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, including its data model, query language, and REST API commands.

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 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:

- Architecture: An overview of the Doradus architecture. The Doradus OLAP and Doradus Spider databases are compared.
- 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: Describes core Doradus data model concepts and describes the extensions specific to Doradus Spider.
- Doradus Query Language (DQL): A detailed description of DQL concepts including extensions specific to Doradus Spider databases.
- Spider Object Queries: Describes the Object Query type including parameters and output results in XML and JSON.
- Spider Aggregate Queries: Describes the Aggregate Query type including parameters and output results in XML and JSON.
- Spider REST Commands: Describes the REST commands supported by Spider databases for application management, updates, queries, and task management.

1.1 Recent Changes

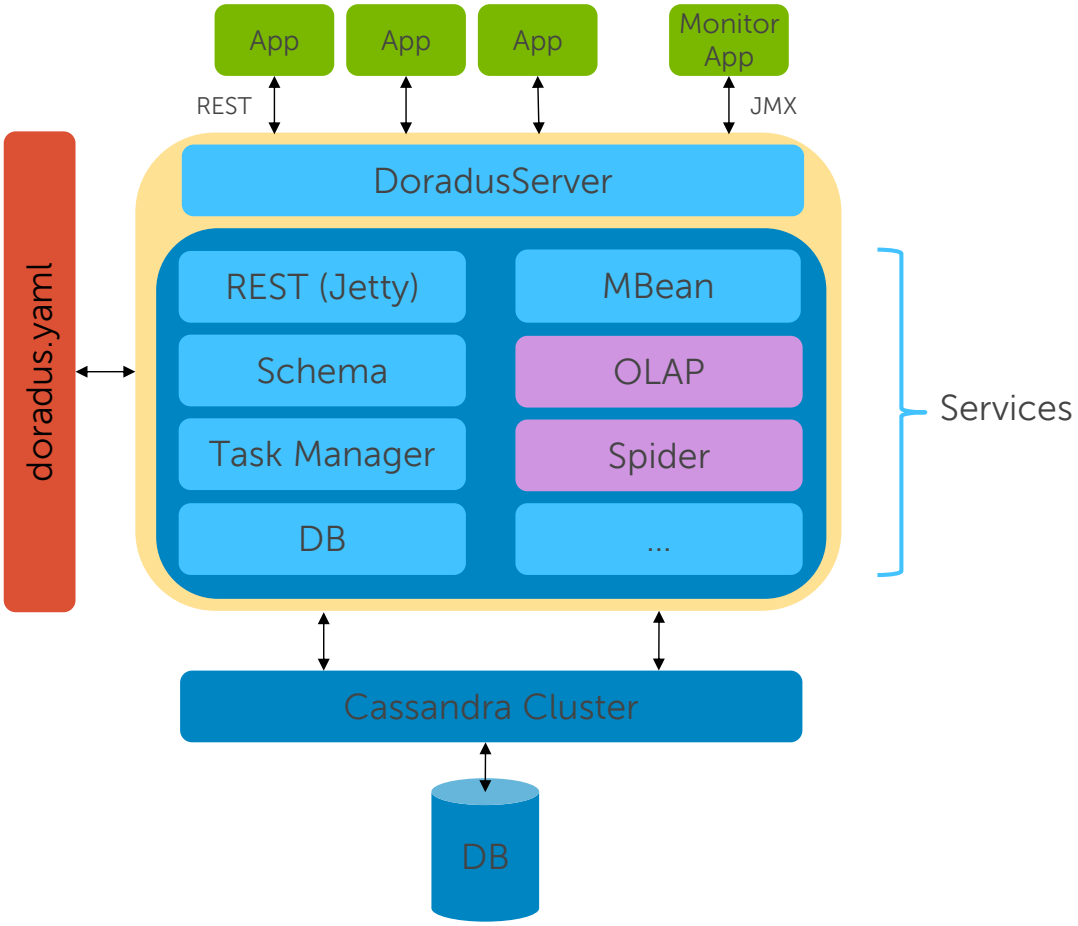
The following changes were made to reflect new features for the v2.2 release:

- The `TOP` and `BOTTOM` functions allow the limit parameter to be 0, which causes all groups to be returned. This allows all groups to be returned in metric-value order instead of grouping-field-value order.
- Link paths in query selection expressions can now begin with a `WHERE` filter. Such outer or "zero level" `WHERE` filters are applied to perspective objects.
- The semantics of using `IS NULL` clauses with link paths has changed. The section **Quantifiers with IS NULL** reflects the new behavior.
- In an aggregate function that uses a text grouping field, the `INCLUDE` and `EXCLUDE` functions now treat the included/excluded values as case-insensitive. Also, text values can use wildcards `?` and `*`.

2. Architecture

Doradus is a Java server application that leverages and extends the Cassandra NoSQL database. At a high level, it is a REST service that sits between applications and a Cassandra cluster, adding powerful features to—and hiding complexities in—the underlying database. This allows applications to leverage the benefits of NoSQL such as horizontal scalability, replication, and failover while enjoying rich features such as full text searching, bi-directional relationships, and powerful analytic queries.

An overview of Doradus architecture is depicted below:



Key components of this architecture are summarized below:

- **Apps**: One or more applications access a Doradus server instance using a simple **REST** API. A **JMX** API is available to monitor Doradus and perform administrative functions.
- **DoradusServer**: This core component controls server startup, shutdown, and services. Entry points are provided to run the server as a stand-alone application, as a Windows service (via procrun), or embedded within another application.
- **Services**: Doradus' architecture encapsulates functions within service modules. Services are initialized based on the server's **doradus.yaml** configuration. Services provide functions such as the

REST API (an embedded Jetty server), **Schema** processing, and physical **DB** access. A special class of *storage services* provide storage and access features for specific application types. Doradus currently provides two storage services:

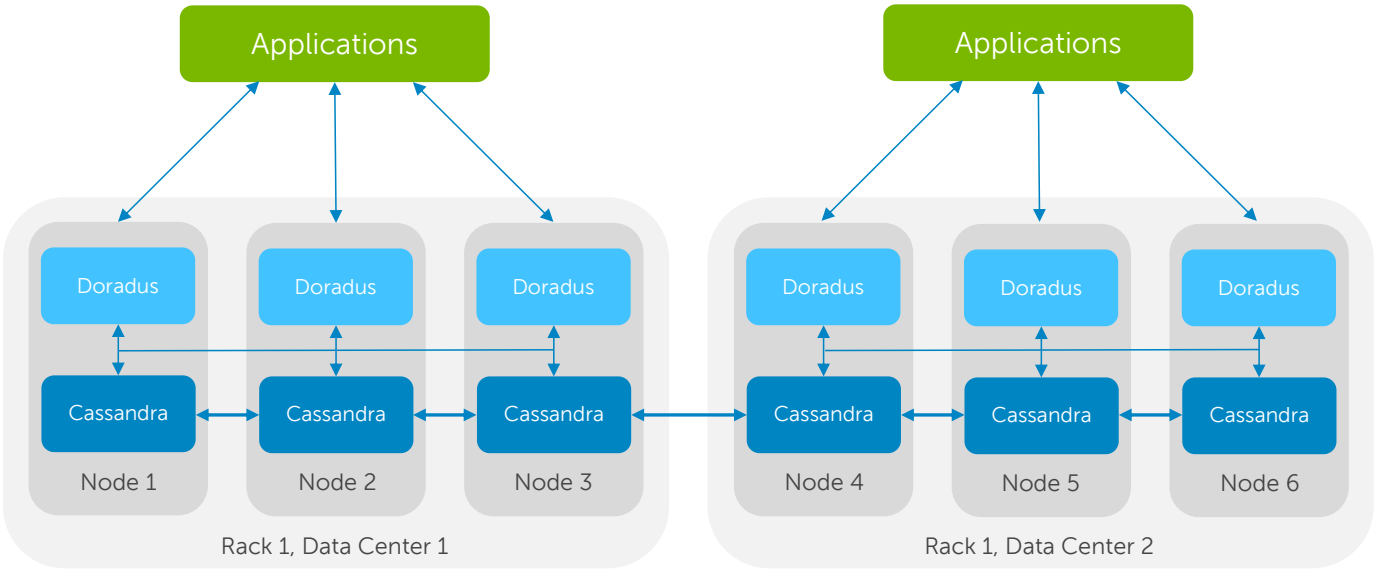
- **OLAP Service**: A Doradus database configured to use the OLAP storage service is termed a *Doradus OLAP Database*. OLAP uses online analytical processing techniques to provide dense storage and very fast processing of analytical queries. This service is ideal for applications that use immutable or semi-mutable time-series data.
- **Spider Service**: A Doradus database configured to use the Spider storage service is termed a *Doradus Spider Database*. The Spider service supports schemaless applications, fully inverted indexing, fine-grained updates, table-level sharding, and other features that support applications that use highly mutable and/or variable data.

Doradus can be configured to use both storage services in a single instance.

- **Cassandra Cluster**: Doradus currently uses the Apache Cassandra NoSQL database for persistence. Future releases are intended to use other data stores. Cassandra performs the "heavy lifting" in terms of persistent, replication, load balancing, replication, and more.

The minimal deployment configuration is a single Doradus instance and a single Cassandra instance running on the same machine. On Windows, these instances can be installed as services. The Doradus server can also be embedded in the same JVM as an application.

Multiple Doradus and Cassandra instances can be deployed to scale a cluster horizontally. An example of a Doradus/Cassandra multi-node cluster is shown below:



This example demonstrates several deployment features:

- One Doradus instance and one Cassandra instance are typically deployed on each node.

- Doradus instances are *peers*, hence an application can submit requests to any Doradus instance in the cluster.
- Each Doradus instance is typically configured to use all *network near* Cassandra instances. This allows it to distribute requests to local Cassandra instances, providing automatic failover should a Cassandra instance fail.
- Cassandra can be configured to know which nodes are in the same *rack* and which racks are in the same *data center*. With this knowledge, Cassandra uses replication strategies to balance network bandwidth and recoverability from node-, rack-, and data center-level failures.

Details on installing and configuring Doradus/Cassandra clusters are provided in the **Doradus Administration** document.

3. Spider Database Overview

This section describes usage and unique features of the Spider Database.

3.1 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"},
   ...
}
```

3.2 Spider 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 automatically 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.

3.3 When Spider Works Best

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

- **Fine-grained searching**: Spider provides rich any-field and field-specific searches for terms, phrases, and wildcards. Efficient equality and range searches are provided for all scalar fields, even with large object populations. Indexing can be disabled for stored-only fields. With these features, Spider is ideal for applications that require fine-grained multi-field searching.

- **Rich relationships**: Links support bi-directional relationships between objects in the same or different tables. DQL path expressions make it easy to navigate relationships with filtering, quantifiers, and transitive searching. This makes Spider ideal for applications whose data uses rich relationships that must be easily queryable.

- **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.

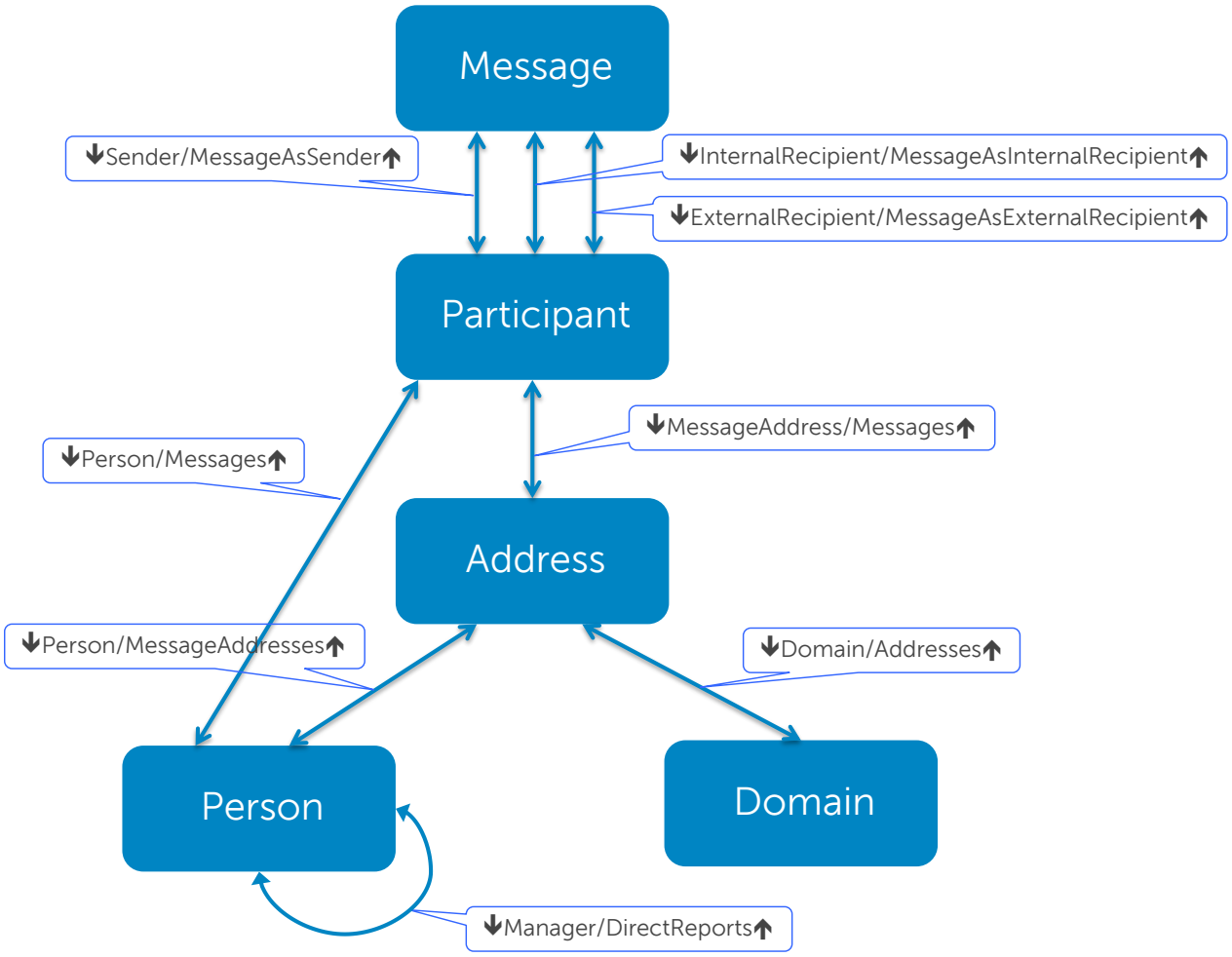
Conversely, Doradus Spider is not a good choice in the following scenarios:

- **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.

3.4 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".

4. Spider Data Model

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

4.1 Motivation

Doradus began as a REST API with Lucene-like searching on top of Cassandra. As a result, its *scalar* data model is similar to Lucene's: named data fields are typed as text, numbers, timestamps, etc. But as Doradus evolved, we wanted a strong linking feature, so we added bi-directional relationships via *links*. We also wanted to leverage the best of the NoSQL world: horizontal scalability, idempotent updates, etc. The result is a unique blend of features from Lucene, graph databases, and NoSQL databases.

4.2 Identifiers

Applications, tables, fields, and other definitions must have a name, called an *identifier*. With Doradus, identifiers must begin with a letter and consist only of letters, numbers, and underscores (_). Letters can be upper- or lower-case, however identifiers are case-sensitive (e.g., `Email`, `Log_Data`, `H1L4`). An exception to the first-letter rule are *alias* names, which must begin with a dollar sign (e.g., `$SalesEmails`).

Pre-defined *system identifiers* begin with an underscore. Because user-defined identifiers cannot begin with an underscore, all system identifiers are reserved. Example system identifiers are `_ID`, `_all`, and `_applications`.

4.3 Application

An application is a *tenant* hosted in a Doradus cluster. An application's name is a unique identifier. An application's data is stored in *tables*, which are isolated from other applications. A cluster can host multiple applications, and each application uses unique URIs to access its data. Example application names are `Email` and `Magellan_1`.

Each application is defined in a *schema*. When the schema is first used to create the application, it is assigned to a specific *storage manager*. Depending on the Doradus server's configuration, multiple storage managers may be available. An application's schema can use core Doradus data model features plus extensions provided by the assigned storage service. Application schemas have the following components:

- **Key**: A user-defined string that acts as a secondary identifier. The key is required to modify the schema or delete the application, hence it acts as an extra safety mechanism.
- **Options**: Application-level options (described below).
- **Tables**: Tables and their fields that the application owns.
- **Task Schedules**: Schedule definitions for application background tasks such as data aging.

The general structure of a schema definition in XML is shown below:

```
<application name="Msgs">
```

```
    <key>MsgsKey</key>
    <options>
       // options
    </options>
    <tables>
       // table definitions
    </tables>
    <schedules>
       // schedule definitions
    </schedules>
</application>
```

The general structure of a schema definition in JSON is shown below:

```
{"Msgs": {
    "key": "MsgsKey",
    "options": {
       // options
    },
    "tables": {
       // table definitions
    },
    "schedules": [
       // schedule definitions
    ]
}}
```

4.3.1 Application Options

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

- **`StorageService`**: 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.

- **`AutoTables`**: Doradus Spider supports an 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>
```

4.4 Table

A table is a named set of objects. Table names are identifiers and must be unique within the same application. Example table names are: `Message`, `LogSnapshot`, and `Security_4xx_Events`.

A table can include the following components:

- Options: Table-level options (described below).
- Fields: Definitions of scalar, link, and group fields that the table uses.
- Aliases: Alias definitions, which are schema-defined expressions that can then be used in queries.

The general structure of a table definition in XML is shown below:

```
<tables>
   <table name="Message">
      <fields>
         // fields
      </fields>
      <aliases>
         // aliases
      </aliases>
   </table>
   ...
</tables>
```

In same structure in JSON is shown below:

```
"tables": {
   "Message": {
      "fields": {
         // fields
      },
      "aliases": {
         // aliases
      }
   },
```

```
    ...
}
```

4.4.1 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>
```

4.4.1.1 Data Aging Options

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.

4.4.1.2 Table Sharding Options

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**.

4.5 Objects

The addressable member of a table is an *object*. Every object has an addressable key called its *ID*, which uniquely identifies the object within its owning table. An object's data is stored within one or more *fields*.

4.6 Object IDs

Every object has a unique, immutable ID, stored in a system-defined field called `_ID`. Object IDs behave as follows:

- IDs are variable length. Objects in the same table can have different ID lengths as long as every value is unique.
- User applications set an object's ID by assigning the `_ID` field when the object is created. It is the application's responsibility to generate unique ID values. If two objects are added with the same ID, they are *merged* into a single object. That is, one "add" is treated as an "update" of an existing object.
- To Doradus, IDs are Unicode text strings. To use binary (non-Unicode) ID values, user applications must convert them to a Unicode string, e.g., using hex or base64 encoding.
- When IDs are included in XML or JSON messages or in URIs, some characters may need to be escaped. For example, the base64 characters '+' and '/' must be converted to `%2B` and `%2F` respectively in URIs. IDs returned by Doradus in messages are always escaped when needed.

4.7 Fields

All fields other than `_ID` are user-defined. Every field has a *type*, which determines the type of values it holds. Field types fall into three categories: *scalar*, *link*, and *group*.

4.7.1 Scalar Fields

Scalar fields store simple data such as numbers or text. A single-valued (SV) scalar field stores a single value per object. Multi-valued (MV) scalar fields, also called scalar *collections*, can store multiple values per object. The scalar field types supported by Doradus are summarized below:

- **Text**: An arbitrary length string of Unicode characters.
- **Boolean**: A logical *true* or *false* value.
- **Integer** and **Long**: A signed 64-bit integer value. These two types are synonyms.
- **Float**: A 32-bit floating-point value.
- **Double**: A 64-bit floating-point value.
- **Timestamp**: A date/time value with millisecond precision. Doradus treats all timestamp values as occurring in the UTC time zone. Timestamp values must be given in the format:

  ```
  yyyy-MM-dd HH:mm:ss.SSS
  ```

  where:

  - `yyyy` is a 4-digit year between 0000 and 9999
  - `MM` is a 1- or 2-digit month between 1 and 12
  - `dd` is a 1- or 2-digit day-of-month between 1 and 31
  - `HH` is a 1- or 2-digit hour between 0 and 23
  - `mm` is a 1- or 2-digit minute between 0 and 59
  - `ss` is a 1- or 2-digit second between 0 and 59

- `SSS` is a 1-to-3 digit millisecond between 0 and 999

Only the year component is required. All other components can be omitted (along with their preceding separator character) in right-to-left order. Omitted time elements default to 0; omitted date elements default to 1. Hence, the following timestamp values are all valid:

```
2011-02-01 08:50:01.123
2011-02-01 08:50:01           // same as 2011-02-01 08:50:01.000
2011-02-01 08:50              // same as 2011-02-01 08:50:00.000
2011-02-01 08                 // same as 2011-02-01 08:00:00.000
2011-02-01                    // same as 2011-02-01 00:00:00.000
2011-02                       // same as 2011-02-01 00:00:00.000
2011                          // same as 2011-01-01 00:00:00.000
```

- **Binary**: An arbitrary length sequence of bytes. A binary field definition includes an *encoding* (`Base64` or `Hex`) that defines how values are encoded when sent to or returned by Doradus.

When type names are used in schema definitions, they are case-insensitive (e.g., `integer` or `INTEGER`).

4.7.1.1 The `Collection` Property

By default, scalar fields are SV. Assigning an SV scalar field for an existing object replaces the existing value, if present. A scalar field can be declared MV by setting its `collection` property to `true`. Example:

```
<field name="Tags" collection="true" type="TEXT"/>
```

MV scalar field values are added and removed individually and treated as a *set*: duplicate values are not stored. This preserves idempotent update semantics: adding the same value twice is a no-op.

4.7.1.2 The `Analyzer` Property

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.

4.7.1.2.1 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.

4.7.1.2.2 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.

4.7.1.2.3 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.

4.7.1.2.4 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.

4.7.1.2.5 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`.

4.7.1.2.6 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`.

4.7.1.2.7 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
```

4.7.2 Link Fields

Link fields are *pointers* that create inter-object relationships. All relationships are bi-directional, therefore every link has an *inverse* link that defines the same relationship from the opposite direction. A link and its inverse link can be in the same table or they can reside in different tables. An example link declaration in XML is shown below:

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

In this example, the link field `MessageAddress` is owned by the `Participant` table and points to the `Address` table, whose inverse link is called `Messages`. The table to which a link points is called the link's *extent*.

Link fields are always MV: the `collection` property, if set, is ignored. A link's values are IDs of objects that belong to its extent table. Relationships are created or deleted by adding IDs to or removing IDs from the link field. Like MV scalar fields, link values are *sets*, hence duplicates are ignored.

Because relationships are bi-directional, when a link is updated, its inverse link is automatically updated at the same time. For example, if an object ID is added to `MessageAddress`, connecting the owning participant object to a specific `Address` object, the `Messages` link for that address object is updated to point back to the same participant.

One side-effect of this referential integrity behavior is that objects can be *implicitly* created: if an object ID is added to `MessageAddress` and the corresponding person doesn't exist, it is created. An implicitly-created object will only have an `_ID` and automatically-updated link field value(s).

If a link's owner and extent are the same table, the relationship is *reflexive*. An example reflexive relationship is the `Manager`/`DirectReports` relationship in the `Person` table:

```
<table name="Person">
   <fields>
      <field name="DirectReports" table="Person" type="LINK" inverse="Manager"/>
      <field name="Manager" table="Person" type="LINK" inverse="DirectReports"/>
      ...
   </fields>
</table>
```

A link can also be its own inverse: such relationships are called *self-reflexive*. For example, we could define *spouse* and *friends* as self-reflexive relationships (though some may argue friendship is not always reciprocal).

4.7.2.1 The `Sharded` Property

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.

4.7.3 Group Fields

A group field is a named field that contains one or more other fields. The contained fields are called *nested* fields and may be scalar, link, or group fields. The group field itself does not hold any values; values are stored by the contained *leaf* scalar and link fields. Note that all field names within a table must be unique, even for those contained within a group field. An example group field is shown below:

```
<table name="Person">
    <fields>                    <!-- fields belonging to Person -->
        <field name="Location">
            <fields>            <!-- fields belong to Location -->
                <field name="Department" type="TEXT"/>
                <field name="Office" type="TEXT"/>
            </fields>
        </field>
        ...
    </fields>
</table>
```

Here, the group field `Location` contains the nested text fields `Department` and `Office`. In a query, the values of both nested fields are returned when the group field `Location` is requested. Below is another example consisting of link fields:

```
<table name="Message">
    <fields>
        <field name="Participants">
            <fields>
                <field name="Sender" type="LINK" table="Participant" inverse="MessageAsSender"/>
                <field name="Recipients">
                    <fields>
                        <field name="ExternalRecipients" type="link" table="Participant"
                            inverse="MessageAsExternalRecipient"/>
                        <field name="InternalRecipients" type="link" table="Participant"
                            inverse="MessageAsInternalRecipient"/>
                    </fields>
                </field>
```

```
            </fields>
         </field>
         ...
      </fields>
   </table>
```

In this example, the group field `Participants` contains a link called `Sender` and a nested group called `Recipients`, which has two additional links `ExternalRecipients` and `InternalRecipients`. The values of all three link fields can be retrieved by requesting the `Participants` in a query.

4.7.4 Dynamically-Added 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.

4.8 Aliases

An alias is a table-specific *derived field*, in which a path expression is assigned to a name that is declared in the schema. In an object query, the alias name can be used in the query or fields parameters. In an aggregate query, the alias name can be used in the query or grouping parameters. Each occurrence of an alias name is replaced by its expression text, analogous to a macro.

Alias names must begin with a '$', must be at least 2 characters in length, and secondary characters can be letters, digits, or underscores ('_'). Alias names must also be unique within the application.

Below is an example alias declaration:

```
<table name="Message">
   <aliases>
      <alias name="$SalesEmails" expression="Sender.Person.WHERE(Department:sales)"/>
   <aliases>
   ...
</table>
```

Here, the alias named `$SalesEmails` is assigned the expression `Sender.Person.WHERE(Department:sales)`. If the following object query is submitted:

```
GET /Msgs/Message/_query?q=$SalesEmails.LastName:powell
```

The alias name `$Sender` it replaced with its expression text, which causes the query to be parsed as:

```
GET /Msgs/Message/_query?q=Sender.Person.WHERE(Department:sales).LastName:powell
```

Because the expression text is expanded and evaluated in the context where it is used, alias declarations are not evaluated at schema definition time. When used in queries, an alias will generate an error if its expansion results in an invalid query.

4.9 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.

4.10 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.

5. Doradus Query Language (DQL)

The Doradus query language (DQL) is used in object and aggregate queries. DQL is analogous to full text languages used by search engines such as Lucene and 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.

This section describes DQL, including extensions supported by Doradus Spider.

5.1 Query Perspective

A DQL instance is a Boolean expression that selects objects from a *perspective* table. Logically, the query expression is evaluated against each perspective object; if the expression evaluates to "true", the query selects the object.

The query expression may include clauses that reference fields from objects that are linked to a perspective object. For example, if the perspective table is `Person`, the clause `Manager.Name:John` is true if a person's `Manager` points to an object whose `Name` field includes the term `John`.

In object queries, requested fields are returned by the query for each selected perspective object. In aggregate queries, selected objects are included in one or more metric computations.

5.2 Clauses

A DQL query is comprised of one or more Boolean expressions called *clauses*. Each clause examines a field that is related to perspective objects. The clause evaluates to true for a given object if the examined field matches the clause's condition. Example clauses are shown below:

```
Name:Smith
FirstName=John
SendDate:[2001 TO "2005-06-31"]
NOT Name:Jo*
ALL(InternalRecipients).ANY(MessageAddress).Domain.Name = "dell.com"
Manager^(3).LastName:Wright
```

These examples show a variety of comparison types: qualified and unqualified fields, *contains* and *equals*, ranges, pattern matching, quantifiers, and transitive searches.

5.3 Clause Negation: NOT

Any clause can be negated by prefixing it with the uppercase keyword NOT. To use "not" as a literal term instead of a keyword, either enclose it in single or double quotes or use non-uppercase. Double negation (`NOT NOT`) and any even number of NOT prefixes cancel out.

5.4 Clause Connectors: AND, OR, and Parentheses

Clauses are *implicitly* connected with "and" semantics by separating them with at least one whitespace character. For example:

```
LastName:Wright Department:Sales
```

This query consists of two clauses and is identical to the following query expression:

```
LastName:Wright AND Department:Sales
```

Clauses can be *explicitly* connected with the uppercase keywords `AND` and `OR`. When a DQL query has a mixture of `AND` and `OR` clauses, the `AND` clauses have higher evaluation precedence. For example:

```
Origin=3 AND Platform=2 OR SendDate='2011-10-01' AND Size=1000
```

This is evaluated as if the following parentheses were used:

```
(Origin=3 AND Platform=2) OR (SendDate='2011-10-01' AND Size=1000)
```

Parentheses can be used to change the evaluation order. For example:

```
Origin=3 AND (Platform=2 OR SendDate='2011-10-01') AND Size=1000
```

5.5 Literals

Most clauses contain a comparison that include a literal. When the comparison uses a predefined field whose type is known, the literal *should* use a format compatible with that type. However, Doradus does not strictly enforce this. For example, if `Size` is declared as an integer, the following clause is allowed:

```
Size=Foo
```

Of course, because `Size` is declared as an integer, it cannot match the value `Foo`, so no objects will be selected by the clause.

The types of literals supported by Doradus are described below.

5.5.1 Booleans

A Boolean literal is the keyword `true` or `false` (case-insensitive).

5.5.2 Numbers

Doradus supports both integer and floating-point numeric literals. An integer literal is all digits, preceded with a minus sign (-) for negative values. Integer literals are considered long (64-bit). Floating-point constants use the same format supported by Java: an optional preceding sign, at least one digit, an optional fractional part, and an optional exponent, which may be signed. Example floating-point literals are shown below:

```
0
-123
3.14
10E12
2.718e-6
```

5.5.3 Timestamps

Timestamp literals have the general form:

```
"YYYY-MM-DD hh:mm:ss.fff"
```

Notes:

- In DQL expressions, timestamp values should be enclosed in single or double quotes.
- All time and date elements except for the year can be omitted from right-to-left if the preceding separator character ('.', ':', or '-') is also eliminated.
- When a timestamp value consists of a year only, the enclosing quotes can be omitted.
- Omitted time components default to 0; omitted date components default to 1.

5.5.4 Terms

Text fields can be evaluated as a set of tokens called *terms*. Depending on the underlying storage service, the terms may be generated and indexed when a text field is stored, or they may be computed dynamically. Generally, terms are alphanumeric character sequences separated by whitespace and/or punctuation.

In search clauses, multiple terms can be used, and each term can contain wildcards. Example search terms are shown below:

```
John
Fo*
a?c
Event_413
```

Note that terms are not quoted. A term is a sequence of characters consisting of letters, digits, or any of these characters:

```
? (question mark)
* (asterisk)
_ (underscore)
@ (at symbol)
# (hash or pound)
- (dash or minus)
```

In a term, the characters `_`, `@`, `#`, and `-` are treated as term *separators*, creating multiple terms that are part of the same literal. For example, the following literals all define two terms, `john` and `smith` – the separator character is not part of either term:

```
john_smith
john@smith
john#smith
john-smith
```

The characters `?` and `*` are single- and multi-character wildcards: `?` matches any single character and `*` matches any sequence of characters.

See the discussion on **Contains Clauses** for more details about using terms in searches.

5.5.5 Strings

Some clauses compare to a full text string instead of a term. When the text string conforms to the syntax of a term, it can be provided unquoted. Example:

```
Name=Smith
```

The value `Smith` is a string because "=" follows the field name; if the field name was followed by ":", `Smith` would be treated as a term. See **Equality Clauses** described later.

When the string consists of multiple terms or contains characters not allowed in terms, the string must be enclosed in single or double quotes. Example:

```
Name="John Smith"
```

Note that to Doradus, object IDs are also strings. Therefore, when used in DQL queries, object ID literals may be provided as a single unquoted term or as a quoted string. Examples:

```
_ID=ISBN1234
Sender='cihSptpZrCM6oXaVQH6dwA=='
```

In a string, the characters `?` and `*` are treated as wildcards as they are in terms. If a string needs a `?` or `*` that must be used literally, they must be escaped. Escaping can also be used for non-printable or other characters. Doradus uses the backslash for escaping, and there are two escape formats:

- `\x`: Where `x` can be one of these characters: `t` (tab character), `b` (backspace), `n` (newline or LF), `r` (carriage return), `f` (form feed), `'` (single quote), `"` (double quote), or `\` (backslash).
- `\uNNNN`: This escape sequence can be used for any Unicode character. `NNNN` must consist of four hex characters 0-F.

For example, the string `"Hello!World"` can also be specified as `"Hello\u0021World"`.

5.6 Null Values

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. This means that `NOT(clause)` will be true if `clause` references a null field.

For Boolean fields, this may be a little unintuitive: assume a Boolean field `IsInternal` is null for a given object. Then:

```
IsInternal=true         // false
IsInternal=false        // false
NOT IsInternal=true     // true
NOT IsInternal=false    // true
```

In other words, a null Boolean field is neither true nor false, so comparison to either value will always be false, and negating such a comparison will always be true.

5.7 IS NULL Clause

A scalar or link field can be tested for nullity by using the clause *field* `IS NULL`. Examples:

```
LastName IS NULL
InternalRecipients IS NULL
Sender.MessageAddress.Person.Manager IS NULL
```

The last example above uses a link path (described later). See the section Quantifiers with IS NULL for examples of how `IS NULL` interacts with quantified link paths.

An `IS NULL` clause can be negated with `NOT` as a clause prefix:

```
NOT LastName IS NULL
NOT InternalRecipients IS NULL
NOT Sender.MessageAddress.Person.Manager IS NULL
```

5.8 Text *Contains* Clauses

Text fields can be queried for values that *contain* specific terms. In contrast, equality clauses (described later) compare the entire field value.

5.8.1 Term Clauses

A *term clause* searches a specific field for one or more terms. To designate it as a *contains* clause, the field name must be followed by a colon. Example:

```
Name:Smith
```

This clause searches perspective objects' `Name` field for the term `Smith` (case-insensitive). To specify multiple terms, enclose them in parentheses. Example:

```
Name:(John Smith)
```

To match this clause, an object's `Name` field must contain both `John` and `Smith`, but they can appear in any order and be separated by other terms.

Be sure to enclose multi-term clauses in parentheses! The following query looks like it searches for `John` and `Smith` in the `Name` field:

```
Name:John Smith      // doesn't do what you think!
```

But, this sequence is actually interpreted as two clauses that are AND-ed together:

```
Name:John AND *:Smith
```

Matching objects must have `John` in the `Name` field and `Smith` in any field.

5.8.2 Phrase Clauses

A *phrase clause* is a *contains* clause that searches for a field for a specific term *sequence*. Its terms are enclosed in single or double quotes. For example:

```
Name:"John Sm*th"
```

This phrase clause searches the `Name` field for the term `John` immediately followed by a term that matches the pattern `Sm*th`. The matching terms may be preceded or followed by other terms, but they must be in the specified order and with no intervening terms. As with term clauses, phrases clauses can use wildcards, and searches are performed without case sensitivity.

5.8.3 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.

5.9 Range Clauses

Range clauses can be used to select a scalar field whose value falls within a specific range. (Range clauses are not allowed on link fields or the `_ID` field.) The math operators `=`, `<`, `<=`, and `>=` are allowed for numeric fields:

```
Size > 10000
LastName <= Q
```

Doradus also allows *bracketed* ranges with both *inclusive* ([ ]) and *exclusive* ({ }) bounds. For example:

```
Size = [1000 TO 50000}
```

This is shorthand for:

```
Size >= 1000 AND Size < 50000
```

Text fields can also use bracketed range clauses:

```
FirstName={Anne TO Fred]
```

This clause selects all objects whose `FirstName` is greater than but not equal `Ann` and less than or equal to `Fred`.

For range clauses, ":" and "=" are identical – both perform an *equals* search. Hence, the previous example is the same as:

```
FirstName:{Anne TO Fred]
```

5.10 Equality Clauses

An equality clause searches a field for a value that matches a literal constant, or, for text fields, a pattern. The equals sign (=) is used to search for an exact field value. The right hand side must be a literal value. Example:

```
Name="John Smith"
```

This searches the `Name` field for objects that exactly match the string "John Smith". The search is case-insensitive, so an object will match if its `Name` field is "`john smith`", "`JOHN SMITH`", etc.

For text fields, wildcards can be used to define patterns. For example:

```
Name="J* Sm?th"
```

This clause matches "`John Smith`", "`Joe Smyth`", etc.

Boolean, numeric and timestamp fields cannot use wildcards. Examples:

```
HasBeenSent=false
Size=1024
ModifiedDate="2012-11-16 17:19:12.134"
```

In a timestamp literal, elements can be omitted from right-to-left, but the clause still compares to an exact value. Omitted time elements default to 0; omitted date elements default to 1. For example:

```
ModifiedDate="2010-08-05 05:40"
```

This clause actually searches for the exact timestamp value `2010-08-05 05:40:00.000`. However, timestamps can be used in range clauses. See also the section Subfield Clauses, which describes clauses that search for timestamp subfields.

Link fields and the `_ID` field can also be used in equal searches by comparing to an object ID. For example:

```
Manager=sqs284
_ID='kUNaqNJ2ymmb07jHY9OPOw=='
```

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

```
NOT Size=1024
```

5.11 IN Clause

A field can be tested for membership in a set of values using the `IN` clause. Examples:

```
Size IN (512,1024,2048,4096,8192,16384,32768,65536)
LastName IN (Jones, Smi*, Vledick)
Manager IN (shf637, dhs729, fjj901)
_ID IN (sjh373,whs873,shc729)
```

As shown, values are enclosed in parentheses and separated by commas. If the comparison field is a text field, values can use wildcards. Link fields and the `_ID` field are compared to object IDs. Literals that are not a simple term must be enclosed in single or double quotes.

The `IN` clause is a shorthand for a series of `OR` clauses. For example, the following clause:

```
LastName IN (Jones, Smi*, Vledick)
```

is the same as the following `OR` clauses:

```
LastName = Jones OR LastName = Smi* OR LastName = Vledick
```

As a synonym for the `IN` keyword, Doradus also allows the syntax *`field`*`=(`*`list`*`)`. For example, the following two clauses are equivalent:

```
LastName IN (Jones, Smi*, Vledick)
LastName=(Jones, Smi*, Vledick)
```

5.12 Timestamp Clauses

This section describes special clauses that can be used with timestamp fields.

5.12.1 Subfield Clauses

Timestamp fields possess date/time *subfields* that can be used in equality clauses. A subfield is accessed using "dot" notation and an upper-case mnemonic. Each subfield is an integer value and can be compared to an integer constant. Only equality (=) comparisons are allowed for subfields. 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)
```

The recognized subfields of a timestamp field and their possible range values are:

- `YEAR`: any integer
- `MONTH`: 1 to 12
- `DAY`: 1 to 31
- `HOUR`: 0 to 23
- `MINUTE`: 0 to 59
- `SECOND`: 0 to 59

Though timestamp fields will store sub-second precision, there are no subfields that allow querying the sub-second portion of specific values.

5.12.2 NOW Function

A timestamp field can be compared to the current time, optionally adjusted by an offset, using the `NOW` function. The `NOW` function dynamically computes a timestamp value, which can be used anywhere a timestamp literal value can be used. The `NOW` function uses the general format:

```
NOW([<timezone> | <GMT offset>] [<unit adjust>])
```

The basic formats supported by the `NOW` function are:

- **`NOW()`**: Without any parameters, `NOW` creates a timestamp equal to the current time in the UTC (GMT) time zone. (Remember that Doradus considers all timestamp fields values as belonging to the UTC time zone.)

- **`NOW(<timezone>)`**: A time zone mnemonic (`PST`) or name (`US/Pacific`) can be passed, which creates a timestamp equal to the current time in the given time zone. The values supported for the <timezone> parameter are those returned by the Java function `java.util.TimeZone.getAvailableIDs()`.

- **`NOW(<GMT offset>)`**: This format creates a timestamp equal to the current in UTC, offset by a specific hour/minute value. The <GMT offset> must use the format:

  ```
  GMT<sign><hours>[:<minutes>]
  ```

where <sign> is a plus ('+') or minus ('-') sign and <hours> is an integer. If provided, <minutes> is an integer preceded by a colon.

- **NOW(<unit adjust>)**: This format creates a timestamp relative to the current UTC time and adjusts a single unit by a specific amount. The <unit adjust> parameter has the format:

  ```
  <sign><amount><unit>
  ```

  where <sign> is a plus ('+') or minus ('-') sign, <amount> is an integer, and <unit> is a singular or plural time/date mnemonic (uppercase). Recognized values (in plural form) are SECONDS, MINUTES, HOURS, DAYS, MONTHS, or YEARS.

- **NOW(<timezone> <unit adjust>)** and **NOW(<GMT offset> <unit adjust>)**: Both the <timezone> and <GMT offset> parameters can be combined with a <unit adjust>. In this case, the current UTC time is first adjusted to the specified timezone or GMT offset and then adjusted by the given unit adjustment.

Below are example NOW functions and the values they generate. For illustrative purposes, assume that the current time on the Doradus server to which the NOW function is submitted is 2013-12-04 01:24:35.986 UTC.

Function	Timestamp Created	Comments
NOW()	2013-12-04 01:24:35.986	Current UTC time (no adjustment)
NOW(PST)	2013-12-03 17:24:35.986	Pacific Standard Time (-8 hours)
NOW(Europe/Moscow)	2013-12-04 05:24:35.986	Moscow Standard Time (+4 hours)
NOW(GMT+3:15)	2013-12-04 04:39:35.986	UTC incremented by 3 hours, 15 minutes
NOW(GMT-2)	2013-12-03 23:24:35.986	UTC decremented by 2 hours
NOW(+1 DAY)	2013-12-05 01:24:35.986	UTC incremented by 1 day
NOW(+1 MONTH)	2014-01-04 01:24:35.986	UTC incremented by 1 month
NOW(GMT-3:00 +1 YEAR)	2014-12-03 22:24:35.986	UTC decremented by 3 hours, incremented by 1 year
NOW(ACT -6 MONTHS)	2013-06-04 11:24:35.986	FMT adjusted to Australian Capitol Territory Time (+10 hours) then decremented by 6 months

(Remember to escape the '+' sign as %2B in URIs since, un-escaped, it is interpreted as a space.)

The value generated by the NOW function can be used wherever a literal timestamp value can appear. Below are some examples:

```
SendDate > NOW(-1 YEAR)
SendDate >= NOW(PST +9 MONTHS)
ReceiptDate = [NOW() TO NOW(+1 YEAR)]
```

```
ReceiptDate = ["2013-01-01" TO NOW(Europe/Moscow)]
```

Because the NOW function computes a timestamp relative to the time the query is executed, successive executions or the same query could create different results. This could also affect the results of paged queries.

5.12.3 PERIOD Function

The PERIOD function generates a timestamp value range, computed relative to the current time. It is a shortcut for commonly-used range clauses that occur close to (or relative to) the current date/time. A timestamp field can be compared to a PERIOD function to see if its value falls within the corresponding range. A timestamp range clause using the PERIOD function uses the following form:

```
field = PERIOD([<timezone>]).<range>
```

With no parameter, the PERIOD function computes a timestamp range relative to a snapshot of the current time in UTC. If a <timezone> parameter is provided, the UTC time is adjusted up or down to the specified time zone. The <timezone> can be an abbreviation (PST) or a name (America/Los_Angeles). The allowable values for a <timezone> abbreviation or name are those recognized by the Java function java.util.TimeZone.getAvailableIDs().

The <range> parameter is a mnemonic that chooses how the range is computed relative to the "now" snapshot. There are two types of range mnemonics: THIS mnemonics and LAST mnemonics. All mnemonics must be uppercase.

THIS mnemonics compute a range *around* the current time, that is a range that includes the current time. The recognized THIS mnemonics are:

```
THISMINUTE
THISHOUR
TODAY
THISWEEK
THISMONTH
THISYEAR
```

Note that TODAY is used as the mnemonic for "this day". THIS mnemonics use no additional parameters. They define a timeframe (minute, hour, day, week, month, or year) that includes the current time. The timeframe is inclusive of the timeframe start but exclusive of the timeframe end. For example, if the current time is 2013-12-17 12:40:13, the function PERIOD.THISHOUR defines the range:

```
["2013-12-17 12:00:00" TO "2013-12-17 13:00:00"}
```

Note the exclusive bracket (}) on the right hand side.

LAST mnemonics compute a range that *ends* at the current time. That is, they choose a timeframe that leads up to "now", going back an exact number of units. The recognized LAST mnemonics are:

```
LASTMINUTE
LASTHOUR
```

```
LASTDAY
LASTWEEK
LASTMONTH
LASTYEAR
```

By default, the LAST mnemonics reach back one unit: 1 minute, 1 hour, etc. Optionally, they allow an integral parameter that extends the timeframe back a whole number of units. For example, LASTMINUTE(2) means "within the last 2 minutes", LASTMONTH(3) means "within the last 3 months", etc. LAST periods are inclusive at both ends of the range. For example, if the current time is 2013-12-17 12:40:13, the function PERIOD.LASTHOUR defines the range:

```
["2013-12-17 11:40:13" TO "2013-12-17 12:40:13"]
```

Example timestamp clauses using the PERIOD function are shown below:

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

Example ranges generated by each mnemonic are given below. For illustrative purposes, assume that the current time on the Doradus server to which the PERIOD function is submitted is 2013-12-04 01:24:35 UTC. If a <timezone> parameter is included, the "now" value would first be adjusted to the corresponding timezone, and the range would be computed relative to the adjusted value.

<range> Mnemonic	Timestamp Range	Comments
THISMINUTE	["2013-12-04 01:24:00" TO "2013-12-04 01:25:00"}	Same year, month, day, hour, and minute as now.
LASTMINUTE	["2013-12-04 01:23:35" TO "2013-12-04 01:24:35"]	Within the last minute (60 seconds).
THISHOUR	["2013-12-04 01:00:00" TO "2013-12-04 02:00:00"}	Same year, month, day, and hour as now.
LASTHOUR	["2013-12-04 00:24:35" TO "2013-12-04 01:24:35"]	Within the last hour (60 minutes).
TODAY	["2013-12-04 00:00:00" TO "2013-12-05 00:00:00"}	Same year, month, and day as now.
LASTDAY	["2013-12-03 01:24:35" TO "2013-12-04 01:24:35"]	Within the last day (24 hours).
THISWEEK	["2013-12-02 00:00:00" TO "2013-12-09 00:00:00"}	Same week as now (based on ISO 8601).
LASTWEEK	["2013-11-27 01:24:35" TO "2013-12-04 01:24:35"]	Within the last 7 days.
THISMONTH	["2013-12-00 00:00:00" TO "2014-01-01 00:00:00"}	Same year and month as now.
LASTMONTH	["2013-11-04 01:24:35" TO "2013-12-04 01:24:35"]	Within the last calendar month.
THISYEAR	["2013-01-01 00:00:00" TO "2014-01-01 00:00:00"}	Same year as now.

LASTYEAR	["2012-12-04 01:24:35" TO "2013-12-04 01:24:35"]	Within the last year.

5.13 Link Clauses

Link fields can be compared for a single value using an object ID as the value. Example:

```
Manager=def413
```

A link can also be tested for membership in a set of values using the IN operator:

```
DirectReports IN (zyxw098, ghj780)
DirectReports = (zyxw098, ghj780)
```

The two examples above are equivalent. Inequalities and range functions are not allowed for link fields.

Links can also be used in *path expressions*, which are described in the following sections.

5.13.1 Link Path Clauses

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

```
field1.field2...fieldN
```

The field names used in a link path must follow these rules:

- The first field (`field1`) must be a link field defined in the query's perspective table.
- All fields in the path must be link fields except for the last field, which can be a link or scalar field.
- Each secondary field (`field2` through `fieldN`) must be a field that belongs to the extent table of the prior (immediate left) link field. That is, `field2` must be a field owned by the extent table of `field1`, `field3` must be owned by extent table of `field2`, and so forth.
- If the second-to-last field (`fieldN-1`) is a timestamp field, the last field can be a timestamp subfield (`YEAR`, `HOUR`, etc.)

The right-most field in the link path is the comparison field. The type of the target value(s) in the clause must match the type of the comparison field. For example, if the comparison field is an integer, the target values must be integers; if the comparison field is a link, the target values must be object IDs; etc. Implicit quantification occurs for every field in the path. Consider these examples:

```
Manager.Name : Fred
Sender.MessageAddress.Domain.Name = 'hotels.com'
DirectReports.DirectReports.FirstName = [Fred TO Wilma}
```

In order, these examples are interpreted as follows:

- A perspective object (a `Person`) is selected if at least one of its manager's name contains the term `Fred`.

- A perspective object (a `Message`) is selected if it has at lease one sender with at least one address with at lease one domain named `hotels.com`.
- A perspective object (a `Person`) is selected if at least one direct report has at least one second-level direct report whose `FirstName` is >= `Fred` but < `Wilma`.

5.13.2 WHERE Filter

Sometimes we need multiple selection clauses for the objects in a link path, but we need the clauses to be "bound" to the same instances. To illustrate this concept, consider this query: Suppose we want to find messages where an internal recipient is within the R&D department and in an office in Kanata. We might try to write the query like this:

```
// Doesn't do what we want
GET /Msgs/Message/_query?q=InternalRecipients.Person.Department='R&D' AND
    InternalRecipients.Person.Office='Kanata'
```

But the problem is that the two `InternalRecipients.Person` paths are separately quantified with `ANY`, so the query will return messages that have at least one internal recipient in R&D (but not necessarily in Kanata) while another internal recipient is in Kanata (but not necessarily in R&D). It might be tempting to quantify the two `InternalRecipient.Person` paths with `ALL`:

```
// Still not what we want
GET /Msgs/Message/_query?q=ALL(InternalRecipients.Person).Department='R&D' AND
    ALL(InternalRecipients.Person).Office='Kanata'
```

Now the problem is that the query won't select messages that have one or more internal recipients who are not in R&D/Kanata, even though there might be another recipient who is.

What we really need is for the two `InternalRecipient.Person` clauses to be *bound*, meaning they apply to the same instances and are not separately quantified.

The `WHERE` filter can be used for this scenario, as shown below:

```
GET /Msgs/Message/_query?q=InternalRecipients.Person.WHERE(Department='R&D' AND Office='Kanata')
```

The `WHERE` function is appended to the portion of the link path for which we need multiple selection clauses, and the clauses are specified as a parameter. The field names referenced in the `WHERE` expression are qualified to the object to the left of the `WHERE` clause. In the example above, `Department` and `Office` are qualified to `Person`, so they must be fields belonging to those objects. Note that implicit quantification takes places in the example above, hence it is identical to the following query:

```
GET /Msgs/Message/_query?q=ANY(InternalRecipients.Person).WHERE(Department='R&D' AND
    Office='Kanata')
```

5.13.3 Outer WHERE Filter

The `WHERE` filter usually follows a link name to select *related* objects connected via that link. However, a link path can begin when a `WHERE` filter, in which case it selects *perspective* objects. For example:

```
GET /Msgs/Person/_query?q=WHERE(Department:sales)
```

This `WHERE` filter selects perspective objects, in this case `Person`. The query above is identical to the following:

```
GET /Msgs/Person/_query?q=Department:sales
```

The *scope* of an outer `WHERE` filter remains at the perspective object. Hence, multiple, outer `WHERE` filters can be chained together as in the following example:

```
GET /Msgs/Person/_query?q=WHERE(Department:sales).WHERE(Office:aliso)
```

The outer `WHERE` filters are AND-ed together, so the example above is identical to this query:

```
GET /Msgs/Person/_query?q=Department:sales AND Office:aliso
```

Outer `WHERE` filters allow aliases to be defined as link paths that can be used in multiple contexts. For example, assume the following alias is defined:

```
"Person": {
   aliases: {
      "$SalesPeople": {"expression": "WHERE(Department:sales)"}
   }
   ...
}
```

The alias `$SalesPeople` can be used as a selection expression or link filter whenever the expression scope is `Person`. For example, the alias can be used in the following queries:

```
GET /Msgs/Person/_query?q=$SalesPeople
GET /Msgs/Person/_query?q=$SalesPeople.WHERE(Office:aliso)
GET /Msgs/Message/_query?q=Sender.Person.$SalesPeople
```

In the first two cases, the aliases expression `WHERE(Department:sales)` filters perspective `Person` objects. In the third case, the expression filters `Person` objects connected to a perspective `Message` object via `Sender.Person`.

5.14 Quantifier Functions

A quantifier clause tests a set of values that are related to a particular perspective object. For the clause to be true, the values in the set must satisfy the clause's condition *quantitatively*, that is, in the right quantity.

5.14.1 Overview: ANY, ALL, and NONE

When a comparison field is multi-valued with respect to a query's perspective table, it is possible that all, some, or none of its values related to a given perspective object will match the target value or range. By default, sets are implicitly compared using "any" quantification. However, the explicit quantifiers `ANY`, `ALL`, and `NONE` can be used. Here is how these quantifiers work:

- Every clause can be thought of as having the general form <field path> = {target}, which means the values produced by the <field path> must *match* the values in the {target} set. The {target} set is defined by the comparison operator and values specified in the clause (e.g., `Size > 10`, `Size = [1000 TO 10000]`). How the link path *matches* the {target} set depends on how the <field path> is quantified.

- When a field path has no explicit quantification, such as `A.B.C`, then the path is implicitly quantified with "any". This means that at least one value in the set {A.B.C} must match the target value set. Logically, the set is constructed by gathering all C's for all B's for all A's into a single set; if the intersection between this set and the {target} set is not empty, the quantified expression is true.

- The explicitly quantified link path `ANY(A.B.C)` is identical to the implicitly quantified link path `A.B.C`.

- If a set implicitly or explicitly quantified with `ANY` is empty, it does not match the {target} set. Hence, if there are no A values, or no A's have a B value, or no B's have a C value, the set is empty and the clause cannot match the {target} set.

- The explicit quantifier `ALL` requires that the <field path> is not an empty set and that every member is contained in the {target} set. For `ALL(A.B.C)`, some A's might not have a B value and some B's might not have a C value, but that's OK – as long as the set {A.B.C} is not be empty and every C value in the set matches the {target} set, the clause is true.

- The explicit quantifier `NONE` requires that no member of the <field path> is contained in the {target} set. Unlike `ANY` and `ALL`, this means `NONE` matches the {target} set if the <field path> set is empty. Otherwise, no member of the set {A.B.C} must match a {target} set value for the clause to be true. Semantically, `NONE` is the same as `NOT ANY`.

- A <field path> can use multiple quantifiers, up to one per field. For example `ALL(A).ANY(B).NONE(C)` can be interpreted as "No C's for any B for every A can match a target value". Put another way, for the quantified <field path> to be true for a perspective object P, all of P's A values must have at least one B value for which none of its C values match the {target} set. The same existence criteria applies as described above: if a given B has no C values, `NONE(C)` is true for that B; if a given A has no B values, `ANY(B)` is false for that A; if a given object P has no A values, `ALL(A)` is false for that P.

- Implicit "any" quantification is used for any link "sub-path" this is not explicitly quantified. For example, `ALL(A).B.C` is the same as `ALL(A).ANY(B.C)`. Similarly, `A.B.NONE(C)` is the same as `ANY(A.B).NONE(C)`.

- Note that `ALL(A.B)` is not the same as `ALL(A).ALL(B)`. This is because in the first case, we collect the set of all B's for all A's and test the set – if a given A has no B's, that's OK as long as the set {A.B} is not empty and every value matches the {target} set. However, in `ALL(A).ALL(B)`, if a given A has no B values, `ALL(A)` fails for that A, therefore the clause is false for the corresponding perspective object.

- However, `ANY(A.B)` is effectively the same as `ANY(A).ANY(B)` because in both cases we only need one A to have one B that matches the {target} set. Similarly, `NONE(A.B)` is effectively the same as `NONE(A).NONE(B)` because `NONE(X)` is the same as `NOT ANY(X)`.

- Nested quantifiers are not allowed (e.g., `ANY(A.ALL(B))`).

The following sections look more closely at explicit quantifiers in certain instances.

5.14.2 Quantifiers on MV Scalar Fields

Explicit quantifiers can be used with a single MV scalar as shown in these examples:

```
ANY(Tags) = Confidential
ALL(Tags) : (Priority, Internal)
NONE(Tags) = "Do Not Forward"
```

In the first example, the `ANY` quantifier acts the same as if no explicit quantifier was given. That is, an object is selected if *at least one* `Tags` value is `Confidential`. In the second example, *all* of an object's `Tags` values must have one of the terms in the set {`Priority`, `Internal`} in order to be selected. In the last example, *none* of the object's `Tags` can equal the value `"Do Not Forward"` (case-insensitive). The `NONE` quantifier is true if the quantified field is null.

5.14.3 Quantifiers on SV Scalar Fields

Explicit quantification is allowed on SV scalars. An SV scalar is treated as a set of zero or one value, otherwise it is treated the same as an MV scalar. Strictly speaking, quantifiers are not needed on SV scalars because simple comparisons produce the same results. For example:

```
ANY(Name) = Fred        // same as Name = Fred
ALL(Name) = Fred        // same as Name = Fred
NONE(Name) = Fred       // same as NOT Name = Fred
```

5.14.4 Quantifiers on Link Fields

When a clause's comparison field is a single link field, explicit quantifiers have similar semantics as with MV scalars. Examples:

```
ANY(Manager) = ABC
ALL(DirectReports) = (DEF, GHI) // same as ALL(DirectReports) IN (DEF, GHI)
NONE(MessageAddresses) = XYZ
```

The first case is the same as implicit quantification: an object is selected if it has at least one `Manager` whose object ID is `ABC`. In the second case, the object is selected all `DirectReports` values point to either of the objects with IDs `DEF` or `GHI`. In the third example, the object must not have any `MessageAddresses` values equal to `XYZ`.

5.14.5 Quantifiers with IS NULL

Special semantics are applied when a link path is used with the `IS NULL` clause. When quantified with `ANY`, a link path is null if the resulting value set is empty. In the following example, each link is implicitly qualified with `ANY`:

```
InternalRecipients.Person.LastName IS NULL
```

This clause is true for a perspective object if:

- `InternalRecipients` is null, or
- `Person` is null for every `InternalRecipients` object, or

- `LastName` is null for ever `Person` object.

In other words, if at least one `InternalRecipients.Person.LastName` exists, `IS NULL` is false. Since `ANY` is associative, any portion of the link path can be explicitly quantified and the result is the same.

If a link path used with `IS NULL` is quantified with `ALL`, the clause is true only if (1) the quantified portion of the path is not empty but (2) the remainder of the path produces an empty set. For example:

```
ALL(InternalRecipients).Person.LastName IS NULL
```

This clause is true if `InternalRecipients` is not null but `Person.LastName` is null for every `InternalRecipients` value. Either `Person` can be null or a `Person`'s `LastName` can be null to satisfy the second condition. Alternatively, consider this `ALL` quantification:

```
ALL(InternalRecipients.Person).LastName IS NULL
```

In this case, `InternalRecipients` cannot be null, at least one `InternalRecipients` must have a `Person`, but no `Person` objects have a `LastName`. Essentially, `ALL` adds an existential requirement to the quantified portion of the link path.

If the quantifier `NONE` is used, the quantified portion of the link path can be empty. For example:

```
NONE(InternalRecipients).Person.LastName IS NULL
```

This clause is true for an object if either `InternalRecipients` is null or if `LastName` is not null for every `InternalRecipients.Person`.

When a quantified `IS NULL` clause is negated, it selects the opposite objects than without negation. This means that all objects selected by the following clause:

```
Q(X).Y IS NULL
```

are selected by the clause:

```
NOT Q(X).Y IS NULL
```

where `Q` is a quantifier and `X` and `Y` are field paths.

5.14.6 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`.

5.15 Transitive Function

The transitive function causes a *reflexive* link to be traversed recursively, creating a set of objects for evaluation. Consider this clause:

```
DirectReports.Name = Fred
```

This clause selects people that have at least one `DirectReports` value whose `Name` is `Fred`. Each clause considers the immediate `DirectReports` of each object.

We can change the search to *transitively* follow `DirectReports` links by adding the transitive operator '^' after the reflexive link's name:

```
DirectReports^.Name = Fred
```

This selects people with a direct report named `Fred` or a direct report that has a second-level direct report named `Fred`, recursively down the org chart. The transitive operator '^' expands the set of objects evaluated in the place where the reflexive link (`DirectReports`) occurs recursively. The recursion stops when either:

- A cycle is detected: If an object is found that was already visited in evaluating the `DirectReports^` expression, that object is not evaluated again.
- A null is detected: The recursion stops at the "edges" of the graph, in this case someone with no direct reports.

In some cases, we know that a reflexive graph is not very deep for any given object, so we can let the recursion continue until the leaf nodes are reached. But in some cases, the graph may be arbitrarily large and we need to restrict the number of levels searched. In this case, a *limit* can be passed to the transitive operator in parentheses:

```
DirectReports^(5).LastName = Smith
```

This clause selects people with any direct report up to 5 levels deep whose last name is `Smith`.

The transitive function can be combined with quantifiers as shown below:

```
ALL(InternalRecipient.Person).Manager^.Name : Fred
```

This clause selects messages whose every internal recipient has a superior (manager, manager's manager, etc.) named `Fred`.

5.16 COUNT(<link path>)

The `COUNT` function can be used on a link field or a field path ending with a link. It produces a count of the link values relative to each perspective object; the count can then be used in a comparison clause. For example:

```
COUNT(DirectReports) > 0
```

This clause returns true for a perspective object if it has at least one `DirectReports` value.

When a link path is used, the number of *leaf* link values are counted. For example:

```
COUNT(InternalRecipients.MessageAddress.Person) > 5
```

This clause returns true if the total of all `InternalRecipients.MessageAddress.Person` values is > 5. Some `InternalRecipients` may not have a `MessageAddress` value, and some `MessageAddress` objects may not have a `Person` value.

The link values to be included in a `COUNT` function can be filtered by using the `WHERE` function. For example:

```
COUNT(DirectReports.WHERE(LastName=Smith)) > 0
```

This clause returns true for objects that have at least one direct report whose last name is `Smith`. Note that the `WHERE` expression is specified immediately after the link name to be filtered. The parameter to the `WHERE`

function is an expression relative to the link field. In this example, `LastName` must be a scalar field belong to `DirectReports`' extent table.

The parameter to the `WHERE` function can be a link path. For example:

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

This clause returns true for messages that have at least 5 internal recipients whose last name is Smith.

6. Spider Object Queries

An *object query* is a DQL query that selects and returns selected fields for objects in the perspective table. This section describes object query features including extensions supported by the Spider storage service.

6.1 Query Parameters

An object query allows the following parameters:

- **Query** (required): A DQL query expression that selects objects in the perspective table.
- **Page size** (optional): Limits the number of objects returned. If absent, the page size defaults to `search_default_page_size` 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.
- **Fields** (optional): A comma-separated list of fields to return for each object. Several formats and options are allowed for this parameter as described later in the section Fields Parameter. By default, all scalar fields for selected objects are returned.
- **Order** (optional): Orders the objects returned by 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.
- **Skip** (optional): Causes the specified number of objects to be skipped. Without this parameter, the first page of objects is returned.

How query parameters are passed depends on the REST command. Spider supports object queries submitted in two different ways:

- **URI command**: All query parameters are passed in the URI of the command.
- **Entity command**: All query parameters are passed in an input entity (XML or JSON document) submitted with the command.

REST commands and how query parameters are passed are described in the section Spider REST Commands. The examples in this section use the URI command format.

6.2 Fields Parameter

When the **fields** parameter is omitted, all scalar fields of selected objects are returned. When it is provided, there are many options for selecting fields to be returned. These are described below.

6.2.1 Basic Formats

The **fields** parameter has the following general formats:

- `*`: Using the value "*" (without quotes) is a synonym for omitting the fields parameter. All scalar fields are returned for each object.
- `_all`: This keyword requests all scalar fields of each perspective object and the scalar fields of each first-level object connected via a link value.
- `_local`: This keyword requests all scalar field values and the `_ID` value of all link fields for each object.
- `<scalar field>`: When a scalar field is requested, its value (SV) or values (MV), if any, are returned.
- `<link field>`: When a link field is requested, the _ID of each linked object is returned.
- `<group field>`: When a group field is requested, all *leaf* scalar and link fields within the group and its nested groups, recursively, are returned.
- `f1,f2,f3`: The fields parameter can be a comma-separated list of field names. This example is a simple list of names: only fields f1, f2, and f3 are returned. Each field can be a scalar, link, or group field.
- `f1,f2.f3,f2.f4.f5`: This example uses *dotted notation* to define link paths. This example requests three fields: `f1`, which belongs to the perspective object; `f3`, which is connected to the perspective object via link `f2`; and `f5`, which is related via links `f2` to `f4`. When dotted notation is used, each field to the left of a dot must be a link belonging to the object on its left.
- `f=f1,f2(f3,f4(f5))`: This is the same example as above using *parenthetical qualification* instead of dotted notation. Parenthetical and dotted notations can be mixed within a single fields parameter list. Example: `f=Name,Manager(Name,Manager.Name)`.
- `f1[s1],f2[s2].f3[s3],f2.f4[s4].f5`: This is a dotted notation example where *size limits* are placed on specific link fields. If `f1` is a link field, the number of `f1` values returned for each object will be no more than `s1`, which must be a number. Similarly, link `f2` is limited to `s2` values per object; `f3` is limited to `s3` values for each `f2` value; and `f4` is limited to `s4` values per `f2` value. Since it has no bracketed parameter, field `f5` is not limited in the number of values returned. Link size limits are explained further later.
- `f=f1[s1],f2[s2](f3[s3],f4[s4](f5))`: This is the same *size limits* example as above using parenthetical qualification instead of dotted notation. Link size limits are explained further later.

6.2.2 WHERE Filters

When link paths are used in the **fields** parameter, values can be filtered using a `WHERE` expression. For example, consider this URI query:

```
.../_query?q=Size>1000&f=InternalRecipients.WHERE(Person.Department:support)
```

This returns only InternalRecipients whose `Person.Department` contains the term "support".

The WHERE clause does not have to be applied to the terminal field in the link path. For example:

```
.../_query?q=Size>1000&f=InternalRecipients.Person.WHERE(Department:support).Office
```

This returns `InternalRecipients.Person.Office` values for each message but only for `InternalRecipients.Person` objects whose Department contains the term "support".

6.2.3 Size Limits on Link Fields

When the **fields** parameter includes link fields, the maximum number of values returned for each link can be controlled individually. As an example, consider the following link path:

```
A -> B -> (C, D)
```

That is, A is a link from the perspective object, B is a link from A's extent, and B's extent has two link fields C and D. If the query includes a query-level **page size** parameter, it controls the maximum number of objects returned, but not the number of link field values returned. For example, the following queries are all identical, showing different forms of dotted and parenthetical qualification currently allowed for the f parameter:

```
.../_query?f=A(B(C,D)))&s=10&q=...
.../_query?f=A.B(C,D)&s=10&q=...
.../_query?f=A.B.C,A.B.D&s=10&q=...
```

In these queries, the maximum number of objects returned is 10, but the number of A, B, C, or D values returned for any object is unlimited.

To limit the maximum number of values returned for a link field, a size limit can be given in square brackets immediately after the link field name. Suppose we want to limit the number of A values returned to 5. This can be done with either of the following queries:

```
.../_query?f=A[5](B(C,D)))&s=10&q=...
.../_query?f=A[5].B(C,D))&s=10&q=...
```

In these queries, the maximum objects returned is 10, the number of values returned for B, C, and D is unlimited, and the maximum number of A values returned for each object is 5.

To control the maximum values of both A and B, either of the following syntaxes can be used:

```
.../_query?f=A[5](B[4](C,D))&s=10&q=...    // alternative #1
.../_query?f=A[5].B[4](C,D)&s=10&q=...     // alternative #2
```

These two syntax variations are identical, limiting A to 5 values for each perspective object and B to 4 values for each A value. The maximum object limits is still 10, and the field value limits for C and D is unlimited.

To limit the number of values for C and D (but not A or B), we can use any of the following syntax variations:

```
.../_query?f=A(B(C[4],D[3]))&s=10&q=...    // alternative #1
.../_query?f=A.B(C[4],D[3])&s=10&q=...     // alternative #2
```

```
.../_query?f=A.B.C[4],A.B.D[3]&s=10&q=... // alternative #3
```

When a WHERE filter is used (see previous section), a size limit for the same field should be provided after the WHERE clause. The following examples show the same query with alternate syntax variations:

```
.../_query?f=A.WHERE(foo=bar)[5](B.WHERE(foogle=true)(C[4],D[3]))&s=10&q=...
.../_query?f=A.WHERE(foo=bar)[5].B.WHERE(foogle=true)(C[4],D[3])&s=10&q=...
```

In this example, link A is filtered and limited to 5 values maximum; B is filtered but has no value limit; C is not filtered but limited to 4 values; and D is also not filtered but limited to 3 values.

The WHERE clause is always qualified with a dot, whereas field names can be qualified with a dot or within parentheses. Placing the size limit after the WHERE clause helps to signify that field values are first filtered by the WHERE condition; the filtered set is then limited by the size limit. Also as shown, parenthetical qualification is preferable when multiple extended fields are listed after a link that uses a WHERE filter. (Using purely dotted notation, the WHERE condition would have to be repeated.)

Keeping with current conventions, an explicit size value of zero means “unlimited”. So, for example:

```
.../_query?f=A[5](B(C,D[3]))&s=0&q=...
```

This query places no limits on the number of objects returned as well as the number of B values returned for each A, or the number of C values returned for each B. But a maximum of 5 A values are returned for each object, and a maximum of 3 D values are returned for each B.

Note that size limits only apply to link fields: when an MV scalar field is requested, all values are returned.

6.3 Query Results

An object query always returns an output entity even if there are no objects matching the query request. The outer element is results, which contains a single docs element, which contains one doc element for each object that matched the query expression. Examples for various field types are shown below.

6.3.1 Empty Results

If the query returns no results, the docs element is empty. In XML:

```
<results>
   <docs/>
</results>
```

In JSON:

```
{"results": {
   "docs": []
}}
```

6.3.2 SV Scalar Fields

This object query requests all scalar fields of Person objects whose LastName is Garn:

```
GET /Msgs/Person/_query?q=LastName:Garn&f=*
```

In XML, the result looks like this:

```
<results>
   <docs>
      <doc>
         <field name="Department">Field Sales</field>
         <field name="FirstName">Chris</field>
         <field name="LastName">Garn</field>
         <field name="Name">Chris Garn</field>
         <field name="Office">Aliso Viejo 5</field>
         <field name="_ID">07Z094KNjmEsqMoV/yNI0g==</field>
      </doc>
      <doc>
         <field name="Department">Sales Operations</field>
         <field name="FirstName">Jim</field>
         <field name="LastName">Garn</field>
         <field name="Name">Jim Garn</field>
         <field name="Office">Aliso Viejo 5</field>
         <field name="_ID">kUNaqNJ2ymmb07jHY9OPOw==</field>
      </doc>
      <doc>
         <field name="Department">Admin</field>
         <field name="FirstName">Doug</field>
         <field name="LastName">Garn</field>
         <field name="Name">Doug Garn</field>
         <field name="Office">Aliso Viejo 5</field>
         <field name="_ID">m1yYabbtytmjw+e8OCz1dg==</field>
      </doc>
   </docs>
</results>
```

In JSON:

```
{"results": {
   "docs": [
      {"doc": {
         "Department": "Field Sales",
         "FirstName": "Chris",
         "LastName": "Garn",
         "Name": "Chris Garn",
         "Office": "Aliso Viejo 5",
         "_ID": "07Z094KNjmEsqMoV/yNI0g=="
      }},
      {"doc": {
         "Department": "Sales Operations",
         "FirstName": "Jim",
         "LastName": "Garn",
         "Name": "Jim Garn",
         "Office": "Aliso Viejo 5",
```

```
            "_ID": "kUNaqNJ2ymmb07jHY9OPOw=="
        }},
        {"doc": {
            "Department": "Admin",
            "FirstName": "Doug",
            "LastName": "Garn",
            "Name": "Doug Garn",
            "Office": "Aliso Viejo 5",
            "_ID": "m1yYabbtytmjw+e8OCz1dg=="
        }}
    ]
}}
```

The `_ID` field of each object is always included. SV scalar fields are returned only if they have values. If a group contains any leaf fields with values, they are returned at the outer (`doc`) level: the group field is not included.

When timestamp fields are returned, the fractional component of a value is suppressed when it is zero. For example:

```
2012-01-06 19:59:51
```

This value means that the seconds component is a whole value (51). If a seconds component has a fractional value, it is displayed with 3 digits to the right of the decimal place. Example:

```
2012-01-06 19:59:51.385
```

6.3.3 MV Scalar Fields

The following object query requests the MV scalar field `Tags`:

```
GET /Msgs/Message/_query?q=*&f=Tags
```

A typical result is shown below:

```
<results>
    <docs>
        <doc>
            <field name="Tags">
                <value>AfterHours</value>
            </field>
            <field name="_ID">+/pz/q4Jf8Rc2HK9Cg08TA==</field>
        </doc>
        <doc>
            <field name="Tags">
                <value>Customer</value>
                <value>AfterHours</value>
            </field>
            <field name="_ID">+/wqUBY1WsGtb7zjpKYf7w==</field>
        </doc>
        <doc>
```

```
            <field name="Tags"/>
            <field name="_ID">+1ZQASSaJei0HoGz6GdINA==</field>
        </doc>
    </docs>
</results>
```

The same request in JSON is shown below:

```
{"results": {
    "docs": [
        {"doc": {
            "Tags": ["AfterHours"],
            "_ID": "+/pz/q4Jf8Rc2HK9Cg08TA=="
        }},
        {"doc": {
            "Tags": ["Customer","AfterHours"],
            "_ID": "+/wqUBY1WsGtb7zjpKYf7w=="
        }},
        {"doc": {
            "Tags": [],
            "_ID": "+1ZQASSaJei0HoGz6GdINA=="
        }}
    ],
}}
```

As shown, all values of the `Tags` field are returned, and an element is included even when it is null, as it is for the third object.

6.3.4 Link Fields

When a query has no **fields** parameter or explicitly requests "*", only scalar fields of perspective objects are returned. When the fields parameter includes a link field, by default only the `_ID` field of each linked object is returned. If a link field is requested that has no values, an empty list is returned. For example, consider this object query:

```
GET /Msgs/Person/_query?q=LastName=Powell&f=Manager,DirectReports
```

This query searches for people whose `LastName` is `Powell` and requests the `Manager` and `DirectReports` links. An example result in XML:

```
<results>
    <docs>
        <doc>
            <field name="_ID">gfNqhYF7LgBAtKTdIx3BKw==</field>
            <field name="DirectReports">
                <doc>
                    <field name="_ID">mKjYJmmLPoTVxJu2xdFmUg==</field>
                </doc>
            </field>
            <field name="Manager">
                <doc>
```

```
                <field name="_ID">nLOCpa7aH/Y3zDrnMqG6Fw==</field>
            </doc>
        </field>
    </doc>
    <doc>
        <field name="_ID">sHUm0PEKu3gQDDNIHHWv1g==</field>
        <field name="DirectReports"/>
        <field name="Manager">
            <doc>
                <field name="_ID">tkSQlrRqaeHsGvRU65g9HQ==</field>
            </doc>
        </field>
    </doc>
  </docs>
</results>
```

In JSON:

```
{"results": {
    "docs": [
        {"doc": {
            "_ID": "gfNqhYF7LgBAtKTdIx3BKw==",
            "DirectReports": [
                {"doc": {
                    "_ID": "mKjYJmmLPoTVxJu2xdFmUg=="
                }}
            ],
            "Manager": [
                {"doc": {
                    "_ID": "nLOCpa7aH/Y3zDrnMqG6Fw=="
                }}
            ]
        }},
        {"doc": {
            "_ID": "sHUm0PEKu3gQDDNIHHWv1g==",
            "DirectReports": [],
            "Manager": [
                {"doc": {
                    "_ID": "tkSQlrRqaeHsGvRU65g9HQ=="
                }}
            ]
        }}
    ]
}}
```

As shown, requested link fields are returned even if they have no values. By default, only the `_ID` values of linked objects are included.

7. Spider Aggregate Queries

Aggregate queries perform metric calculations across objects selected from the perspective table. Compared to an object query, which returns fields for selected objects, an aggregate query only returns the metric calculations. This section describes aggregate query parameters, commands, and result formats for various grouping options.

7.1 Aggregate Parameters Overview

Aggregate queries use the following parameters:

- **Metric** (required): Defines one or more functions to calculate for selected objects. Metric functions such as `COUNT`, `SUM`, and `MAX` are supported. Each function is computed across selected objects, optionally subdivided into groups as defined by the grouping parameter. See the section Metric Parameter for details.
- **Query** (optional): A DQL query expression that selects objects in the perspective table. When this parameter is omitted, all objects in the table are included in metric computations.
- **Grouping** (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. A wide range of grouping expressions are supported as described in the section Grouping Parameter.

How parameters are passed depends on the REST command. Spider supports two commands for submitting aggregate queries:

- **URI command**: All parameters are passed in the URI of the command.
- **Entity command**: All parameters are passed in an entity (XML or JSON document) submitted with the command.

Details of each command are described in the section Spider REST Commands. In the examples used in this section, the URI command is used. The following sections describe the more complex parameters used by aggregate queries.

7.2 Metric Parameter

The metric parameter is a comma-separated list of *metric functions*. Each function performs a statistical calculation on a scalar or link field. The general syntax of a metric function is:

```
function(field)
```

Where *function* is a metric function name and *field* defines what the metric function is computed on. The field must be a scalar or link defined in the application's schema. It can belong to the perspective table, or it

can be a path to a field linked to the perspective table (e.g., `DirectReports.Name`). The supported metric functions are summarized below:

- **`COUNT(`*`field`*`)`**: Counts the values for the specified *field*. If the field is multi-valued with respect to the perspective object, all values are counted for each selected object. For example, `COUNT(Tags)` tallies all `Tags` values of all objects. The COUNT function also allows the special value "*", which counts the selected objects in the perspective table regardless of any fields. That is, `COUNT(*)` counts objects instead of values.

- **`DISTINCT(`*`field`*`)`**: This metric is similar to COUNT except that it totals unique values for the given *field*. For example, `COUNT(Size)` finds the total number of values of the `Size` field, whereas `DISTINCT(Size)` finds the number of unique `Size` values.

- **`SUM(`*`field`*`)`**: Sums the non-null values for the given numeric *field*. The field's type must be `integer`, `long`, `float`, or `double`.

- **`AVERAGE(`*`field`*`)`**: Computes the average value for the given *field*, which must be `integer`, `long`, `float`, `double`, or `timestamp`. Note that the `AVERAGE` function uses SQL *null-elimination* semantics. This means that objects for which the metric field does not have a value are not considered for computation even though the object itself was selected. As an example, consider an aggregate query that computes `AVERAGE(foo)` for four selected objects, whose value for `foo` are 2, 4, 6, and null. The value computed will be 4 ((2+4+6)/3) not 3 ((2+4+6+0)/4) because the object with the null field is eliminated from the computation.

- **`MIN(`*`field`*`)`**: Computes the minimum value for the given *field*. For scalar fields, `MIN` computes the lowest value found based on the field type's natural order. For link fields, `MIN` computes the lowest object ID found in the link field's values based on the string form of the object ID.

- **`MAX(`*`field`*`)`**: Computes the maximum value for the given *field*, which must be a predefined scalar or link field.

Example metric functions are shown below:

```
COUNT(*)
DISTINCT(Tags)
MAX(Sender.Person.LastName)
AVERAGE(Size)
MAXCOUNT(Sender.Person.DirectReports)
```

The metric parameter can be a comma-separated list of metric functions. All functions are computed concurrently as objects are selected. An example metric parameter with multiple functions is shown below:

```
MIN(Size),MAX(Size),COUNT(InternalRecipients)
```

The results of *multi-metric* aggregate queries are described later.

7.3 Grouping Parameter

When a grouping parameter is provided, it causes the aggregate query to compute multiple values, one per *group value* as defined by the grouping expression. Either of two methods can be used to pass a grouping parameter depending upon whether or not composite grouping is desired. A wide range of grouping expressions are allowed as described in the following sections.

7.3.1 Global Aggregates: No Grouping Parameter

Without a grouping parameter, an aggregate query returns a single value: the metric function computed across all selected objects. Consider the following aggregate query URI REST command:

```
GET /Msgs/Message/_aggregate?m=MAX(Size)
```

This aggregate query returns the largest `Size` value among all messages. The response in XML is:

```
<results>
   <aggregate metric="MAX(Size)"/>
   <value>16796009</value>
</results>
```

In JSON:

```
{"results": {
   "aggregate": {"metric": "MAX(Size)"},
   "value": "16796009"
}}
```

As shown, an `aggregate` element lists the parameters used for the aggregate query. In this case, only a metric parameter was provided. For a global aggregate, the metric value is provided in the `value` element.

When the grouping field is multi-valued with respect to the perspective table, the metric is applied across all values for each perspective object. For example, in this query:

```
GET /Msgs/Message/_aggregate?m=COUNT(Tags)
```

If `Tags` is an MV scalar, all values for each message are counted, so the total returned may be more than the number of objects. Furthermore, an object related to a perspective object may be processed more than once. Consider this query:

```
GET /Msgs/Message/_aggregate?m=COUNT(ExternalRecipients.MessageAddress.Domain)
```

Some messages are likely to have multiple external recipients linked to the same domain. Therefore, `ExternalRecipients.MessageAddress.Domain` will count the same domain multiple times for those messages.

7.3.2 Single-level Grouping

When the grouping parameter consists of a single grouping field, objects are divided into sets based on the distinct values found for the grouping field. A separate metric value is computed for each group. For example, this aggregate query uses a single-valued scalar as the grouping field:

```
GET /Msgs/Message/_aggregate?m=MAX(Size)&f=Tags
```

The `Tags` field logically partitions objects into groups: one for each field value. Each object is included in each group for which is has a `Tags` value. If an object has no `Tags` value, it is placed in a `(null)` group. The maximum `Size` is then computed for each group. A typical result in XML is shown below:

```
<results>
   <aggregate metric="MAX(Size)" group="Tags"/>
   <totalobjects>6030</totalobjects>
   <summary>16796009</summary>
   <groups>
      <group>
         <metric>4875</metric>
         <field name="Tags">(null)</field>
      </group>
      <group>
         <metric>16796009</metric>
         <field name="Tags">AfterHours</field>
      </group>
      <group>
         <metric>16796009</metric>
         <field name="Tags">Customer</field>
      </group>
   </groups>
</results>
```

In JSON:

```
{"results": {
   "aggregate": {"metric": "MAX(Size)", "group": "Tags"},
   "totalobjects": "6030",
   "summary": "16796009",
   "groups": [
      {"group": {
         "metric": "4875",
         "field": {"Tags": "(null)"}
      }},
      {"group": {
         "metric": "16796009",
         "field": {"Tags": "AfterHours"}
      }},
      {"group": {
         "metric": "16796009",
         "field": {"Tags":"Customer"}
      }}
   ]
}}
```

For grouped aggregate queries, the `results` element contains a `groups` element, which contains one `group` for each group value. Each `group` contains the `field` name and value for that group and the corresponding `metric` value. The `totalobjects` value computes the number of objects selected in the computation. The

`summary` value computes the metric value across all selected objects independent of groups. For the `AVERAGE` function, this provides the *true average*, not the *average of averages*.

7.3.3 Grouping Field Aliases

By default, aggregate query group results use the fully-qualified path name of each grouping field. You can shorten the output result by using an *alias* for each grouping field. The alias name is used instead of the fully-qualified name. For example:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)&f=TOP(3,Sender.Person.Name)
```

This query produces a result such as the following:

```
<results>
   <aggregate metric="COUNT(*)" group="TOP(3,Sender.Person.Name)"/>
   <totalobjects>6030</totalobjects>
   <summary>6030</summary>
   <totalgroups>186</totalgroups>
   <groups>
      <group>
         <metric>5256</metric>
         <field name="Sender.Person.Name">(null)</field>
      </group>
      <group>
         <metric>82</metric>
         <field name="Sender.Person.Name">Quest Support</field>
      </group>
      <group>
         <metric>80</metric>
         <field name="Sender.Person.Name">spb_setupbuilder</field>
      </group>
   </groups>
</results>
```

The fully-qualified field name `Sender.Person.Name` is used for the field parameter in each group element. An alias can be substituted for a grouping field by appending `AS` *`name`* to the grouping field. Example:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)&f=TOP(3,Sender.Person.Name) AS Name
```

The alias `Name` is used in the output as shown below:

```
<results>
   <aggregate metric="COUNT(*)" group="TOP(3,Sender.Person.Name)"/>
   <totalobjects>6030</totalobjects>
   <summary>6030</summary>
   <totalgroups>186</totalgroups>
   <groups>
      <group>
         <metric>5256</metric>
         <field name="Name">(null)</field>
      </group>
```

```
        <group>
            <metric>82</metric>
            <field name="Name">Quest Support</field>
        </group>
        <group>
            <metric>80</metric>
            <field name="Name">spb_setupbuilder</field>
        </group>
    </groups>
</results>
```

When many groups are returned and/or the grouping field name is long, the alias name can significantly shorten the output results. An alias name can also improve the identity of the groups.

7.3.4 Multi-level Grouping

The grouping parameter can list multiple grouping expressions to form multi-level grouping. Each grouping expression must be a *path* from the perspective table to a link or scalar field. Below is an example multi-level aggregate query:

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

In this example, `TRUNCATE(SendDate,DAY)` is the top-level grouping field and `Tags` is the second-level grouping field. The query creates groups based on the cross-product of all grouping field values, and a metric is computed for each combination for which at least one object has a value. A perspective object is included in the metric computation for each group for which it has actual values. A `summary` value is computed for each non-leaf group, and a `totalobjects` value is computed for the top-level group. An example result in XML is shown below:

```
<results>
    <aggregate metric="COUNT(*)" group="TRUNCATE(SendDate,DAY),Tags"/>
    <totalobjects>6030</totalobjects>
    <summary>6030</summary>
    <groups>
        <group>
            <summary>4752</summary>
            <field name="SendDate">2010-07-17 00:00:00</field>
            <groups>
                <group>
                    <metric>1</metric>
                    <field name="Tags">(null)</field>
                </group>
                <group>
                    <metric>4751</metric>
                    <field name="Tags">AfterHours</field>
                </group>
                <group>
                    <metric>1524</metric>
                    <field name="Tags">Customer</field>
                </group>
```

```
            </groups>
        </group>
        <group>
            <summary>1278</summary>
            <field name="SendDate">2010-07-18 00:00:00</field>
            <groups>
                <group>
                    <metric>1278</metric>
                    <field name="Tags">AfterHours</field>
                </group>
                <group>
                    <metric>700</metric>
                    <field name="Tags">Customer</field>
                </group>
            </groups>
        </group>
    </groups>
</results>
```

The same response in JSON:

```
{"results": {
    "aggregate": {"metric": "COUNT(*)", "group": "TRUNCATE(SendDate,DAY),Tags"},
    "totalobjects": "6030",
    "summary": "6030",
    "groups": [
        {"group": {
            "summary": "4752",
            "field": {"SendDate":"2010-07-17 00:00:00"},
            "groups": [
                {"group": {
                    "metric": "1",
                    "field": {"Tags": "(null)"}
                }},
                {"group": {
                    "metric": "4751",
                    "field": {"Tags":"AfterHours"}
                }},
                {"group": {
                    "metric": "1524",
                    "field": {"Tags":"Customer"}
                }}
            ]
        }},
        {"group": {
            "summary": "1278",
            "field": {"SendDate": "2010-07-18 00:00:00"},
            "groups": [
                {"group": {
                    "metric": "1278",
                    "field": {"Tags":"AfterHours"}
```

```
                }},
                {"group": {
                    "metric": "700",
                    "field": {"Tags":"Customer"}
                }}
            ]
        }}
    ]
}}
```

Each non-leaf `group` has an inner `groups` element containing its corresponding lower-level `group` elements and a `summary` element that provides a group-level metric value. Leaf-level groups have a `metric` element. This structure is recursive if there are more than two grouping levels.

7.3.5 Group Fields as Grouping Parameters

Doradus Spider allows a group field to 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.

7.3.6 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
```

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.

7.3.7 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)
    &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 "*".

7.3.8 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.

7.3.9 Special Grouping Functions

This section describes special functions that provide enhanced behavior for the grouping parameter.

7.3.9.1 BATCH Function

The `BATCH` function divides a scalar field's values into specific ranges. Each range becomes a grouping field value, and objects contribute to the metric computation for the ranges for which it has values. The `BATCH` function's first value must be a scalar field. The remaining values must be literal values compatible with the field's type (text, timestamp, or numeric), and they must be given in ascending order. Example:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)&f=BATCH(Size,100,1000,10000,100000)
```

This query counts messages grouped by specific ranges of the `Size` field. The ranges are divided at the given literal values: `100`, `1000`, `10000`, and `100000`. The lowest value implicitly creates an extra *less than* group; the highest value is open-ended and creates a *greater than or equal to* group. The query in the example above defines the following 5 groups:

```
Group 1: Size < 100
Group 2: Size >= 100 AND Size < 1000
Group 3: Size >= 1000 AND Size < 10000
Group 4: Size >= 10000 AND Size < 100000
Group 5: Size >= 100000
```

The example above returns a result such as the following:

```
<results>
    <aggregate metric="COUNT(Size)" group="BATCH(Size,100,1000,10000,100000)"/>
    <groups>
        <group>
            <field name="Size">&lt;100</field>
            <metric>0</metric>
        </group>
        <group>
            <field name="Size">100-1000</field>
```

```
            <metric>125</metric>
        </group>
        <group>
            <field name="Size">1000-10000</field>
            <metric>4651</metric>
        </group>
        <group>
            <field name="Size">10000-100000</field>
            <metric>1149</metric>
        </group>
        <group>
            <field name="Size">&gt;=100000</field>
            <metric>105</metric>
        </group>
    </groups>
    <summary>6030</summary>
</results>
```

In JSON:

```
{"results": {
    "aggregate": {"group": "BATCH(Size,100,1000,10000,100000)", "metric": "COUNT(Size)"},
    "groups": [
        {"group": {
            "field": {"Size": "<100"},
            "metric": "0"
        }},
        {"group": {
            "field": {"Size": "100-1000"},
            "metric": "125"
        }},
        {"group": {
            "field": {"Size": "1000-10000"},
            "metric": "4651"
        }},
        {"group": {
            "field": {"Size":"10000-100000"},
            "metric": "1149"
        }},
        {"group": {
            "field": {"Size": ">=100000"},
            "metric": "105"
        }}
    ],
    "summary":"6030"
}}
```

As shown above in the `<100` group, if no selected object has a value that falls into one of the specified groups, that group is still returned: the group's metric is 0 for the `COUNT` function and empty for all other metric functions. As with all grouped aggregate queries, a `summary` value is returned that applies the metric function across all groups.

7.3.9.2 INCLUDE and EXCLUDE Functions

In an aggregate query, normally all values of a scalar grouping field are used to create groups. For example:

```
GET /Email/Message/_aggregate?f=Tags&...
```

All values of the `Tags` field for selected objects are used to create grouping fields. To eliminate specific values from being used for grouping—without affecting the selection of the owning object—the `EXCLUDE` function can be used:

```
GET /Email/Message/_aggregate?f=Tags.EXCLUDE('Confidential, 'Internal')&...
```

When the grouping field is a text field, the values passed to the `EXCLUDE` function are whole, case-insensitive values—not terms—and must be enclosed in quotes. In the example above, groups matching the value `Confidential` or `Internal` or case variations of these are excluded. The values used for text scalars can contain wildcards `?` and `*`. For example:

```
GET /Email/Message/_aggregate?f=Tags.EXCLUDE('*sam?')&...
```

This aggregate query excludes all groups that end with the `samx`, where `x` is any letter, or case variations of this sequence.

To generate only groups that match specific scalar values—without affecting the selection of the owning object—the `INCLUDE` function can be used:

```
GET /Email/Message/_aggregate?f=Tags.INCLUDE('Confidential, 'Internal')&...
```

The only groups generated are those matching `Confidential` and `Internal` and case variations of these; all other values are skipped. Again, when the grouping field is a text scalar, the value must be enclosed in quotes, and it can contain wildcards `?` and `*`.

The values passed to `INCLUDE` and `EXCLUDE` must be compatible with the corresponding scalar type field: integers for `integer` or `long` fields, Booleans for `boolean` fields, etc. Additionally, the keyword `NULL` (uppercase) can be used to include or exclude the `(null)` group normally generated when at least one object has a null value for the grouping field. Example:

```
GET /Email/Message/_aggregate?f=Tags.EXCLUDE(NULL)&...
```

7.3.9.3 TERMS Function

Groups can be created from the *terms* used within a specific field. The general format of the `TERMS` function is:

```
TERMS(<field name> [, (<stop term 1> <stop term 2> ...)])
```

Any predefined scalar field can be used, but `TERMS` is most effective with text fields. The optional *stop term* list is a list of terms, enclosed in parentheses, that are excluded from the unique set of terms found within the specified field.

For example, the following request fetches the COUNT of messages grouped by terms found within the Subject field for a particular sender. For brevity, the TERMS function is wrapped by the TOP function to limit the results to the five groups with the highest counts:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)&q=Sender.MessageAddress.Person.Name:Support
   &f=TOP(5,TERMS(Subject))
```

Similar to the BATCH function, the TERMS function creates dynamic groups from a text field based on the terms it uses. To do this, as objects matching the query parameter (if any) are found, the field passed to TERMS is parsed into alphanumeric terms, and a group is created for each unique term. Each contributes to the group metric computation for each term it contains. If a term appears multiple times within a field (e.g., "plan for a plan"), the object is only counted once. An example result in XML is shown below:

```
<results>
   <aggregate query="Sender.MessageAddress.Person.Name:Support" metric="COUNT(*)"
      group="TOP(5,TERMS(Subject))"/>
   <groups>
      <group>
         <field name="Subject">dilemmas</field>
         <metric>14</metric>
      </group>
      <group>
         <field name="Subject">unchary</field>
         <metric>14</metric>
      </group>
      <group>
         <field name="Subject">sundae</field>
         <metric>14</metric>
      </group>
      <group>
         <field name="Subject">tillage</field>
         <metric>14</metric>
      </group>
      <group>
         <field name="Subject">infernal</field>
         <metric>14</metric>
      </group>
   </groups>
   <summary>82</summary>
   <totalgroups>85</totalgroups>
</results>
```

In JSON:

```
{"results": {
   "aggregate": {
      "group": "TOP(5,TERMS(Subject))",
      "metric": "COUNT(*)",
      "query": "Sender.MessageAddress.Person.Name:Support"
   },
```

```
    "groups": [
       {"group": {
           "field": {"Subject": "dilemmas"},
           "metric": "14"
       }},
       {"group": {
           "field": {"Subject": "unchary"},
           "metric": "14"
       }},
       {"group": {
           "field": {"Subject": "sundae"},
           "metric": "14"
       }},
       {"group": {
           "field": {"Subject": "tillage"},
           "metric": "14"
       }},
       {"group": {
           "field": {"Subject": "infernal"},
           "metric": "14"
       }}
    ],
    "summary": "82",
    "totalgroups": "85"
}}
```

If the terms “sundae” and “tillage” were considered uninteresting, they could be eliminated from the results by listing them as stop terms in a second parenthetical parameter to the `TERMS` function:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)&q=Sender.MessageAddress.Person.Name='Quest Support'
   &f=TOP(5,TERMS(Subject,(sundae tillage)))
```

7.3.9.4 TOP and BOTTOM Functions

By default, all group values are returned at each grouping level, and the groups are returned in ascending order of the grouping field value. The `TOP` and `BOTTOM` functions can be used to return groups in the order of the metric value. Optionally, they can also be used to limit the number of groups returned to the *highest* or *lowest* metric values. The `TOP` and `BOTTOM` functions *wrap* a grouping field expression and specify a *limit* parameter. For example:

```
GET /Msgs/Message/_aggregate?m=SUM(Size)&f=TOP(3,InternalRecipients.Person.Name)
```

The first parameter to TOP/BOTTOM is the limit value; the second parameter is the grouping field expression. This aggregate query sums the `Size` field of message objects, grouped by internal recipient names, but it only returns the groups with the three highest `SUM` values. Typical results for the example above in XML:

```
<results>
   <aggregate metric="SUM(Size)" group="TOP(3,InternalRecipients.Person.Name)"/>
   <totalobjects>6030</totalobjects>
   <summary>190643320</summary>
```

```
    <totalgroups>836</totalgroups>
    <groups>
       <group>
          <metric>103198352</metric>
          <field name="InternalRecipients.Person.Name">(null)</field>
       </group>
       <group>
          <metric>20060808</metric>
          <field name="InternalRecipients.Person.Name">Marc Bourauel</field>
       </group>
       <group>
          <metric>16798901</metric>
          <field name="InternalRecipients.Person.Name">Nina Cantauw</field>
       </group>
    </groups>
</results>
```

In JSON:

```
{"results": {
    "aggregate": {
       "metric": "SUM(Size)",
       "group": "TOP(3,InternalRecipients.Person.Name)"
    },
    "totalobjects": "6030",
    "summary": "190643320",
    "totalgroups": "836",
    "groups": [
       {"group": {
          "metric": "103198352",
          "field": {"InternalRecipients.Person.Name": "(null)"}
       }},
       {"group": {
          "metric": "20060808",
          "field": {"InternalRecipients.Person.Name": "Marc Bourauel"}
       }},
       {"group": {
          "metric": "16798901",
          "field": {"InternalRecipients.Person.Name": "Nina Cantauw"}
       }}
    ]
}}
```

The `BOTTOM` parameter works the same way but returns the groups with the lowest metric values. When either the `TOP` or `BOTTOM` function is used, the total number of groups that were actually computed is returned in the element `totalgroups`, as shown above.

When the limit parameter is 0, all groups are returned, but they are returned in metric-computation order. Using 0 for the limit parameter essentially means *unlimited*.

When the aggregate query has multiple grouping fields, a TOP or BOTTOM function can be used with each grouping field. In secondary groups, TOP and BOTTOM return groups whose metric values are computed relative to their parent groups. Below is an example aggregate query 2-level grouping using TOP for the outer level and BOTTOM for the inner level:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)&f=TOP(3,Sender.Person),BOTTOM(2,TRUNCATE(SendDate,DAY))
```

7.3.9.5 TRUNCATE Function

The TRUNCATE function truncates a timestamp field to a given granularity, yielding a value that can be used as a grouping field. Before the timestamp field is truncated, the TRUNCATE function can optionally *shift* the value to another time first. The syntax for the function is:

```
TRUNCATE(<timestamp field>, <precision> [, <time shift>])
```

For example:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)&f=TRUNCATE(SendDate,DAY,GMT-2)
    &q=SendDate >= "2010-07-17"
```

This query finds all messages whose SendDate is >= "2010-07-17". For each one, it subtracts 2 hours from the SendDate value and then truncates ("rounds down") to the nearest day. The count of all objects for each modified timestamp is computed in a separate group.

The <precision> value must be one of the following mnemonics:

Precision	Meaning
SECOND	The milliseconds component of the timestamp is set to 0.
MINUTE	The milliseconds and seconds components are set to 0.
HOUR	The milliseconds, seconds, and minutes components are set to 0.
DAY	All time components are set to 0.
WEEK	All time components are set to 0, and the date components are set to the Monday of the calendar week in which the timestamp falls (as defined by ISO 8601). For example 2010-01-02 is truncated to the week 2009-12-28.
MONTH	All time components are set to 0, and the day component is set to 1.
QUARTER	All time components are set to 0, the day component is set to 1, and the month component is rounded "down" to January, April, July, or October.
YEAR	All time components are set to 0 and the day and month components are set to 1.

The optional <time shift> parameter adds or subtracts a specific amount to each object's timestamp value before truncating it to the requested granularity. Optionally, the parameter can be quoted in single or double quotes. The syntax of the <time shift> parameter is:

```
<timezone> | <GMT offset>
```

Where <GMT offset> uses the same format as the NOW function:

```
GMT<sign><hours>[:<minutes>]
```

The meaning of each format is summarized below:

- `<timezone>`: A timezone abbreviation (e.g., `"PST"`) or name (e.g., `"America/Los_Angeles"`) can be given. Each object's timestamp value is assumed to be in GMT (UTC) time and adjusted by the necessary amount to reflect the equivalent value in the given timezone. The allowable values for a <timezone> abbreviation or name are those recognized by the Java function `java.util.TimeZone.getAvailableIDs()`.

- `GMT+<hour>` or `GMT-<hour>`: The term GMT followed by a plus or minus signed followed by an integer hour value adjust each object's timestamp up or down by the given number of hours.

- `GMT+<hour>:<minute>` or `GMT-<hour>:<minute>`: This is the same as the previous format except that each object's timestamp is adjusted up or down by the given hour and minute value.

Note that in the GMT versions, the sign ('+' or '-') is required, and in URIs, the '+' sign must be escaped as `%2B`.

The timestamp field passed to the `TRUNCATE` function can belong to the perspective table, or it can be at the end of a field path (e.g., `TRUNCATE(Messages.SendDate)`).

When a grouping field uses the `TRUNCATE` function, the truncated value is used for the field value within each group. An example in XML is shown below:

```
<results>
   <aggregate query="SendDate >= 2010-07-17" metric="COUNT(*)" group="TRUNCATE(SendDate,HOUR)"/>
   <groups>
      <group>
         <field name="SendDate">2010-07-17 00:00:00</field>
         <metric>5</metric>
      </group>
      <group>
         <field name="SendDate">2010-07-17 01:00:00</field>
         <metric>4</metric>
      </group>
      <group>
         <field name="SendDate">2010-07-17 02:00:00</field>
         <metric>4</metric>
      </group>
      ...
   </groups>
   <summary>6030</summary>
</results>
```

In JSON:

```
{"results": {
   "aggregate": {
      "group": "TRUNCATE(SendDate,HOUR)",
      "metric": "COUNT(*)",
```

```
        "query": "SendDate >= 2010-07-17"
    },
    "groups": [
        {"group": {
            "field": {"SendDate": "2010-07-17 00:00:00"},
            "metric": "5"
        }},
        {"group": {
            "field": {"SendDate": "2010-07-17 01:00:00"},
            "metric": "4"
        }},
        {"group": {
            "field": {"SendDate": "2010-07-17 02:00:00"},
            "metric": "4"
        }},
        ...
    ]
}}
```

7.3.9.6 UPPER and LOWER Functions

When a text field is used as a grouping field in an aggregate query, *actual* field values are used to form each group value. For example, in this query:

```
.../_aggregate?m=COUNT(*)&f=Extension&...
```

If the field `Extension` has identical but differently-cased values such as ".jpg" and ".JPG", a group is created for each one and the metric function (`COUNT`) is applied to each one.

When a text field is used as the grouping field, values can be case-normalized as they are used as grouping field values. This can be done with the `UPPER` and `LOWER` functions, which translate each text field accordingly as it is sorted into aggregated groups. Example:

```
.../_aggregate?m=COUNT(*)&f=LOWER(Extension)&...
```

This causes the `Extension` field to be down-cased before it is sorted into its metric group. Hence, both values ".jpg" and ".JPG" are counted in a single group.

7.3.9.7 WHERE Function

The `WHERE` function can be used to provide *filtering* on a path used in a grouping expression. Most importantly, it can be used for multi-clause expressions that are *bound* to the same objects. To illustrate why the `WHERE` clause is needed and how it is used, here's an example.

Suppose we want to count messages grouped by the domain name of each message's recipients, but we only want recipients that received the message after a certain date and the recipient's address is considered external. As an example, this aggregate query won't work:

```
// Doesn't do what we want
GET /Msgs/Message/_aggregate?m=COUNT(*)
    &f=Recipients.MessageAddress.Domain.Name
    &q=Recipients.ReceiptDate > "2014-01-01" AND Recipients.MessageAddress.Domain.IsInternal=false
```

This query doesn't work because it selects messages for which at least one recipient's `ReceiptDate` is > `"2014-01-01"`, and at least one recipient has an external domain. Every such message is then counted in all of its `Recipients.MessageAddress.Domain.Name` values, even for those that don't really qualify.

Using the `WHERE` filter for query expressions, we could bind the two query clauses to the same `Recipients` instances. But this query still doesn't work:

```
// Still not what we want
GET /Msgs/Message/_aggregate?m=COUNT(*)
    &f=Recipients.MessageAddress.Domain.Name
    &q=Recipients.WHERE(ReceiptDate > "2014-01-01" AND MessageAddress.Domain.IsInternal=false)
```

This causes the correct objects to be selected, but it still counts them in all `Recipients.MessageAddress.Domain.Name` groups, not just those found with the query expression.

For this scenario, we can use the `WHERE` function in the grouping parameter instead of the query parameter. In a grouping parameter, the `WHERE` function filters out group values we don't want. And, when the object selection criteria lies solely in the choice of groups, we don't need a separate query parameter. The solution to the previous problem can be expressed as follows:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)
    &f=Recipients.WHERE(ReceiptDate > "2014-01-01" AND
        MessageAddress.Domain.IsInternal=false).MessageAddress.Domain.Name
```

The grouping field is still `Recipients.MessageAddress.Domain.Name`, but the `WHERE` function inserted after `Recipients` filters values used for grouping. The first field in each `WHERE` clause (`ReceiptDate` and `MessageAddress`) must be members of the same table as `Recipients`, thereby filtering the recipients in some manner. In this case, only recipients whose `ReceiptDate` is > `"2014-01-01"` and whose `MessageAddress.Domain.IsInternal` is false. Groups are created by domains of recipients that match those constraints, and only objects within those group values are counted.

But wait! It gets better! The `WHERE` function can be applied to multiple components of the same grouping path as long as each subquery is qualified to the path component to which it is attached. Exploiting this, we can factor out the redundant specification of `MessageAddress.Domain` with this shorter but equivalent expression:

```
GET /Msgs/Message/_aggregate?m=COUNT(*)
    &f=Recipients.WHERE(ReceiptDate > "2014-01-01").Address.Domain.WHERE(IsInternal=false).Name
```

Neat, yes?

7.4 Multi-metric Aggregate Queries

The metric parameter can use a comma-separated list of metric functions. Such *multi-metric* queries perform multiple metric computations in a single pass through the data. The simplest case uses no grouping parameter. For example:

```
.../_aggregate?m=COUNT(*),MAX(Size),AVERAGE(Size)
```

This query requests a count of all objects and the maximum and average values for the `Size` field. Multi-metric query results use an outer `groupsets` element containing one `groupset` for each metric function. The query above returns results such as the following:

```
<results>
   <aggregate metric="COUNT(*),MAX(Size),AVERAGE(Size)"/>
   <groupsets>
      <groupset metric="COUNT(*)">
         <value>6030</value>
      </groupset>
      <groupset metric="MAX(Size)">
         <value>16796009</value>
      </groupset>
      <groupset metric="AVERAGE(Size)">
         <value>31615.808</value>
      </groupset>
   </groupsets>
</results>
```

In JSON:

```
{"results": {
   "aggregate": {"metric": "COUNT(*),MAX(Size),AVERAGE(Size)"},
   "groupsets": [
      {"groupset": {
         "metric": "COUNT(*)",
         "value": "6030"
      }},
      {"groupset": {
         "metric": "MAX(Size)",
         "value": "16796009"
      }},
      {"groupset": {
         "metric": "AVERAGE(Size)",
         "value": "31615.808"
      }}
   ]
}}
```

As shown, one `groupset` is provided for each metric function. Since there is no grouping parameter, the format of each `groupset` matches that of a global aggregate query: a `metric` element identifies which metric function is computed, and a `value` element provides the function's value.

For grouped, multi-metric aggregate queries, each metric function is computed for all inner and outer group levels. Any mix of metric functions can be used except for the `DISTINCT` function, which cannot be used in a multi-metric queries. Grouped queries produce one `groupset` result for each metric function as in the non-grouped case. However, each `groupset` will contain groups that decompose the metric computations in the appropriate groups.

When a multi-metric aggregate query uses the TOP or BOTTOM function in the outer grouping field, the TOP or BOTTOM limit is derived from the first metric function. The outer and inner groups are selected on this metric function, and each metric function is performed for those groups. For example:

```
.../_aggregate?m=COUNT(*),MAX(Size),AVERAGE(Size)&f=TOP(2,Tags),TRUNCATE(SendDate,MONTH)
```

The COUNT(*) metric determines which outer groups are included in the TOP(2) grouping; the COUNT(*), MAX(Size), and AVERAGE(Size) functions are returned for all these 2 outer groups and the corresponding inner groups. A typical response to the query above in XML:

```
<results>
   <aggregate metric="COUNT(*),MAX(Size),AVERAGE(Size)"
      group="TOP(2,Tags),TRUNCATE(SendDate,MONTH)"/>
   <groupsets>
      <groupset group="TOP(2,Tags),TRUNCATE(SendDate,MONTH)" metric="COUNT(*)">
         <summary>6030</summary>
         <totalgroups>3</totalgroups>
         <groups>
            ...
         </groups>
      </groupset>
      <groupset group="TOP(2,Tags),TRUNCATE(SendDate,MONTH)" metric="MAX(Size)">
         <summary>16796009</summary>
         <totalgroups>3</totalgroups>
         <groups>
            ...
         </groups>
      </groupset>
      <groupset group="TOP(2,Tags),TRUNCATE(SendDate,MONTH)" metric="AVERAGE(Size)">
         <summary>31615.808</summary>
         <totalgroups>3</totalgroups>
         <groups>
            ...
         </groups>
      </groupset>
   </groupsets>
</results>
```

Because the outer grouping level used the TOP function, a totalgroups value is provided for each outer grouping level. Here's the same response in JSON:

```
{"results": {
   "aggregate": {
      "group": "TOP(2,Tags),TRUNCATE(SendDate,MONTH)",
      "metric": "COUNT(*),MAX(Size),AVERAGE(Size)"
   },
   "groupsets": [
      {"groupset": {
         "metric": "COUNT(*)",
         "group": "TOP(2,Tags),TRUNCATE(SendDate,MONTH)",
```

```
            "groups": [
                ...
            ],
            "summary": "6030",
            "totalgroups": "2"
        }},
        {"groupset": {
            "metric": "MAX(Size)",
            "group": "TOP(2,Tags),TRUNCATE(SendDate,MONTH)",
            "groups": [
                ...
            ],
            "summary": "16796009",
            "totalgroups": "2"
        }},
        {"groupset": {
            "metric": "AVERAGE(Size)",
            "group": "TOP(2,Tags),TRUNCATE(SendDate,MONTH)",
            "groups": [
                ...
            ],
            "summary": "31615",
            "totalgroups": "2"
        }}
    ]
}}
```

8. Spider REST Commands

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

8.1 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.

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

8.1.1 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.

8.1.2 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
```

8.1.3 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.

8.1.4 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: 43
Content-type: Text/plain

Database is not reachable. Waiting to retry
```

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.

8.2 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`
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}`

REST Command	Method and URI
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.

8.3 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.

8.3.1 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.

8.3.2 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.

8.3.3 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.

8.3.4 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.

8.4 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.

8.4.1 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
```

8.4.2 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.

8.4.3 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.

8.5 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"
    }
}}
```

8.6 Object Query Commands

Doradus Spider supports two commands for submitting object queries. Details of object query parameters and the output formats returned by object queries are described in the section Spider Object Queries.

8.6.1 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.

8.6.2 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"
}}
```

8.7 Aggregate Query Commands

Doradus Spider supports two commands for performing aggregate queries.

8.7.1 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)
```

8.7.2 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)"
}}
```

8.8 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.

8.8.1 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.

8.8.2 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.

8.8.3 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.

8.8.3.1 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.

8.8.3.2 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.

8.8.3.3 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.

8.9 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 executed 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.

8.9.1 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.

8.9.1.1 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`.

8.9.1.2 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.

8.9.1.3 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.

8.9.2 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.

8.9.3 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.

8.9.4 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.