```

The processing log should be used only for testing or debugging purposes. In the production there are usually so much events that log writing could slow down the processing.

The processing log can be enabled in the grid engines main configuration file.

```
content-server :
{
    processing-log :
    {
        enabled         = false
        file            = "/tmp/contentServer_processing.log"
        maxSize         = 100000000
        truncateSize    = 20000000
    }
}
```

The "maxSize" parameter is the maximum size limit for the log file. After that it is automatically truncated, which means that the old data in the beginning of the file is automatically removed. The "truncateSize" is the preferred size of the log file after the truncate operation.

The debug log is used for debugging purposes. The debug log can contain whatever debug information that the developer has wanted to print into it (parameter values, code phase indicators, warnings, etc.). It is configured in the same way as the processing log.

```
content-server :
{
    debug-log :
    {
        enabled         = false
        file            = "/tmp/contentServer_debug.log"
        maxSize         = 100000000
        truncateSize    = 20000000
    }
}
```

Notice that also the Data Server and the Query Server have processing and debug logs. So, make sure that you are configuring correct logs.

2.3 Data Server

The Data Server is one of the grid engine's main modules. The Data Server can be seen as a data service that other component are using in order for fetching parameter values from the actual grid files. A typical fetch operation requires that requester gives the exact location (= file-id and message-index) of the requested grid and also the information position (= coordinates) in the grid.

Technically also the Data Server can be distributed in such way that it is running in a different server (= CORBA server). In this case the Data Server is "remote" from the grid engine's point. However, in the most of the cases the Data Server is an embedded part of the grid engine, which means that its remote usage is disabled in the grid engines main configuration file. The local caching is disabled and there is no need for IOR (International Object Reference) that is used in order to define CORBA connections.

```
data-server :
{
    Remote      = false
    caching     = false
    ior         = ""
}
```

2.3.1 Grid storage

The Data Server takes care of the memory mapping and releasing of the grid files according to the information that it gets from the Content Server. In practice, the Data Server "listens" Content Server events (= event_fileAdded, event_fileDeleted, etc.) and update its grid file mappings according to these events.

The grid filenames that come from the Content Server can usually directly used also in the Data Server. However, it is possible that these filenames do not contain the full directory paths, which means that the Data Server should complete these paths. That's why the grid engine's configuration file contains the "grid-storage.directory" parameter. The value of this parameter is added into the beginning of the filenames that come from the Content Server.

```
data-server :
{
    grid-storage :
    {
        directory                = ""
        memoryMapCheckEnabled    = false
        preloadEnabled           = false
        preloadFile              = "%(DIR)/preload.csv"
        preloadMemoryLock        = false
    }
}
```

The "memoryMapCheckedEnabled" parameter is an experimental parameter that was used in order to solve a situation where a memory mapped file was removed from the disk when it was still memory mapped. The idea was that the Data Server tried to check whether the current memory mapped file exist before it tried to access it Unfortunately this did not work well, and that's why it is usually disabled.

All preload parameters are also experimental parameters that should not be used. They probably work, but the benefits for preloading grids into memory or locking them into memory are very minimal meanwhile the drawbacks are quite significant.

2.3.2 Virtual grids

The Data Server can create "virtual grids" that can be accessed as they were real grid. The idea is that the Data Server can use existing grids in order to calculate new grids on the fly. For example, we might have grids that contains wind vectors (U and V). We can use these in order to create a new grid that contains wind speed values.

The Data Server automatically registers virtual grids into the Content Storage when the required input grids are available. After that they can be accessed as they would be real grids. On the other hand, the Data Server automatically removes these registrations when the input grids (= grids that were used for calculations) are removed from the Content Storage.

Virtual grids are a little bit slower to use and to maintain than physical grids. That's why it is not recommended to use them in heavy loaded production. However, for individual research purposes virtual grids could be useful. For example, it is possible to define virtual grids that is counted from multiple ensemble grids (for example, a probability that the temperature value is below 0 Celsius).

Virtual grids are defined in a separate configuration file, which name should be found from the grid engine's main configuration file. The definition file is a CSV file that might look like this:

```
# Wind speed
VIRT-WIND-SPEED:SMARTMET:1096:6:10;U-MS:SMARTMET:1096:6:10,V-MS:SMARTMET:1096:6:10;WIND_SPEED;9;;

# Min values
VIRT-TG-K-MIN:ECBSF:5009:0:7:3:1;TG-K:ECBSF:5009:0:7:3:1-50;MIN;9;;

# Values inside the range 273..283
VIRT-TG-K-IN-273-283:ECBSF:5009:0:7:3:1;TG-K:ECBSF:5009:0:7:3:1-50;IN_PRCNT;9;273,283;
```

The first field defines all details related to the new virtual grid parameter. The second field defines the grid parameters that are required for creating the new virtual grid. The third field is the name of the function and the fourth field is the type of the function that is used for creating the virtual grid. The fifth field contains additional parameters that are given to the current function. The idea is that we use the functions for defining multiple virtual grids.

Most common functions used for creating virtual files are implemented (for performance reasons) with C++ language. A normal user cannot create own C++ functions, but he/she can define LUA functions instead. These functions might be a little bit slower that C++ functions, but this is not significant if they are not used in mass production.

LUA functions can be written in multiple LUA files. The names of these files should be listed in the grid engine's main configuration file.

```
data-server :
{
    luaFiles = ["%(DIR)/vff_convert.lua"];
}
```

Notice that these LUA files are used in the Data Server and only for virtual file creation. The Query Server also uses own LUA files, but they are used when data is queried.

2.3.3 Logs

The Data Server has also the processing log and the debug log. These logs are congfigured in the same way as the logs used by the Content Server.

```
data-server :
{
    processing-log :
    {
        enabled         = false
        file            = "/tmp/dataServer_processing.log"
        maxSize         = 100000000
        truncateSize    = 20000000
    }

    debug-log :
    {
        enabled         = false
        file            = "/tmp/dataServer_debug.log"
        maxSize         = 100000000
        truncateSize    = 20000000
    }
}
```

2.4 Query Server

The Query Server is one of the grid engine's main modules. If offers a simple way to query information from grid files. It uses the Content Server and the Data Server in order to fetch the requested data. In practice, the Query Server contains a lot of logic that is needed in order to find correct data or to convert it into correct form. This logic needs quite a lot configuration.

The Query Server can be distributed in such way that it is running in a different server (= CORBA server). In this case the Query Server is "remote" from the grid engine's point. However, in the most of the cases the Query Server is an embedded part of the grid engine, which means that its remote usage is disabled in the grid engines main configuration file. In this case there is no need for IOR (International Object Reference) that is used in order to define CORBA connections.

```
query-server :
{
    remote = false
    ior          = ""
}
```

2.4.1 Data querying

Technically it is very easy to fetch data even without the Query Server. However, this is the case only when we know exactly what we are looking for.

In order to create an exact query, we need to define the following query parameters:

1. Producer (who is the producer of the requested data)
2. Generation (which generation we should use)
3. Parameter (which forecast parameter we are looking for)
4. Level type id (the level type id used to indicate parameter levels)
5. Level (the level value in the given level type)
6. Forecast type and number
7. Geometry (which grid geometry we should use if there are several geometries available)
8. Forecast time (time or time range of the requested data)
9. Location (the coordinates of the requested data)

Unfortunately, most of the queries contain only part of this information. That's why the most important functionality of the Query Server is to fulfill incomplete queries before fetching any data. This fulfilling is based on several configuration files.

Currently there are only two requirements for information queries. They need to contain 1) a valid parameter name and 2) valid data coordinates. Everything else can be missing.

In the case of missing information, the Query Server has different approaches depending on which information is missing.

1. Producer and/or geometry information missing

It is quite common that a user or an application does not define any producer or any geometry when it is requesting data. In this case, the Query Server needs to do this selection.

If a query does not contain any geometry definition, then the Query Server first find out all geometries that contain the requested coordinates. Otherwise, it uses given geometries.

If a query does not contain any producer definition the Query Server goes through the producer search file and selects the first producer that support one of the found geometries. Otherwise, it selects the first valid geometry available for the current producer.

The producer search file is a simple CSV-file that looks like this:

```
SMARTMET;1096;;
SMARTMETMTA;1096;;
ECG;1007;;
ECGMTA;1007;;
ECG;1008;;
ECGMTA;1008;;
```

The first field is the name of the producer and the second field is the geometry identifier. Notice that the same producer might support several different geometries. This file is automatically uploaded if it changes, which means that there is no need to restart the server.

The actual name and location of this file is defined in the grid engine's main configuration file.

```
query-server :
{
    producerFile = "%(DIR)/grid-engine/producers.csv"
}
```

2. Generation information missing

If a query does not contain any generation information, then the Query Server tries to use the newest possible generation. If the newest generation is not complete the Query Server might use the second newest generation instead. In this case, the end time parameter in the query has a significant role. The idea is that we do not fetch data from the newest generation if the newest generation does not contain data at the given query end time meanwhile the older generation does. In this case we fetch data from the older generation in spite of that the new generation might already contain some data timesteps.

Let's assume that the newest generation contains data from 04:00 - 12:00 and the second newest generation might contain data from 00:00 - 20:00. If the end time of query is in range 04:00 - 12:00 the Query Server uses the newest generation. However, if the end time is in range 12:00 - 20:00 then the Query Server uses the second oldest generation.

The Query Server picks data so that it starts from the query end time and move towards the query start time. Each picked data must come from the same or from the older generation than the previously picked data. For example, if a query requests data from time range 00:00 - 10:00 then the result contains data from the second newest generation (0:00 - 03:59) and from the newest generation (04:00 - 10:00).

If the generation information (= origintime / analysis time) is given then the data is fetched only from the current generation.

Notice that this proceeding might change in future so that the incomplete generations cannot be queried if this is configured so.

3. Level type or/and level information missing

Notice that at this point the producer, the geometry and the generation is already selected and the parameter name is defined.

If a query does not contain any level type information but has a level value, then the Query Server searches **parameter mapping files** and and tries to find a mapping that has the matching producer name, the geometry, the parameter name and the level value. If it finds this kind of mapping and if the mapping's "search match" field is enabled, then the level type information is picked from this mapping.

If the query has the level type information but not any level value, then the Query Server searches the parameter mapping files, but in this case the level value is ignored and the level type should match.

If both the level type and the level value is missing, then the Query Server searches the parameter mappings files and ignores these values.

It is important to understand that if the level information is incomplete then the mapping line order in the mapping files is very significant.

Parameter mapping files are described in more details in the chapter "Parameter mappings".

The grid engine's main configuration file contains the list of parameter mapping files that it uses.

```
query-server :
{
    mappingFiles =
    [
        "%(DIR)/mapping_fmi.csv",
        "%(DIR)/mapping_fmi_auto.csv",
        "%(DIR)/mapping_newbase.csv",
        "%(DIR)/mapping_newbase_auto.csv",
        "%(DIR)/mapping_netCdf.csv",
        "%(DIR)/mapping_netCdf_auto.csv",
        "%(DIR)/mapping_virtual.csv"
    ];
}
```

4. Forecast type or/and forecast number

The forecast type and the number is needed when there are multiple forecasts available belonging into the same generation and having the same geometry, the same parameter name and the same forecast time (=> ensemble forecasts). If this is not the case then the query does not need to contain this information.

If the queried data contains ensemble forecasts but there is no forecast type or number defined in the query, then the data can come from any grid belonging to the ensemble.

2.4.2 Query cache

The Query Server is able to cache query results into the memory. This might be useful if exactly same query is repeated continuously. Unfortunately, this does not seem to be so common in real production in spite of that we could cache thousands of query results. In other words, the benefits of caching query results are not usually any significant compared to the increased memory consumption. That' why the query cache is usually disabled. A query result is automatically removed from the cache if it has not been accessed in the given "maxAge" time (= seconds).

```
queryCache :
{
    enabled = false
    maxAge = 300
}
```

2.4.3 LUA functions

When parameters are requested from the Query Server, the requested parameter can contain a LUA function. This means that the value of the actual parameter is sent to a LUA function and this function returns a new value for the parameter.

For example, when we request the "T-K" parameter from the Query Server, we get temperature value in Kelvins. If we want to get temperature value in Celsius, we can request the same parameter with the "K2C" function that does the required conversion.

The current “K2C” function is defined like this:

```
function K2C(numOfParams,params)
    local result = {};
    If (numOfParams == 1) then
        result.message = 'OK';
        if (params[1] ~= ParamValueMissing) then
            result.value = params[1] - 273.15;
        else
            result.value = ParamValueMissing;
        end
    else
        result.message = 'Invalid number of parameters given ('..numOfParams..')!';
        result.value = 0;
    end
    return result.value,result.message;
end
```

LUA functions can be defined in multiple files. If these files change then updates LUA files are automatically loaded into the server. There is no need to restart the server.

The grid-engine's main configuration file contains the list of LUA files that it uses. This list is not automatically uploaded, which means that the modification of this list requires that the server is restarted. However, modification of the actual LUA files does not require the restart.

```
query-server :
{
    luaFiles =
    [
        "%(DIR)/function_basic.lua",
        "%(DIR)/function_conversion.lua"
    ];
}
```

2.4.4 Parameter aliases

Sometimes query parameters can be quite complex and long. Especially if a parameter contains function calls. That's why we are able to define alias names for this kind of parameters in separate alias files. An alias file is a simple text file that might look like this:

```
Temperature:K2C{T-K:::6:2}
Humidity:RH-PRCNT:SMARTMET:1096:6:2

SSI:SummerSimmerIndex{$Humidity;$Temperature}
```

The first field is the alias name (Temperature, Humidity, SSI) and the rest of the line is the actual content of the current alias. In other words, the alias name is replaced with this content.

The Query Server automatically uploads alias files in few seconds after they have been updated. There is no need to restart the server. The grid-engine's main configuration file contains the list of alias files that it uses. This list is not automatically uploaded, which means that the modification of this list requires that the server is restarted. However, modification of the actual alias files does not require the restart.

```
query-server :
{
    aliasFiles =
    [
        "%(DIR)/alias_demo.cfg",
        "%(DIR)/alias_newbase_extension.cfg"
    ];
}
```

2.4.5 Logs

The Query Server has also the processing log and the debug log. These logs are configured in the same way as the logs used by the Content Server and the Data Server.

```
query-server :
{
    processing-log :
    {
        enabled         = false
        file            = "/tmp/queryServer_processing.log"
        maxSize         = 100000000
        truncateSize    = 20000000
    }

    debug-log :
    {
        enabled         = false
        file            = "/tmp/queryServer_debug.log"
        maxSize         = 100000000
        truncateSize    = 20000000
    }
}
```

3 PARAMETER MAPPINGS

3.1 Introduction

Parameter mappings are defined in the parameter mapping files. Parameter mappings are needed for several purposes.

1. Mapping parameter names to the names that are used in the Content Storage

It's quite common that users and application want to use different parameter names for querying information rather than the names that were used in the Content Storage. However, if we want to search parameters then we need to use the same identifiers that are used in the Content Storage. This means that if we want to use different parameter names (like Newbase of NetCDF names) then we should map these names to the names that are used in the Content Storage (=> FMI names).

2. Defining conversions used with parameters mappings

Data conversions are needed if there is not exactly same data available that we want to request. For example, we might want to request temperature values in Celsius, but there is only temperature values in Kelvins available. In this case we should define a conversion function so that the Query Engine can convert data values "on-the-fly" when the data is requested.

3. Defining interpolation methods used with different parameters

Parameter data can usually be requested at any geographical point, at any level and at any time. In practice, we do not have this kind of data, but we can interpolate it from the closest grid points. We can interpolate data in three different ways: we can do 1) area interpolation, 2) level interpolation and 3) time interpolation. Different parameters are interpolated in different ways. For example, "Temperature" values might be interpolated linearly, meanwhile the "SoilType" parameter should not be interpolated at all, because the soil type values are integer numbers that identifies the soil type. If we interpolate these kinds of values, we get decimal values like “3.52”, which is not a soil type. In mapping files, we can define default interpolation methods for each parameter.

4. Defining climatological parameters

The parameters mapping files can be used for defining some parameters as "climatological" parameters, which means that the Query Server ignores the year value when these parameters are queried. In other words, in spite of that the climatological data is stored as year "yyyy" data, it can be queried by using any year value in the request.

5. Defining parameter search order

It is quite common that a user or an application requests parameters without any details (= geometry, level type, level, etc.). In this case the Query Server returns the first matching parameter if the current parameter has its "search match" field enabled. This means that also the order of the mapping lines might be important is some cases.

6. Defining default precisions for different parameters

When a parameter value is printed it is useful to know the preferred precision of the current parameter. This can be defined in the parameter mapping files.

3.2 Mapping files

All mapping files are CSV files that contains multiple fields separated by ';' character. Each mapping lines contains the following fields:

```
1. The name of the producer (ECG).
2. The name of the parameter that we want to map (for example "Temperature")
3. The type of the target parameter (2 = FMI name)
4. The name of the target parameter (for example "T-K").
5. Geometry identifier (for example 1007)
6. The level id type (1 = FMI)
7. The level id (1 = surface, 2 = pressure, 3 = hybrid, 4 = altitude, etc.)
8. The level value
9.  Area interpolation method (0 = none, 1 = linear, 2 = nearest, 3 = min, 4 = max, 9 = landscape)
10. Time interpolation method (0 = none, 1 = linear, 2 = nearest, 3 = min, 4 = max)
11. Area interpolation method (0 = none, 1 = linear, 2 = nearest, 3 = min, 4 = max, 5 = logarithmic)
12. Group flags (bit 0: climatological parameter)
13. Search match (E = enabled, D = disabled, I = ignore)
14. Conversion function (for example, "SUM($,-273.15)")
15. Reverse conversion function (for example, "SUM($,273.15)")
16. Default precision
```

A typical mapping file looks like this.

```
ECG;Temperature;2;T-K;1007;1;1;00000;1;1;1;0;E;SUM{$,-273.15};SUM{$,273.15};1;
ECG;Temperature;2;T-K;1007;1;2;25000;1;1;2;0;D;SUM{$,-273.15};SUM{$,273.15};1;
ECG;TemperatureSea;2;TSEA-K;1007;1;1;00000;1;1;1;0;E;SUM{$,-273.15};SUM{$,273.15};1;
ECG;TemperatureSea;2;TSEA-K;1008;1;1;00000;1;1;1;0;E;SUM{$,-273.15};SUM{$,273.15};1;
ECG;TotalColumnWater;2;TCW-KGM2;1007;1;1;00000;1;1;1;0;E;;;;
```

In this example (= the first line), we define that the producer ECG has a parameter named as "Temperature", which is actually the parameter "T-K" that needs to be converted from Kelvins to Celsius.

The only differences between the first and the second line is that the first line defines mapping for the ground level temperature (levelType = 1, level = 0) meanwhile the second line defines mapping for the temperature at the pressure level 25000. In this case the level interpolation is set to 2 (= nearest).

In the last line the parameter name "TotalColumnWater" is directly mapped into the "TCW-KGM2" parameter without any conversions.

3.3 Automated mapping file generation

Creating thousands of parameter mappings manually does not make any sense. That's why the grid engine automatically generates basic mapping files according to the parameter information it gets from the Content Server. Currently the grid engine generates automatically three different parameter mapping files. This is defined in the grid engine's main configuration file.

```
mappingTargetKeyType = 2

mappingUpdateFile :
{
    fmi         = "%(DIR)/mapping_fmi_auto.csv"
    newbase     = "%(DIR)/mapping_newbase_auto.csv"
    netCdf      = "%(DIR)/mapping_netCdf_auto.csv"
}
```

The "fmi" file contains all FMI parameters found from the Content Server. The primary purpose of this file is to tell what parameters are available. From the mapping point of view, it is not so informative, because the mapping names are same as the parameter names. On the other hands, this file is needed because it also defines which interpolation methods should be used when the current parameters are requested.

The "newbase" file contains mappings for Newbase parameter names. These mappings are based on mappings that are defined in the "smartmet-library-grid-files" module, where these mappings are needed in order to identify parameters in grib, netcdf and querydata files. This information is used in order to generate parameter mappings files needed by the grid engine. The mapping in the "netCdf" file is created in the same way.

These automatically generated mapping files are not necessary fully correct, or there might be missing some information that we want to define by ourselves (like interpolation methods or conversion functions). On the other hand, these files are overwritten every time when new generation is created (about every 3-5 minutes). That's why we should move parameter mappings from these automatically generated files to more permanent mapping files. When a mapping is move into a permanent mapping file it disappears from the automatically generated file.

4 NEWBASE MAPPINGS

4.1 Introduction

Finnish Meteorological Institute (FMI) has a lot of applications that are using so called Newbase producer and parameter names that do not match the producer and the parameter names that are used in the Radon database (= FMI's internal database). However, in order to keep these applications running there must be a way to fetch information by using these Newbase names. If you do not have these kinds on requirements, we recommend that you pass this section, because this is very complex issue to understand.

4.2 Producer and parameter mapping

The Newbase producer names contain encoded information about the producer, the geometry and the level type (for example, "ecwmf_europe_surface", "ecwmf_world_pressure", etc.), meanwhile the Radon producer names do not contain this kind of encoding. For example, there is just one ECG producer that contains multiple geometries and multiple level types. Technically this means that when using the Newbase names, the producer, the geometry and the level type is automatically given. However, when using the Radon names, the level type and the geometry should be given separately.

On the other hand, the Radon database uses different producer names in order to separate the raw data from the postprocessed data (for example, "ECG", "ECGMTA"), meanwhile there is no similar separation in Newbase producers. This all means that we have to define "many-to-many" mappings if we want to continue using Newbase producer and parameter names. For example, we should define mappings between the following Newbase and Radon producers.

Newbase	**Radon**
ecwmf_scandinavia_surface	ECG
ecwmf_europe_surface	ECGMTA
ecwmf_world_surface	
ecwmf_scandinavia_pressure	
ecwmf_europe_pressure	
ecwmf_world_pressure	

In practice, we have to define mappings for each Newbase producer and parameter. For example, the Newbase producer "ecwmf_europe_pressure" has "Temperature" and "WindSpeedMS" parameters, which must be mapped to different Radon producers:

```
ecmwf_europe_pressure:Temperature             => ECG:Temperature
ecmwf_europe_pressure:WindSpeedMS             => ECGMTA:WindSpeedMS
```

Unfortunately, this is not enough. The grid data (= in Querydata files) used by the Newbase parameters do not always match to the original data (= in GRIB files) in spite of the Querydata files are usually generated from the GRIB files. For example, there is no "Temperature" (= temperature in Celsius) parameters in the Radon database. However, there is a parameter called "T-K" (= temperature in Kelvins). This means that we should be able to convert the original grid values into units used with the Newbase parameters. This all requires a lot of configuration.

In addition, it is also possible that different producers express the same information in different values. For example, usually "relative humidity" is expressed in per cents. Some producers think that this is a number between 0 ... 100, meanwhile other producers might think that it is a number between 0.0 ... 1.0. Usually the parameter name should tell, which units or number ranges are used. Unfortunately, this is not always the case. That's why we must be able to "tune" individual parameters for individual producers.

4.3 Configuration

Newbase producer and parameter mappings can be divided into three phases. First, we define mappings from the Newbase producers' parameters into correct FMI producers, geometries, level types and levels. Notice that in this phase we do not define mappings between the actual parameters (Temperature => T-C). We just define from where these parameters should be found (producer, geometry, level type, level, forecast type, forecast number).

For example, the Newbase producer "**ecmwf_europe_pressure**" has the "**Temperature**" parameter which should be found from the "**ECG"** producer and the "**WindSpeedMS**" parameter which should be found from the "**ECGMTA**" producer. In both cases the geometry is 1008 (= Europe) and the level type identifier is 2 (= pressure level). In this case there is no need to map any other information (like the actual level, forecast type or forecast number).

The Newbase parameter mappings are usually defined in multiple mapping files. Typically, each Newbase producer has own mapping file. In the previous example, we could have a mapping file called "pm_ecmwf_europe_pressure.cfg" that contains the required mappings.

```
# ecmwf_europe_pressure:Temperature => producer=ECG,geometry=1008 (europe),levelType=2 (pressure)
ecmwf_eurooppa_pressure;Temperature:ECG;1008;2;;;;

# ecmwf_europe_pressure:WindSpeedMS => producer=ECG,geometry=1008 (europe),levelType=2 (pressure)
ecmwf_eurooppa_pressure;WindSpeedMS:ECGMTA;1008;2;;;;
```

It is possible that the current producer has also some alias names. For example, "**ecmwf_eurooppa_painepinta**" is a Finnish version of "**ecmwf_europe_pressure**". If we do not want to define the same parameter mappings multiple times, then we should add alias definitions into the same mappings file.

```
ecmwf_eurooppa_painepinta:ecmwf_eurooppa_pressure
```

Notice that all used mapping files should be listed in the grid-engine's main configuration file.

```
query-server :
{
     producerMappingFiles =
     [
          "%(DIR)/grid-engine/pm_ecmwf_europe_pressure.cfg",
          "%(DIR)/grid-engine/pm_ecmwf_world_surface.cfg"
     ];
}
```

So far, we know where (= FMI producer, geometry, level type identifier, etc.) to look at the requested Newbase parameters. For example, we know that the "ecmwf_europe_pressure" producer's "Temperature" parameter should be found from the "ECG" producer's geometry 1008 and the level type identifier 2. On the other hand, we know that there is no such FMI parameter as "Temperature".

In the second phase, we should make sure that there are parameter mappings from these Newbase parameter names to FMI parameter names. This is something we already did in the previous chapter.

In the last phase, we define rules for available Newbase producers and generations. If a Newbase producer is mapped into multiple FMI producers, all these FMI producers should have the same generations / analysis times available. The point is that it does not make sense to map parameters if they are coming from different generations / analysis times. That's why we need rules that define which Newbase producers and generations are available.

For example, if the FMI producers “ECG” and “ECGMTA” have the same generations available, then these generations are available also for the Newbase producer “ecmwf_world_surface”. On the other hand, if a generation is available only for one of these FMI producers, then it is not available for the current Newbase producer. This availability information is usually used for triggering different kinds of scripts (for example, data transfers, image generations, etc.).

These availability rules are defined in a simple CSV-file, which is shown below. The first field is the name of the Newbase producer and the rest of the fields define the related FMI producers.

```
ecwmf_world_surface;ECG;ECGMTA
ecwmf_world_pressure;ECG;ECGMTA
pal_scandinavia;SMARTMET;SMARTMETMTA
```

The name of this CSV-file is defined in the grid-engine’s main configuration file.

```
query-server :
{
    producerStatusFile = "%(DIR)/grid-engine/stat_newbase_producers.cfg"
}
```