Doradus Spider Database

1. Introduction

This document describes the Doradus Spider database, which is a Doradus server configured to use the Spider storage service to manage data. The Spider storage manager provides flexible indexing and update features that benefit specific types of applications. This document describes the unique features of the Spider service includes its data model, query language, and REST API.

Doradus Spider builds upon the Doradus core data model and query language (DQL). This document provides an overview of these topics and provides Spider-specific examples. The following documents are also available:

- **Doradus Data Model and Query Language**: Provides a detailed description of the core Doradus data model, query language (DQL), and object and aggregate query commands.
- **Doradus OLAP Database**: Describes the features, data model extensions, and REST commands specific to Doradus OLAP, which is an alternative storage service to Doradus Spider.
- **Doradus Administration**: Describes how to install and configure Doradus for various deployment scenarios, and operational topics such as security, monitoring, and the JMX API.

This document is organized into the following sections:

- **Spider Database Overview**: An overview of the Spider database including its unique features and the type of applications it is best suited for.
- **Spider Data Model**: Summarizes core Doradus data model and describes the extensions specific to Doradus Spider.
- **Doradus Query Language**: Provides a reference of core DQL concepts and describes extensions specific to Doradus Spider databases.
- **Spider REST Commands**: Describes the REST commands supported by Spider databases for application management, updates, queries, and task management.

2. Spider Database Overview

The motivation and general architecture of Doradus is described in the document **Doradus Data Model and Query Language**. Familiarity with that document is assumed here, which describes the usage and unique features of Doradus Spider.

2.1 Starting a Spider Database

The Doradus server is configured as a Spider database when the following option is used in the doradus.yaml file:

```
storage_services:
    - com.dell.doradus.service.spider.SpiderService
```

When the Doradus server starts, this option initializes the `SpiderService` and places new applications under the control of the Spider storage service by default. Details of installing and configuring the Doradus server are covered in the **Doradus Administration** document, but here's a review of three ways to start Doradus:

- **Console app**: The Doradus server can be started as a console app with the command line:

  ```
  java com.dell.doradus.core.DoradusServer [arguments]
  ```

  where `[arguments]` override values in the doradus.yaml file. For example, the argument "`-restport 5711`" set the REST API port to 5711.

- **Windows service**: Doradus can be started as a Windows service by using the procrun package from Apache Commons. See the **Doradus Administration** document for details.

- **Embedded app**: Doradus can be embedded in another JVM process. It is started by calling the following method:

  ```
  com.dell.doradus.core.DoradusServer.startEmbedded(String[] args, String[] services)
  ```

  where:

  - `args` is the same arguments parameter passed to `main()` when started as a console application. For example `{"-restport", "5711"}` sets the REST port to 5711.

  - `services` is the list of Doradus services to initialize. Each string must be the full package name of a service. The current set of available services is listed below:

    ```
    com.dell.doradus.service.db.DBService: Database layer (required)
    com.dell.doradus.service.schema.SchemaService: Schema services (required)
    com.dell.doradus.mbeans.MBeanService: JMX interface
    com.dell.doradus.service.rest.RESTService: REST API
    com.dell.doradus.service.taskmanager.TaskManagerService: Background task execution
    com.dell.doradus.service.spider.SpiderService: Spider storage service
    com.dell.doradus.service.olap.OLAPService: OLAP storage service
    ```

Required services are automatically included. An embedded application must list at least one storage service. Other services should be listed when the corresponding functionality is needed.

If the Doradus server is configured to use multiple storage services, or if Spider is not the default storage service (the first one listed in doradus.yaml), an application can explicitly choose Spider in its schema by setting the application-level `StorageService` option to `SpiderService`. For example, in JSON:

```
{"Msgs": {
   "options": {"StorageService": "SpiderService"},
   ...
}
```

2.2 Spider Feature Overview

Doradus is a server application that runs on top of a NoSQL database, providing high-value features that make the database easier to use and more valuable to applications. Doradus currently runs on top of the Cassandra database, though its architecture allows it to use other persistence engines. The architecture also allows different *storage services* to be used, each offering indexing, storage, and query techniques that benefit specific application types.

Spider is a one of the available storage services: the other is the OLAP service. Spider supports the core Doradus data model and query language (DQL) and extends these with the following unique features:

- **Fully-inverted indexing**: Spider offers *field analyzers* that can index all scalar fields of all objects. Dictionaries and term vectors are used for text fields, and trie trees are used for numeric and timestamp fields. These techniques provide efficient searching for full and partial values as well as range searching.
- **Stored-only fields**: Indexing can selectively be disabled for scalar fields so they can be stored and retrieved without consuming indexing space.
- **Dynamic fields**: Fields do not have to be predefined in the schema and can be added dynamically on a per-object basis. Dynamically-added fields are fully indexed. Objects in the same table can have different sets of fields. An object can even have millions of fields.
- **Query extensions**: Spider fully supports DQL object and aggregate queries and provides extensions that leverage fully-inverted indexing. For example, in addition to field-specific *term*, *phrase*, and *equality* clauses, Spider allows "any field" clauses that perform these searches on all fields within a table, including dynamically-added fields.
- **Extended aggregate grouping**: Spider extends Doradus aggregate queries with special *compound* and *composite grouping* features. Compound grouping allows one or more metric functions to be computed with multiple grouping sets using a single pass through the data. Composite grouping performs group-level "roll-up" metric computations for non-leaf groups.

- **Statistics**: Spider allows aggregate queries to be predefined in the schema as *statistics*, which are computed and refreshed in background tasks. Whole or partial statistic sets can then be retrieved quickly via the REST API. This allows complex, longer-running aggregate queries to be computed asynchronously while allowing applications to quickly retrieve the latest values.

- **Table-level sharding**: Traditional inverted indexing techniques experience performance issues when the number of indexed objects grows to millions and beyond. Spider offers table-level, time-based *sharding*, which splits indexing records into time-based shards, which maintain efficient search performance for time-oriented queries. Both scalar and link fields can leverage time-based sharding.

- **Table-level aging**: Spider allows each table to define a timestamp *aging* field, which is used to automatically delete expired objects. Data aging is performed in background tasks with definable schedules.

- **Background tasks**: In addition to statistics refreshing and data aging, Spider uses background tasks to perform data integrity and management tasks. For example, tasks can be requested to re-index a field whose type has changed or to delete obsolete data for a deleted field. All background tasks have independent cron-based schedules, and REST commands are provided to start, stop, and modify individual tasks.

2.3 When Spider Works Best

Doradus Spider databases are designed to support applications with one or more of the following needs:

- **Variable structure**: Data can vary from structured to highly unstructured. For example, an application that harvests data from web pages, emails, or file servers may dynamically discover fields it wants to store and index. Even for structured data, in which all required fields are predefined in the schema, Spider only consumes space for those fields that actually have values. Both predefined and dynamically-defined fields can be queried via DQL.

- **Immediate indexing**: Fields are indexed as they are stored, making them immediately visible to queries.

- **Document management**: Compared to Doradus OLAP, Spider is better suited for storing and indexing large content objects such as documents, files, and messages. Text and binary fields up to 10MB or more should work well.

- **Fine-grained updates**: Both batch and single-object updates are efficient. Frequent updates to single objects and even single fields are quick and immediately reflected in indexes. Data aging allows expiration of each object based on its own schedule.

- **Complex aggregate queries**: DQL extensions such as compound/composite grouping and stored queries (statistics) provide a wide range of ways in which aggregate queries can be used.

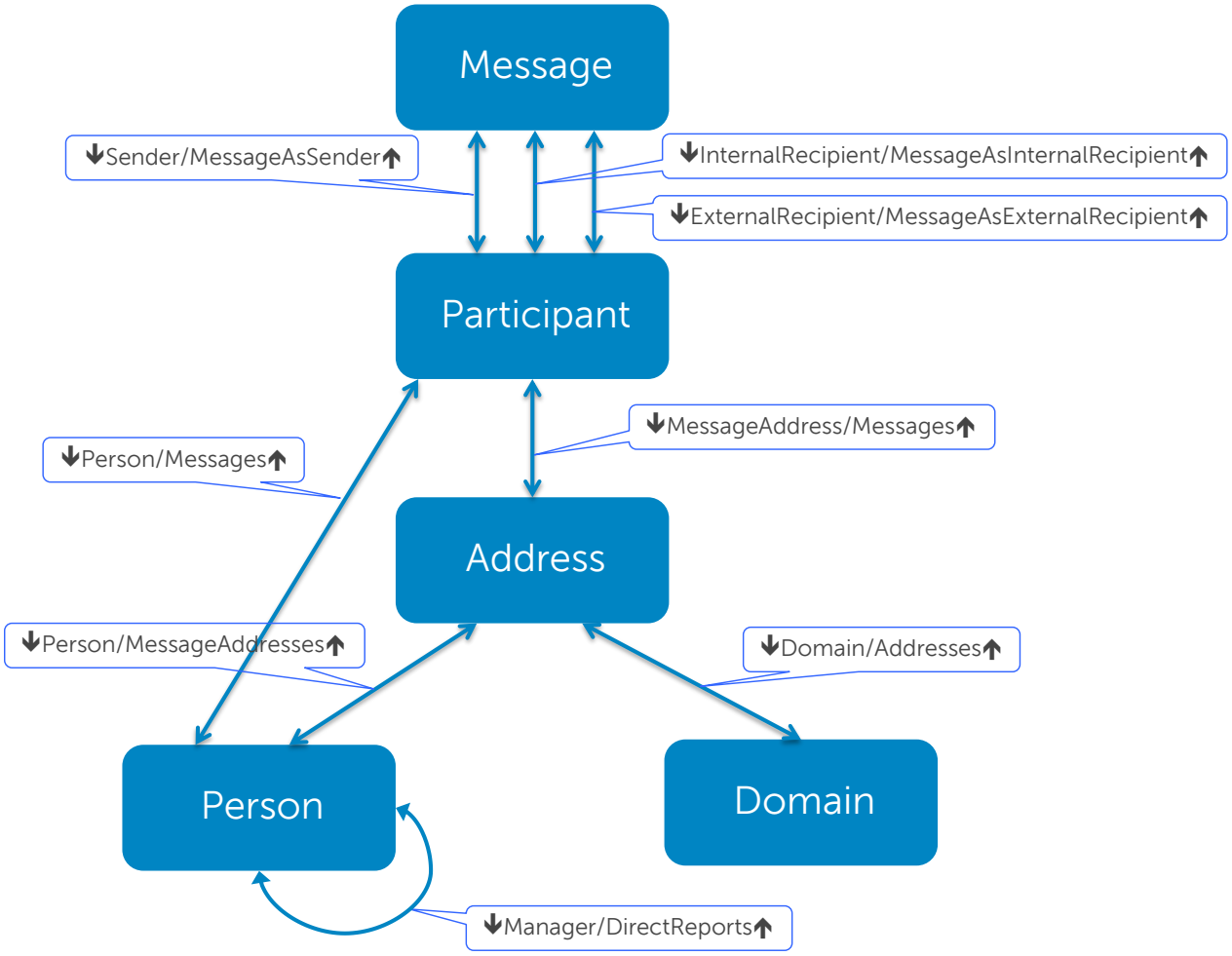
Conversely, Doradus Spider is not a good choice in the following scenarios:

- **Immutable, structured data**: Spider supports these applications, but Doradus OLAP provides faster queries and denser space storage for this scenario. Unless immediate indexing or fine-grained updates are required, OLAP is a better choice for immutable or semi-mutable, structured data.

- **NoSQL Anti-patterns**: Spider is not a good choice when a simpler database such as a persistent hash table will suffice, nor for NoSQL anti-patterns such as applications that need ACID transactions. Applications that want a *persistent queue* are not a good fit since objects are not intended to be short-lived. Very large object (BLOB) storage is also not a good fit since each field is stored in a column, and Cassandra loads whole column values into memory – streaming is not supported.

2.4 The `Msgs` Sample Application

To illustrate features, a sample application called `Msgs` is used. The `Msgs` application schema is depicted below:



The tables in the Msgs application are used as follows:

- Message: Holds one object per *sent* email. Each object stores values such as Body, Subject, Size, and SendDate and links to Participant objects in three different roles: sender, internal recipient, and external recipient.
- Participant: Represents a sender or receiver of the linked Message. Holds the ReceiptDate timestamp for that participant and links to identifying Person and Address objects.
- Address: Stores each participant's email address, a redundant link to Person, and a link to a Domain object.
- Person: Stores person Directory Server properties such as a Name, Department, and Office.
- Domain: Stores unique domain names such as "yahoo.com".

In the diagram, relationships are represented by their link name pairs with arrows pointing to each link's *extent*. For example, ↓Sender is a link owned by Message, pointing to Participant, and MessageAsSender↑ is the inverse link of the same relationship. The Manager and DirectReports links form a *reflexive* relationship within Person, representing an org chart.

The schema for the Msgs application is shown below in XML:

```
<application name="Msgs">
    <key>MsgsKey</key>
    <options>
        <option name="AutoTables">false</option>
        <option name="StorageService">SpiderService</option>
    </options>
    <tables>
        <table name="Address">
            <fields>
                <field name="Domain" type="LINK" inverse="Addresses" table="Domain"/>
                <field name="Messages" type="LINK" inverse="MessageAddress" table="Participant"
                    sharded="true"/>
                <field name="Name" type="TEXT"/>
                <field name="Person" type="LINK" inverse="MessageAddresses" table="Person"/>
            </fields>
        </table>
        <table name="Domain">
            <fields>
                <field name="Addresses" type="LINK" inverse="Domain" table="Address"/>
                <field name="InternalID" type="TEXT" analyzer="NullAnalyzer"/>
                <field name="IsInternal" type="BOOLEAN"/>
                <field name="Name" type="TEXT"/>
            </fields>
        </table>
        <table name="Message">
            <options>
```

```
            <option name="sharding-field">SendDate</option>
            <option name="sharding-granularity">DAY</option>
            <option name="retention-age">5 YEARS</option>
            <option name="sharding-start">2010-07-17</option>
            <option name="aging-field">SendDate</option>
        </options>
        <fields>
            <field name="Body" type="TEXT"/>
            <field name="Participants">
                <fields>
                    <field name="Recipients">
                        <fields>
                            <field name="ExternalRecipients" type="LINK"
                                inverse="MessageAsExternalRecipient" table="Participant"/>
                            <field name="InternalRecipients" type="LINK"
                                inverse="MessageAsInternalRecipient" table="Participant"/>
                        </fields>
                    </field>
                    <field name="Sender" type="LINK" inverse="MessageAsSender"
                        table="Participant"/>
                </fields>
            </field>
            <field name="SendDate" type="TIMESTAMP"/>
            <field name="Size" type="INTEGER"/>
            <field name="Subject" type="TEXT"/>
            <field name="Tags" type="TEXT" collection="true"/>
            <field name="ThreadID" type="TEXT" analyzer="OpaqueTextAnalyzer"/>
        </fields>
        <aliases>
            <alias name="$SalesEmails" expression="Sender.Person.WHERE(Department:Sales)" />
        </aliases>
        <statistics>
            <statistic name="DepartmentTrend" metric="COUNT(*)"
                group="Sender.Person.Department,TRUNCATE(SendDate,DAY)" />
            <statistic name="EmailSizes" metric="COUNT(*)"
                group="BATCH(Size,1000,10000,50000,100000,1000000)" />
            <statistic name="EmailsPerDay" metric="COUNT(*)" group="TRUNCATE(SendDate,DAY)" />
            <statistic name="TotalCount" metric="COUNT(*)"/>
        </statistics>
    </table>
    <table name="Participant">
        <options>
            <option name="sharding-field">ReceiptDate</option>
            <option name="sharding-granularity">DAY</option>
            <option name="sharding-start">2012-07-17</option>
        </options>
        <fields>
            <field name="MessageAddress" type="LINK" inverse="Messages" table="Address"/>
            <field name="MessageAsExternalRecipient" type="LINK" inverse="ExternalRecipients"
                table="Message"/>
```

```
            <field name="MessageAsInternalRecipient" type="LINK" inverse="InternalRecipients"
               table="Message"/>
            <field name="MessageAsSender" type="LINK" inverse="Sender" table="Message"/>
            <field name="Person" type="LINK" inverse="Messages" table="Person"/>
            <field name="ReceiptDate" type="TIMESTAMP"/>
         </fields>
      </table>
      <table name="Person">
         <fields>
            <field name="DirectReports" type="LINK" inverse="Manager" table="Person"/>
            <field name="FirstName" type="TEXT"/>
            <field name="LastName" type="TEXT"/>
            <field name="Location">
               <fields>
                  <field name="Department" type="TEXT"/>
                  <field name="Office" type="TEXT"/>
               </fields>
            </field>
            <field name="Manager" type="LINK" inverse="DirectReports" table="Person"/>
            <field name="MessageAddresses" type="LINK" inverse="Person" table="Address"/>
            <field name="Messages" type="LINK" inverse="Person" table="Participant"/>
            <field name="Name" type="TEXT"/>
         </fields>
      </table>
   </tables>
   <schedules>
      <schedule type="data-aging" value="0 3 * * SAT" table="Message"/>
      <schedule type="stat-refresh" value="*/30 * * * *" table="Message"/>
   </schedules>
</application>
```

The same schema in JSON is shown below:

```
{"Msgs": {
   "key": "MsgsKey",
   "options": {
      "StorageService": "SpiderService",
      "AutoTables": "false"
   },
   "tables": {
      "Message": {
         "options": {
            "sharding-field": "SendDate",
            "sharding-granularity": "DAY",
            "sharding-start": "2010-07-17",
            "aging-field": "SendDate",
            "retention-age": "5 YEARS"
         },
         "fields": {
            "Body": {"type": "TEXT"},
            "Participants": {
```

```
            "fields": {
                "Sender": {"type": "LINK", "table": "Participant", "inverse":
                    "MessageAsSender"},
                "Recipients": {
                    "fields": {
                        "ExternalRecipients": {"type": "LINK", "table": "Participant",
                            "inverse": "MessageAsExternalRecipient"},
                        "InternalRecipients": {"type": "LINK", "table": "Participant",
                            "inverse": "MessageAsInternalRecipient"}
                    }
                }
            }
        },
        "SendDate": {"type": "TIMESTAMP"},
        "Size": {"type": "INTEGER"},
        "Subject": {"type": "TEXT"},
        "Tags": {"collection": "true", "type": "TEXT"},
        "ThreadID": {"type": "TEXT", "analyzer": "OpaqueTextAnalyzer"}
    },
    "aliases": {
        "$SalesEmails": {"expression": "Sender.Person.WHERE(Department:Sales)"}
    },
    "statistics": {
        "TotalCount": {"metric": "COUNT(*)"},
        "EmailSizes": {"metric": "COUNT(*)", "group":
            "BATCH(Size,1000,10000,50000,100000,1000000)"},
        "EmailsPerDay": {"metric": "COUNT(*)", "group": "TRUNCATE(SendDate,DAY)"},
        "DepartmentTrend": {"metric": "COUNT(*)", "group":
            "Sender.Person.Department,TRUNCATE(SendDate,DAY)"}
        }
    },
    "Participant": {
        "options": {
            "sharding-field": "ReceiptDate",
            "sharding-granularity": "DAY",
            "sharding-start": "2012-07-17"
        },
        "fields": {
            "MessageAddress": {"type": "LINK", "table": "Address", "inverse": "Messages"},
            "MessageAsExternalRecipient": {"type": "LINK", "table": "Message", "inverse":
                "ExternalRecipients"},
            "MessageAsInternalRecipient": {"type": "LINK", "table": "Message", "inverse":
                "InternalRecipients"},
            "MessageAsSender": {"type": "LINK", "table": "Message", "inverse": "Sender"},
            "Person": {"type": "LINK", "table": "Person", "inverse": "Messages"},
            "ReceiptDate": {"type": "TIMESTAMP"}
        }
    },
    "Address": {
        "fields": {
            "Domain": {"type": "LINK", "table": "Domain", "inverse": "Addresses"},
```

```
            "Messages": {"type": "LINK", "table": "Participant", "inverse": "MessageAddress",
               "sharded": "true"},
            "Name": {"type": "TEXT"},
            "Person": {"type": "LINK", "table": "Person", "inverse": "MessageAddresses"}
         }
      },
      "Person": {
         "fields": {
            "DirectReports": {"type": "LINK", "table": "Person", "inverse": "Manager"},
            "FirstName": {"type": "TEXT"},
            "LastName": {"type": "TEXT"},
            "Location": {
               "fields": {
                  "Department": {"type": "TEXT"},
                  "Office": {"type": "TEXT"}
               }
            },
            "Manager": {"type": "LINK", "table": "Person", "inverse": "DirectReports"},
            "MessageAddresses": {"type": "LINK", "table": "Address", "inverse": "Person"},
            "Messages": {"type": "LINK", "table": "Participant", "inverse": "Person"},
            "Name": {"type": "TEXT"}
         }
      },
      "Domain": {
         "fields": {
            "Addresses": {"type": "LINK", "table": "Address", "inverse": "Domain"},
            "InternalID": {"type": "text", "analyzer": "NullAnalyzer"},
            "IsInternal": {"type": "BOOLEAN"},
            "Name": {"type": "TEXT"}
         }
      }
   },
   "schedules": [
      {"schedule": {"table": "Message", "type": "data-aging", "value": "0 3 * * SAT"}},
      {"schedule": {"table": "Message", "type": "stat-refresh", "value": "*/30 * * * *"}}
   ]
}}
```

Some highlights of this schema (details described later):

- The application-level option `StorageService` explicitly assigns the application to the `SpiderService`. The `AutoTables` option is disabled.
- The `Message` table uses these unique features:
    - Table-level sharding is enabled via the `sharding-field`, `sharding-granularity`, and `sharding-start` options, and automatic data aging is enabled via the `aging-field` and `retention-age` options.

    - A group field called `Participants` contains the link field `Sender` and a second-level group called `Recipients`, which contains the links `InternalRecipients` and `ExternalRecipients`. All three links point to the `Participant` table, allowing the `Participants` and `Recipients` group fields to be used in DQL quantifiers.
    - The `ThreadID` field is assigned the non-default `OpaqueTextAnalyzer`, which means it can only be searched for whole values (not terms).
    - An alias called `$SalesEmails` is assigned the expression `Sender.Person.WHERE(Department:Sales)`. `$SalesEmails` is dynamically expanded when used in DQL queries.
    - Four different statistics (stored aggregate queries) are defined: `TotalCount`, `EmailSizes`, `EmailsPerDay`, and `DepartmentTrend`.
- The `Participant` table uses table-level sharding similarly as the `Message` table.
- The `Address` table declares the `Messages` link as `sharded` to leverage its target table's sharding.
- The `Domain` table declares the `InternalID` field with the non-default `NullAnalyzer`. This means the `InternalID` can be returned in queries but is not indexed and cannot be searched.
- The application defines a `schedule` for the `data-aging` task in the `Message` table, assigning its schedule the cron expression "`0 0 3 * *`", which means "every day at 03:00". The `stat-refresh` task schedule is set to "`*/30 * * * *`", which means "every 30 minutes".

3. Spider Data Model

Doradus Spider extends the core Doradus data model with unique features. This section summarizes core data model concepts and describes the extended Spider features.

3.1 Core Data Model

The core Doradus data model is described in detail in the document **Doradus Data Model and Query Language**. For review, the core concepts and terms are summarized below:

- A Doradus *cluster* can host multiple tenants, called *applications*.
- Each application has a unique name within the cluster (e.g., `Msgs`). Its schema defines its *tables*, which are private to the application.
- A table's name is unique within its application (e.g., `Message`). A table's addressable members are called *objects*.
- An object consists of named *fields*, whose names are unique within the table (e.g., `SendDate`, `Subject`). There are three kinds of fields:
    - A *scalar* field is `text`, `integer` (same as `long`), `boolean`, `timestamp`, or `binary`. By default, scalar fields are *single-valued* (*SV*). A scalar field can be *multi-valued* (*MV*) by declaring it with a `collection` property equal to `true`.
    - A *link* field is a pointer to related objects in the same or another table. Every link has an *inverse* link that defines the same relationship in the opposite direction. Link fields are always multi-valued (MV).
    - A *group* field holds nested scalar, link, and group fields.
- Every object has a unique *object ID*, which is an opaque string value. The object ID is held in a system-defined field called `_ID`.
- A table can define *aliases*, which are link path expressions assigned a name that can be used in queries. An alias is like a macro that is expanded where it is referenced.
- Application, table, and field names must be *identifiers*, which begin with a letter and consist of letters, digits, or underscores (_). Alias names must be begin with a dollar sign ($) and consist of letters, digits, or underscores.
- An application can define *schedules*, which control how often background tasks related to the application are executed.

3.2 Application Options

Doradus Spider supports two application-level options, described below.

3.2.1 StorageService Option

All applications can explicitly choose their storage service by defining the application-level option `StorageService`. The storage service name for Spider applications is `SpiderService`. Example:

```
<application name="Msgs">
    <options>
        <option name="StorageService" value="SpiderService"/>
    </options>
    ...
</application>
```

If an application does not set the `StorageService` option, it is assigned the Doradus server default. Once assigned, the `StorageService` option cannot be changed.

3.2.2 AutoTables Option

Doradus Spider supports am application-level Boolean option called `AutoTables`, which is set to `true` by default. When enabled, `AutoTables` allows tables to be implicitly created when objects are added to them. For example, suppose the following Add Batch command is submitted:

```
POST /Msgs/blogs
```

If the `blogs` table does not exist and the `Msgs` application's `AutoTable` option is true, the table is automatically created and then the Add Batch request is processed. An automatically-created table has no predefined fields, hence all field assignments are treated as dynamic Text fields. To disallow the automatic creation of tables, `AutoTables` must be set to `false`. Example:

```
<application name="Msgs">
    <key>MsgsKey</key>
    <options>
        <option name="AutoTables" value="false"/>
    </options>
    ...
</application>
```

3.3 Table Options

Spider applications support table-level options, which engage two different features: automatic *data aging* and *sharding*. Below is an example in XML:

```
<table name="Message">
    <options>
        <option name="aging-field">SendDate</option>
        <option name="retention-age">5 YEARS</option>
        <option name="sharding-field">SendDate</option>
        <option name="sharding-granularity">DAY</option>
        <option name="sharding-start">2010-07-17</option>
    </options>
    ...
</table>
```

3.3.1 Data Aging

Data aging causes objects within the table to be deleted when a timestamp field reaches a defined age. Aging is performed in a background task whose schedule can be controlled. An object is deleted when the data-aging task executes and finds it is equal to or greater than the defined age.

Data aging is controlled by the following `options`:

- **`aging-field`**: Defines the field to use for data aging. It is required if a non-zero `retention-age` is specified. The aging field must be defined in the table's schema, and its type must be `timestamp`.
- **`retention-age`**: Enables data aging and defines the retention age. It must be specified in the format:

  ```
  <value> [<units>]
  ```

  Where `<value>` is a positive integer and `<units>`, if provided, is `days`, `months`, or `years`; `days` is the default. An object's age is the difference between "now" (when the aging task executes) and the object's aging field value. When this age is greater than the `retention-age`, the object is deleted. If `retention-age` is set to 0, aging is disabled.

When data aging is enabled, each object's aging field can be modified at any time. An object is deleted only when the aging field has a value.

3.3.2 Table Sharding

Table sharding improves the performance of certain queries for tables with large populations (millions of objects or more). To benefit from sharding, a table must meet the following conditions:

- Objects have a timestamp field whose value is stable, meaning it is rarely modified. In the example schema, the `Message` table's `SendDate` field works well because it is rarely modified once a message is created. This timestamp field is used as the *sharding* field.
- To benefit from a sharded table, queries must include an *equality clause* or *range clause* that uses the sharding field. For example, both of the following queries select objects in specific time frames:

  ```
  GET /Msgs/Message/_query?q=SendDate=PERIOD().LASTWEEK AND ...
  GET /Msgs/Message/_query?q=SendDate=[2014-01-01 TO 2014-03-01] AND ...
  ```

Normally, Doradus Spider creates a single *term vector* for each field/term combination. For example, the term vector with key "`Body/the`" holds references to all objects that use the term "the" in the field `Body`. For common terms, the term vector may point to every object in the table, and very large term vectors slow query performance. When sharding is enabled, separate term vectors are created for objects in each *shard*. Faster searching occurs when the sharding field is then included in queries.

Sharding is enabled with the following table-level `options`:

- **`sharding-field`**: This option enables sharding and identifies the sharding field. Its value must be a `timestamp` field defined in the schema.

- `sharding-granularity`: This option specifies what time period causes objects to be assigned to a new shard. It can be `HOUR`, `DAY`, `WEEK`, or `MONTH`. If not specified, it defaults to `MONTH`. The value should be chosen so that each shard as a reasonable number of objects (< 1 million).

- `sharding-start`: This option specifies the date on which sharding begins for the table. Objects whose sharding-field value is null or less than the `sharding-start` value are considered "un-sharded" and assigned to shard #0. Objects whose sharding field is greater than or equal to the `sharding-start` value are assigned a shard number based on the difference between the two values and the `sharding-granularity`. If not explicitly assigned, `sharding-start` defaults to "now", meaning the timestamp of the schema change that enables sharding.

Each object's sharding field value can be modified at any time. If the modified value does not cause the object to be assigned a new shard number, the update is efficient. However, if the sharding field is assigned a value that changes the object's shard number, the update is slower since the object's fields are re-indexed.

Table sharding can also benefit certain links that have very high *fan-outs*. See the description later on **Sharded Links**.

3.4 Scalar Field Analyzers

Doradus Spider uses *analyzers* to determine how to store and index scalar fields. The analyzer controls two aspects of scalar field management:

- **Indexing**: The analyzer determines whether or not the field's value are indexed.

- **Term extraction**: If values are indexed, the analyzer generates one or more *terms* from each field value that are used to create index records. In other words, the analyzer *tokenizes* the field for indexing.

Every scalar field defined in the schema is assigned a default analyzer that is optimized for the scalar's field type. Optionally, the field can explicitly declare an `analyzer` property that assigns the default or an alternate analyzer as allowed by the chart below:

Field Type	Default Analyzer	Alternate Analyzers
Text	`TextAnalyzer`	`NullAnalyzer, OpaqueTextAnalyzer, HTMLAnalyzer`
Integer	`IntegerAnalyzer`	`NullAnalyzer`
Long	`IntegerAnalyzer`	`NullAnalyzer`
Boolean	`BooleanAnalyzer`	`NullAnalyzer`
Timestamp	`DateAnalyzer`	`NullAnalyzer`
Binary	`NullAnalyzer`	

Every scalar field type can declare the `NullAnalyzer` to prevent the field from being indexed. Text fields can assign the `OpaqueTextAnalyzer` to index the entire field as a single value instead of being tokenized into

terms, or they can assign the `HTMLAnalyzer` to index the field as HTML text. Binary fields are not indexed and must always use the `NullAnalyzer`.

When a scalar field is MV, all values are indexed. For example, consider the following declaration:

```
<field name="Quotes" type="Text" collection="true" analyzer="TextAnalyzer"/>
```

Suppose an object is inserted with the following three `Quotes` values:

```
{"What time is it?", "Life is a Beach", "It's Happy Hour!"}
```

Using the `TextAnalyzer`, each unique term is indexed once ("a", "happy", "is", etc.), and each whole value ("life is a beach") is also indexed. Therefore phrase clauses such as `Quotes:Happy` will select the object, as will equality searches such as `Quotes="life is a beach"`. If the analyzer is set to `OpaqueTextAnalyzer`, only whole values are indexed, hence the equality clause will select the object but the term clause won't. Details of each analyzer are described in the next sections.

3.4.1 The `TextAnalyzer`

The `TextAnalyzer` assumes that text values are "plain text" (i.e., no metadata or mark-up). For each field value it tokenizes, the `TextAnalyzer` generates zero or more terms as lowercased letter/digit/apostrophe sequences separated by consecutive whitespace/punctuation sequences. That is, each contiguous sequence of letters, digits, and/or apostrophes becomes a term. A term can *contain* an apostrophe, but it cannot begin or end with an apostrophe. For example, the apostrophe is included in *doesn't*, but outer apostrophes in the sequence *'tough'* are excluded, yielding the term *tough*. An apostrophe is any of the following characters:

- The Unicode APOSTROPHE (0x27)
- The Windows right single quote (0x92)
- The Unicode RIGHT SINGLE QUOTATATION MARK (0x2019)

As example of how text fields are tokenized by the `TextAnalyzer`, suppose an email object is created with the following text field values, all indexed with `TextAnalyzer`:

```
From:     John Smith
To:       Betty Sue
Subject:  The Office Move
Body:     Hi Betty,
          Just a reminder that you're scheduled to move to your "fancy" new office tomorrow, number
          B413. If you have any questions, please let me know.
          Thanks, John.
```

The `TextAnalyzer` indexes these fields to generate the following terms:

Field Name	Terms
From	john smith
To	betty sue
Subject	move office the
Body	a any b413 betty fancy have hi if john just know let me move new number office please

	questions reminder scheduled thanks that to tomorrow you your you're

As shown, terms are extracted in lowercase, and punctuation and whitespace are removed. As part of down-casing, the `TextAnalyzer` converts any apostrophe retained within a term to the "straight apostrophe" character (0x27.) Although a term may appear multiple times within a field, it is indexed but once.

Though not shown above, the `TextAnalyzer` also creates a term equal to the term's entire field value, down-cased, and enclosed in single quotes. This value is used as an optimization for equality searches. For example, the whole-field value for the field `From` is `'john smith'`. For text fields with large values, this "whole field" value is created as an MD5 value instead of the literal text.

The terms generated by the `TextAnalyzer` allow efficient execution of a wide range of full text queries: single terms, phrases, wildcard terms, range clauses, etc. Searches are performed without case sensitivity: for example the phrase "You're Scheduled to MOVE" will match the `Body` field shown above.

3.4.2 The `IntegerAnalyzer`

The `IntegerAnalyzer` is the default analyzer for Integer/Long scalar fields. It creates terms using *trie values*, which allow efficient searching of value ranges in addition to exact value matching. For example, consider the following range clause:

```
Size:[500 TO 10000]
```

Text-based searches don't work on numeric fields: "500" is actually greater than "10000" when compared as text. If only exact values were stored for `Size`, the range clause above would require searching 9501 separate values. But using trie values, this range clause requires searching no more than 77 lookups.

3.4.3 The `BooleanAnalyzer`

The `BooleanAnalyzer` is the default analyzer for Boolean scalar fields. It creates one term record that indexes all fields with a "true" value and another term record for "false" values. This allows all objects with either value to be quickly found.

3.4.4 The `DateAnalyzer`

The `DateAnalyzer` is the default analyzer for Timestamp scalar fields. Similar to the `IntegerAnalyzer`, it indexes timestamp values by creating trie values, which allow efficient execution of range clauses. The trie values are chosen to efficiently find timestamp values that match values with various granularities. For example, a search for all field values that fall in a given year, month, date, or date+time requires only one lookup.

3.4.5 The `NullAnalyzer`

Any scalar field can be assigned the `NullAnalyzer`, which prevents the corresponding field from being indexed. Values for the field are stored as-is and are not indexed, thereby reducing storage space. Searching is not allowed for un-indexed fields. Hence, fields assigned the `NullAnalyzer` are "stored only". Note that binary fields are not indexed and must use the `NullAnalyzer`.

3.4.6 The OpaqueTextAnalyzer

Text fields can optionally be assigned the `OpaqueTextAnalyzer`. This causes the corresponding field values to be indexed as a single, opaque, down-cased value instead of as a series of terms. For example, if the text field `UserDomain` is assigned the `OpaqueTextAnalyzer`, the field value "`NT AUTHORITY`" will only match exact value searches for "`nt authority`", "`NT authority`", etc., but not term clauses for `NT` or `authority`.

3.4.7 The HTMLAnalyzer

Text fields can optionally be assigned the `HTMLAnalyzer`. This analyzer indexes the element content of any HTML elements it finds in the text. Tag names and element attributes are ignored. The content of all elements is logically concatenated into a single text field, which is then indexed with the same process as the `TextAnalyzer`. For example, consider the following HTML document:

```
<html xmlns="http://www.w3.org/1999/xhtml" dir="ltr" lang="en-US">
   <body>
      <div name="Body">Hi Betty,
      <p>Just a reminder that you're scheduled to move to your "fancy" new office
         tomorrow, number <b>B413</b>. If you have any questions, please let me know.
         Thanks, John.
      </div>
   </body>
</html>
```

If this HTML document is the value of a field named `Body`, it generates the exact same terms as in the plain text example shown for the `TextAnalyzer`. Specifically, the terms generated are:

```
a any b413 betty fancy have hi if john just know let me move new number office please questions
reminder scheduled thanks that to tomorrow you your you're
```

3.5 Dynamic Fields

Doradus Spider allows fields to be dynamically added to objects. That is, objects in Add Batch and Update Batch commands can assign values to fields not declared in the corresponding table. Dynamic fields are treated as text and indexed using the `TextAnalyzer`. Dynamic fields can be referenced in query clauses and returned in query results, though they cannot be used as grouping fields in aggregate queries.

3.6 Sharded Links

The table sharding feature described earlier can benefit certain *high fan-out* links. These are links that have a large number of values (thousands to millions). When a high fan-out link points to a sharded table, the link can also be declared as *sharded* by settings its `sharded` property to true. Example:

```
<table name="Address">
   <fields>
      <field name="Name" type="TEXT"/>
      <field name="Messages" type="LINK" inverse="MessageAddress" table="Participant"
         sharded="true"/>
      ...
   </fields>
```

```
</table>
```

In this example, the `Messages` link connects each `Address` object to related `Participant` objects. Since a participant exists for each sender or recipient of every message, some address objects could be linked to thousands or millions of participant objects. Because the `Messages` extent table, `Address`, is declared as sharded, the `Messages` link can be declared as sharded. That is, only links whose target table is sharded can be declared as sharded.

A sharded link's values are stored in sharded term vectors similarly as sharded scalar fields. This improves performance for queries that include a selection clause on the link's owning table and a clause that uses the referenced table's sharding field. Example:

```
GET /Msgs/Address/_query?q=Name:dell AND Messages.ReceiptDate=[2013-01-01 TO 2013-01-31]
```

This query searches for `Address` objects whose `Name` contains the term "dell" and that are connected to participants whose `ReceiptDate` is in January, 2013. In other words, this query lists all addresses that sent or received a message in a specific time frame. Queries that fit this pattern are more efficient for high fan-out links declared as sharded.

3.7 Statistics

Doradus Spider supports predefined aggregate queries called *statistics*. A statistic belongs to a specific table and performs an aggregate query on the objects in that table. Each query is executed in the background, and its results are persisted in the database. This allows a statistic's computations to be retrieved quickly even when the aggregate query takes much longer to compute. Statistics can be recomputed or *refreshed* upon request or automatically based on a schedule. Example statistic declarations are shown below:

```
<table name="Message">
    <statistics>
        <statistic name="DepartmentTrend" metric="COUNT(*)"
            group="Sender.Person.Department,TRUNCATE(SendDate,DAY)"/>
        <statistic name="EmailSizes" metric="COUNT(*)"
            group="BATCH(Size,1000,10000,50000,100000,1000000)"/>
        <statistic name="EmailsPerDay" metric="COUNT(*)" group="TRUNCATE(SendDate,DAY)"/>
        <statistic name="TotalCount" metric="COUNT(*)"/>
    </statistics>
    ...
</table>
```

In this example, `TotalCount` is a global statistic that provides a count of all `Message` objects. `EmailSizes` and `EmailsPerDay` are single-level grouped statistics. `DepartmentTrend` is a 2-level grouped statistic. A statistic definition has the following four properties:

- **`name`** (required): The name used to reference the statistic in REST commands. The name is an identifier and must be unique among all statistics owned by the same table.
- **`metric`** (required): Defines the statistical function and field to use. The same metric functions and syntax are allowed as for aggregate queries.

- **group** (optional): Defines a grouping expression. When the group property is omitted, the statistic is a "global statistic". When present, it defines the statistic as a single- or multi-level statistic. The group parameter has the same syntax as the grouping parameter for aggregate queries. However, statistics cannot use compound grouping (multiple GROUP sets) since separate statistics can be declared to accomplish the same thing. Similarly, statistics do not support composite grouping since the same computations can be made with a separate statistic declaration.

- **query** (optional): Defines a DQL expression that selects the objects to be included in the metric computation. When the query property is omitted, all objects in the table are included in the metric. The query property has the same syntax as the query parameter for aggregate queries.

Note that when statistics are first declared, they do not immediately have a value. Instead, they must be explicitly refreshed via REST commands or by a scheduled background task.

3.8 Task Schedules

Doradus Spider applications use background *tasks* to perform functions such as data aging and statistics refreshing. By default, tasks are unscheduled and executed only when requested via a REST command or assigned a schedule in the schema. Schedules are defined in the schema at the application level as in the following example:

```
<application name="Msgs">
   <schedules>
      <schedule type="data-aging" value="0 3 * * SAT" table="Message"/>
      <schedule type="stat-refresh" value="*/30 * * * *" table="Message"/>
   </schedules>
   ...
</application>
```

In this example, the schedule for data-aging task for the Message table is assigned the cron expression "0 0 3 * *", which means "every day at 03:00". The Message table's stat-refresh task schedule is set to "*/30 * * * *", which means "every 30 minutes".

A schedule declaration has the following properties:

- type (required): Defines the type of task being scheduled. Possible values are:

    - app-default: Defines a default schedule for the application. The table property cannot be specified. All tasks use this schedule unless a more specific schedule applies to the task.

    - table-default: Defines a default schedule for a table, which must be specified via the table property. The table schedule overrides the app-default schedule, if present, and becomes the default schedule for all tasks in the table unless a more specific schedule applies.

    - stat-refresh: Defines a schedule for statistic refresh tasks. The table property is required. If the statistic property is not specified, it becomes the default for all statistic refresh tasks in the table. Otherwise it applies to the named statistic only.

    - `data-aging`: Defines a schedule for a table's data aging tasks. The `table` property is required.
    - `data-checks`: Defines a schedule for an optional background data integrity task. The `table` property is required. The `data-checks` task is described in more detail in the section **Task Management Commands**.

- `value` (required): Defines the task schedule as a five-part cron expression (described below).
- `table`: The `table` property is required for all schedule types except for `app-default`. It declares the table to which the schedule applies.
- `statistic`: This property is only allowed for `stat-refresh` schedules. When present, it assigns the schedule to the named statistic. When the statistic property is absent for a `stat-refresh` schedule, it becomes the default schedule for all statistic refresh tasks in the table.

The schedule `value` property is a cron expression that use the following general format:

```
<minute pattern> <hour pattern> <month day pattern> <month pattern> <week day pattern>
```

Each of the five patterns defines at which values the corresponding unit matches. A task is started when the current time matches all five of its schedule's pattern parts (if the task is not already executing). The following rules apply to all patterns:

- An asterisk (`*`) pattern matches all possible values (every minute, every hour, etc.)
- A pattern can be a single number (`5`), a comma-separated list of numbers (`1,3,5`), or a dashed number range (`0-4`). A pattern can also mix of comma-separated and dashed ranges (`1-15,17,20-25`). All numbers must be within range for the corresponding unit (e.g., 0 to 59 for minutes).
- A pattern can optionally define a numeric *interval* adding the suffix `/<interval>`, where `<interval>` is a number. The corresponding part matches every value in range but no more often than the specified interval. For example, the minute pattern `*/15` means every 15 minutes. The hour pattern `3-18/5` matches the 3rd, 8th, 13th, and 18th hour.

Rules for specific patterns are defined below:

- `<minute pattern>`: Values must be in the range `0` to `59`.
- `<hour pattern>`: Values must be in the range `0` to `23`.
- `<month day pattern>`: Values must be in the range `1` to `31`. The special value `L` (uppercase "l") denotes the last day of the month.
- `<month pattern>`: Values must be in the range `1` (January) to `12` (December). Alternatively, 3-letter aliases can be used: `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, and `Dec`.
- `<week day pattern>`: Values must be in the range `0` (Sunday) to `6` (Monday). Alternatively, 3-letter aliases: can be used: `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`, and `Sun`.

Below are some example cron expressions and their meaning:

Cron Expression	Meaning
`* * * * *`	Every minute.
`5 * * * *`	Every 5th minute (00:05, 00:10, etc.)
`* 12 * * Mon`	Every minute during the 12th hour of every Monday.
`59 11 * * 1,2,3,4,5`	11:59AM on every Monday, Tuesday, Wednesday, Thursday and Friday.
`*/15 9-17 * * *`	Every 15 minutes between the 9th and 17th hour of the day (9:00, 9:15, 9:30, 9:45 and so on). The last execution will be at 17:45.
`* 12 1-15,17,20-25 * *`	Every minute during the 12th hour of the day, but the day of the month must be between the 1st and the 15th, the 17th, or between the 20th and the 25th.

4. Doradus Query Language (DQL)

The Doradus Query Language (DQL) will be familiar to those who have used Apache Lucene or other full text languages from which DQL borrows concepts such as *terms*, *phrases*, and *ranges*. To these DQL adds *link paths*, *quantifiers*, and a *transitive* function to support graph-based searches. DQL is described in detail in the document **Doradus Data Model and Query Language**. This section first provides a summary of the core concepts and terms. It then describes extensions to DQL supported by Doradus Spider.

4.1 Core DQL Overview

4.1.1 Query Perspective

- Every DQL query has a *perspective*, which is a table in which objects will be searched.

4.1.2 Clauses

- A DQL query is a Boolean expression consisting of one or more *clauses*.
- Clauses are implicitly AND-ed but can be explicitly joined via `AND`, `OR`, and parentheses. Examples:

```
X AND Y
X AND Y OR Z
(X AND Y) OR (Q AND P)
```

- A clause is negated by prefixing it with `NOT`.
- A *term clause* searches for one or more *terms*, optionally using wildcards (* and ?), within a text field. A colon (:) separates the field name and search term(s). The search is case-insensitive. Example term clauses are shown below:

```
LastName:Smith
NOT FirstName:Jo*
Name:(Smith John)
```

- A *phrase clause* searches for an exact sequence of terms within a text field. The phrase is enclosed in quotes and is case-insensitive. Examples:

```
Name:"john smith"
```

- A field that has no value for a given object is *null*. Doradus treats null as a "value" that will not match any literal. For example, the clause `Size=0` will be false if the `Size` field is null.
- Any scalar or link field can be tested for nullity by using the `IS NULL` clause. Examples:

```
Size IS NULL
NOT LastName IS NULL
```

- An *equality clause* compares entire field values instead of terms. An equal sign (=) follows the field name. Multi-term values must be enclosed in single or double quotes. Text comparisons are case-insensitive. Examples:

```
Name="John Smith"
Name="J* Sm?th"
HasBeenSent=false
Size=1024
ModifiedDate="2012-11-16 17:19:12.134"
Manager=sqs284                // Manager is a link; sqs284 is an object ID
_ID='kUNaqNJ2ymmb07jHY9OPOw==' // Enclose strings in single or double quotes
```

- A *range clause* searches a scalar field for values within a specific range. The normal math operators (=, <, <=, and >=) are allowed for numeric fields. Examples:

```
Size > 10000
ReceiptDate <= '2012-04-13'
```

- A *bracketed* range clause uses square (inclusive) or curly (exclusive) brackets to search a scalar field for a range of values. Examples:

```
Size = [1000 TO 50000}     // same as Size >= 1000 AND Size < 50000
LastName={Anteater TO Fox] // same as Title > Anteater and Title <= Fox
LastName:{Anteater TO Fox] // ':' is the same as '=' for bracketed ranges
```

- *Not equals* is written by negating an equality clause with the keyword `NOT`. Example:

```
NOT Size=1024
```

- A *set membership clause* tests a field for membership in a set of values using the IN operator. Examples:

```
Size IN (100,500,1000,200000)
LastName IN (Jones, Smi*, Vledick)
Manager IN (shf637, dhs729, fjj901)
_ID IN (sjh373,whs873,shc729)
LastName=(Jones, Smi*, Vledick) // alternate syntax for LastName IN (Jones, Smi*, Vledick)
```

- Timestamps possess date/time *subfields* that can be used in equality clauses. Examples:

```
ReceiptDate.MONTH = 2    // month = February, any year
SendDate.DAY = 15        // day-of-month is the 15th
NOT SendDate.HOUR = 12   // hour other than 12 (noon)
```

- A timestamp field can be compared to the current time, optionally adjusted by an offset, using the `NOW` function. Examples:

```
SendDate > NOW(-1 YEAR)
SendDate >= NOW(PST +9 MONTHS)
ReceiptDate = [NOW() TO NOW(+1 YEAR)]
ReceiptDate = [2013-01-01 TO NOW(Europe/Moscow)]
```

- The PERIOD function is used to compare a timestamp field to a value range computed relative to the current time. Examples:

```
SendDate = PERIOD().TODAY                    // Field has the same year/month/and date as now
SendDate = PERIOD().LASTWEEK                 // Field is in the last week (UTC)
SendDate = PERIOD(PST).LASTMONTH(2)          // Field is in the last 2 months (PST)
SendDate = PERIOD(Europe/Moscow).LASTYEAR(3) // Field is in the last 3 years (Moscow time)
```

- Link fields can be compared to a single object ID or tested for membership in a set of IDs. Examples:

```
Manager=def413
DirectReports IN (zyxw098, ghj780)
DirectReports = (zyxw098, ghj780)  // same as above
```

4.1.3 Link Paths

- A clause can search a field of an object that is related to the perspective object by using a *link path*. Examples:

```
Manager.Name : Fred
Manager.Manager.LastName = Smith
Sender.MessageAddress.Domain.Name : yahoo
```

- A link path can use a WHERE filter to define multiple clauses that are *bound* to the same objects in a link path. Example:

```
InternalRecipients.Person.WHERE(Department='R&D' AND Office='Kanata')
```

- The COUNT function can be used on a path ending with a link. Example:

```
COUNT(DirectReports) > 5
```

- WHERE filtering can be combined with the COUNT function. Examples:

```
COUNT(DirectReports.WHERE(LastName=Smith)) > 0
COUNT(InternalRecipients.WHERE(MessageAddress.Person.LastName=Smith)) > 5
```

4.1.4 Quantifiers

- Link paths without an explicit quantifier are implicitly quantified with ANY. That is:

```
InternalRecipients.Person.LastName = smith
```

  Is the same as each of these:

```
ANY(InternalRecipients.Person.LastName) = smith
ANY(InternalRecipients).Person.LastName = smith
ANY(InternalRecipients).ANY(Person).LastName = smith
```

- The ANY quantifier requires that the value set created by the argument is not empty and at least one value matches the value set on the right hand side.

- The ALL quantifier requires that the value set created by the argument is not an empty set and every member matches the RHS value set. Examples:

```
ALL(InternalRecipients).ALL(Person).LastName = smith   // Neither link can be null
ALL(InternalRecipients.Person).LastName = smith        // The set {InternalRecipients.Person}
                                                       // cannot be null
```

- The NONE quantifier requires that no member of the value set created by the argument is matches the RHS value set. Unlike ANY and ALL, the value set created by the argument to NONE can be empty for the clause to be true. Semantically, NONE is the same as NOT ANY. Example:

```
NONE(InternalRecipients.Person).LastName = smith// true if {InternalRecipients.Person} is
null
```

- Quantifiers can be used on scalar fields. Examples:

```
ANY(Tags) = Customer                // at least one Tags must be Customer
ALL(Tags) = (Customer, Competitor)  // all Tags must be Customer or Competitor
NONE(Tags) = "DO NOT FORWARD"       // no Tags can be "DO NOT FORWARD"
```

4.1.5 Transitive Function

- The transitive function (^) causes a *reflexive* link to be traversed recursively, creating a set of objects for evaluation. Example:

```
DirectReports^.Name:Doug
```

- A *limit* can be passed to the transitive operator in parentheses. Example:

```
DirectReports^(5).Name:Doug
```

4.1.6 Aliases in Queries

- A schema-defined *alias* can be used anywhere in a DQL query. Its defined expression is expanded in place, and the expanded DQL query is parsed. For example, the following clause:

```
$SalesEmails.WHERE(Office:Maidenhead)
```

  Is expanded and parsed as this DQL clause:

```
Sender.Person.WHERE(Department:Sales).WHERE(Office:Maidenhead)
```

4.2 Spider DQL Extensions

Doradus Spider extends DQL with features described in this section.

4.2.1 Any-field Clauses

Term and phrase clauses can search all fields of the perspective table by eliminating the field name qualifier. The simplest form of an any-field clause is a term clause:

```
John
```

This clause searches all scalar fields of the perspective table for a value contains the term `John` (case-insensitive). All indexed scalar fields are searched, which means those whose analyzer is other than `NullAnalyzer`. Fields indexed with the `OpaqueTextAnalyzer` will match only if the field's value *equals* `John`. Fields that use other analyzers will match if they *contain* the term `John`. However, Boolean, timestamp, and numeric fields do not contain text terms, so they will never contain a text term such as `John`.

However, suppose we perform an any-field search for an integer value:

```
12226
```

If an object has a text field called `MessageID` with the value "`Host1-12226-Alpha1`", 12226 is one the terms generated by the field, hence the above clause would match the field. If another object has an integer field called `Size` whose value is 12226, it would also match since the `IntegerAnalyzer` generates a term equal to the field's value. Hence, objects could match a term clause based on the type of terms generated by their scalar field analyzers.

Multiple terms can be provided by separating them by at least one space. For example:

```
John Smith
```

This multi-term clause searches for the terms `John` and `Smith` in any order and across any field. For example, an object will match if it has a `FirstName` field that contains `John` and a `LastName` field that contains `Smith`. If all terms are contained in the same field, they can appear in any order and be separated by other terms. For example, an object will match the example above if it has a `Name` field whose value is "`John Smith`", "`John Q. Smith`", or "`Smith, John`".

An any-field clause can be requested using the field qualifier syntax by using the field name "*". Example:

```
*:(John Smith)
```

This is the same as the unqualified term clause:

```
John Smith
```

Phrase clauses can also use the field qualifier syntax and request an any-field search:

```
*:"John Smith"
```

This query finds object where the terms `John` and `Smith` appear in succession in any field. The phrase can be preceded or followed by other terms.

4.2.2 Quantifiers on Group Fields

Doradus Spider allows quantifiers (`ANY`, `ALL`, and `NONE`) on group fields. Group field quantification works as follows: Assume a group field G with *leaf* fields F1, F2, ... Fn. Quantifiers can be used on the group field G if all fields Fi are of the same type:

- If the fields are scalars, they must be all of the same scalar type (e.g., integer or text).

- If the fields are links, they must all have the same extent (target table).

When a group field is quantified, the quantifier is applied to the union of the leaf field values. That is, the quantifier Q on the group field G:

```
Q(G)
```

This is interpreted as:

```
Q(union(F1, F2, ..., Fn))
```

In our example `Msgs` schema, the `Message` table's group field `Participants` contains three links that all refer to the `Participant` table: `ExternalRecipients`, `InternalRecipients`, and `Sender`. Consider the following query:

```
ANY(Participants.ReceiptDate) = [2013-01-01 TO 2013-01-31]
```

This is evaluated as the following equivalent expression:

```
ANY(union(ExternalRecipients, InternalRecipients, Sender).ReceiptDate) = [2013-01-01 TO 2013-01-31]
```

The "union" function does not actually exist – it is shown for illustrative purposes. It semantically combines all link values into a single set. If any linked object has a `ReceiptDate` within the given range, the entire expression is true. The `ANY` quantifier is false when all three links are null since `ANY` on an empty set is false.

If `ALL` is used instead of `ANY`, the overall expression is true only if all values in the set have a `ReceiptDate` within the given range and the set is not empty.

If `NONE` is used, the overall expression is true if none of the objects in the set have a `ReceiptDate` within the given range or if the set is empty. Unlike `ANY` and `ALL`, `NONE` quantification returns true if the set is empty.

Quantifiers can also be used on group fields whose leaf fields are scalars of the same type. In the example schema, the `Person` table's `Location` field is a group containing the scalar text fields `Department` and `Office`. The following query:

```
GET /Msgs/Person/_query?q=NONE(Location):Sales
```

Finds people for which neither `Department` nor `Office` contains the term `Sales`.

4.3 Spider Aggregate Query Extensions

This section describes extensions to aggregate queries supported by Spider applications.

4.3.1 Group Fields as Grouping Parameters

A group field can be used as a grouping expression if all of its leaf fields are links. For example, in the example schema the group field `Participants` contains links `Sender`, `InternalRecipients`, and `ExternalRecipients`, all pointing to the `Participant` table. If the `Participants` field is used as a grouping parameter, the values (object IDs) for all three links are combined into a set for each perspective object. The object is included in the metric computation for each group for which it has a value. If a perspective object has no values for any of the links, it is included in the `(null)` group.

Link paths and WHERE filters can be used on the group field with the same syntax as allowed for links. For example:

```
f=Participants.Person.Name
f=Participants.WHERE(ReceiptDate > 2014-02-01).Person.Name
f=Participants.Person.WHERE(Department:Sales).Name
```

In the first example, the combined Person.Name values of each link (ExternalRecipients, InternalRecipients, and Sender) form the grouping sets; each perspective object is included in each set in which it participates. If a perspective object has no Participants.Person.Name values, it is included in the null group.

The second and third examples using the same grouping parameter: Participants.Person.Name. However, the presence of the WHERE filter causes nulls to be handled differently:

- In the second example, a perspective object is simply skipped if it has no participants or none of its participants are selected by the expression ReceiptDate > 2014-02-01. When a perspective is selected by the WHERE expression but it is has no Person.Name values, it is included in the (null) group.
- In the third example, a perspective object is skipped if it has no participants, no participants have Person values, or no Person.Department field includes the term Sales. Only if a perspective object is chosen but has no Name values is it included in the (null) group.

4.3.2 Composite Grouping

Doradus Spider supports a special grouping feature called *composite* grouping. It is meaningful only for aggregate queries with 2 or more grouping levels. It causes the metric function(s) to be computed for parent (non-leaf) groups in addition to leaf-most groups. The extra computations are returned as composite results within the corresponding parent groups.

Composite grouping is requested by using a special grouping parameter instead of the normal grouping parameter. For example, consider the following URI aggregate query:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)&f=TRUNCATE(SendDate,DAY),Tags
```

Composite grouping can be requested for this 2-level query by using the &cf parameter instead of &f:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)&cf=TRUNCATE(SendDate,DAY),Tags
```

Example query results with composite grouping are described later under REST commands.

4.3.3 Compound Grouping: GROUP Sets

Doradus Spider allows the aggregate query grouping parameter to consist of multiple *grouping sets*. Each grouping set is enclosed in a GROUP function; multiple grouping sets are separated by commas. This feature is known as *compound* grouping. The general syntax is:

```
GROUP(<expression 1>),GROUP(<expression 2>),...,GROUP(<expression n>)
```

Each <expression n> parameter must use one of the following forms:

- A "*" can be used to compute a global aggregate (i.e., `GROUP(*)`). The metric function is computed for all selected objects just as in an aggregate query with no grouping parameter. The `GROUP(*)` function should be specified at most once since there is only one metric value for a global aggregate.
- A single-level grouping expression, consisting of a single scalar field or a field path (e.g., `GROUP(Tags)`).
- A multi-level grouping expression, consisting of a comma-separated list of scalar fields and/or field paths (e.g., `GROUP(TRUNCATE(SendDate,DAY),Tags)`).

Each single- and multi-level grouping expression must be relative to the perspective table. The same set of objects selected by the aggregate query is passed to each grouping set, and separate metric computations are performed for each grouping set. Aggregate queries that use compound grouping perform a single pass through the selected objects and computes multiple grouping sets at the same time.

Consider this aggregate query:

```
GET /Msgs/Message/_aggregate?m=MAX(Size)
    &f=GROUP(*),GROUP(Tags),GROUP(TOP(3,Sender.Person.Office),TOP(3,Sender.Person.Department))
```

This compound grouping request computes the following:

1) The maximum `Size` value of all messages (`GROUP(*)`).
2) The maximum `Size` of messages grouped by `Tags`.
3) The maximum `Size` of messages grouped first by the top 3 sender's offices and then by the top 3 sender's departments.

Example query results with compound grouping are described later under REST commands.

5. Spider REST Commands

This section provides and overview of the Doradus REST API and describes REST commands supported by Doradus Spider.

5.1 REST API Overview

The Doradus REST API is managed by an embedded Jetty server. All REST commands support XML and JSON messages for requests and/or responses as used by the command. By default, Doradus uses unsecured HTTP, but HTTP over TLS (HTTPS) can be configured, optionally with mandatory client authentication. See the **Doradus Administration** document for details on configuring TLS.

The REST API is accessible by virtually all programming languages and platforms. GET commands can also be entered by a browser, though a plug-in may be required to format JSON or XML results. The `curl` command-line tool is also useful for testing REST commands.

Some example REST commands supported by Spider are shown below:

- List all application schemas:

```
GET /_applications
```

- Query the `Msgs` application's `Person` table for objects whose `LastName` is `Smith`:

```
GET /Msgs/Person/_query?LastName=Smith
```

- Submit a batch of objects to the `Message` table:

```
POST /Msgs/Message
{"batch": {
    "docs": [
        "doc": {
            "_ID": "58275782",
            "FirstName": "John",
            "LastName": "Smith",
            ...
        },
        "doc": {
            ...
        },
        ...
    ]
}}
```

- Perform an aggregate query that returns the top 5 departments with the most people, sub-grouped by the top 3 offices:

```
GET /Msgs/Person/_aggregate?m=COUNT(*)&f=TOP(5,Department),TOP(3,Office)
```

Unless otherwise specified, all REST commands are synchronous and block until they are complete. Object queries can use stateless paging for large result sets.

5.1.1 Common REST Headers

Most REST calls require extra HTTP headers. The most common headers used by Doradus are:

- **`Content-Type`**: Describes the MIME type of the input entity, optionally with a `Charset` parameter. The MIME types supported by Doradus are `text/xml` (the default) and `application/json`.
- **`Content-Length`**: Identifies the length of the input entity in bytes. The input entity cannot be longer than `max_request_size`, defined in the doradus.yaml file.
- **`Accept`**: Indicates the desired MIME type of an output (response) entity. If no `Accept` header is provided, it defaults to input entity's MIME type, or `text/xml` if there is no input entity.
- **`Content-Encoding`**: Specifies that the input entity is compressed. Only `Content-Encoding: gzip` is supported.
- **`Accept-Encoding`**: Requests the output entity to be compressed. Only `Accept-Encoding: gzip` is supported. When Doradus compresses the output entity, the response includes the header `Content-Encoding: gzip`.
- **`X-API-Version`**: Requests a specific API version for the command. Currently, `X-API-Version: 2` is supported.

Header names and values are case-insensitive.

5.1.2 Common REST URI Parameters

Mainly for testing REST commands in a browser, the following URI query parameters can be used:

- **`api=N`**: Requests a specific API version for the command. Currently, `api=2` is supported. This parameter overrides the `X-API-Version` header if present.
- **`format=[json|xml]`**: Requests the output message format in JSON or XML, overriding the `Accept` header if present.

These parameters can be used together or independently. They can be added to any other parameters already used by the REST command, if any. Examples:

```
GET /_applications?format=json
GET /Msgs/Message/_statistics/_status?format=xml&api=2
GET /Msgs/Person/_query?q=LastName:Smith&s=5&api=2&format=json
```

5.1.3 Common JSON Rules

In JSON, Boolean values can be text or JSON Boolean constants. In both cases, values are case-insensitive. The following two members are considered identical:

```
"AutoTables": "true"
"AutoTables": TRUE
```

Numeric values can be provided as text literals or numeric constants. The following two members are considered identical:

```
"Size": 70392
"Size": "70392"
```

Null or empty values can be provided using either the JSON keyword `NULL` (case-insensitive) or an empty string. For example:

```
"Occupation": null
"Occupation": ""
```

In JSON output messages, Doradus always quotes literal values, including Booleans and numbers. Null values are always represented by a pair of empty quotes.

5.1.4 Common REST Responses

When the Doradus Server starts, it listens to its REST port and accepts commands right away. However, if the underlying Cassandra database cannot be contacted (e.g., it is still starting and not yet accepting commands), REST commands that use the database will return a `503 Service Unavailable` response such as the following:

```
HTTP/1.1 503 Service Unavailable
Content-length: 65
Content-type: Text/plain

Services are not yet available (waiting for Cassandra connection)
```

When a REST command succeeds, a `200 OK` or `201 Created` response is typically returned. Whether the response includes a message entity depends on the command.

When a command fails due to user error, the response is usually `400 Bad Request` or `404 Not Found`. These responses usually include a plain text error message (similar to the 503 response shown above).

When a command fails due to a server error, the response is typically `500 Internal Server Error`. The response includes a plain text message and may include a stack trace of the error.

5.2 Spider REST Command Summary

The REST API commands supported by Doradus Spider are summarized below:

REST Command	Method and URI
Application Management Commands	
Create Application	`POST /_applications`
Modify Application	`PUT /_applications/{application}`
List All Applications	`GET /_applications`

REST Command	Method and URI
List Application	GET /_applications/{application}
Delete Application	DELETE /_applications/{application}/{key}
Object Update Commands	
Add Batch	POST /{application}/{table}
Update Batch	PUT /{application}/{table}
Delete Batch	DELETE /{application}/{table}
Query Commands	
Get Object by ID	GET /{application}/{table}/{ID}
Object Query via URI	GET /{application}/{table}/_query?{params}
Object Query via Entity	GET /{application}/{table}/_query PUT /{application}/{table}/_query
Aggregate Query via URI	GET /{application}/{table}/_aggregate?{params}
Aggregate Query via Entity	GET /{application}/{table}/_aggregate PUT /{application}/{table}/_aggregate
Statistics Commands	
Refresh Table Statistics	PUT /{application}/{table}/_statistics/_refresh
Refresh Single Statistic	PUT /{application}/{table}/_statistics/{stat}/_refresh
Get Refresh Status	GET /{application}/{table}/_statistics/_status
Query Statistic	GET /{application}/{table}/_statistics/{stat}?{params}
Task Management Commands	
Get All Task Status	GET /_tasks
Get Application Task Status	GET /_tasks/{application}
Get Table Task Status	GET /_tasks/{application}/{table}
Get Table Task Type Status	GET /_tasks/{application}/{table}/{task}
Modify Application Task Status	PUT /_tasks/{application}?{command}
Modify Table Tasks Status	PUT /_tasks/{application}/{table}?{command}
Modify Table Task Type Status	PUT /_tasks/{application}/{table}/{task}?{command}
Start Data Cleanup Task	POST /_tasks/{application}/{table}/{task}/{field}[?{params}]
Stop Data Cleanup Task	DELETE /_tasks/{application}/{table}/{task}/[{field}]

Details on each command are described in the following sections.

5.3 Application Management Commands

REST commands that create, modify, and list applications are sent to the `_applications` resource. Application management REST commands supported by Doradus Spider are described in this section.

5.3.1 Create Application

A new application is created by sending a POST request to the `_applications` resource:

```
POST /_applications
```

The request must include the application's schema as an input entity in XML or JSON format. If the request is successful, a `200 OK` response is returned with no message body.

Because Doradus uses *idempotent* update semantics, using this command for an existing application is not an error and treated as a Modify Application command. If the identical schema is added twice, the second command is treated as a no-op.

See **The `Msgs` Sample Application** section for an example schema in XML and JSON.

5.3.2 Modify Application

An existing application's schema is modified with the following REST command:

```
POST /_application/{application}
```

where `{application}` is the application's name. The request must include the modified schema in XML or JSON as specified by the request's `content-type` header. Because an application's name cannot be changed, {application} must match the application name in the schema. Note that the application's key cannot be changed either. If the request is successful, a `200 OK` response is returned with no message body.

Modifying an application *replaces* its current schema. All schema changes are allowed, including adding and removing any schema component type, although there is no way to rename a schema component. However, minimal updates are made to accommodate changes to existing data. Some changes require explicit requests for *data cleanup* tasks to remove or re-index obsolete data. Various schema change scenarios and their data cleanup implications are summarized below:

- **Adding a table**: When a new table is added to the schema, the underlying stores (ColumnFamilies) are automatically created. Objects can be added to the table immediately after the schema change.
- **Adding a new field**: When a new scalar or link field is added, all existing objects will have a null value for the field. Data can be added to the field immediately after the schema change.
- **Deleting a table**: When an existing table is deleted, the corresponding stores (ColumnFamilies) are automatically deleted. However, if the table contains a link field whose extent table is not also deleted, inverse link data is not deleted. The obsolete link data, if present, is not returned in queries, but a link cleanup task must be requested to remove obsolete link data.
- **Changing a field definition**: All field modifications are allowed, but Spider does not automatically reorganize data to match the new field definition. For example, if a field's type is changed from text to timestamp, existing data will remain indexed with the previous text-based analyzer. In this case, a field cleanup task should be requested to re-index existing data using the field's new analyzer. If a field is changed from a scalar type to a link field (with a suitable inverse), all existing data will be obsolete since scalar fields are stored in a different format than link fields. In this case, a cleanup task should be requested to delete the obsolete scalar data.
- **Deleting a field**: When an existing scalar field is deleted, existing data is not disturbed, so it acts like an undefined field. The field's existing values can be returned in queries, but the field can no longer

be used as a grouping field in aggregate queries. When a link field (and its inverse) are deleted, existing link values are not deleted. For these cases, a field cleanup task should be requested to remove obsolete data.

Requesting a field cleanup task is discussed in the section **Task Management Commands**.

5.3.3 List Application

A list of all application schemas is obtained with the following command:

```
GET /_applications
```

The schemas are returned in the format specified by the `Accept` header.

The schema of a specific application is obtained with the following command:

```
GET /_applications/{application}
```

where `{application}` is the application's name.

5.3.4 Delete Application

An existing application—including all of its data—is deleted with the following command:

```
DELETE /_applications/{application}/{key}
```

where `{application}` is the application's name. The `{key}` must match the application's defined key as a safety mechanism. When an application is deleted, all of its underlying stores (ColumnFamilies) are deleted. No data cleanup tasks are required.

5.4 Object Update Commands

This section describes REST commands for adding, updating, and deleting objects in Spider applications. Doradus uses idempotent update semantics, which means repeating an update is a no-op. If a REST update command fails due to a network failure or similar error, it is safe to perform the same command again.

5.4.1 Add Batch

A batch of new objects is added to a specific table in an application using the following REST command:

```
POST /{application}/{table}
```

where `{application}` is the application name and `{table}` is the table in which the objects are to be added. If the given table name does not exist but the application's `AutoTables` option is true, the table is implicitly created before the batch is added. If `AutoTables` is false and the table is unknown, an error is returned.

The command must include an input entity that contains the objects to be added. The format of an example input message in XML as shown below:

```
<batch>
   <docs>
```

```
        <doc>
            <field name="_ID">AAFE9rf++BCa3bQ4HgAA</field>
            <field name="SendDate">2010-07-18 05:22:35</field>
            <field name="Size">3654</field>
            <field name="Subject">RE: synopses copens silk seagull citizens</field>
            <field name="Tags">
                <add>
                    <value>AfterHours</value>
                    <value>Customer</value>
                </add>
            </field>
            <field name="InternalRecipients">
                <add>
                    <value>KMTkYYrkL4MmrHxj//ZVhQ==</value>
                </add>
            </field>
            <field name="Sender">wWR7yZik2p1rrI6/qSepXg==</field>
            ...
        </doc>
        <doc>
            <field name="_ID">AAFE9rf++BCa3bQ4HgAB</field>
            <field name="SendDate">2010-07-17 10:37:03</field>
            <field name="Size">15830</field>
            <field name="Subject">bloodily douches spadones poinds vestals</field>
            <field name="Tags">AfterHours</field>
            <field name="InternalRecipients">Ii91O7qHb8rPhzvaihzTqw==</field>
            <field name="Sender">fiJOQPhAJqQJeuEOD+iH8Q==</field>
            ...
        </doc>
        ...
    </docs>
</batch>
```

In JSON:

```
{"batch": {
    "docs": [
        {"doc": {
            "_ID": "AAFE9rf++BCa3bQ4HgAA",
            "SendDate": "2010-07-18 05:22:35",
            "Size": "3654",
            "Subject": "RE: synopses copens silk seagull citizens",
            "Tags": {
                "add": ["AfterHours", "Customer"]
            },
            "InternalRecipients": {
                "add": ["KMTkYYrkL4MmrHxj//ZVhQ=="]
            },
            "Sender": "wWR7yZik2p1rrI6/qSepXg==",
            ...
        }},
```

```
        {"doc": {
            "_ID": "AAFE9rf++BCa3bQ4HgAB",
            "SendDate": "2010-07-17 10:37:03",
            "Size": "15830",
            "Subject": "bloodily douches spadones poinds vestals",
            "Tags": "AfterHours",
            "InternalRecipients": "Ii91O7qHb8rPhzvaihzTqw==",
            "Sender": "fiJOQPhAJqQJeuEOD+iH8Q==",
            ...
        }},
        ...
    ]
}}
```

Semantics about adding batches are described below:

- **Object ID**: If the `_ID` field is not defined for an object, Spider assigns a unique ID based (e.g., "`AAFE95dAqRCa3bQ4HgAA`"). The ID is a base 64-encoded 120-bit value that is generated in a way to ensure uniqueness even in a multi-node cluster.

- **MV scalars and link fields**: When an MV scalar or link field is assigned a single value, it can use the same syntax as an SV scalar field. For example, the link `Sender` is assigned a single value and uses the simplified name/value syntax. But when an MV scalar or link field is assigned multiple values, they must be enclosed in an `add` group. See for example the MV scalar `Tags`, which is assigned 2 values. An MV field can always use the `add` group syntax even if it being assigned a single value. See for example the link field `InternalRecipients`.

- **Batch size**: Batches can be arbitrarily large, but large batches require more memory since the entire parsed batch is held in memory. Also, mutations are flushed when their count reaches `batch_mutation_threshold`, defined in doradus.yaml. Hence, large batches can be stored as multiple "sub-batch" transactions.

- **Existing IDs**: If an object in an Add Batch command is given an ID that corresponds to an existing object, the existing object is updated. This preserves the idempotent update semantics of Add Batch commands: if the same object is added twice, the second "add" is treated as an "update" and, since all of the values will be the same, the second "add" will be a no-op.

- **Binary fields**: Binary field values must be encoded as declared using the `encoding` declared in the schema (Base64 or Hex).

- **Group fields**: Group fields can be ignored, and assignments can be made to leaf fields directly. However, leaf fields can also be qualified via the owning group. For example, in the `doc` group below, `Participants` and `Recipients` are both group fields:

```
    <doc>
        <field name="Participants">
            <field name="Recipients">
                <field name="InternalRecipients">KMTkYYrkL4MmrHxj//ZVhQ==</field>
```

```
            </field>
            <field name="Sender">wWR7yZik2p1rrI6/qSepXg==</field>
        </field>
        ...
    </doc>
```

If the Add Batch command is successful, the request returns a `201 Created` response. The response contains a `doc` element for each object in the batch:

```
<batch-result>
    <status>OK</status>
    <has_updates>true</has_updates>
    <docs>
        <doc>
            <updated>true</updated>
            <status>OK</status>
            <field name="_ID">AAFE9rf++BCa3bQ4HgAA</field>
        </doc>
        <doc>
        ...
        </doc>
    </docs>
</batch-result>
```

In JSON:

```
{"batch-result": {
    "status": "OK"
    "has_updates": "true",
    "docs": [
        {"doc": {
            "updated": "true",
            "status": "OK",
            "_ID": "AAFE9rf++BCa3bQ4HgAA"
        }},
        {"doc": {
            ...
        }}
    ]
}}
```

As shown, a `doc` element is given with an `_ID` value for every object added or updated including objects whose `_ID` was set by Doradus. The `status` element indicates if the update for that object was a valid request, and the `updated` element indicates if a change was actually made to the database for that object. If at least one object in the batch was updated, the `has_updates` element is included with a value of `true`. If an individual object update failed, its `status` will be `Error`, and an error message will be included in its `doc` element. If no objects in the batch were updated (because the existing objects already existed and had the requested values), the `has_updates` element will be absent. For example:

```
<batch-result>
```

```
    <status>OK</status>
    <docs>
        <doc>
            <updated>false</updated>
            <status>OK</status>
            <comment>No updates made</comment>
            <field name="_ID">AAFE9rf++BCa3bQ4HgAA</field>
        </doc>
        <doc>
            ...
        </doc>
    </docs>
</batch-result>
```

In JSON:

```
{"batch-result": {
    "status": "OK"
    "docs": [
        {"doc": {
            "updated": "false",
            "status": "OK",
            "comment": "No updates made",
            "_ID": "AAFE9rf++BCa3bQ4HgAA"
        }},
        {"doc": {
            ...
        }}
    ]
}}
```

If the entire batch is rejected, e.g., due to a syntax error in the parsed input message, a `400 Bad Request` is returned along with a plain text message. For example:

```
HTTP/1.1 400 Bad Request
Content-length: 62
Content-type: Text/plain

Child of link field update node must be 'add' or 'remove': foo
```

5.4.2 Update Batch

A batch of objects can be updated by issuing a PUT request to the appropriate application and table:

```
PUT /{application}/{table}
```

An input entity must be provided in JSON or XML as specified by the command's content-type. The message has the same format as the Add Batch command with the following differences:

- **Object IDs**: Every object in the batch must be assigned an ID. If the `_ID` field value is missing within a `doc` group, that update is skipped.

- **Nullifying SV scalars**: An object's SV scalar field can be set to null by giving it an empty value. Example:

```
<doc>
    <field name="SendDate"></field>
    ...
</doc>
```

  In JSON, the keyword `NULL` or an empty string ("") can be assigned to nullify an SV scalar.

- **Removing MV field values**: MV scalar and link values can be removed with a `remove` group. Example:

```
<doc>
    <field name="Tags">
        <remove>
            <value>AfterHours</value>
        </remove>
    </field>
    <field name="InternalRecipients">
        <remove>
            <value>Ii91O7qHb8rPhzvaihzTqw==</value>
        </remove>
    </field>
    ...
</doc>
```

  Values can be added and removed for the same field in the same `doc` group.

If the PUT request is successful, a `200 OK` response is returned with a `batch-result` group. It contains a `doc` element for each object updated, including `_ID`, `status`, and other elements as needed.

Updating an object is an idempotent operation: if an update does not actually change any of the object's fields, the update is a no-op. Hence, performing the same update in succession is safe.

5.4.3 Delete Batch

A batch of one or more objects can be deleted by issuing a DELETE command using the appropriate application and table:

```
DELETE /{application}/{table}
```

An input entity is required using the same syntax as for the Add Batch and Update Batch commands except that only the `_ID` field should be set for each `doc` group. All other field assignments are ignored. When an object is deleted that has link fields, inverse link values are fixed-up in the same request.

A `200 OK` response with a `batch-result` group is returned with a `doc` element for each requested _ID. The `doc` element indicates if the object was actually found or not. Example:

```
<batch-result>
    <status>OK</status>
    <docs>
```

```
        <doc>
            <status>OK</status>
            <comment>Object not found</comment>
            <field name="_ID">28dj2716rgq</field>
         </doc>
    </docs>
</batch-result>
```

Like other updates, deletes are idempotent: It is not an error to delete an already-deleted object.

5.5 Get Object Command

All fields of a single object can be fetched with a GET request using the application name, table name, and ID of the desired object:

```
GET /{application}/{table}/{ID}
```

The object's ID should not be quoted, but it must be URL-encoded. If the specified object is not found, a `404 Not Found` response is returned. Otherwise, all scalar and link fields of the object are returned as a `200 OK` response. Leaf fields are qualified within their owning group fields. For example:

```
GET /Msgs/Person/UdjrbyhvF%2FfPGV6fT4fewg%3D%3D
```

The response message is the same used for Add Batch and Update Batch commands. MV scalar and link fields with a single value are returned without the enclosing `add` group. For example in XML:

```
<doc>
    <field name="Name">Damien Munro</field>
    <field name="DirectReports">
        <add>
            <value>+/0sWHX9YiVnWYT+kqwxig==</value>
            <value>0iaYNVln92Blc0HFxyMMOA==</value>
            <value>aZcnXyb1AJj1OLO92drpjA==</value>
            <value>let9N7as5W/N+9Lib0woKQ==</value>
            <value>qBbk9tqZ5sF+/1x/xsC9JA==</value>
            <value>tIlJFIruDFOoStlzzB9vag==</value>
            <value>zqd7seWR5a+78JRKc8ztDQ==</value>
        </add>
    </field>
    <field name="Manager">VTuT6K8SlRMnNtdscYDwLQ==</field>
    <field name="FirstName">Damien</field>
    <field name="LastName">Munro</field>
    <field name="_ID">UdjrbyhvF/fPGV6fT4fewg==</field>
    <field name="Location">
        <field name="Department">Management-Sales</field>
        <field name="Office">Melbourne</field>
    </field>
</doc>
```

In JSON, the same response is:

```
{"doc": {
    "Name": "Damien Munro",
    "DirectReports": {
        "add": [
            "+/0sWHX9YiVnWYT+kqwxig==",
            "0iaYNVln92Blc0HFxyMMOA==",
            "aZcnXyb1AJj1OLO92drpjA==",
            "let9N7as5W/N+9Lib0woKQ==",
            "qBbk9tqZ5sF+/1x/xsC9JA==",
            "tIlJFIruDFOoStlzzB9vag==",
            "zqd7seWR5a+78JRKc8ztDQ=="
        ]
    },
    "Manager": "VTuT6K8SlRMnNtdscYDwLQ==",
    "FirstName": "Damien",
    "LastName": "Munro",
    "_ID": "UdjrbyhvF/fPGV6fT4fewg==",
    "Location": {
        "Department": "Management-Sales",
        "Office": "Melbourne"
    }
}}
```

5.6 Object Query Commands

Core concepts and command parameters for object queries are described in the document **Doradus Data Model and Query Language**. This document describes object query commands specific to Doradus Spider.

5.6.1 Object Query via URI

An object query can submit all query parameters in the URI of a GET request. The general form is:

```
GET /{application}/{table}/_query?{params}
```

where `{application}` is the application name, `{table}` is the perspective table to be queried, and `{params}` are URI parameters, separated by ampersands (&) and encoded as necessary. The following parameters are supported:

- **`q=text`** (required): A DQL query expression that defines which objects to select. Examples:

  ```
  q=*    // selects all objects
  q=FirstName:Doug
  q=NONE(InternalRecipients.Person.WHERE(LastName=Smith)).Department=Sales
  ```

- **`s=size`** (optional): Limits the number of objects returned. If absent, the page size defaults to the `search_default_page_size` option in the doradus.yaml file. The page size can be set to 0 to disable paging, causing all results to be returned in a single page. Examples:

  ```
  s=50
  s=0    // return all objects
  ```

- **f=*fields*** (optional): A comma-separated list of fields to return for each selected object. Without this parameter, all scalar fields of each object are returned. Link paths can use parenthetical or dotted qualification. Link fields can use WHERE filtering and per-object maximum value limits in square brackets. Examples:

```
f=*
f=_local
f=_all
f=Size,Sender.Person.*
f=InternalRecipients[3].Person.DirectReports.WHERE(LastName=Smith)[5].FirstName
f=Name,Manager(Name,Manager(Name))
```

- **o=*field* [ASC|DESC]** (optional): Orders the results by the specified *field*, which must be a scalar field belonging to the perspective table. Without this parameter, objects are returned in an internally-defined order. When an order field is specified, by default objects are sorted in ascending of the field's value. Optionally, the field name can be followed by ASC to explicitly request ascending order or DESC to request descending order. Examples:

```
o=FirstName
o=LastName DESC
```

- **k=*count*** (optional): Causes the first *count* objects in the query results to be skipped. This parameter can only be used in conjunction with the &o (order) parameter. If the &k parameter is omitted or 0, the first page of objects is returned. Examples:

```
k=100
k=0    // returns first page
```

- g=*token* (optional): Returns a secondary page of objects starting with the first object *greater* than the given continuation token. The continuation token must be one return in a previous query. The &g parameter can only be used for queries that are not sorted (no &o parameter). The &g and &e parameters cannot both be used in the same query.

- e=*token* (optional): Returns a secondary page of objects starting with the first object *equal* to the given continuation token. The continuation token must be one return in a previous query. The &e parameter can only be used for queries that are not sorted (no &o parameter). The &g and &e parameters cannot both be used in the same query.

Query results are *paged* using either of two methods depending on whether or not the results are sorted with the &o parameter:

- **Sorted queries**: To retrieve a secondary page, the same query text should be submitted along with the &k parameter to specify the number of objects to skip. Doradus Spider will re-execute the query and skip the specified number of objects.

- **Unsorted queries**: To retrieve a secondary page, the same query text should be submitted using either the &g or &e parameter to pass the continuation token from the previous page. This form of

paging is more efficient since Doradus Spider does not re-execute the entire query but instead *continues* the query from the continuation object.

The following object query selects people whose `LastName` is `Powell` and returns their full name, their manager's name, and their direct reports' name. The query is limited to 2 results:

```
GET /Msgs/Person/_query?q=LastName=Powell&f=Name,Manager(Name),DirectReports(Name)&s=2
```

A typical result in XML is shown below:

```
<results>
   <docs>
      <doc>
         <field name="Name">Karen Powell</field>
         <field name="_ID">gfNqhYF7LgBAtKTdIx3BKw==</field>
         <field name="DirectReports">
            <doc>
               <field name="Name">PartnerMarketing EMEA</field>
               <field name="_ID">mKjYJmmLPoTVxJu2xdFmUg==</field>
            </doc>
         </field>
         <field name="Manager">
            <doc>
               <field name="Name">David Cuss</field>
               <field name="_ID">nLOCpa7aH/Y3zDrnMqG6Fw==</field>
            </doc>
         </field>
      </doc>
      <doc>
         <field name="Name">Rob Powell</field>
         <field name="_ID">sHUm0PEKu3gQDDNIHHWv1g==</field>
         <field name="DirectReports"/>
         <field name="Manager">
            <doc>
               <field name="Name">Bill Stomiany</field>
               <field name="_ID">tkSQlrRqaeHsGvRU65g9HQ==</field>
            </doc>
         </field>
      </doc>
   </docs>
   <continue>sHUm0PEKu3gQDDNIHHWv1g==</continue>
</results>
```

The same result in JSON:

```
{"results": {
   "docs": [
      {"doc": {
         "Name": "Karen Powell",
         "_ID": "gfNqhYF7LgBAtKTdIx3BKw==",
         "DirectReports": [
```

```
                {"doc": {
                    "Name": "PartnerMarketing EMEA",
                    "_ID": "mKjYJmmLPoTVxJu2xdFmUg=="
                }}
            ],
            "Manager": [
                {"doc": {
                    "Name": "David Cuss",
                    "_ID": "nLOCpa7aH/Y3zDrnMqG6Fw=="
                }}
            ]
        }},
        {"doc": {
            "Name": "Rob Powell",
            "_ID": "sHUm0PEKu3gQDDNIHHWv1g==",
            "DirectReports" :[],
            "Manager": [
                {"doc": {
                    "Name": "Bill Stomiany",
                    "_ID": "tkSQlrRqaeHsGvRU65g9HQ=="
                }}
            ]
        }},
        "continue": "sHUm0PEKu3gQDDNIHHWv1g=="
    ]
}}
```

As shown, requested link fields are returned even if when they are empty. Because the query results were limited via the `&s` parameter, a `continue` token is provided for retrieving the next page.

5.6.2 Object Query via Entity

An object query can be performed by passing all parameters in an input entity. Because some clients do not support HTTP GET-with-entity, the PUT method can be used instead, even though no modifications are made. Both these examples are equivalent:

```
GET /{application}/{table}/_query
PUT /{application}/{table}/_query
```

The entity passed in the command must be a JSON or XML document whose root element is `search`. The query parameters are given as child elements. URI query parameters map to the following element names:

URI Parameter	Element name
e	continue-at
f	fields
g	continue-after
k	skip
o	order
q	query
s	size

Here is an example `search` document in XML:

```
<search>
    <query>LastName=Powell</query>
    <fields>Name,Manager(Name),DirectReports(Name)</fields>
    <size>2</size>
</search>
```

The same example in JSON:

```
{"search": {
    "query": "LastName=Powell",
    "fields": "Name,Manager(Name),DirectReports(Name)",
    "size": "2"
}}
```

5.7 Aggregate Query Commands

Core concepts and command parameters for aggregate queries are described in the document **Doradus Data Model and Query Language**. This section describes aggregate query commands specific to Doradus OLAP.

5.7.1 Aggregate Query via URI

An aggregate query can submit all parameters in the URI of a GET request. The REST command is:

```
GET /{application}/{table}/_aggregate?{params}
```

where `{application}` is the application name, `{table}` is the perspective table, and `{params}` are URI parameters separated by ampersands (&). The following parameters are supported:

- `m=`***`metric function list`*** (required): A list of one or more metric functions to calculate for selected objects. Each function is computed across selected objects, optionally subdivided into groups as defined by the grouping parameter. Example metric function:

  ```
  m=COUNT(*)
  m=DISTINCT(Name)
  m=SUM(Size)
  m=MAX(Sender.Person.LastName)
  m=AVERAGE(SendDate)
  m=MAX(Size), MIN(Size), AVERAGE(Size), COUNT(*)     // 4 metric functions
  ```

- `q=`***`text`*** (optional): A DQL query expression that defines which objects to include in metric computations. If omitted, all objects are selected (same as `q=*`). Examples:

  ```
  q=*
  q=LastName=Smith
  q=ALL(InternalRecipients.Person.Domain).IsInternal = false
  ```

- **f=*grouping list*** (optional): A list of one or more grouping expressions, which divide computations into single- or multi-level groups. When this parameter is omitted, the corresponding *global query* computes a single value for each metric function. When provided, the corresponding *grouped query* computes a value for each group value/metric function combination. Multiple grouping sets can be specified via the `GROUP` function. Examples:

```
f=Tags
f=TOP(3,Sender.Person.Department)
f=BATCH(Size,1000,10000,100000),TOP(5,ExternalRecipients.MessageAddress.Domain.Name)
f=GROUP(*),GROUP(Tags),GROUP(TOP(3,Sender.Person.Office),TOP(3,Sender.Person.Department))
```

- **cf=*grouping list*** (optional): For multi-level grouped queries, Doradus spider supports a special feature called *composite* grouping. It is requested by using `&cf` instead of `&f`, and it is meaningful only for aggregate queries with 2 or more grouping levels. The `&cf` parameter's value is the same as for `&f`, but it causes extra composite values to be computed for intermediate grouping levels.

Below is an example aggregate query using URI parameters:

```
GET /Msgs/Message/_aggregate?q=Size>10000&m=COUNT(*)&f=TOP(3,Sender.Person.Department)
```

An example response in XML is:

```
<results>
   <aggregate metric="COUNT(*)" query="Size>10000" group="TOP(3,Sender.Person.Department)"/>
   <totalobjects>1254</totalobjects>
   <summary>1254</summary>
   <totalgroups>37</totalgroups>
   <groups>
      <group>
         <metric>926</metric>
         <field name="Sender.Person.Department">(null)</field>
      </group>
      <group>
         <metric>82</metric>
         <field name="Sender.Person.Department">HR</field>
      </group>
      <group>
         <metric>45</metric>
         <field name="Sender.Person.Department">Sales Technical Specialists</field>
      </group>
   </groups>
</results>
```

In JSON:

```
{"results": {
   "aggregate": {
      "metric": "COUNT(*)",
      "query": "Size>10000",
      "group": "TOP(3,Sender.Person.Department)"
```

```
    },
    "totalobjects": "1254",
    "summary": "1254",
    "totalgroups": "37",
    "groups": [
       {"group": {
          "metric": "926",
          "field": {"Sender.Person.Department": "(null)"}
       }},
       {"group": {
          "metric": "82",
          "field": {"Sender.Person.Department": "HR"}
       }},
       {"group": {
          "metric": "45",
          "field": {"Sender.Person.Department": "Sales Technical Specialists"}
       }}
    ]
}}
```

5.7.2 Aggregate Query via Entity

Aggregate query parameters can be provided in an input entity instead of URI parameters. The same REST command is used except that no URI parameters are provided. Because some browsers/HTTP frameworks do not support HTTP GET-with-entity, this command also supports the PUT method even though no modifications are made. Both of the following are equivalent:

```
GET /{application}/{table}/_aggregate
PUT /{application}/{table}/_aggregate
```

where `{application}` is the application name and `{table}` is the perspective table. The input entity must be a JSON or XML document whose root element is `aggregate-search`. Aggregate query parameters are given as child elements. URI query parameters map to the following element names:

URI Parameter	Element name
cf	composite-fields
f	grouping-fields
m	metric
q	query

Below is an example aggregate query request entity in XML:

```
<aggregate-search>
   <query>Size&gt;10000</query>
   <metric>COUNT(*)</metric>
   <grouping-fields>TOP(3,Sender.Person.Department)</grouping-fields>
</aggregate-search>
```

In JSON:

```
{"aggregate-search": {
    "query": "Size>10000",
    "metric": "COUNT(*)",
    "grouping-fields": "TOP(3,Sender.Person.Department)"
}}
```

5.7.3 Composite Grouping Results

Composite grouping is requested by using the `&cf` parameter instead of the `&f` parameter. This generates an extra computation for each parent (non-leaf) group called a *composite* value. For example:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)&cf=TRUNCATE(SendDate,DAY),Tags
```

A typical result for this 2-level aggregate query in XML is shown below:

```
<results>
   <aggregate metric="COUNT(*)" group="TRUNCATE(SendDate,DAY),Tags"/>
   <totalobjects>6032</totalobjects>
   <summary>6032</summary>
   <groups>
      <group>
         <summary>4753</summary>
         <field name="SendDate">2010-07-17 00:00:00</field>
         <groups>
            <group>
               <metric>1</metric>
               <field name="Tags">(null)</field>
            </group>
            <group>
               <metric>4752</metric>
               <field name="Tags">AfterHours</field>
            </group>
            <group>
               <metric>1524</metric>
               <field name="Tags">Customer</field>
            </group>
         </groups>
      </group>
      <group>
         <summary>1279</summary>
         <field name="SendDate">2010-07-18 00:00:00</field>
         <groups>
            <group>
               <metric>1279</metric>
               <field name="Tags">AfterHours</field>
            </group>
            <group>
               <metric>701</metric>
               <field name="Tags">Customer</field>
            </group>
         </groups>
```

```
        </group>
        <group composite="true">
            <field name="SendDate">*</field>
            <groups>
                <group>
                    <metric>1</metric>
                    <field name="Tags">(null)</field>
                </group>
                <group>
                    <metric>6031</metric>
                    <field name="Tags">AfterHours</field>
                </group>
                <group>
                    <metric>2225</metric>
                    <field name="Tags">Customer</field>
                </group>
            </groups>
        </group>
    </groups>
</results>
```

The same response in JSON is shown below:

```
{"results": {
    "aggregate": {"metric": "COUNT(*)", "group": "TRUNCATE(SendDate,DAY),Tags"},
    "totalobjects": "6032",
    "summary": "6032",
    "groups": [
        {"group": {
            "summary": "4753",
            "field": {"SendDate": "2010-07-17 00:00:00"},
            "groups": [
                {"group": {
                    "metric": "1",
                    "field": {"Tags": "(null)"}
                }},
                {"group": {
                    "metric": "4752",
                    "field": {"Tags": "AfterHours"}
                }},
                {"group": {
                    "metric": "1524",
                    "field": {"Tags": "Customer"}
                }}
            ]
        }},
        {"group": {
            "summary": "1279",
            "field": {"SendDate": "2010-07-18 00:00:00"},
            "groups": [
                {"group": {
```

```
                "metric": "1279",
                "field": {"Tags": "AfterHours"}
            }},
            {"group": {
                "metric": "701",
                "field": {"Tags": "Customer"}
            }}
        ]
    }},
    {"group": {
        "composite": "true",
        "field": {"SendDate": "*"},
        "groups": [
            {"group": {
                "metric": "1",
                "field": {"Tags": "(null)"}
            }},
            {"group": {
                "metric": "6031",
                "field": {"Tags": "AfterHours"}
            }},
            {"group": {
                "metric": "2225",
                "field": {"Tags":"Customer"}
            }}
        ]
    }}
  ]
}}
```

As shown, composite grouping produces an extra group for non-leaf grouping levels. This group is marked with a `composite` property of `true`, and the value for its `field` element is "*". Within the composite group, lower-level metric groups are provided for each lower-level grouping field value, however, these lower-level metrics are computed across all objects at the composite grouping level. In the example above, the composite group computes the metric function (`COUNT(*)`) for all second-level groups (`Tags`) across all first-level group values (`SendDate`).

Composite grouping is only meaningful for multi-level grouping.

5.7.4 Compound Grouping Results

Compound grouping is requested by using a comma-separated list of `GROUP` functions within the grouping parameter. Each `GROUP` function is called a *grouping set*, and its parameter can denote a global, single-level, or multi-level grouping expression. If the grouping parameter is passed as a composite grouping parameter (`&cf`), a composite group is created for each non-leaf group within each group set. Consider this example:

```
GET /Msgs/Message/_aggregate?m=MAX(Size)
    &cf=GROUP(*),GROUP(TRUNCATE(SendDate,WEEK)),GROUP(TOP(2,TERMS(Subject)),Tags)
    &q=SendDate > 2013-10-15
```

This compound grouping aggregate query selects messages whose `SendDate` is >= `2013-10-15`, and it computes the following:

- The maximum `Size` value of selected messages (`GROUP(*)`).
- The maximum `Size` of selected messages grouped by `SendDate` truncated to WEEK granularity (`GROUP(TRUNCATE(SendDate,WEEK))`).
- The maximum `Size` of selected messages grouped first by the top 2 terms used in the `Subject` field and then by the `Tags` field (`GROUP(TOP(2,TERMS(Subject)),Tags)`). Because composite grouping was requested (`&cf`), this multi-level grouping expression uses the composite grouping technique.

Compound aggregate queries compute all grouping sets in a single pass. The query above returns XML results such as the following:

```
<results>
   <aggregate metric="MAX(Size)" query="SendDate > 2009-10-15"
      group="GROUP(*),GROUP(TRUNCATE(SendDate,WEEK)),GROUP(TOP(2,TERMS(Subject)),Tags)"/>
   <totalobjects>6032</totalobjects>
   <groupsets>
      <groupset>
         <value>16796009</value>
      </groupset>
      <groupset group="TRUNCATE(SendDate,WEEK)">
         <summary>16796009</summary>
         <groups>
            <group>
               <field name="SendDate">2010-07-12 00:00:00</field>
               <metric>965230</metric>
            </group>
            ...
         </groups>
      </groupset>
      <groupset group="TOP(2,TERMS(Subject)),Tags">
         <summary>16796009</summary>
         <totalgroups>15267</totalgroups>
         <groups>
            <group>
               <summary>16796009</summary>
               <field name="Subject">scalepan</field>
               <groups>
                  <group>
                     <metric>16796009</metric>
                     <field name="Tags">AfterHours</field>
                  </group>
                  ...
               </groups>
            </group>
            ...
            <group composite="true">
```

```
                <field name="Subject">*</field>
                <groups>
                    <group>
                        <metric>7317</metric>
                        <field name="Tags">(null)</field>
                    </group>
                    ...
                </groups>
            </group>
        </groups>
    </groupset>
</groupsets>
</results>
```

In JSON:

```
{"results": {
    "aggregate": {
        "group": "GROUP(*),GROUP(TRUNCATE(SendDate,WEEK)),GROUP(TOP(2,TERMS(Subject)),Tags)",
        "query": "SendDate > 2009-10-15",
        "metric":"MAX(Size)"
    },
    "totalobjects": "6032",
    "groupsets": [
        {"groupset": {
            "value": "16796009"
        }},
        {"groupset": {
            "group": "TRUNCATE(SendDate,WEEK)",
            "summary": "16796009",
            "groups": [
                {"group": {
                    "field": {"SendDate": "2010-07-12 00:00:00"},
                    "metric": "16796009"
                }},
                ...
            ]
        }},
        {"groupset": {
            "group": " TOP(2,TERMS(Subject)),Tags ",
            "summary": "16796009",
            "totalgroups": "15267",
            "groups": [
                {"group": {
                    "summary": "16796009",
                    "field": {"Subject": "scalepan"},
                    "groups": [
                        {"group": {
                            "metric": "16796009",
                            "field": {"Tags": "AfterHours"}
                        }},
```

```
                    {"group": {
                        "metric": "16796009",
                        "field": {"Tags": "Customer"}
                    }}
                ]
            }},
            ...
            {"group": {
                "composite": "true",
                "field": {"Origin": "*"},
                "groups": [
                    {"group": {
                        "metric": "7317",
                        "field": {"Tags": "(null)"}
                    }},
                    ...
                ]
            }}
        ]
    }}
  ]
}}
```

Notable aspects of a compound group result:

- As with all aggregate queries, the outer `results` element contains an `aggregate` element that confirms the aggregate query parameters.
- The `results` element also contains a `groupsets` element, which contains one `groupset` element per grouping set, that is, for each `GROUP` function.
- The contents of each `groupset` element follows the format applicable for global, single-level, or multi-level aggregate queries, except that they do not contain an `aggregate` element.
- As with all grouped aggregate queries, each `groupset` and each non-leaf group contains a `summary` value.
- Only multi-level grouping sets can contain a composite group, denoted by a `composite=true` element and a `field` value of "*".

5.7.5 Compound/Multi-metric Grouping Results

Multi-metric aggregate queries can also use compound grouping. That is, a single aggregate query can specify multiple metric functions in the `&m` parameter and multiple `GROUP` functions in the `&f` or `&cf` parameter. Such queries compute multiple metric functions, like a multi-metric query, and provide multiple grouping operations for each metric, all in a single query.

The results of compound/multi-metric queries are returned using `groupset` elements for each combination of metric function and `GROUP` function. If there are *m* metric functions and *n* `GROUP` functions, the result will

contain *m x n* `groupset` elements. Each `groupset` identifies the metric and grouping parameter for which it provides results.

As an example, the following compound/multi-metric aggregate query has 3 metric functions and 2 `GROUP` functions:

```
GET /Msgs/Message/_aggregate?m=COUNT(*),MAX(Size),AVERAGE(Size)
    &cf=GROUP(TOP(2,Tags),Subject),GROUP(TRUNCATE(SendDate,DAY))
```

This means the result will contain 6 `groupset` elements, as shown in the following XML outline:

```
<results>
   <aggregate metric="COUNT(*),MAX(Size),AVERAGE(Size)"
      group="GROUP(TOP(2,Tags),Subject),GROUP(TRUNCATE(SendDate,DAY))"/>
   <groupsets>
   <groupset group="TOP(2,Tags),Subject" metric="COUNT(*)">...</groupset>
   <groupset group="TRUNCATE(SendDate,DAY)" metric="COUNT(*)">...</groupset>
   <groupset group="TOP(2,Tags),Subject" metric="MAX(Size)">...</groupset>
   <groupset group="TRUNCATE(SendDate,DAY)" metric="MAX(Size)">...</groupset>
   <groupset group="TOP(2,Tags),Subject" metric="AVERAGE(Size)">...</groupset>
   <groupset group="TRUNCATE(SendDate,DAY)" metric="AVERAGE(Size)">...</groupset>
   </groupsets>
</results>
```

In JSON:

```
{"results": {
   "aggregate": {
      "group": "GROUP(TOP(2,Tags),Origin),GROUP(TRUNCATE(SendDate,DAY))",
      "metric": "COUNT(*),MAX(Size),AVERAGE(Size)"
   },
   "groupsets": [
      {"groupset": {"group": "TOP(2,Subject),Origin", "metric": "COUNT(*)", ...}},
      {"groupset": {"group": "TRUNCATE(SendDate,DAY)", "metric": "COUNT(*)", ...}},
      {"groupset": {"group": "TOP(2,Subject),Origin", "metric": "MAX(Size)", ...}},
      {"groupset": {"group": "TRUNCATE(SendDate,DAY)", "metric": "MAX(Size)", ...},
      {"groupset": {"group": "TOP(2,Subject),Origin", "metric": "AVERAGE(Size)", ...}},
      {"groupset": {"group": "TRUNCATE(SendDate,DAY)", "metric": "AVERAGE(Size)", ...}}
   ]
}}
```

Though not shown here, each `groupset` will contain `groups`, `summary`, and `totalgroups` elements as required by each GROUP function. If the query requests composite grouping (`&cf`), a `groupset` with multi-level grouping will contain a composite group for non-leaf groups.

5.8 Statistic Commands

Once declared, statistics do not have a value until they are automatically refreshed by a scheduled task or explicitly refreshed via a REST command. REST commands are also used to view the refresh status of

statistics and to retrieve their metric computation(s). This section describes the REST commands that Doradus Spider provides for statistics.

5.8.1 Refresh Statistics

All statistics owned by a given table can be explicitly refreshed by issuing the following REST command:

```
PUT /{application}/{table}/_statistics/_refresh
```

Where `{application}` is the Spider application name and `{table}` is the table name whose statistics are to be refreshed. This command is allowed on all Spider applications, though it is a no-op if the table does not own any statistics. It returns a `200 OK` response with no response message.

Alternatively, a single statistic can be explicitly refreshed using the following command:

```
PUT /{application}/{table}/_statistics/{stat}_refresh
```

Where `{stat}` is the statistic's name. An error is returned if the given statistic name is unknown. Otherwise, it returns a `200 OK` response.

For both commands, the affected statistic(s) are refreshed asynchronously in a background task, hence the command returns quickly.

5.8.2 Get Refresh Status

The current refresh status of all statistics in a given table can be requested with the following REST command:

```
GET /{application}/{table}/_statistics/_status
```

Where `{application}` is the Spider application name and `{table}` is the table name whose refresh status is desired. A response is always `200 OK` though the response entity will be empty for tables that have no statistics. An example for a table with statistics is shown below:

```
<stats-status>
   <tables>
      <table name="Message">
         <statistics>
            <statistic name="DepartmentTrend">
               Statistic has not been scheduled for recalculation
            </statistic>
            <statistic name="EmailSizes">
               Recalculation finished at: 2014-03-25 16:00:10; recalculation time: 0 seconds
            </statistic>
            <statistic name="EmailsPerDay">
               Recalculation finished at: 2014-03-25 16:00:10; recalculation time: 0 seconds
            </statistic>
            <statistic name="TotalCount">
               Recalculation finished at: 2014-03-25 16:00:10; recalculation time: 0 seconds
            </statistic>
         </statistics>
```

```
        </table>
    </tables>
</stats-status>
```

The same response in JSON is shown below:

```
{"stats-status": {
    "tables": {
        "Message": {
            "statistics": {
                "DepartmentTrend":
                    "Statistic has not been scheduled for recalculation",
                "EmailSizes":
                    "Recalculation finished at: 2014-03-25 16:00:10; recalculation time: 0 seconds",
                "EmailsPerDay":
                    "Recalculation finished at: 2014-03-25 16:00:10; recalculation time: 0 seconds",
                "TotalCount":
                    "Recalculation finished at: 2014-03-25 16:00:10; recalculation time: 0 seconds"
            }
        }
    }
}}
```

As shown, the response indicates if each statistic has been refreshed and if so the time and duration with which it was last refreshed.

5.8.3 Query Statistic

A statistic's metric value(s) are retrieved with the following REST command:

```
GET /{application}/{table}/_statistics/{stat}?{params}
```

Where `{application}` is the Spider application name, `{table}` is the table name that owns the statistic, and `{stat}` is the statistic's name. If the `{params}` parameter is omitted, all of the statistic's metric values are retrieved. If the statistic has a grouping parameter, `{params}` can be used to retrieve a subset of the available metrics corresponding to specific groups. The value for `{params}` is an ampersand-separated list of `GROUP` functions that use one of the following patterns:

```
GROUP(n):<range>
```

or:

```
GROUP(n)=<value>
```

where `n` is a group level and `:<range>` and `=<value>` are expressions to select a range of values or a single value for the corresponding groups.

If a statistic is queried that has not yet been computed, the result contains the `statistic` definition but no other elements. Example:

```
<results>
```

```
    <statistic name="TotalCount" metric="COUNT(*)"/>
</results>
```

In JSON:

```
{"results": {
    "statistic": {"name": "TotalCount", "metric": "COUNT(*)"}
}}
```

Examples for fetching global and grouped statistics that have values are shown in the next sections.

5.8.3.1 Fetching Global Statistics

If the statistic has no grouping parameter, no `{params}` are allowed; the single metric value computed is returned. For example:

```
GET /Msgs/Message/_statistics/TotalCount
```

This command returns the statistic's single metric value in a `results` group as shown below in XML:

```
<results>
    <statistic metric="COUNT(*)" name="TotalCount"/>
    <value>6030</value>
</results>
```

In JSON:

```
{"results": {
    "statistic": {"metric": "COUNT(*)", "name": "TotalCount"},
    "value":"6030"
}}
```

As shown, the `statistic` definition is returned for reference. The global metric value is returned in a `value` element.

5.8.3.2 Fetching Statistics With a Single Grouping Level

When a statistic has a grouping parameter, by default the metric values for all group levels are returned in the response. For example:

```
GET /Msgs/Message/_statistics/EmailsPerDay
```

This returns all groups in the `EmailsPerDay` statistics such as those shown below in XML:

```
<results>
    <statistic name="EmailsPerDay" metric="COUNT(*)" group="TRUNCATE(SendDate,DAY)"/>
    <groups>
        <group>
            <field name="SendDate">2010-07-17 00:00:00</field>
            <metric>4752</metric>
        </group>
        <group>
            <field name="SendDate">2010-07-18 00:00:00</field>
```

```
            <metric>1278</metric>
         </group>
         ...
      </groups>
      <summary>6030</summary>
   </results>
```

In JSON:

```
{"results": {
    "statistic": {"name": "EmailsPerDay", "metric": "COUNT(*)", "group": "TRUNCATE(SendDate,DAY)"},
    "groups": [
       {"group": {
           "field": {"SendDate": "2010-07-17 00:00:00"},
           "metric": "4752"
       }},
       {"group": {
           "field": {"SendDate": "2010-07-18 00:00:00"},
           "metric": "1278"
       }},
       ...
    ],
    "summary": "6030"
}}
```

These results are very similar to those returned if the equivalent aggregate query is performed dynamically.

A subset of the available groups can be fetched by passing a `GROUP` parameter that selects the desired group(s). For example, to fetch the computation of the group "2010-07-17 00:00:00" only:

```
GET /Msgs/Message/_statistics/EmailsPerDay?GROUP(1)="2010-07-17 00:00:00"
```

The value of the `GROUP` function must be compatible with the grouping field type. In this case, it is a timestamp value because the grouping field is a timestamp field. A subset response is returned such as the following:

```
<results>
   <statistic name="EmailsPerDay" metric="COUNT(*)" group="TRUNCATE(SendDate,DAY)"/>
   <groups>
      <group>
         <field name="SendDate">2010-07-17 00:00:00</field>
         <metric>4752</metric>
      </group>
   </groups>
   <summary>6030</summary>
</results>
```

Note that the `summary` value is returned even when a subset of the outer groups is returned.

If no groups exist that match the `GROUP` parameter, the response includes the summary value but no `groups` element. Therefore, it appears as a global statistic. Example:

```
<results>
   <statistic name="EmailsPerDay" metric="COUNT(*)" group="TRUNCATE(SendDate,DAY)"/>
   <value>6030</value>
</results>
```

Alternatively, a range of group values can also be selected, which is shown in the next example.

5.8.3.3 Fetching Statistics With Multiple Grouping Levels

When a statistic has multiple grouping levels, a limited set of values can be selected for each group level. The first `GROUP` function follows the query parameter "?"; subsequent `GROUP` functions should be separated by ampersands (&). Each `GROUP` function should include a unique group level. Example:

```
GET /Msgs/Message/_statistics/DepartmentTrend
   ?GROUP(1)='BU Product Management'
   &GROUP(2):['2010-07-17' TO '2010-07-18']
```

The `DepartmentTrend` statistic is grouped first by `Sender.Person.Department` (a text field) and then by `TRUNCATE(SendDate.DAY)` (a timestamp field). The Query Statistic request above requests metrics for:

- A single level 1 group: the department named "BU Product Management", and
- A range of level 2 groups: those whose value falls between 2010-07-17 and 2010-07-18, inclusively.

Note that the right hand side of the `GROUP` parameter must be compatible with the field type of the corresponding grouping field. The `GROUP` parameter can be compared to a single value using an equality operator (=) or to a range of values using the range clause (:). Each `GROUP` parameter level can be used only once.

When multiple `GROUP` parameters are used, only metric values for the matching outer and inner group values are returned. A typical result for the previous example is shown below:

```
<results>
   <statistic name="DepartmentTrend" metric="COUNT(*)"
      group="Sender.Person.Department,TRUNCATE(SendDate,DAY)"/>
   <groups>
      <group>
         <field name="Sender">BU Management/Admin</field>
         <groups>
            <group>
               <field name="SendDate">2010-07-17 00:00:00</field>
               <metric>3</metric>
            </group>
         </groups>
         <summary>3</summary>
      </group>
   </groups>
</results>
<summary>6030</summary>
```

As shown, the `summary` values for all non-leaf groups are always included.

5.9 Task Management Commands

Doradus Spider uses two kinds of background tasks. *Scheduled* tasks such as data aging and statistics refreshing are defined in an application's schema and execute automatically based on their assigned schedule. Through REST commands, scheduled tasks can be temporarily suspended from normal execution and later resumed. A scheduled task can also be started immediately, or, if it currently executing, it can be stopped.

On-demand tasks are requested via REST commands and execute once to perform a data cleanup function. Once started, an on-demand task can be stopped before it finishes via a REST command.

Both scheduled and on-demand tasks can be monitored via REST commands. This section describes the task management REST commands supported by Doradus Spider.

On multi-node clusters, tasks are distributed between Doradus server instances so that the background task workload is shared. Task execution is influenced by parameters defined in the doradus.yaml file. See the **Doradus Administration** document for information about configuring these parameters.

5.9.1 Start Data Cleanup Task

As described in the section **Modify Application**, Doradus Spider adds and deletes stores (ColumnFamilies) as needed for new and obsolete tables. However, Spider does not automatically reorganize data for some changes such as removing a field or changing its type. Leaving obsolete data in the database may be acceptable for some situations. However, if you decide that obsolete data should be removed or re-indexed, you can use a REST command to request a one-time (on-demand) data cleanup task.

A data cleanup task is started by issuing the following REST command:

```
POST /_tasks/{application}/{table}/{task}/{field}[?{params}]
```

Where:

- `{application}` is the Spider application name,
- `{table}` is the table name that owns the data to be cleaned-up,
- `{task}` is the type of data cleanup task desired,
- `{field}` is the name of the existing or deleted field to be cleaned-up,
- `{params}` is an optional set of parameters passed for some cleanup tasks.

If the POST request is successful, a `200 OK` is returned with a simple text message such as "Task added". The on-demand task can then be monitored with the Get Task Status command or stopped with a Stop Data Cleanup Task command (described later).

The possible values for `{task}` and the associated cleanup task process are described in the following sections.

5.9.1.1 Delete-link Task

Examples:

```
POST /_tasks/Msgs/Message/delete-link/ExternalRecipients
```

or:

```
POST /_tasks/Msgs/Message/delete-link/ExternalRecipients
   ?inverse=MessageAsExternalRecipient&table=Participant
```

This cleanup task deletes all values for a deleted link field. It is not needed when a table containing a link is deleted at the same time as the link's inverse table is deleted. However, it is needed in the following two cases:

1) The table containing a link field is deleted, but the inverse link's table was not deleted. In this case, old data remains for the inverse link. This task can be used to delete the obsolete data for the inverse link.

2) Both a link field and its inverse field were deleted, but neither link's owning table was deleted. In this case, old data remains for both links. This task can be used to delete obsolete data for both links.

For case 1), the delete-link task should be requested without `{params}` (first example above). The cleanup task scans the table for values belonging to the given link name and deletes them, but it does not attempt to find and delete inverse values.

For case 2), the delete-link task should be given one link as the `{field}` parameter, and `{params}` should be used to specify the `inverse` link name and its owning `table` name (second example). The cleanup task scans the perspective table for link values and deletes them. For each value found, it also deletes the link inverse value with the given `inverse` name from the given `table`.

5.9.1.2 Delete-scalar Task

Example:

```
POST /_tasks/Msgs/Message/delete-scalar/ThreadID
```

This cleanup task deletes all values and all index data for a scalar field. It can be used to remove obsolete data for a deleted scalar, or it can be used to nullify values for a current scalar (e.g., if its type has changed). The specified `{field}` does not have to exist in the current schema. The cleanup tasks scans all objects in the table and removes both field values and index data found for the field.

5.9.1.3 Re-index Task

Example:

```
POST /_tasks/Msgs/Message/re-index/Tags
```

This cleanup task can be used for a scalar field whose type has changed. This includes scalar fields that were originally undeclared in the schema, assigned values (which were indexed with the `TextAnalyzer`), and later declared in the schema with a type other than Text. The specified `{field}` must be defined in the

current schema, and it must be a scalar field. The cleanup task scans all existing object values and re-indexes the field with the analyzer currently defined for it.

5.9.2 Stop Data Cleanup Task

An executing cleanup task can be stopped with either of the following REST commands:

```
DELETE /_tasks/{application}/{table}/{task-type}
DELETE /_tasks/{application}/{table}/{task-type}/{field}
```

Where `{application}` is the name of a Spider application, `{table}` is the name of a table the application owns, and `{task-type}` is `delete-link`, `delete-scalar`, or `re-index`. If there are multiple cleanup tasks of the same type running for the same table, the cleanup task's `{field}` name can be provided to stop only the corresponding task. Otherwise, all tasks of the given `{task-type}` are stopped.

If one or more tasks were found and stopped, a `200 OK` response is returned with a plain text message "Task(s) interrupted". If no executing tasks were found to stop, a `404 Not Found` response is returned.

5.9.3 Get Task Status

The following REST commands list the status, respectively, of (1) all tasks defined for the Doradus instance, (2) all tasks defined for a given application, (3) all tasks for a given table, and (4) all tasks of a given type for a specified table:

```
GET /_tasks
GET /_tasks/{application}
GET /_tasks/{application}/{table}
GET /_tasks/{application}/{table}/{task}
```

Where `{application}` is the name of an application, `{table}` is the name of a table owned by the application, and `{task}` is a scheduled task type owned by the table. When provided, `{task}` must be one of `data-aging`, `stat-refresh`, or `data-checks`. Tasks matching the given request, if any, are returned in a response such as the following. In XML:

```
<tasks>
   <task name="Msgs/Message/data-aging">
      <schedule>0 3 * * SAT</schedule>
      <state>Undefined</state>
   </task>
      <task name="Msgs/Message/stat-refresh">
      <schedule>*/30 * * * *</schedule>
      <state>Succeeded</state>
      <last-scheduled-time>Wed Mar 26 14:00:00 PDT 2014</last-scheduled-time>
      <last-started-time>Wed Mar 26 14:00:06 PDT 2014</last-started-time>
      <last-finished-time>Wed Mar 26 14:00:07 PDT 2014</last-finished-time>
   </task>
      <task name="Msgs/Message/delete-scalar/MessageID">
      <state>Succeeded</state>
      <last-scheduled-time>Wed Mar 26 14:47:00 PDT 2014</last-scheduled-time>
      <last-started-time>Wed Mar 26 14:47:14 PDT 2014</last-started-time>
```

```
        <last-finished-time>Wed Mar 26 14:47:16 PDT 2014</last-finished-time>
    </task>
</tasks>
```

In JSON:

```
{"tasks": {
    "Msgs/Message/data-aging": {
        "schedule":"0 3 * * SAT",
        "state": "Undefined"
    },
    "Msgs/Message/stat-refresh": {
        "schedule": "*/30 * * * *",
        "state": "Succeeded",
        "last-scheduled-time": "Wed Mar 26 14:00:00 PDT 2014",
        "last-started-time": "Wed Mar 26 14:00:06 PDT 2014",
        "last-finished-time": "Wed Mar 26 14:00:07 PDT 2014"
    },
    "Msgs/Message/delete-scalar/MessageID": {
        "state": "Succeeded",
        "last-scheduled-time": "Wed Mar 26 14:47:00 PDT 2014",
        "last-started-time": "Wed Mar 26 14:47:14 PDT 2014",
        "last-finished-time": "Wed Mar 26 14:47:16 PDT 2014"
    }
}}
```

As shown, each task has a hierarchical name that follows the format:

```
{application}/{table}/{task}[/{name}]
```

`{task}` can be one any of the scheduled task types (`data-aging`, `data-checks`, or `stat-refresh`) or an on-demand task type (`delete-link`, `delete-scalar`, or `re-index`). The `{name}` is a statistic name for the `stat-refresh` task or the scalar or link field name for an on-demand cleanup task. For schedulable tasks, the respons includes the `schedule` definition of each task.

5.9.4 Modify Task

The following REST commands modify the current status of, respectively, (1) all commands owned by a given application, (2) all tasks owned by a given table, and (3) a table-level task of a given type:

```
PUT /_tasks/{application}?{command}
PUT /_tasks/{application}/{table}?{command}
PUT /_tasks/{application}/{table}/{task}?{command}
```

Where `{application}` is the name of an application, `{table}` is the name of a table owned by the application, and `{task}` is a scheduled task type owned by the table. When provided, `{task}` must be one of `data-aging`, `stat-refresh`, or `data-checks`. The given `{command}` must be one of the following:

- `start`: Immediately starts the affected task if it is not already running. This command can only be used for a single task at a time.

- `interrupt` (synonym: `stop`): Sends a signal to the affected task(s) to stop running. Actual behavior may depend on the task itself, but normally the task should check the interruption flag from time to time and stop execution when interrupted.
- `suspend`: Stops scheduling of the affected task(s). Already-running tasks keep running until finished or interrupted.
- `resume`: Resumes scheduling of the affected task(s) according to their defined schedules.

If successful, the Modify Task command returns a `200 OK` response. A `400 Bad Request` response is returned if the command is not recognized.