EGEE

R-GMA System Specification Version

6.2.0

July 2, 2010

Abstract: This document presents the System Specification for the Information and Monitoring Services middleware component (R-GMA) in sufficient detail to support design verification, detailed design and test specification. It specifies precisely the interfaces and externally visible behaviour of all the R-GMA services.

The document is structured with a single chapter containing a technical overview of the system, followed by a separate chapter for each of the R-GMA services, with a standard set of headings in each of these chapters. Chapters on security, SQL language support, service data types and service parameters complete the document.

CONTENTS

1 INTRODUCTION

1.1 PURPOSE AND STRUCTURE OF THIS DOCUMENT

This document presents the System Specification for an Information and Monitoring Services middleware component (R-GMA), in sufficient detail to support the following activities:

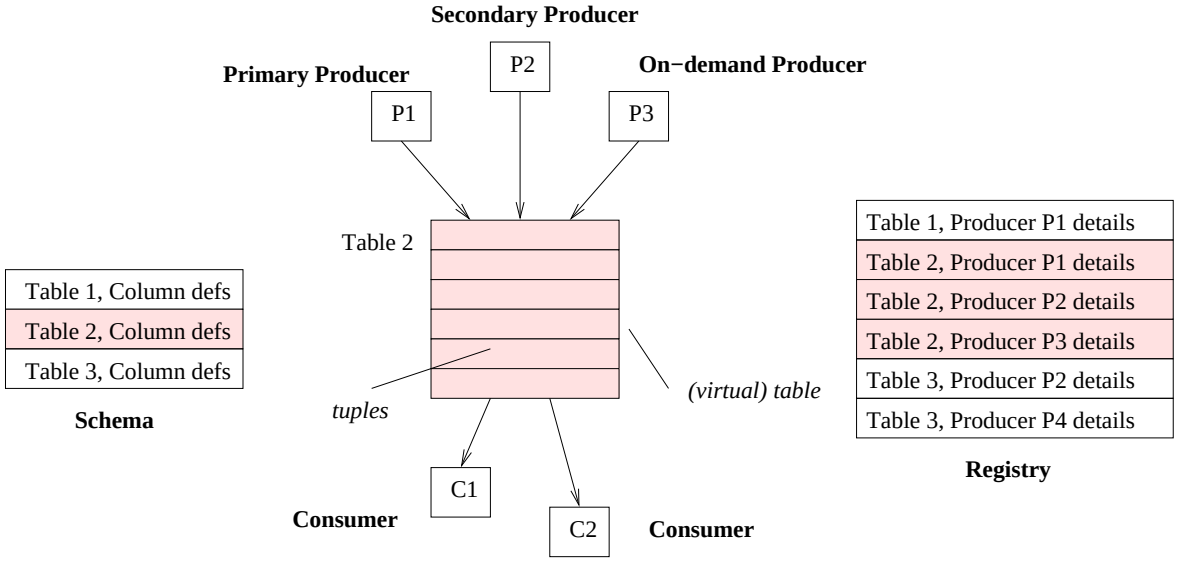
- Design verification: This document sets out precisely *what R-GMA will do*, so that it can be verified that it will provide the required services to other middleware components and Grid users.
- System design: Detailed designs will show exactly *how* the system specified here will be implemented.
- System test: This document is the primary input to test specification activities.

This chapter is followed by one describing message formats then a separate chapter for each of the R-GMA services, with a standard set of headings in each chapter. Chapters on security, SQL language support, service data types and service parameters complete the document.

1.2 CONCEPTS

1.2.1 VIRTUAL DATABASE

R-GMA enables Grid users to share information in a *virtual database (VDB)*. To the ordinary user a virtual database looks much like a real one. It organises its data into tables, and data can be inserted and queried using standard SQL constructs. It even supports indexes and views. However there is no central repository holding the data for each table. A virtual database just consists of a list of table definitions (called a *schema*), a list of data providers (called a *registry*) and a set of rules for deciding which data providers to contact for any given query. These rules are fixed by R-GMA, and are encoded into a hidden component called the *mediator* and described in section 7.



The picture above shows the principal components of R-GMA. Data is written into the virtual database by *producers* and read from it by *consumers*. The model used by R-GMA, where consumers contact the registry to obtain a list of producers for their query then contact the producers directly to get the data, is

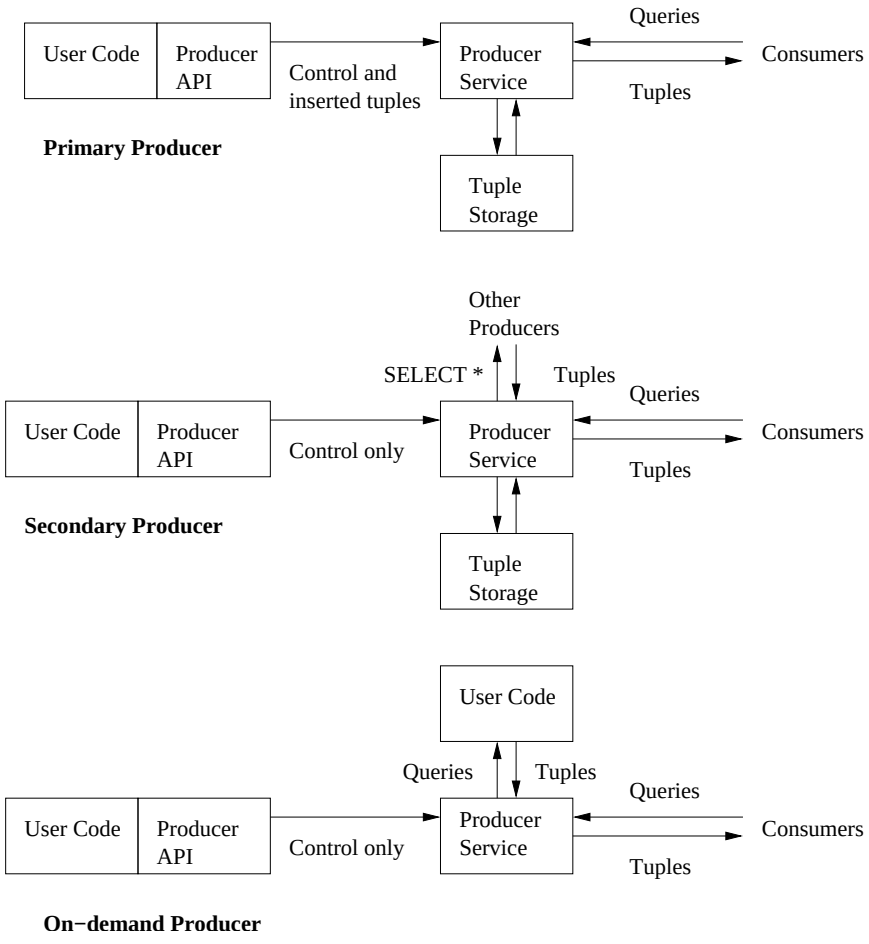
known as the Grid Monitoring Architecture (GMA) and was first described by the Open Grid Forum (or Global Grid Forum as it as at that time).

A single Grid can support any number of virtual databases. Each virtual database has a Grid-wide unique name. Typically, a virtual database will be owned and managed by a *virtual organization* (VO), but R-GMA doesn't require this to be the case.

R-GMA is not a distributed database management system. Instead, it provides a useful and predictable information system built on a much looser coupling of data providers across a Grid. This document provides a precise description of that information system.

1.2.2 PRODUCERS

Producers are the data providers for the virtual database. Writing data into the virtual database is known as *publishing*, and data is always published in complete rows, known as *tuples*. There are three classes of producer: *Primary*, *Secondary* and *On-demand*. Each is created by a user application and returns tuples in response to queries from consumers. As the picture below shows, the main difference is in where the tuples come from.



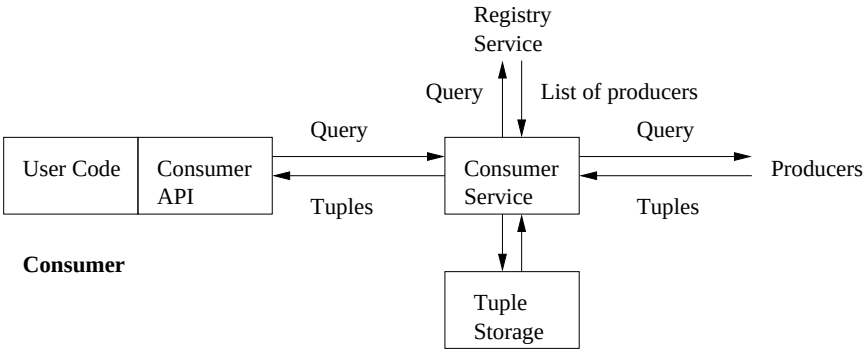
The *producer service* in these pictures is a process running on a server, acting on behalf of the user code (this is explained in the Services Architecture in section 1.3). In a Primary Producer, the user code periodically inserts tuples into storage maintained internally by the producer service. The producer service autonomously answers consumer queries from this storage. The Secondary Producer service also answers queries from its internal storage, but it populates this storage itself by running its own

query against the virtual database: the user code sets the process running and the tuples come from other producers. In the On-demand Producer, there is no internal storage; data is provided by the user code in direct response to a query forwarded to it by the producer service.

The tuple storage maintained by Primary and Secondary producers can either be in memory or in a real database. Producers that use memory storage are optimized to answer simple queries quickly, but they must still be able to answer the same queries as producers using database storage, even if that means creating an in-memory database on the fly. In an On-demand producer, tuple storage, if any, is outside of R-GMA and is the responsibility of the user code.

1.2.3 CONSUMERS

In R-GMA, each consumer represents a single SQL SELECT query on the virtual database. The request is initiated by user code, but the *consumer service* carries out all of the work on its behalf. The query is first matched against the list of available producers in the Registry and a set of producers capable of answering the query is selected, in a process called *mediation*. The consumer service then sends the query directly to each selected producer to obtain the answer tuples.



1.2.4 QUERY TYPES

There are four types of query: *continuous*, *latest*, *history* and *static*. The set of queries that a particular producer supports is recorded in the registry. All query types except static can take an optional time interval parameter.

All four types of query cause tuples to be streamed from producer services to the consumer service that is running the query on the consumer application's behalf. The consumer application can then pull the tuples from the consumer service in its own time.

Continuous queries are subscriptions with producers to receive all *new* tuples that match the query; tuples are streamed to the consumer service automatically as they are inserted into the virtual database by the Primary producers. Streaming continues until the consumer requests it to stop. Continuous consumers are also registered in the Registry so they can be notified if the list of relevant producers changes during the lifetime of the query. If a time interval is specified, the producers selected for streaming will additionally send any tuples they already have, that are not older than the time interval. There is no guarantee that tuples are time-ordered. All Primary and Secondary producers support continuous queries. On-demand producers do not.

Latest and history queries are *one-time* queries: they are evaluated against the current contents of the virtual database, the results are streamed back to the consumer service, and then they terminate. In a history-query, all versions of any matching tuples are returned; in a latest-query, only those representing the "current state" (see below) are returned. In both cases, a time interval may be specified with the

query, to limit the age of the tuples returned, as in a continuous query. Primary and Secondary Producers may optionally support one-time queries. On-demand producers do not.

Static queries are one-off database queries and are only supported by On-demand producers. The purpose of On-demand producers is to make external data stores accessible through the R-GMA infrastructure so it's unlikely that an On-demand producer will publish to the same table as Primary and Secondary producers.

1.2.5 TUPLE MANAGEMENT

All tables in an R-GMA schema have several metadata columns added to the table definition by R-GMA when the table is created in the schema (see 3.3.4 for the column definitions). These columns are filled in when tuples are inserted into a Primary producer and include a time-stamp added by R-GMA if the user has not already done so. Secondary producers do not modify tuple metadata in the tuples they republish. On-demand producers put null values into the metadata columns of all tuples they publish.

Primary and Secondary producers have one or two tuple stores, a *history tuple store* against which history queries are evaluated (and from which tuples are retrieved for continuous queries), and an optional *latest tuple store* against which latest queries are evaluated. On-demand producers do not store tuples.

Each tuple inserted into a primary or secondary producer added to the producer's history tuple store, regardless of what is already there: no attempt is made to detect duplicates. If applicable, it is also added to the producer's latest tuple store, provided it is not older than any previous version already there, which means its time-stamp must be no earlier than any tuple with the same *primary key* that is already in the tuple-store. If an older version exists, it is replaced by the new one. The primary key is a subset of the columns in the table, and is defined when the table is first created in the schema. This is the only place in R-GMA where the the primary key is used.

Primary producers declare a *Latest Retention Period (LRP)* on each table to which they publish tuples. The LRP is used by the primary producer to calculate a *latest retention time* for each tuple, by adding the LRP to the tuple's time-stamp. This is stored in the tuple's metadata and remains unaltered when the tuple is republished by a Secondary Producer. A latest-query is evaluated against only the most recent versions of tuples, and only those tuples that have not exceeded their latest retention time (this is the definition of current state).

Both primary and secondary producers declare a *History Retention Period (HRP)* on each table to which they publish tuples. The HRP is considered to be relative to the time at which the tuple is stored in the producer and is not relative to the time stamp on a tuple. A history-query is evaluated against whatever is available, but that is guaranteed to include at least all versions of tuples that have not exceeded the producer's HRP for the table.

Retention periods do not affect the delivery of tuples to continuous consumers: it is fundamental to the mediation logic of R-GMA that its producer services ensure that all new tuples are streamed to all subscribed consumers, unless there is an unrecoverable fault that prevents it.

Producers are obliged to periodically remove expired tuples from their tuple stores so that they do not run out of storage space. The exact details of how retention periods work for each producer type are specified in the corresponding chapters in this document.

1.3 SERVICE ARCHITECTURE

1.3.1 SERVICES

R-GMA is implemented as a set of six types of *service* running on one or more servers. They are Primary Producer, Secondary Producer, On-demand Producer, Consumer, Registry and Schema. The virtual

database is realised by the interaction of these services; users contact their local services to publish data or run queries on the virtual database, and the service translates this into a sequence of operations to be carried out locally, or by contacting other R-GMA services as necessary, on their behalf. Each service has a well defined set of *operations* that are requested by applications through an exchange of messages with the service. The semantics of the operations and their parameters are defined in subsequent chapters in this document. The operations in each service are grouped into *user operations* and *system operations*. User operations are used by client applications and (some) by other R-GMA services. System operations are used by other R-GMA services only.

R-GMA uses https calls in a request/response pattern, for all user-to-service and service-to-service communications apart from streaming. Streaming tuples to continuous consumers uses a lower level communication protocol, for efficiency (see section 3.3.6).

1.3.2 RESOURCES

Connections to R-GMA services only last for the duration of a single operation, but in producer and consumer services, R-GMA needs to retain private data *between* operations, for each producer or consumer instance currently being managed by the service. In a Primary or Secondary producer, for example, this includes the tuple stores. R-GMA stores the private data associated with each producer and consumer in a *resource*. It is created by the service when a user calls a "create" operation and is given an identifier that is passed back to the client. The client then includes the resource identifier with all subsequent requests relating to this resource. Resource identifiers are positive integers that are unique within any given service and are not re-used. A resource is normally closed/destroyed at the explicit request of the user, but in order to protect itself from an accumulation of redundant resources, an R-GMA service requires the user to periodically contact the service to keep the resource alive. A resource has a *termination interval*; if the service doesn't hear from the user for any period exceeding the termination interval, the resource is closed by the service. For most purposes the APIs hide the need to know about termination intervals - though the server can be queried to discover what termination intervals are being used.

The registry protects itself in a similar way, against producers and consumers that register then disappear, so a periodic message is also sent to the registry by the consumer and producer services, on the user's behalf to keep the producers and consumers registered.

1.3.3 SERVICE APIS

R-GMA provides APIs for Java, C++, C and Python languages, to make it easier for user applications to interact with the R-GMA services. The APIs are independent of each other and are designed to present an apropriate interface to R-GMA for each supported language. Each API contains a method for each user operation of every service, and that method simply packages up its parameters into an https request and sends it to the service for execution. Any return values or errors are passed back to the caller. The API transparently manages any authentication required by the server, and looks after the resource identifier.

All R-GMA services report errors by sending exceptions, which are described fully in 1.4.5. Each API presents service and API exceptions to the user in a manner appropriate to the language, for example as Java Exceptions. The absence of an exception indicates success. The results of SQL queries (tuples) are returned in *tuple sets*.

The server may report to the API that a resource is no longer available. In this case it is the responsibility of the API to create a new resource in the same state as the old one. In the case of a consumer a warning is attached to the next tuple set so that the user is made aware as some data loss may have occured.

1.3.4 SERVICE SECURITY

Security for R-GMA is discussed in detail, in section 13.

1.4 FAULT TOLERANCE

1.4.1 REGISTRY AND SCHEMA REPLICATION

The schema and registry represent the only single point of failure in a virtual database, so multiple replicas are maintained in different Registry and Schema services in the Grid. Each service can host the registries and schemas of multiple virtual databases. Replicas are synchronized according to the rules in sections 9.3.3 and 10.2.3. Users, producer services and consumer services always contact their local Registry and Schema services, and it's the responsibility of those services to locate working replicas and switch as necessary following failure.

1.4.2 SERVICE FAILURE AND RECOVERY

R-GMA producer and consumer resources are always destroyed when the service hosting them stops or restarts. All calls to services in R-GMA, either from the user or another service, must therefore be prepared to discover that a resource no longer exists, or is no longer registered in the registry, and to handle the error gracefully. The use of time-outs (soft-state registration) in the registry ensures entries for resources that no longer exist are automatically removed within a reasonable length of time.

Those R-GMA services that use permanent storage (registry service, schema service and primary and secondary producers with permanent tuple stores (see, for example, 3.3.2)) do have some degree of resilience, because new resources can reconnect to existing storage, provided the storage itself is recoverable. In the case of the registry and schema, the database is brought up to date by the replication mechanisms. In the case of a permanent tuple-store, a producer created using the existing store will automatically make tuples already in the tuple-store available to consumers, and they will remain in the store until they exceed their retention period, in the usual way. Backing up permanent storage (e.g. database backups) is outside the scope of R-GMA's responsibility.

1.4.3 TRAFFIC JAMS

Processing a single operation in R-GMA can involve interactions between several services. In addition, each service can expect to handle a large number of simultaneous client requests. It can also be expected that at any one time there may be a number of clients and services that are slow to respond or even block completely. R-GMA service implementations must ensure therefore, through the use of queues, thread pools, connection timeouts and other devices, that all correctly functioning clients and services get a reasonable response, and a small number of blocked connections are not able to seriously disrupt the overall operation of the service. Deliberate attempts to disrupt a service (denial-of-service attacks) are discussed in the chapter on security.

1.4.4 CLOCK SYNCHRONIZATION

As far as possible relative rather than absolute times are passed between R-GMA servers however some absolute times are passed so any servers running R-GMA services should keep their clocks synchronized using a protocol such as NTP.

1.4.5 ERROR REPORTING

R-GMA services use the following exceptions for reporting all errors.

RGMAPermanentException Any kind of error when it is unlikely that repeating the call will lead to success. This might be caused for example by bad parameters in the call from the user or a corrupt database.

RGMATemporaryException An error where it is probable that repeating the call may be sucessul. This might be caused by memory being low or the server becoming too heavily loaded.

UnknownResourceException Resource identifier is unknown. This will not be passed back to the user via the API but the API will create a new resource.

RGMAPermanentException and RGMATemporaryException are implemented as subclasses of RGMAException by those API languages that support the concept.

In the case of fatal errors, external resources (especially database and network connections) are freed up and any threads associated with the resource are terminated as cleanly as possible.

Warnings are only issued in R-GMA in response to consumer queries, and only to indicate possible inadequacies in the results. They are appended to the query's result set.

1.5 BOOTSTRAPPING

R-GMA servers run all R-GMA services. Users only have to be able to locate and connect to a single server, usually at their own site to avoid firewall problems, in order to use R-GMA. All APIs locate these services using a file containing their URLs that is configured when the APIs are installed, and is unlikely to change.

Producer and Consumer services only connect to other Producer and Consumer services through URLs passed to them in system calls, or to the Registry and Schema services on their own server. The Registry and Schema services will act as proxies if requests need to be forwarded to other Registry and Schema services.

Registry and Schema services need to be able to locate other Registry and Schema services, firstly to be able to forward requests to virtual databases that they're not hosting locally, and also to be able to replicate the registries and schemas that they are hosting locally. They obtain this information from files that maps virtual database names to Registry and Schema service URLs for all virtual databases supported at that site.

2 MESSAGE FORMATS

2.1 DATA TYPES

An important part of these chapters is the list of operations and their input and output parameters. Input parameters are prefixed by $\Rightarrow$ and output parameters by $\Leftarrow$. Parameters may be omitted or repeated as indicated by the notation:

(1..1) Exactly once - the default

(0..1) May be omitted

(1..*) At least one

(0..*) Any number

after the basic type name. Types take the following values:

xsd:string a sequence of Unicode characters

xsd:boolean the literals *true* or *false* encoded as strings

xsd:int an integer in the range $[-2^{31}, 2^{31}-1]$

xsd:long an integer in the range $[-2^{63}, 2^{63}-1]$

Tuple a tuple. This is followed by a list of the fields within the tuple. These can be of any of the types mentioned - except for another tuple.

2.2 REQUEST

Simple requests (such as xsd:string) values are sent as http request parameters (either POST or GET) and Tuples are encoded as one or more result sets within an XML string sent as a single http parameter.

2.3 RESPONSE

The output is sent as an XML encoding of one or more R-GMA tuple sets.

Errors are also encoded as XML.

2.4 XML FORMATS

2.4.1 TUPLE SET

A single result set looks like:

```
<r m="Be warned" r="2" c="2">
  <v>Row 1 Col 1</v>
  <v>Row 1 Col 2</v>
  <n/>
  <v></v>
  <e/>
</r>
```

The result set has an (optional) and has 2 rows and 2 columns as indicate by the r and c attributes which default to 1. The data values then follow inside <v></v> or <n/> to indicate the null value. The <e/> indicates that there is no more data.

With these rules a simple "OK" message is just

```
<r><v>OK</v><e/></r>
```

Multiple tuple sets must be wrapped in <s></s> as shown below:

```
<s>
  <r><v>OK</v><e/></r>
  <r><v>OK</v><e/></r>
</s>
```

2.4.2 ERRORS

A temporary exception is represented as:

```
<t m="This may recover" o="1"/>
```

where the o attribute shows the number of succesfgul operations and defaults to 0.

A permanent exception uses p instead of t. For example:

```
<p m="This will not recover"/>
```

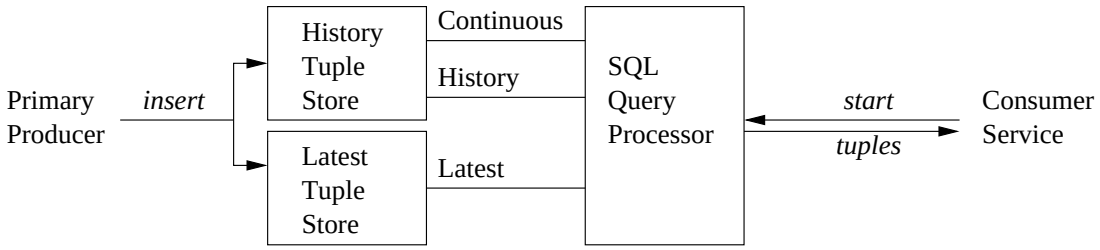
The unknown resource message is very simple - it is just:

```
<u/>
```

3 PRIMARY PRODUCER SERVICE

3.1 DESCRIPTION

Primary Producer resources are created by the Primary Producer Service at the request of a user who wishes to publish tuples into to one or more tables in a virtual database. The principal components of a Primary Producer resource are shown in the picture below. They contain tuple stores to hold tuples inserted by the user, and they have an SQL query processor to run consumer queries against their tuple stores. They are the primary source of data in a virtual database. They all support continuous queries, and can be configured to support any combination of latest and history queries as well.



The Primary Producer Service is responsible for authenticating all users and services that connect to it, and for authorizing all operations and all requests to access its tuple stores, as specified in chapter 13.

3.2 INTERFACE

3.2.1 USER INTERFACE

createPrimaryProducer

⇒	xsd:boolean isHistory	If history queries are supported
⇒	xsd:boolean isLatest	If latest queries are supported
⇒	xsd:string type	DATABASE or MEMORY
⇒	xsd:string logicalName	The logical name should not be be specified for type MEMORY. For type DATABASE it is optional.
⇐	Tuple(1..1)	
	xsd:int connectionId	connectionId of new Primary Producer resource.

Creates a new Primary Producer resource and returns its endpoint. The Primary Producer is not added to a Registry until *declareTable* is called. The user cannot prevent Primary Producers from supporting continuous queries, but support for latest and/or history queries is optional. Primary Producers can use any type of storage, regardless of the query types they support. Tuple stores can be made permanent by providing a *logical name* for the tuple store as described in 3.3.2. The termination interval is described in 3.3.1.

declareTable

⇒	xsd:int connectionId	Primary Producer resource identifier.
⇒	xsd:string tableName	VDBTable to register.
⇒	xsd:string predicate	Producer's predicate.
⇒	xsd:int hrpSec	History Retention Period in seconds.
⇒	xsd:int lrpSec	Latest Retention Period in seconds.
⇐	Tuple(1..1)	
	xsd:string status	OK

Adds a table to the list of tables to which this producer may publish tuples, as described in 3.3.3 below. The table name must have an explicit virtual database name prefix (separated from it by a dot). The format of the predicate is specified in 14.8, and the retention periods are described in 3.3.5 (the latest retention period can be overridden in *insert*).

insert

⇒	xsd:int connectionId	Primary Producer resource identifier.
⇒	xsd:string insert	SQL INSERT statement.
⇒	xsd:int(0..1) lrpSec	Latest Retention Period in seconds.
⇐	Tuple(1..1)	
	xsd:string status	OK

Inserts one or more tuples into a primary producer resource's tuple stores, as described in 3.3.4. The format of the insert statement is specified in 14.7; as with *declareTable*, the table name must have an explicit virtual database name prefix. Inserted tuples must match the producer's predicate and the table schema or they will be rejected. If the latest retention period is omitted, the default specified in the call to *declareTable* will be used. If a tuple is found to be invalid or cannot be inserted for any reason, the operation will stop and throw an RGMAException indicating the number of tuples successfully inserted in its *numSuccessfulOps* field (those tuples will remain in the tuple stores).

getLatestRetentionPeriod

⇒	xsd:int connectionId	Primary Producer resource identifier.
⇒	xsd:string tableName	VDBTable name.
⇐	Tuple(1..1)	
	xsd:int hrpSec	Latest Retention Period in seconds.

Returns a Primary Producer resource's declared Latest Retention Period for a given table.

See also the common producer service operations: getHistoryRetentionperiod 6.1.1.

See also the common resource management service operations: close 8.1.1 and destroy 8.1.1.

See also the operation common to all services: getProperty 11.1.1.

3.2.2 SYSTEM INTERFACE

See the common producer service operations: start 6.1.2 and abort 6.1.2.

See the common resource management service operation: ping 8.1.2.

3.3 DETAILS

3.3.1 CREATING AND DESTROYING PRIMARY PRODUCERS

A new Primary Producer Resource is created when a user calls the *createPrimaryProducer* operation and is destroyed when the user calls the *close* or *destroy* operations. The special processing for *close* is desribed below. In addition, if the service does not hear from the user for a period exceeding the *termination interval*, the service will initiate a *close* operation on the resource. A call to any user operation on the resource is sufficient to keep it alive.

3.3.2 TUPLE STORES

The Primary Producer's history and latest tuple stores were described in 1.2.5. They may be physically in memory or database storage. The memory storage may also make use of an RDBMS (typically memory resident) but is transient. A single producer uses only one type of storage for all types of queries that it supports.

The Primary and Secondary Producer services only uses stores created and managed by themselves. Tuple stores are normally temporary and are destroyed along with the producer resource, but users can make atuple store with database storage permanent by specifying a *logical name* for the tuple store when they create a new Primary (or Secondary) producer. If the service is running in secure mode, the user's Distinguished Name (DN) is prefixed to the logical name. In insecure mode, the user must make the name unique within the service by some other means. Permanent tuple stores are not destroyed when a producer resource is destroyed, so they can be re-used by creating a new producer and naming the same store, provided there are no other producers using it already. The user can then re-declare any table that is already in the store (with a compatible predicate - see 14.8) and any existing tuples will be automatically recovered. The names and some information about tuple stores can be obtained by calling *listTupleStores*. Permanent tuple stores are deleted by calling *dropTupleStore*. R-GMA only allows the the *owners* of existing tuple stores to re-use or list them.

3.3.3 DECLARING TABLES

Producers must declare their intention to publish to a table by calling *declareTable* before they can insert tuples. The table definition must already exist in the schema (see 14.3). The Primary Producer Service obtains the table definition from the schema by calling *getFullTableDetails* and creates the corresponding table in the resource's tuple stores (or if it already exists in a named tuple store, checks its structure). If the schema specifies that the table should be indexed, the producer creates the indexes if possible, and also indexes the *RgmaTimestamp* column (see below). It then registers the producer as a publisher for that table, by calling the registry's *registerProducerTable* operation. The service must be ready to service consumer queries against the resource's tuple stores as soon as this notification is sent. This registry operation returns a list of any relevant continuous consumers and the producer service must send an *addProducer* message to each of them to notify them about the new producer. The producer service will also periodically re-send the *registerProducerTable* to the registry on the user's behalf, to maintain the producer's entries in the registry throughout its lifetime. The list of consumers returned is used to help check that consumers are still alive (see 3.3.7). The user can declare more than one table in a single producer, and these can even be in different virtual databases. All the tables are treated independently by the producer service, except for processing "join" queries.

3.3.4 INSERTING TUPLES

Tuples are inserted into the producer service's tuple storage by calling the *insert* or *insertList* operations. They are checked for type against the schema, and for content against the producer's declared predicate. The following metadata is added to each tuple by the service:

MeasurementDate	DATE	This and the next column will be supported for a while for backward compatibility. At some stage they will be turned into user columns so that they can be eliminated.
MeasurementTime	TIME	See above
RgmaTimestamp	TIMESTAMP(9)	UTC tuple time-stamp: only added by R-GMA if not already filled in by the user; those added by R-GMA will only have a resolution of 1ms (see 14.4 for the format of a TIMESTAMP)
RgmaLRT	TIMESTAMP(6)	UTC Latest Retention Time (see below)
RgmaOriginalServer	VARCHAR(255)	Hostname (including domain name) of server publishing the information.
RgmaOriginalClient	VARCHAR(255)	Hostname (including domain name) of client publishing the information.

Tuples are inserted to the tuple stores and streamed to subscribed consumers as described in 1.2.5. If one of the producer's tuple stores fills up, the service temporarily blocks inserts by returning an RGMABufferFullException until tuples expire and can be removed, as described below.

A Producer making use of a named tuple store should avoid sending tuples that have already been sent by another producer that used the store previously.

3.3.5 REMOVING TUPLES

The user must set a *History Retention Period (HRP)* and a *Latest Retention Period (LRP)* for each declared table. The HRP is recorded in the registry indicating to the mediator the maximum age of tuples it guarantees to maintain in its history tuple store. The Primary Producer Service adjusts the HRP it sends to the registry to take account of the actual age of its history tuple store (which will be very short for a new producer), and updates it in the periodic calls to *registerProducerTable* up to the value set by the user as time progresses. The LRP is used to calculate a *Latest Retention Time (LRT)* for each tuple by adding it to the tuple's RgmaTimestamp (the table-default LRP may be overridden in each call to *insert*). The LRT is written into the tuple's metadata because Secondary Producers must also respect it. The two retention periods are independent. A history tuple store records the time of insertion of each tuple into the tuple store and uses this to work out when the tuples should be expired. The same rule applies to a tuple arriving in a tuple store managed by a secondary producer.

Tuples that have expired in either tuple store can be removed by the service, and the service is obliged to periodically clean up: it is not allowed to run out of storage and block inserts if it could free up storage by deleting expired tuples. In addition, tuples in the latest tuple store that have exceeded their LRT must *never* take part in latest queries.

The distinction between *close* and *destroy* on a Primary Producer is that a *close* will wait for tuples to be streamed to any continuous consumers that have not yet received them, and will also wait for any tuples in a memory based history tuple store to expire before the producer is destroyed. Note that the wait for tuples to expire does not apply to the latest tuple store nor to tuple stores using database storage. During this time, the producer will remain registered and still accept new history queries and latest (but not new continuous queries) for each declared table until all tuples for the table have expired from the history tuple store, at which point the corresponding table is unregistered, by calling *unregisterProducerTable*. When all tables have been unregistered and all tuples have been streamed, the producer resource is destroyed.

3.3.6 PROCESSING QUERIES

Consumer services send a *start* message to a Primary Producer to request it to execute a query and start streaming the resulting tuples back to the consumer service. Continuous queries receive all old tuples available in the producer's history store since the start time specified in the *start* call, followed by all new tuples as they are inserted to the producer. One-time queries execute on the current contents of the history and latest tuple stores only and terminate when they've returned all of the results to the consumer service.

The streaming protocol and the streaming server in the consumer service are described in section 7.3.4. As described there, tuples are streamed in *chunks*, with the last chunk for a one-time query distinguished by an *end-of-results* flag. The connection details of the streaming server, and the maximum number of tuples it will accept in a single chunk, are passed in the *start* call. It is the responsibility of the implementation to ensure that the results of a one-time query are evaluated just once and are immune to changes to the tuple stores while the results are being streamed.

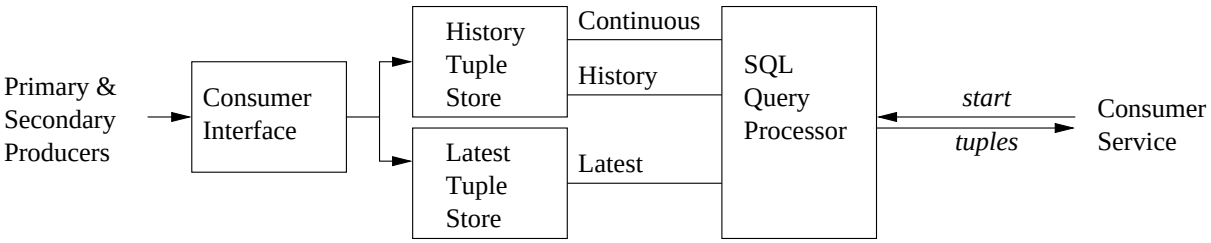
3.3.7 STOPPING QUERIES

One-time queries automatically terminate when the last tuple has been streamed to the consumer. All queries may be terminated by running for longer than the query *timeout* specified in the call to *start*.

4 SECONDARY PRODUCER SERVICE

4.1 DESCRIPTION

Secondary Producer resources are created by the Secondary Producer Service at the request of a user to *republish* one or more tables in a virtual database. Republishing means running a "SELECT * WHERE" query against a table in the virtual database and publishing the resulting tuples back to the same table. The mediator ensures this isn't recursive. As the picture below shows, the principal components of a Secondary Producer are the same as a Primary Producer. All Secondary Producers support continuous queries, and can be configured to support any combination of latest and history queries as well.



The value of a Secondary Producer is that it creates real tables from virtual ones, because it collects all tuples inserted into the virtual database for any table it's republishing, into its own tuple stores. This means it can act as an archiver for the tables, can answer queries involving joins between them (even tables in different VDBs) because they're all in one database, and can be used to reduce the load on other producers.

The Secondary Producer Service is responsible for authenticating all users and services that connect to it, and for authorizing all operations and all requests to access its tuple stores, as specified in chapter 13.

4.2 INTERFACE

4.2.1 USER INTERFACE

createSecondaryProducer

⇒	xsd:boolean isHistory	If history queries are supported
⇒	xsd:boolean isLatest	If latest queries are supported
⇒	xsd:string type	DATABASE or MEMORY
⇒	xsd:string logicalName	The logical name should not be be specified for type MEMORY. For type DATABASE it is optional.
⇐	Tuple(1..1)	
	xsd:int connectionId	connectionId of new Secondary Producer resource.

Creates a new Secondary Producer resource and returns its endpoint. The Secondary Producer is not added to a Registry until *declareTable* is called. The user cannot prevent Secondary Producers from supporting continuous queries, but support for latest and/or history queries is optional. Secondary Producers can use any type of storage, regardless of the query types they support. Tuple stores can be made permanent by providing a *logical name* for the tuple store as described in 4.3.2. The termination interval is described in 4.3.1.

declareTable

⇒	xsd:int connectionId	Secondary Producer resource identifier.
⇒	xsd:string tableName	VDBTable to register.
⇒	xsd:string predicate	Producer's predicate.
⇒	xsd:int hrpSec	History Retention Period in seconds.
⇐	Tuple(1..1)	
	xsd:string status	OK

Adds a table to the list of tables republished by this producer, as described in 4.3.3. The table name must have an explicit virtual database name prefix (separated from it by a dot). The format of the predicate is specified in 14.8; by construction, a Secondary Producer will republish *all* tuples that match its predicate. The retention period is described in 4.3.6.

showSignOfLife

⇒	xsd:int connectionId	Resource identifier.
⇐	Tuple(1..1)	
	xsd:string status	OK

Sends a sign-of-life request to a resource as a way to keep the resource alive. Any other user operation on the resource will also serve to keep it alive.

See also the common producer service operations: getHistoryRetentionperiod 6.1.1.

See also the common resource management service operations: close 8.1.1 and destroy 8.1.1.

See also the operations common to all services: getProperty 11.1.1.

4.2.2 SYSTEM INTERFACE

secondary

addProducer ⇒	xsd:int connectionId	Consumer resource identifier.
⇒	xsd:string url	Producer URL.
⇒	xsd:int id	Producer resource identifier.
⇒	xsd:string tableName	Table name
⇒	xsd:string predicate	Predicate
⇒	xsd:int hrpSec	History Retention Period in seconds
⇒	xsd:boolean isHistory	If producer supports history queries
⇒	xsd:boolean isLatest	If producer supports latest queries
⇒	xsd:boolean isContinuous	If producer supports continuous queries
⇒	xsd:boolean isStatic	If producer supports static queries
⇒	xsd:boolean isSecondaryProducer	If producer is secondary
⇒	xsd:string qosAttrib	The QOS attribute - not used currently
⇐	Tuple(1..1)	
	xsd:string status	OK

Sent by a producer service to a continuous consumer or secondary producer to notify it about an addition to the list of relevant producers in the registry. Ignored if the query is not currently executing. See the *plan maintenance* section (7.3.3) for how the consumer or secondary producer should react to this.

removeProducer

⇒	xsd:int connectionId	Consumer resource identifier.
⇒	xsd:string url	Producer URL.
⇒	xsd:int id	Producer resource identifier.
⇐	Tuple(1..1)	
	xsd:string status	OK

See the common producer service operations: start 6.1.2 and abort 6.1.2.

See the common resource management service operation: ping 8.1.2.

4.3 DETAILS

4.3.1 CREATING AND DESTROYING SECONDARY PRODUCERS

A new Secondary Producer resource is created when a user calls the *createSecondaryProducer* operation and is destroyed when the user calls the *close* or *destroy* operations. In addition, if the service does not hear from the user for a period exceeding the *termination interval*, the service will initiate a *close* operation on the resource. A call to any user operation on the resource is sufficient to keep it alive.

4.3.2 TUPLE STORES

These are exactly as in the Primary Producer Service (see 3.3.2) but System Administrators should be even more wary of granting direct access to permanent tuple stores maintained by Secondary Producers because Secondary Producers are granted special access to the tuple stores of the producers from which they are consuming, and those producers trust them to enforce the data access rules of the VDB when republishing the data.

4.3.3 DECLARING TABLES (AS A PRODUCER)

The semantics of *declareTable* are identical from the user's point of view to the Primary Producer, except that there is no Latest Retention Period to set. Internally, the service prepares the tuple stores then registers the Secondary Producer resource as a producer by calling *registerProducerTable* and periodically calls this again to keep itself registered.

4.3.4 DECLARING TABLES (AS A CONSUMER)

Unlike a Primary Producer, the Secondary Producer must also act as a continuous consumer for each declared table, by running a "SELECT * FROM *tableName* WHERE *predicate*" query on the virtual database. The procedure for identifying and notifying the producers that will answer this query is exactly as in 7.3.2 and the list of producers is maintained as described in 7.3.3. A side effect of this is to register the Secondary Producer resource as a *secondary* consumer and keep it registered. Flagging the resource as a secondary consumer simply allows the mediator to ensure that it doesn't generate recursive query plans - in all other respects, the mediator and the registry service treat the secondary producer resource like any other continuous consumer and no further distinction is made in this document. Likewise, the secondary producer presents the same streaming interface to producers as described in 7.3.4 for continuous consumers.

4.3.5 INSERTING TUPLES

Tuples are inserted into a Secondary Producer by the service itself (by its streaming server). Users do not insert tuples into a Secondary Producer directly. Secondary Producers do not modify the tuples they store in any way.

4.3.6 REMOVING TUPLES

The rules for History Retention Periods and Latest Retention Times in a secondary producer are identical to a primary producer.

The *close* call, however, is different from a primary producer. Since a secondary producer must stop consuming immediately after a *close* request it must also stop servicing queries too, because it no longer holds a complete set of tuples for the tables that it claims to republish. Therefore *close* means the same as *destroy* in a secondary producer.

4.3.7 PROCESSING QUERIES

Query processing in a Secondary Producer is identical to a Primary Producer (see 3.3.6).

4.3.8 STOPPING QUERIES

This is also identical to a Primary Producer (see 3.3.7).

5 On-demand Producer Service

5.1 Description

On-demand Producer resources are created by the On-demand Producer Service at the request of a user who wishes to make an external data store available through a virtual database. They are registered in the same way as the other producer types and their queries are parsed, validated and authorized like queries to the other producer types, but they only support static queries, are not used by Secondary Producers, have no internal tuple storage, and have no concept of retention periods. As the picture below shows, queries are simply handed off to a user-defined application (*query handler*) that is expected to process the query and return the resulting tuples on demand.

The On-demand Producer Service is responsible for authenticating all users and services that connect to it and for authorizing all operations and requests to access its users' external data stores, as specified in chapter 13.

5.2 Interface

5.2.1 User Interface

createOnDemandProducer

⇒	xsd:string hostName	Hostname of machine to which to connect to get data to answer queries.
⇒	xsd:int port	Port to which to connect to get data to answer queries.
⇐	Tuple(1..1)	
	xsd:int connectionId	connectionId of new On-demand Producer resource.

Creates a new On-demand Producer resource and returns its endpoint. The On-demand Producer is not added to a Registry until *declareTable* is called. On-demand producers have no tuple stores and only support static queries.

declareTable

⇒	xsd:int connectionId	On-demand Producer resource identifier.
⇒	xsd:string tableName	VDBTable to register.
⇒	xsd:string predicate	Producer's predicate.
⇐	Tuple(1..1)	
	xsd:string status	OK

Adds a table to the list of tables for which this producer will return tuples, as described in 5.3.3. The table name must have an explicit virtual database name prefix (separated from it by a dot). The format of the predicate is specified in 14.8; all returned tuples must match this predicate or they will be rejected by the producer.

See also the common producer service operations: getHistoryRetentionperiod 6.1.1.

See also the common resource management service operations: close 8.1.1 and destroy 8.1.1.

See also the operations common to all services: getProperty 11.1.1.

5.2.2 SYSTEM INTERFACE

See the common producer service operations: start 6.1.2 and abort 6.1.2.

See the common resource management service operation: ping 8.1.2.

5.3 DETAILS

5.3.1 CREATING AND DESTROYING ON-DEMAND PRODUCERS

A new on-demand producer resource is created when a user calls the *createOnDemandProducer* operation and is destroyed when the user calls the *close* or *destroy* operations. In addition, if the service does not hear from the user for a period exceeding the *termination interval*, the service will initiate a *close* operation on the resource. A call to any user operation on the resource is sufficient to keep it alive.

5.3.2 THE QUERY HANDLER

The on-demand producer connects to the specifed hostname and port via an SSL socket connection to answer user queries. R-GMA expects the query handler application to be listening for queries throughout the lifetime of the on-demand producer resource.

5.3.3 DECLARING TABLES

On-demand producers must register any tables for which they will answer queries by calling *declareTable*. The On-demand Producer Service registers the producer as a publisher for that table by calling the registry's *registerProducerTable* operation and will periodically re-send the *registerProducerTable* to the registry on the user's behalf, to maintain the producer's entries in the registry throughout its lifetime. The user can declare more than one table in a single producer, and these can even be in different virtual databases. Whether or not "join" queries can be processed depends only on the capabilities of the query handler that answers the producer's queries.

5.3.4 PROCESSING QUERIES

Consumer Services send a *start* message to an On-demand Producer to request it to execute a query and start streaming the resulting tuples back to the consumer service. The query is forwarded to the registered query handler for processing (see below for the protocol), and the tuples (if any) returned by the query handler are streamed by the On-demand Producer service back to the consumer's streaming server, exactly as in a one-time query to any other type of producer (see 7.3.4). A query may be stopped by the consumer service by calling *abort*. Queries will also be automatically aborted by the On-demand Producer Service if they run for more than the *timeout* specified in the call to *start*.

The On-demand producer service will append the standard metadata columns (see 3.3.4) to each tuple for consistency with the table definition in the schema, before streaming them back to the consumer service, but the columns will all be populated with NULLs.

See section 14.1 for information about the character sets used in R-GMA (this will affect both the SQL query sent to the query handler, and its response).

5.3.5 QUERY HANDLER PROTOCOL

The On-demand Producer service contacts the query handler by opening a TCP connection on the host and port specified in the URI and writing the SQL SELECT query and the maximum number of tuples it will allow in a result set, to the port. The two parameters are separated by a semi-colon and the request is terminated by a CR-NL end of line marker. The query handler must respond by sending a series of result sets (chunks) each containing no more than the requested number of tuples, or it must return an exception. The query handler should mark the end of the response by shutting the TCP connection.

The result sets and exceptions returned by the query handler are XML fragments, each containing a single XMLResultSet or XMLException. The XMLResultSet looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <XMLResultSet endOfResults="true">
    <columnMetaData>
      <name>column-name</name>
      <type>NNN</type>
      <table>table-name</table>
    </columnMetaData>
    <row>
      <col>column-value</col>
      <col isNull="true"/>
    </row>
  </XMLResultSet>
```

where *columnMetaData* and *col* are repeated for each column, and *row* is repeated for each row. The valid values for *type* are 4 (INTEGER), 7 (REAL), 8 (DOUBLE PRECISION), 91 (DATE), 92 (TIME), 93 (TIMESTAMP), 1 (CHAR) and 12 (VARCHAR). The *endOfResults* and *isNull* flags both default to *false* if omitted. The *XMLResultSet* element is allowed to be entirely empty. The last result set must contain an *endOfResults* flag set to *true* (even if it is empty).

The XMLException looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
  <XMLException>
    <message>message-string</message>
  <XMLException>
```

The On-demand producer service will shut the TCP connection if the query handler exceeds the tuple limit in a single result set (chunk), or it receives an *abort* request, or if some other error occurs. The query handler must ensure that the result of a query does not change during the time it takes to deliver all of the tuples to the on-demand producer service.

5.4 ERROR HANDLING

The default handling for all errors is to stop processing, make the resource stable if possible and notify the user (see section 1.4.5).

6 PRODUCER OPERATIONS

Operations defined here are are common to all producers.

6.1 INTERFACE

6.1.1 USER INTERFACE

getHistoryRetentionPeriod

⇒	xsd:int connectionId	Producer resource identifier.
⇒	xsd:string tableName	VDBTable name.
⇐	Tuple(1..1)	
	xsd:int hrpSec	History Retention Period in seconds.

Returns a Producer resource's declared History Retention Period for a given table.

6.1.2 SYSTEM INTERFACE

start

⇒	xsd:int connectionId	Producer resource identifier.
⇒	xsd:string select	Consumer's SQL SELECT query.
⇒	xsd:string queryType	"continuous", "latest", "history" or "static"
⇒	xsd:int(0..1) timeIntervalSec	Time interval associated with queryType
⇒	xsd:long timeoutSec	Query timeout in seconds.
⇒	xsd:string consumerURL	Consumer's URL.
⇒	xsd:int consumerId	Consumer's resource ID.
⇒	xsd:string streamingURL	URL for streaming.
⇒	xsd:int streamingPort	Port number for streaming.
⇒	xsd:int bufferSize	Maximum buffer size to use for streaming.
⇒	xsd:int streamingProtocol	Protocol number to use for streaming.
⇒	xsd:string qosAttrib	Consumer's desired Quality of Service.
⇐	Tuple(1:1)	
	xsd:string tableName	
	xsd:long timestamp	

Requests the producer to execute a query and start streaming tuples to the specified consumer. The query will have been generated by the mediator and can be assumed to be valid: both tables and views may be queried (see 13.6.4). It is assumed that there is a streaming server listening on the given endpoint, ready to receive the tuples pushed to it.

abort

⇒	xsd:int connectionId	Producer resource identifier.
⇒	xsd:string consumerURL	Consumer's URL.
⇒	xsd:int consumerId	Consumer's resource ID.
⇐	Tuple(1..1)	
	xsd:string status	OK

Requests the producer to stop streaming tuples to the specified consumer and abort the query.

7 CONSUMER SERVICE

7.1 DESCRIPTION

Consumer resources are created by the Consumer Service at the request of a user who wishes to query one or more tables in the virtual database. Each Consumer represents a single SQL SELECT query. It runs the query on the user's behalf by contacting all of the producers necessary to answer it, and collects the results into an internal tuple buffer from which the user can subsequently retrieve them. Tuples are always *pushed* by producer services to a *streaming server* running in each Consumer Service.

The Consumer Service is responsible for authenticating all users and services that connect to it, and for authorizing all operations, as specified in chapter 13.

The mediator is the component within the consumer which handles its interface to registry instances and maintains the consumer's query plan. All communication between the consumer and the registry is through the mediator.

7.2 INTERFACE

7.2.1 USER INTERFACE

createConsumer

⇒	xsd:string select	Consumer's SQL SELECT query.
⇒	xsd:string queryType	continuous, history, latest or static.
⇒	xsd:int(0..1) timeIntervalSec	Optional time interval in seconds to go back in time for the query.
⇒	xsd:int(0..1) timeoutSec	Optional query timeout in seconds.
⇒	xsd:string(0..*) producerConnections	Optional list of producer endpoints to contact. Each endpoint is encoded as the string representation of an integer producer resource, then a single space followed by the producer resource URL.
⇐	Tuple(1..1)	
	xsd:int connectionId	connectionId of new Consumer resource.

Creates a new Consumer resource and returns its endpoint. The query is not started until the user calls *start* or *startDirected*. The query is validated by calling *getFullTableDetails* in the appropriate schema for each table in the query. The subset of SQL supported by R-GMA for queries is specified in detail in 14.9: both tables and views (see 13.6.4) can be queried. All table and view names must have explicit virtual database name prefixes (separated from them by a dot). The mediator places further restrictions on the complexity of queries supported by each query type. The query type and interval are described in 1.2.4; the interval is relative to the creation time of the consumer resource in the service (an absolute start time is passed to the producer service). The termination interval is described in 7.3.1. Mediates and starts the Consumer's query, as described in 7.3.2 below. The query will be aborted by the consumer service if it is still executing when the timeout is reached. The query will also be aborted if the consumer's tuple buffer fills up. Does not wait for the query to finish before returning. Queries can only be started once.
If there is a user-supplied list of producers this is used instead of obtaining one from the mediator. All listed producers will be contacted. It is the responsibility of the user to ensure the list of producers they provide will be able to answer the query correctly.

abort

⇒	xsd:int connectionId	Consumer resource identifier.
⇐	Tuple(1..1)	
	xsd:string status	OK

Aborts a running query as described in 7.3.2 below. Returns when the query has aborted. Any tuples already held in the consumer may still be retrieved by calling *pop*. Does *not* unregister the consumer.

hasAborted

⇒	xsd:int connectionId	Consumer resource identifier.
⇐	Tuple(1..1)	
	xsd:boolean hasAborted	True if user has aborted the query, or the query timed out.

Checks if the user has aborted a running query, or the query has timed out. Query must have already been started by calling *start* or *startDirected*.

pop

⇒	xsd:int connectionId	Consumer resource identifier.
⇒	xsd:int maxCount	Maximum number of tuples to pop.
⇐	Tuple(0..*)	The results of the query. The fields returned depend upon the query.

Retrieves at most *maxCount* tuples from the Consumer resource. Can be called at any point after a query is first started (even after *abort*). Tuples are returned in a result set with column metadata. Result set will be empty if no tuples are available at the time *pop* is called. An *endOfResults* flag is attached to the final result set, i.e. when the query has finished executing (or has aborted) and *pop* has returned all available tuples. The mediator will attach a warning to a result set if it cannot guarantee the completeness of the results.

See also the common resource management service operations: close 8.1.1 and destroy 8.1.1.

See also the operations common to all services: getProperty 11.1.1.

7.2.2 SYSTEM INTERFACE

addProducer

⇒	xsd:int connectionId	Consumer resource identifier.
⇒	xsd:string url	Producer URL.
⇒	xsd:int id	Producer resource identifier.
⇒	xsd:string tableName	Table name
⇒	xsd:string predicate	Predicate
⇒	xsd:int hrpSec	History Retention Period in seconds
⇒	xsd:boolean isHistory	If producer supports history queries
⇒	xsd:boolean isLatest	If producer supports latest queries
⇒	xsd:boolean isContinuous	If producer supports continuous queries
⇒	xsd:boolean isStatic	If producer supports static queries
⇒	xsd:boolean isSecondaryProducer	If producer is secondary
⇒	xsd:string qosAttrib	The QOS attribute - not used currently
⇐	Tuple(1..1)	
	xsd:string status	OK

Sent by a producer service to a continuous consumer to notify it about an addition to the list of relevant producers in the registry. Ignored if the query is not currently executing. See the *plan maintenance* section (7.3.3) for how the consumer should react to this.

removeProducer

⇒	xsd:int connectionId	Consumer resource identifier.
⇒	xsd:string url	Producer URL.
⇒	xsd:int id	Producer resource identifier.
⇐	Tuple(1..1)	
	xsd:string status	OK

See also the common resource management service operation: ping 8.1.2.

7.3 DETAILS

7.3.1 CREATING AND DESTROYING CONSUMERS

A new Consumer Resource is created when a user calls the Consumer Service's *createConsumer* operation and is destroyed when the user calls the *close* or *destroy* operations. In addition, if the service does not hear from the user for a period exceeding the *termination interval*, the service will initiate a *close* operation on the resource. A call to any user operation on the resource is sufficient to keep it alive. To use the virtual database properly, the user should normally run a mediated query. The query is run on the user's behalf by the consumer service, and the user should call *pop* repeatedly until it returns a result set containing an *endOfResults* flag. If there are no results available in the Consumer resource's buffer at the time at which *pop* is called, an empty result set will be returned, but more tuples may become available if the query is still executing. If the user wants to abort a query, they can call *abort*. A timeout may be specified when a consumer is created in which case the query will be automatically aborted if it is still executing after the time has elapsed (the user can call *hasAborted* to see if this has occurred). The query will also be aborted if the consumer's tuple buffer fills up (the service should be configured with enough overflow buffer space on disk so that a well-behaved consumer is unlikely to hit this limit). The user can

safely pop tuples from the consumer following an abort, but they will only get tuples that are already in the consumer resource's buffer.

7.3.2 STARTING AND STOPPING QUERIES (SERVICE)

The consumer service starts a query by identifying which producers it should contact. For a directed query (*startDirected*), this is specified by the user, and the consumer service will simply contact all listed producers and merge the results, whether or not that makes sense. Consumers running directed queries are not registered in the registry, even for continuous queries. For a mediated query, the consumer service must obtain a *query plan* from the mediator. The mediator forms a layer between the consumer service and the registry services of all VDBs referenced in the consumer's query. A side-effect of this mediator call for continuous consumers is to register them so that they will receive notifications from all new producers. The query plan is essentially a set of instructions to the consumer on which producers to send a *start* message to, and with what query (it may also contain instructions to stop streaming from certain producers). The consumer service merges the results streamed back from all of the producers into the consumer resource's internal tuple buffer, from where they can be popped later by the user. The mediator may also attach warnings to the query plan regarding potential unreliability of results - these must be added to all result sets for the query by the consumer service.

One-time queries automatically stop when the last tuple has been streamed to the consumer, but they will also stop if the user calls *abort* or the query runs for longer than the query timeout specified by the user in the call to *start*. Continuous queries only stop when they are aborted in one of these two ways. Abort messages are forwarded on to the producers (by calling each producer service's *abort* message) so they can clean up any system and database resources associated with the query.

7.3.3 PLAN MAINTENANCE

Since starting a query that involves multiple producers is not an atomic process, some calls to the producers may start while others fail. All types of queries must adapt to this. Continuous queries must also adapt to producers failing, or producers being added or removed during the lifetime of their long-running query. In all cases, the consumer service must request an updated query plan from the mediator and must start or stop streaming from producers as directed by the new plan. The various cases are explained here.

In the case of a failure of a *start* call to a producer, the consumer service must call the mediator to get a new plan that excludes the broken producer.

The consumer service also detects producers that worked initially but have subseqently failed, by periodically checking that they have received either a tuple from each producer, or a plan update (see below) containing that producer. If neither has happened since the last check, the consumer service sends a *ping* to the producer and if the service cannot be contacted or returns an UnknownResourceException, the consumer requests a new plan from the mediator. The mediator may attach a warning to the query plan if it can't be certain that a broken producer is no longer relevant.

Continuous queries are registered in all relevant registries so that the consumer can be notified when new producers are added to the registry. The notification comes in the form of an *addProducer* message from the producer itself. When the consumer service receives an *addProducer*, it must contact the mediator and start or stop streaming from producers as instructed.

Registry replication means that a consumer must also actively ensure its plan remains up to date. It does this as a side-effect of updating its registration via the mediator[1] This operation will query the registries for any new producers, and has the side effect of keeping the consumer registered. As in the other mediation calls, the consumer must act on the instructions of the new plan.

[1] This polling is less efficient than relying on the registry to notify consumers of changes, but is more robust.

7.3.4 STREAMING

Tuples are pushed by producers into a *streaming server* in the consumer service that then delivers them to the tuple buffers of the (local) consumer resources for which they are intended. Note that tuples are not streamed all the way to the user: the user must still ultimately pull them from the consumer by calling *pop*. The streaming server is an implementation-defined server application listening on a well-defined TCP socket, the details of which are passed to each producer in the call to *start*. Tuples are returned in *chunks*, each preceded by a 32-bit network byte ordered integer containing the target consumer's resource identifier, and terminated by an end-of-chunk flag (a single byte with the value 1). The consumer service sets a limit on the maximum number of tuples in a chunk, in the call to *start*. Producers will continue to send chunks for continuous queries until the query is aborted, but a one-time query will terminate after the last chunk, so this is indicated by an end-of-results flag (a single byte with the value 2) after the end-of-chunk flag of the last chunk for the one-time query. Chunks are encoded as XML as defined in 2.4.1.

If a consumer's tuple buffer is full and a new chunk is received for it by the streaming server, the query will be aborted and the connection closed. If a chunk is received for a consumer that does not exist, the streaming server will notify the producer[2]. An encrypted connection is used for streaming (see chapter 13).

[2] Whether this is done in an immediate response or asynchronously remains to be decided.

8 RESOURCE MANAGEMENT OPERATIONS

Operations defined here are are common to all resource management operations - i.e. producers and consumers.

8.1 INTERFACE

8.1.1 USER INTERFACE

close

⇒	xsd:int connectionId	Resource identifier.
⇐	Tuple(1..1)	
	xsd:string status	OK

Schedules a resource for destruction after the resource has completed its work.

destroy

⇒	xsd:int connectionId	Primary Producer resource identifier.
⇐	Tuple(1..1)	
	xsd:string status	OK

Destroys a Primary Producer without any special *close* processing.

8.1.2 SYSTEM INTERFACE

ping

⇒	xsd:int connectionId	Resource identifier.
⇐	Tuple(1..1)	
	xsd:string status	OK

Checks if a resource is still alive (and throws an exception if it is not). Has no side-effects on the resource.

9 REGISTRY SERVICE

9.1 DESCRIPTION

There is exactly one registry per virtual database. It holds the details of all producers that are publishing to tables in the virtual database, and it also holds the details of continuous consumers who wish to be notified of changes to the list of producers. For reasons of resilience and scalability, multiple replicas of the registry can be created for each virtual database. Each replica exists as a Registry Instance in a Registry Service and is created by the service at the request of a user. A single Registry Service can host replicas from multiple virtual databases. The Registry Service also handles queries for registries that it is not hosting, by locating another Registry Service that is hosting a working replica of the requested registry and forwarding the query there. In this way, users, producer services and consumer services only ever need to contact their local Registry Service directly.

The Registry Service is responsible for authenticating all users and services that connect to it, and for authorizing all operations and all requests to access the registries it is hosting, as specified in chapter 13.

9.2 INTERFACE

9.2.1 USER INTERFACE

getAllProducersForTable

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:boolean canForward	True if query can be forwarded.
⇒	xsd:string tableName	Table name.
⇐	Tuple(0..*)	
	xsd:string url	Producer URL
	xsd:int connectionId	Producer resource ID.
	xsd:boolean isSecondaryProducer	If producer is secondary.
	xsd:boolean isContinuous	If producer supports continuous queries.
	xsd:boolean isStatic	If producer supports static queries.
	xsd:boolean isHistory	If producer supports history queries.
	xsd:boolean isLatest	If producer supports latest queries.
	xsd:string predicate	Predicate associated with the producer.
	xsd:int hrpSec	History Retention Period in seconds

Returns an unmediated list of all producers registered for the specified tables, with their connection details and producer types (used by the R-GMA Browser). Processed locally if possible (see 9.3.2).

See also the operations common to all services: getProperty 11.1.1.

9.2.2 SYSTEM INTERFACE

getURL(), getResourceID(), isSecondary(), isContinuous(), isStatic(), isHistory(), isLatest(), getPredicate(),getHistoryRetentionPeriod(), getVdbName(), getTableName()

getMatchingProducersForTables

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:boolean canForward	True if query can be forwarded.
⇒	xsd:string(1..*) tables	Table name
⇒	xsd:string predicate	Predicate associated with the consumer.
⇒	xsd:string queryType	Consumer's Query type.
⇒	xsd:int(0..1) timeIntervalSec	Time interval to look back into the past for matching tuples.
⇒	xsd:boolean isSecondaryConsumer	True for secondary consumer (i.e. consumer part of a secondary producer). Only specified if queryType is continuous.
⇒	xsd:string url	URL of consumer. Only specified if queryType is continuous.
⇒	xsd:int resourceId	Resource id of consumer. Only specified if queryType is continuous.
⇒	xsd:int terminationIntervalSec	Consumer's termination interval in seconds.
⇐	Tuple(0..*)	
	xsd:string url	Producer URL
	xsd:int connectionId	Producer resource ID.
	xsd:boolean isSecondaryProducer	If producer is secondary.
	xsd:boolean isContinuous	If producer supports continuous queries.
	xsd:boolean isStatic	If producer supports static queries.
	xsd:boolean isHistory	If producer supports history queries.
	xsd:boolean isLatest	If producer supports latest queries.
	xsd:string predicate	Predicate associated with the producer.
	xsd:int hrpSec	History Retention Period in seconds
	xsd:string tableName	Table name without VDB prefix.
	xsd:string vdbName	Virtual database name.

Returns a list of all producer-table entries from the registry that match the Consumer's query predicate, for any of the listed tables. This operation is used by the mediator in the consumer service. Continuous consumers are also registered (or updated if they are already registered) by this call so the registry can pass their details to any new, relevant producers. See 9.3.1 for a description of the termination interval. The last three parameters: *isSecondary*, *consumer* and *terminationIntervalSec* are only present for continuous consumers. Processed locally if possible (see 9.3.2).

registerProducerTable

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:boolean canForward	True if query can be forwarded.
⇒	xsd:string url	Producer URL
⇒	xsd:int id	Producer resource ID.
⇒	xsd:string tableName	Table name.
⇒	xsd:string predicate	Producer's predicate.
⇒	xsd:boolean isContinuous	If producer supports continuous queries.
⇒	xsd:boolean isHistory	If producer supports history queries.
⇒	xsd:boolean isLatest	If producer supports latest queries.
⇒	xsd:boolean isStatic	If producer supports static queries.
⇒	xsd:boolean isSecondaryProducer	If producer is secondary.
⇒	xsd:int hrpSec	Table's History Retention Period in seconds.
⇒	xsd:int terminationIntervalSec	Producer's termination interval in seconds.
⇐	Tuple(0..*)	
	xsd:string url	Consumer URL
	xsd:int id	Consumer resource ID.

Adds an entry to the Registry for a producer containing the details passed to this operation or updates them if it is already registered. The producer service will already have validated the predicate before calling this operation. If the new producer supports continuous queries and there are any registered continuous consumers to which this producer is relevant (by comparison of their predicates), then this call returns a list so the producer can send an *addProducer* notification to each of them (the list will be empty for non-continuous producers). The *retentionPeriod* should reflect the actual history available from the producer (which may be very short initially) and should be updated in subsequent calls to this operation. See 9.3.1 for a description of the termination interval. Processed locally if possible (see 9.3.2).

unregisterContinuousConsumer

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:boolean canForward	True if query can be forwarded.
⇒	xsd:string url	Consumer URL
⇒	xsd:int id	Consumer ID.

Removes a Continuous consumer's entry from the Registry. The Consumer Service is expected to notify all producers to which the consumer has subscribed that it is closing - the Registry does not do this. Processed locally if possible (see 9.3.2).

unregisterProducerTable

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:boolean canForward	True if query can be forwarded.
⇒	xsd:string tableName	Table name.
⇒	xsd:string url	Producer's URL
⇒	xsd:int id	Producer's ID.

Removes a producer's entry for a particular table from the Registry. Processed locally if possible (see 9.3.2).

addReplica

⇒	xsd:string replica	String with replication message for a vdb.

Applies updates to the Registry with data from another replica. See section 9.3.3 for details. Always processed locally.

ping

⇒	xsd:int vdbName	Virtual database name
⇐	Tuple(1..1)	
	xsd:string status	OK

Checks if a registry instance is still alive (and throws an exception if it is not). Used by the Registry Service to locate the "closest" working replica for a particular virtual database. Has no side-effects on the registry instance. Always processed locally.

9.3 Details

9.3.1 Registering and unregistering resources

Resources are registered when a service calls the *registerProducerTable* and *getMatchingProducersForTables* operations, and are unregistered when a service calls the *unregisterProducerTable* or *unregisterContinuousConsumer* operations. The registration calls must be repeated periodically with the interval between calls not exceeding the termination interval, otherwise the registry will automatically unregister the resource. If the re-registration arrives late, it will simply put the entry back as if it was a new entry. The registry instance which hosts the replica notes the updates it receives and uses these to generate an *addReplica* call to send updates to all other registries it knows about.

9.3.2 Forwarding operations

Most registry operations specify the name of the virtual database for which they are intended. If the Registry Service is hosting a registry replica for that virtual database, it will process the operation locally. If it is not, it will attempt to locate another Registry Service that is hosting a replica, then check that the replica is still working (trying a different service if it is not), before forwarding the operation to that service and returning the results. This forwarding can be prevented by setting the *canForward* to *false* in those operations that support it: this is meant for use by Registry Services only, to prevent an operation from being forwarded more than once, and is hidden by the User APIs. How the Registry Service obtains URLs of other Registry Services and lists of the virtual databases for which they are hosting replicas, is described in 1.5. The rules about when to switch from one replica to another are given in the next section.

9.3.3 Replication

When a Registry Service receives a request intended for a replica in a virtual database that it has not contacted before, it chooses the "best" replica in some implementation-defined way that is likely to be based on round-trip times for calls to *ping* on each replica. It then uses this replica for all subsequent requests for that virtual database, until it is forced to switch either because the replica fails, or it discovers that new replicas have been added (e.g. for load balancing).

The Registry Service runs a replication cycle, with a site-configurable frequency, for each replica that it is hosting locally. At each cycle, it sends updates made to the replica to all other replicas in the same virtual database, by calling *addReplica*. As far as possible, change-only updates are used to minimize the size of the replication messages.

A new registry replica will be empty initially. A restarted replica will reload any entries it had before. In both cases, the replica will wait until one full replication cycle has been attempted before it will start servicing queries (regardless of whether or not it has heard from all other replicas).

Since replicas will become inconsistent with each other between replication cycles, all R-GMA services must tolerate a registry informing them about producer and consumer resources that no longer exist, or failing to inform them about ones about which it does not yet know.

9.3.4 REGISTRY DATABASE

The Registry maintains two lists, one containing an entry for each table of each producer publishing to the virtual database, and one containing an entry for each continuous consumer querying the virtual database. How these are stored is implementation dependent, but they're likely to be database tables. It is a requirement that their contents can survive the Registry Service being restarted. The lists need to contain just sufficient details for the Registry Service operations to be supported.

10 SCHEMA SERVICE

10.1 DESCRIPTION

There is exactly one schema per virtual database. It holds the names and definitions of all of the tables in the virtual database, and their authorization rules. Each server holds a copy of the schema for each virtual database it supports. Unlike the registry, requests are not forwarded as each server has its own copy. As part of the R-GMA configuration one schema should be designated as master for each virtual database. Each server is configured with a file for each virtual data base it supports identifying the master schema. The slaves poll the master periodically to obtain updates.

The Schema Service is responsible for authenticating all clients and services that connect to it, and for authorizing all operations and all requests to access the schemas it is hosting, as specified in chapter 13.

Note that all operations that change the schema return a boolean flag *changed* which is true if the state of the schema has (or may have been changed). This is for internal use as changes are made first to the master schema and then if changes have been made the local schema is updated from the master. The flag is not exposed by the user API.

10.2 INTERFACE

10.2.1 USER INTERFACE

The following operations are available to client applications.

createTable

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string createTableStatement	SQL CREATE TABLE statement.
⇒	rgma:StringList tableAuthz	Table authorization rules.
⇐	Tuple	
	xsd:boolean changed	True if any changes made.

Adds a new table definition to the schema of the requested virtual database. Rules for naming tables and columns, supported column types and the format of the CREATE TABLE statement are all specified in chapter 14. The authorization rules are described in chapter 13.

dropTable

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string tableName	Table name.
⇐	Tuple	
	xsd:boolean changed	True if any changes made.

Drops a table from the schema of the requested virtual database.

alter

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string torv	TABLE or VIEW.
⇒	xsd:string tableName	Table name.
⇒	xsd:string action	ADD, DROP or MODIFY.
⇒	xsd:string name	Column name.
⇒	xsd:string type (0..1)	Column type.
⇐	Tuple	
	xsd:boolean changed	True if any changes made.

Alter a table defintion. *add* and *drop* are permitted for a table or view and *modify* to a table. The column type must be specified for *add* and *modify* on a table and in no other case.

createIndex

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string createIndexStatement	SQL CREATE INDEX statement.
⇐	Tuple	
	xsd:boolean changed	True if any changes made.

Adds a new index definition for an existing table in a schema of the requested virtual database. The format of the CREATE INDEX statement is specified in 14.5. Indexes cannot be defined for views.

dropIndex

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string indexName	Index name.
⇐	Tuple	
	xsd:boolean changed	True if any changes made.

Drops an index from the schema of the requested virtual database.

createView

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string createViewStatement	SQL CREATE VIEW statement.
⇒	rgma:StringList viewAuthz	View authorization rules.
⇐	Tuple	
	xsd:boolean changed	True if any changes made.

Adds a new view definition on an existing table in a schema of the requested virtual database. Views are described in 13.6.4 and the format of the CREATE VIEW statement is specified in 14.6.

dropView

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string viewName	View name.
⇐	Tuple	
	xsd:boolean changed	True if any changes made.

Drops a view from the schema of the requested virtual database.

getAllTables

⇒	xsd:string vdbName	Virtual database name.
⇐	Tuple (0..*)	
	xsd:string tableName	Table or view name.

Returns a list of all table and view names in the schema of the requested virtual database (no distinction is made between tables and views).

getTableDefinition

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string tableName	Table or view name.
⇐	Tuple (0..*)	
	xsd:string tableName	Table or view name.
	xsd:string columnName	Name of column
	xsd:string columnType	Type of column - e.g. REAL or INTEGER
	xsd:integer columnSize	Size of column
	xsd:boolean columnIsNotNull	If column is NOT NULL
	xsd:boolean columnIsPrimaryKey	If column is PRIMARY KEY
	xsd:string viewFor	

Returns a table's column definitions from the schema of the requested virtual database (used by the R-GMA Browser). If *tableName* is a view, only returns details of those columns in the view.

getTableIndexes

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string tableName	Table name.
⇐	Tuple (0..*)	
	xsd:string indexName	Table or index.
	xsd:string columnName	Name of column

Returns a list of the indexes associated with a table (indexes cannot be associated with a view).

setAuthorizationRules

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string tableName	Table or view name.
⇒	rgma:StringList tableAuthz	Table authorization rules.
⇐	Tuple	
	xsd:boolean changed	True if any changes made.

Replaces a table's or view's read/write authorization rules. The authorization rules are described in chapter 13.

getAuthorizationRules

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string tableName	Table or view name.
⇐	Tuple (0..*)	
	xsd:string authzRule	Authorization rule.

Returns the read/write authorization rules associated with a table or view. Authorization rules are described in chapter 13.

See also the operations common to all services: getProperty 11.1.1.

10.2.2 System Interface

The following operations are available to other R-GMA services only.

getTableTimestamp

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:string tableName	Table name.
⇐	Tuple	
	xsd:long timeStamp	TimeStamp in ms since 1970.

Returns timestamp when table definition was last modified.

getSchemaUpdates

⇒	xsd:string vdbName	Virtual database name.
⇒	xsd:long timeStamp	TimeStamp in ms since 1970.
	xsd:string updates	Table details.

Returns full information for each table that has changed since the specified timestamp - which may be zero.

10.2.3 Replication

Periodically each slave schema sends a request to the master for updates.

10.2.4 Schema Database

The Schema maintains lists of the definitions of all tables in the virtual database, and their access permissions. How these are stored is implementation dependent, but they're likely to be database tables. It is a requirement that their contents can survive the Schema Service being restarted. The lists need to contain just sufficient details for the Schema Service operations to be supported.

11 SERVICE OPERATIONS

Operations defined here are are common to all services.

11.1 INTERFACE

11.1.1 USER INTERFACE

getProperty

⇒	xsd:string name	Property name.
⇒	xsd:string(0..1) parameter	Property parameter.
⇐	xml:Any	Property value.

Returns the current value of the requested service property. The appropriate contents of *parameter* depend on the property name. The list of service properties is extensible and subject to change without warning.

11.1.2 SYSTEM INTERFACE

12 RGMAService Operations

The RGMAService is for functions not directly related to producers, consumers, registry or schema.

12.1 Interface

12.1.1 User Interface

listTupleStores

⇐ Tuple(0..*)

xsd:string logicalName	Logical name of tuple store.
xsd:boolean isHistory	If logical name is associated with a history store.
xsd:boolean isLatest	If logical name is associated with a latest store.

Returns a list of all of the permanent tuple stores. A logical name may be associated with both history and latest stores. Only the user's own tuple stores are listed.

dropTupleStore

⇒	xsd:string logicalName	Logical name of tuple store.
⇐	Tuple(1..1)	
	xsd:string status	OK

Permanently deletes a permanent tuple store, provided it is not currently being used by a producer. Only the user's own tuple stores can be dropped.

getVersion

⇐ Tuple

xsd:string version	Service version.

Returns the version number of the services.

getTerminationInterval

⇐ Tuple)

xsd:int terminationIntervalSec	Termination interval in seconds.

Returns the current termination interval defined by the services.

13 SECURITY

13.1 INTRODUCTION

In a secure installation, R-GMA services are responsible for ensuring the integrity and confidentiality of the virtual databases and all user data held within the service infrastructure. This chapter specifies the security mechanisms used by the services to meet this responsibility. Sites can opt to run their services with these security mechanisms turned off, but other R-GMA services running securely will not communicate with them.

13.2 RISKS TO SERVICES AND DATA

The integrity of a virtual database is compromised if a person maliciously modifies data or causes operations to execute incorrectly so that the virtual database reaches an invalid state or returns invalid results. This can be achieved either by direct attacks on the service infrastructure or by abusing the operations provided by services.

The integrity and confidentiality of user data are compromised if a person can read or modify the data without the owner's consent. User data in R-GMA does not just include tuples, but also query strings (because predicates may contain user data), resource connection details (because these could be used to launch attacks), user identification details stored for security purposes and user names and passwords for external data stores. Data integrity and confidentiality can be compromised by a malicious person making direct attacks on the service infrastructure or abusing the operations provided by services. They can also be compromised by the failure of a service implementation to properly enforce data access rules.

13.3 SECURING SERVICES AND DATA

The service infrastructure consists of the underlying operating system, the servlet container, external storage used by the services (e.g. databases, configuration files, log files), connections between services and external storage (e.g. JDBC links), and network connections to services. With the exception of the network connections, it is beyond the scope of this specification to say how these can be made secure, but it is mandatory for all R-GMA service implementations to address the security of each of these components.

How R-GMA services secure network transfers against interception and modification, how they control connections to services and how they control execute access to operations and read and write access to data stores, are all covered in the remainder of this chapter.

13.4 SECURING NETWORK TRANSERS

All user-to-service and service-to-service connections (including streaming) in a secure R-GMA installation use SSL to encrypt the data as it travels over the network. Data is *not* encrypted by R-GMA within its services. If users don't trust the security of R-GMA's services, they must encrypt the data themselves before sending it to R-GMA.

13.5 AUTHENTICATION

R-GMA services must immediately terminate a user's or other service's attempt to connect to them if their identity cannot be authenticated. If a user chooses to use one of the R-GMA APIs, the API will likewise abandon a connection immediately if it cannot authenticate the service to which it is connecting

(this is *mutual authentication*). The APIs return an appropriate RGMASecurityException in response to authentication failures.

Mutual authentication is based on an exchange of encrypted messages. Each party's identity is proved by their possession of a private key corresponding to the public key embedded in a digital certificate signed by a certificate authority recognised by the other party. Currently R-GMA services accept only X509 format certificates (including proxy certificates generated by grid-proxy-init or voms-proxy-init).

The mechanism used for authentication and for managing Certificate Authority (CA) certificates and their supporting files (revocation lists, signing policies, etc) is implementation-dependent.

13.6 AUTHORIZATION

13.6.1 CREDENTIALS

R-GMA services permit or deny access to resources (in the general sense) to a user or another service, on the basis of *credentials* held by that user or service. Credentials are extracted from the certificate used for authentication in some implementation-defined way. Each credential has a name such as the Distinguished Name (DN) credential found in all certificates[3]. The credentials found in the *VOMS proxy certificates* used by all EGEE services are:

- Membership of virtual organizations (VO);
- Membership of groups within a virtual organization (GROUP);
- Roles held within a virtual organization (ROLE);

and some, such as GROUP, may take more than one value. All authorization rules are defined in terms of combinations of zero or more credentials, and whilst R-GMA service implementations are obliged to support VOMS credentials, they will actually work with any type of credential.

13.6.2 OWNERSHIP

The servers on which R-GMA services run are owned by individual sites, so it is they who decide whether or not to run the services.

Each virtual database has an owner (typically a VO) and registry instances and the master schema are hosted by sites by mutual agreement with the virtual database owner. The virtual database owner owns the registry and schema instances and decides who can read or modify them, but they have to trust the individual sites to implement their policy, and the sites retain overall control.

Virtual database owners control who can create, read or modify information the schema and who can read tuples from their virtual database and who can publish tuples to it, because they own the schema and the data access rules are defined there. Anybody permitted to write to a table defintion in the schema can also control who can read or write to that table within the virtual database.

13.6.3 AUTHORIZATION POINTS

Access to the tuples in the tuple stores is authorized using *read* and *write* rules set for each table and view in the schema. The rule-based mechanisms are described below.

Access to the schema information is controlled by rules stored as part of the VDB in addition to rules stored with a table defintion in the schema.

[3] The DN of host certificates contains the host name.

13.6.4 RESTRICTING ACCESS TO COLUMNS (VIEWS)

Views are defined in the schema as a subset of the columns of a table (the *base table* of the view), created using the Schema service's *createView* operation (see 14.6). Users can query views just as they would real tables, but they are read-only, so producers cannot publish to them, and they do not appear in the registry. Their purpose is to limit access to only certain columns of the base table: tuples read from a view are actually read from the base table, but the query is only allowed to include columns that form part of the view. Authorization rules for row access to views are defined exactly as for tables (see the next section) and completely replace the rules on the base table.

13.6.5 AUTHORIZING READ/WRITE ACCESS TO ROWS IN TABLES OR VIEWS

All consumers (including those used by Secondary Producers) require read authorization to read tuples from views or tables in the virtual database and all producers (Primary, Secondary and On-demand) require write authorization to publish tuples into tables in the virtual database.

These authorization rules are defined in the schema and the virtual database owner decides who can change authorization rules in the schema by authorizing the change operation as described above; the user who creates the table or view sets the initial rules.

The producer services are responsible for enforcing the rules. In a Primary Producer service, the write rules are applied in calls to *insert* and *insertList*, and the read rules are applied in calls to *start*. In a Secondary Producer service, the write rules are ignored but the read rules are applied as in a Primary Producer service. In an On-demand Producer service, the write rules, followed by the read rules, are both applied in calls to *start*. In all cases, the producer services must notice changes to the data access rules and apply the changes to any tuples subsequently sent to a consumer service. Note that enforcing read rules in the producer services requires the consumer services to be able to prove that they are acting on behalf of the user or service to which read access is being granted (this requires *delegation* of credentials).

The default rule is that no user is permitted any access to the table or view. Rules are added to grant access only (not to deny it) and they are cumulative. Read and write rules are formatted identically and have the form "*predicate : credentials : action*", where the three clauses are defined as follows:

predicate Defines the subset of rows of the table or view to which this rule grants access. It is an SQL WHERE clause that compares the values in specified columns with constants, other columns or credential parameters (credential name in square brackets, such as [DN]) that are replaced by the corresponding credentials (or set of credentials) extracted from the user's certificate when the rule is evaluated. The predicate expression is restricted to the following operators: AND, OR, NOT, IN, =, LIKE, <>, <, > and IS NULL. This clause may be empty, in which case the rule applies to all rows in the table.

credentials Defines the set of credentials required for a user to be granted access to the subset of rows defined by this rule. It is a boolean combination of equality constraints of the form *[credential]* = *constant*. This clause may be empty, in which case the rule applies to all authenticated users.

action Defines what any matching user is allowed to do to the subset of rows defined by this rule. The valid values are: *R* (read), *W* (publish) or *RW* (read and publish). This clause must not be empty.

Some examples rules are:

```
WHERE Section = 'Marketing':[GROUP] = 'Marketing' OR [GROUP] = 'Management':RW
```

that grants read-write access to any authenticated user with a GROUP credential of 'Marketing' or 'Management', to those rows that contain the value 'Marketing' in the 'Section' column;

```
WHERE Owner = [DN]::R
```

that grants read-only access to any authenticated user, to those rows that contain the value of their DN credential in the 'Owner' column;

```
WHERE Group = [GROUP] OR Public = 'true'::R
```

that grants read-only access to any authenticated user, to those rows that contains one of their GROUP credential values in the 'Group' column, or have a value of 'true' in the 'Public' column;

```
::R
```

that grants read-only access to any authenticated user, to all rows in the table.

13.6.6 AUTHORIZING ACCESS TO THE SCHEMA

This is controlled by similar rules but without a predicate so the take the form: "*credentials : action*" and so can be distinguised by having only one ":". Possible actions are: *R* (read or list schema entry), *W* (write - i.e. modify an existing schema entry), *C* (create a table) or any permutation of these letters. An empty authorization is not permitted. Rules may be stored with the table defintion in the schema as well as in the defintion of the VDB. The *C* action is only permitted in the VDB definition (there is no meaning to telling a table that it can create itself). Rules only allow operation and do not deny them. As a consequence if a VDB has decided to have a policy of :W then all users of the VDB can modify any table defintion and the creator of a table is unable to restrict this.

13.7 PROVING SERVICES ARE GENUINE

Users determine that R-GMA services are genuine by obtaining their URLs from a trusted source. R-GMA Producer and Consumer services only connect to local Registry and Schema services, and only connect to Producer and Consumer services whose URLs were passed to them by other R-GMA services that they have authenticated and authorized. R-GMA services however communicate with one another and no delagtion mechanism is in use. Instead the set of servers is a part of the VDB definition so the VDB defintion must be carefully controlled.

14 SQL IN R-GMA

R-GMA virtual databases are queried and managed using SQL. It is used for creating and dropping tables, views and indexes, defining producers' predicates, inserting tuples, and querying the virtual database. All SQL statements are parsed for security and validation purposes and are never simply forwarded to an underlying database for execution.

R-GMA aims to be fully compatible with SQL92, but it does not claim compliance. As in SQL92, SQL keywords are not case sensitive in R-GMA.

14.1 CHARACTER SETS

R-GMA only supports the ASCII character set for all character data passed to its services (this includes table, column, index and view names and tuple data). The collation sequence used in consumer queries is the default collation sequence of the underlying database for the ASCII character set.

14.2 NAMING TABLES, COLUMNS, INDEXES AND VIEWS

For compatibility with SQL92, SQL identifiers (table names, column names, index names and view names) defined in an R-GMA schema must consists of at most 128 upper or lower case letters, digits or embedded (not leading or trailing) underscores, and must begin with a letter. Identifiers beginning with *Rgma* are reserved for use by R-GMA in addition to all names reserved by SQL92. R-GMA stores identifiers in their original case, but maps all identifiers to upper case for matching purposes, so you may not, for example, have two table names in the same schema that differ only in case.

Virtual database names (used as prefixes to table and view names) also conform to these rules, except that they are allowed to include embedded dots.

14.3 CREATING TABLES

Tables are created in the virtual database by calling the Schema's *createTable* operation and passing an SQL CREATE TABLE statement conforming to the following specification:

CREATE TABLE *table-name* (*column-name column-type* [*column-qualifier*], . . .)

where *table-name* is a unique table name, *column-name* is a column name and *column-type* is one of the data type names listed below. The optional *column-qualifier* can be either *NOT NULL* indicating that NULL values cannot be inserted to the column, or *PRIMARY KEY* indicating that the column forms part of the primary key for the table (and implies *NOT NULL*). The primary key can also be defined by appending an expression of the following form to the end of the column list.

PRIMARY KEY(*column-name*, . . .)

The primary key is discussed further in 1.2.5. Tables can be dropped using the *dropTable* operation.

14.4 DATA TYPES SUPPORTED

R-GMA supports the following data types:

- INTEGER (signed integer capable of being stored in Java *int* (32-bit two's complement) without loss of precision)

- REAL (floating point number capable of being stored in Java *float* (32-bit IEEE 754) without loss of precision)
- DOUBLE PRECISION (floating point number capable of being stored in Java *double* (64-bit IEEE 754) without loss of precision)
- DATE (Date string in the format 'YYYY-MM-DD')
- TIME(*n*) (UTC time string in the format 'hh:mm:s[.s]', where '.s' means between zero and *n* decimal places; *n* is at most 9 and defaults to 0; leap seconds (60 or 61) are permitted)
- TIMESTAMP(*n*) (UTC timestamp string in the format 'YYYY-MM-DD hh:mm:s[.s]', where the date part is as in DATE and the time part is as in TIME)
- CHAR(*n*) (character string of fixed length *n* where *n* is greater than zero and defaults to 1)
- VARCHAR(*n*) (character string of variable length up to *n*, where *n* is greater than zero and has no default)

For compatibility with SQL92, R-GMA permits the keywords DATE, TIME and TIMESTAMP to appear before literals of those types, but it does not require them.

These are the only type names that may appear in a CREATE TABLE statement and they are the only type names that will appear in the column metadata of a result set. Producer Services must guarantee that they will support the full range of each column type (and the full length of string types) in any table that they publish, or they must fail the call to *declareTable*.

Null values are represented by the unquoted, case-insensitive word NULL: empty strings are *not* considered to be NULL values. Character strings are delimited by single quotes only, and single quotes can only be embedded in them by duplicating them. Timestamps inserted with more precision than a table supports will be rounded to the supported precision. Floating point values may be represented using decimal (e.g. 1.2) or scientific notation (e.g. 1.2E3 or 1.2e3, where the exponent may be positive or negative). Floating point values inserted into columns of type INTEGER will be truncated at the decimal point.

14.5 Creating Indexes

Indexes are created in the virtual database by calling the Schema's *createIndex* operation and passing an SQL CREATE INDEX statement, conforming to the following specification:

CREATE INDEX *index-name* ON *table-name* (*column-name*, ...)

where *index-name* is a unique index name, *table-name* is the table to be indexed and *column-name* is one of the columns to be indexed. Indexes can be dropped using the *dropIndex* operation. Producer Service implementations are encouraged but not obliged to respect table indexes.

14.6 Creating Views

Views are created in the virtual database by calling the Schema's *createView* operation and passing an SQL CREATE VIEW statement, conforming to the following specification:

CREATE VIEW *view-name* AS SELECT *column-name*, ... FROM *table-name*

where *view-name* is the view name, *column-name* is one of the columns in the view and *table-name* is the table on which to create the view. The view name must not be the same as any other view name or table name in the schema. R-GMA only supports a column list, one table and no predicate in a view definition. Views can be dropped using the *dropView* operation. The role of views in R-GMA is explained in 13.6.4.

14.7 Inserting Tuples

The Primary Producer's *insert* operation allows tuples to be inserted into the virtual database. It takes an SQL INSERT statement conforming to the following specification:

INSERT INTO *table-name* (*column-name*, ...) VALUES (*value*, ...)

where *table-name* is the table name, *column-name* is a column name and *value* is its corresponding value, formatted according to the rules in section 14.4. The table name must be be prefixed by a virtual database name, separated from the table name by a dot.

14.8 Producer Predicates

When a producer registers its intention to publish to a table in the virtual database, it must specify a predicate that declares the subset of the table to which it will publish tuples. This may be an empty string (indicating the whole table) or an SQL WHERE clause. To simplify mediation, it is limited to a union of column equality constraints, like this[4]:

WHERE *column* = *constant* AND *column* = *constant* AND ...

Note that a Primary or On-demand Producer's predicate only limits the tuples it can publish; it does *not* mean that it can be used as a single source for tuples matching that predicate. Conversely, by construction, a Secondary Producer *can* be used as a single source for tuples matching its predicate.

14.9 Consumer Queries

Users query the virtual database by creating a consumer using the Consumer Service's *createConsumer* operation and passing an SQL SELECT statement.

For mediation purposes, R-GMA defines two classes of query: *simple* and *complex*. Which class of query is supported at any given time depends on the query type (*continuous*, *latest*, *history* and *static*) and the producers available at the time.

14.9.1 Complex Queries

R-GMA supports a subset of SQL92 in the SQL SELECT queries of its consumers. The BNF defining the set of supported queries is given below:

```
SelectStatement    ::= SelectWithoutOrder [OrderByClause]
SelectWithoutOrder ::= "SELECT" ["ALL"|"DISTINCT"] SelectList FromClause
                       [WhereClause] [GroupByClause] ["HAVING" SQLExpression]
WhereClause        ::= "WHERE" SQLExpression
```

[4] It is intended to extend R-GMA to support any predicate involving columns and constants in disjunctive normal form and to support string ranges (such as "column > 'A' " and "column in ('A', ...)").

```
GroupByClause      ::= "GROUP" "BY" SQLExpressionList
OrderByClause      ::= "ORDER" "BY" SQLSimpleExpression ["ASC"|"DESC"]
                       ("," SQLSimpleExpression ["ASC"|"DESC"])*
SelectList         ::= "*" | "COUNT(*)" | SelectItem ("," SelectItem)* |
                       "COUNT(" ["ALL"|"DISTINCT"] ObjectName ")"
SelectItem         ::= (SQLSimpleExpression | TableColumn) [ ["AS"] Identifier ]
FromClause         ::= "FROM" TableReference (, TableReference)*
TableReference     ::= "(" TableReference ")" |
                       RGMATableName [ ["AS"] Identifier ] JoinedTable
RGMATableName      ::= [ ( "{" ObjectName (, ObjectName)* "}" | ObjectName ) "." ] ObjectName
JoinedTable        ::= ["NATURAL"] ("INNER"|OuterJoinType ["OUTER"]|"UNION") "JOIN"
                       TableReference [JoinSpecification]
OuterJoinType      ::= "LEFT"|"RIGHT"|"FULL"
JoinSpecification  ::= "ON" SQLExpression |
                       "USING" "(" ObjectName ("," ObjectName)* ")"
SQLExpressionList  ::= SQLSimpleExpression ("," SQLSimpleExpression)*

SQLExpression is a boolean expression involving any of:
    OR AND NOT IN BETWEEN LIKE IS NULL <SQLSimpleExpression>

SQLSimpleExpression is an expression involving any of:
    + - || * / ** NULL COUNT SUM AVG MAX MIN <column-name> <number> <string>
```

Table names anywhere in the SELECT statement must be prefixed by a virtual database name, separated from the table name by a dot (as in *DB.Table*), in order to indicate which virtual database contains the required table. Joins across multiple virtual databases are valid queries, although whether or not they can be answered depends on the availability of appropriate producers at the time.

Table names in the FROM clause must be prefixed by a virtual database name, separated from the table name by a dot (as in *DB.Table*), in order to indicate which virtual database contains the required table. The database name must not be used anywhere else in the SELECT statement. If columns in two tables need to be differentiated, an alias for the table should be used.

14.9.2 SIMPLE QUERIES

A simple query is one of the form:

```
SELECT <column-expression> FROM <table-name> WHERE <predicate>
```

where:

- <column-expression> contains only column names, constants and the mathematical operators for addition, subtraction, multiplication and division (+, -, * and /)
- <table-name> contains only one VDB-prefixed table or view name, although the VDB prefix *is* allowed to be a list (in curly brackets as above)
- <predicate> contains only column names, constants, the comparison operators =, >, <, >=, <=, <> and LIKE and the boolean operator AND.

14.10 DATA INTEGRITY

Tuples enter R-GMA in Primary Producers' INSERT statements, and On-Demand Producers' query responses. They leave R-GMA in Consumers' result sets. In all three cases, data values are transported as strings, regardless of their declared SQL data type. Since R-GMA services must internally process and store these values, they may have to convert the data values into native types at various stages of processing, and this has the potential to degrade some types of data. R-GMA makes the following guarantees about data integrity:

- Data values with integer and string types (INTEGER, CHAR, VARCHAR) will pass through R-GMA unchanged.
- Data values with REAL and DOUBLE types will be stored and manipulated using types with at least as many bits as required by the table definition, but some degradation may occur in values that approach the limits of this precision. R-GMA will only throw an exception if precision is lost in the most significant digits (in line with SQL92).
- Data values with TIMESTAMP types will be preserved up to the precision specified for the corresponding table column.

INDEX