Doradus Data Model and Query Language

1. Introduction

This document describes the Doradus data model and query language that are common to the Doradus OLAP and Spider services. The following documents are also available:

- **Doradus OLAP Database**: Describes the features and REST commands of the Doradus OLAP service.
- **Doradus Spider Database**: Describes the features and REST commands of the Doradus Spider service.
- **Doradus Administration**: Describes how to install and configure Doradus for various deployment scenarios, and it describes the JMX commands provided by the Doradus Server.

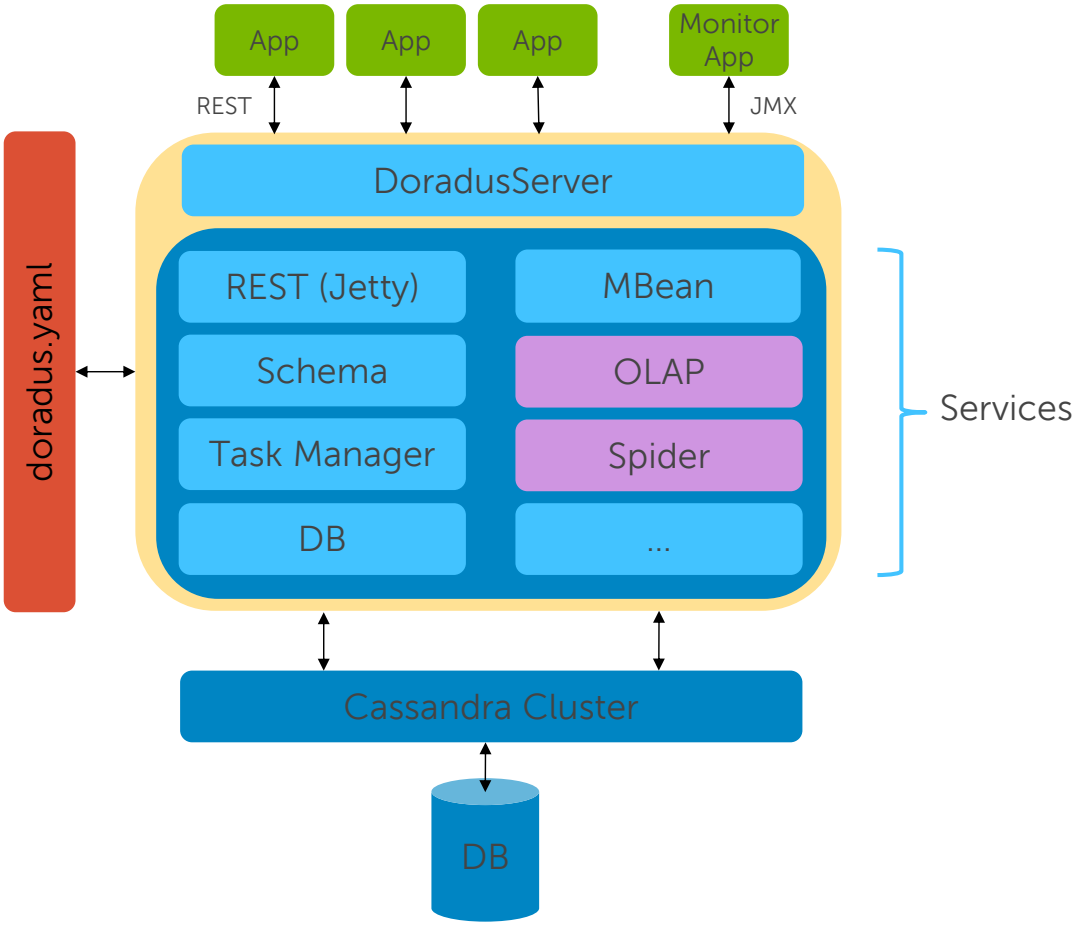
This document is organized into the following sections:

- **Architecture**: An overview of the Doradus architecture. The Doradus OLAP and Doradus Spider databases are compared.
- **Data Model**: Describes the Doradus data model that is common to OLAP and Spider databases.
- **Query Language**: Describes the core Doradus query language (DQL) that is common to OLAP and Spider databases.
- **Object and Aggregate Queries**: Describes these two query types including their parameters and output results in XML and JSON.

2. Architecture

Doradus is a Java server application that leverages and extend 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, a 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. The Doradus **Monitor App** uses this API to provide a graphical administrator client.
- **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 as an embedded component within another application.

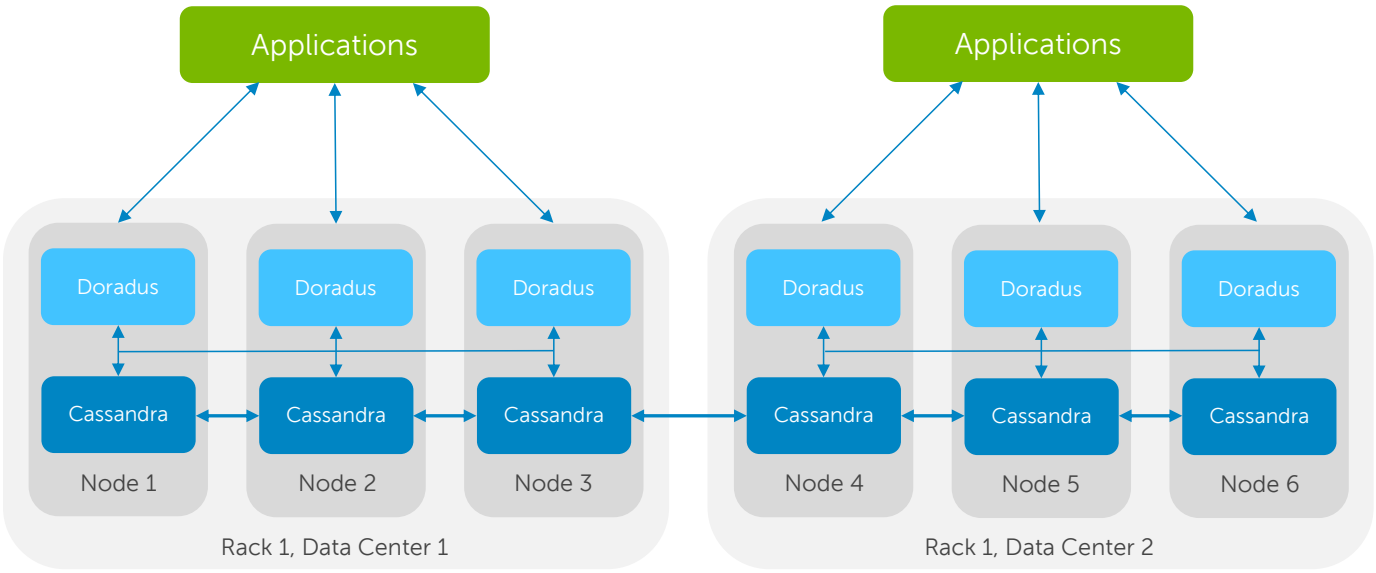
- **Services**: Doradus' architecture encapsulates functions within service modules. Services are initialized based on the server's **doradus.yaml** configuration. Services provide functions such as the **REST** API (an embedded Jetty server), **Schema** processing, and physical **DB** access. A special 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 is provided in the Doradus Administration document.

3. Data Model

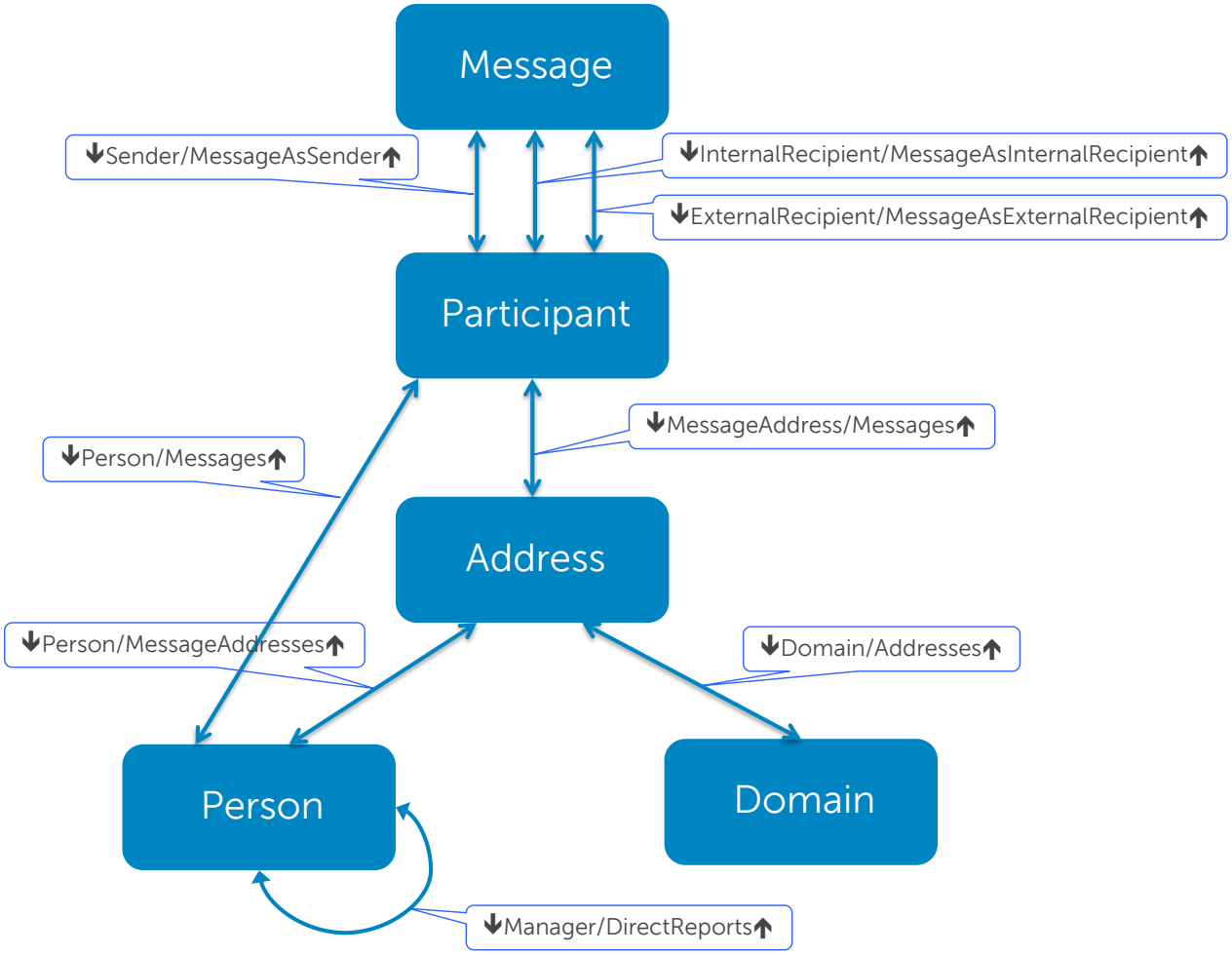
This section describes core Doradus terms and data model concepts.

3.1 Motivation

Doradus started by adding a REST API and Lucene-like fully inverted indexing on top of the Cassandra NoSQL database. 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: schemaless applications, horizontal scalability, idempotent updates, etc. The result is a unique blend of features from Lucene, graph databases, and NoSQL databases.

3.2 The `Email` Sample Application

To illustrate features, a sample application called `Email` will be used. Its schema is depicted below:



The `Email` application uses five tables, which are summarized below:

- `Message`: Holds one object per *sent* email. Each object stores values such as `Size` and `SendDate` timestamp 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 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 Email application tracks email messages and allows analytic queries such as "how many emails are sent daily" and "which department sends the most emails". In practice, an email tracking application would probably use more tables and fields than shown in this sample schema.

The schema for the `Email` application is shown below in XML:

```
<application name="Email">
   <key>EmailKey</key>
   <tables>
      <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>
            <field name="SendDate" type="TIMESTAMP"/>
            <field name="Size" type="INTEGER"/>
            <field name="Subject" type="TEXT"/>
```

```
            <field name="Tags" collection="true" type="TEXT"/>
        </fields>
        <aliases>
            <alias name="$SalesEmails" expression="Sender.Person.WHERE(Department:Sales)"/>
        </aliases>
    </table>
    <table name="Participant">
        <fields>
            <field name="MessageAddress" type="LINK" table="Address" inverse="Messages"/>
            <field name="MessageAsExternalRecipient" type="LINK" table="Message"
                inverse="ExternalRecipients"/>
            <field name="MessageAsInternalRecipient" type="LINK" table="Message"
                inverse="InternalRecipients"/>
            <field name="MessageAsSender" type="LINK" table="Message" inverse="Sender"/>
            <field name="Person" type="LINK" table="Person" inverse="Messages"/>
            <field name="ReceiptDate" type="TIMESTAMP"/>
        </fields>
    </table>
    <table name="Address">
        <fields>
            <field name="Domain" type="LINK" table="Domain" inverse="Addresses"/>
            <field name="Messages" type="LINK" table="Participant" inverse="MessageAddress"/>
            <field name="Name" type="TEXT"/>
            <field name="Person" type="LINK" table="Person" inverse="MessageAddresses"/>
        </fields>
    </table>
    <table name="Person">
        <fields>
            <field name="Location">
                <fields>
                    <field name="Department" type="TEXT"/>
                    <field name="Office" type="TEXT"/>
                </fields>
            </field>
            <field name="DirectReports" type="LINK" table="Person" inverse="Manager"/>
            <field name="FirstName" type="TEXT"/>
            <field name="LastName" type="TEXT"/>
            <field name="Manager" type="LINK" table="Person" inverse="DirectReports"/>
            <field name="MessageAddresses" type="LINK" table="Address" inverse="Person"/>
            <field name="Messages" type="LINK" table="Participant" inverse="Person"/>
            <field name="Name" type="TEXT"/>
        </fields>
    </table>
    <table name="Domain">
        <fields>
            <field name="Addresses" type="LINK" table="Address" inverse="Domain"/>
            <field name="IsInternal" type="BOOLEAN"/>
            <field name="Name" type="TEXT"/>
        </fields>
    </table>
</tables>
```

```
</application>
```

The same schema in JSON is shown below:

```
{"Email": {
    "key": "EmailKey",
    "options": {"StorageService": "OLAPService"},
    "tables": {
        "Message": {
            "fields": {
                "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"}
            },
            "aliases": {
                "$SalesEmails": {"expression": "Sender.Person.WHERE(Department:Sales)"}
            }
        },
        "Participant": {
            "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"},
                "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"},
                "IsInternal": {"type": "BOOLEAN"},
                "Name": {"type": "TEXT"}
            }
        }
    }
}}
```

The constructs in this sample application schema are used as examples throughout this document.

3.3 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`.

3.4 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 such as `StorageService`, which defines the storage service that will manage the application's data.
- 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="Email">
   <key>EmailKey</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:

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

3.5 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**: Depending on the storage manager, some table-level options may be supported.
- **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
      }
   },
   ...
}
```

3.6 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*.

3.7 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.

3.8 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*.

3.8.1 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.
- **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`). Example field definitions in XML are shown below:

```
<fields>
   <field name="SendDate" type="TIMESTAMP"/>
   <field name="Size" type="INTEGER"/>
   <field name="Subject" type="TEXT"/>
   <field name="IsInternal" type="BOOLEAN"/>
</fields>
```

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 defining 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.

3.8.2 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).

3.8.3 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.

3.9 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 /Email/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 /Email/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.

3.10 Tasks and Schedules

Depending upon its assigned storage service, an application may have background *tasks*, which perform functions such as data aging. An application's schema can define schedules that control when its tasks execute. An example schedule declaration is shown below:

```
<application name="Email">
    ...
    <schedules>
        <schedule type="data-aging" value="0 0 3 * *"/>
    </schedules>
</application>
```

In JSON, the schedules group is declared as an array:

```
{"Email": {
    ...
    "schedules": [
        {"schedule": {"type": "data-aging", "value": "0 0 3 * *"}}
    ]
}}
```

This example defines a `data-aging` task to execute with the cron expression "`0 3 * * *`", which means "once per day at 03:00".

3.11 Schema Command Summary

Applications are created, modified, and deleted with REST commands that include a schema as a JSON or XML message. These REST commands use a system-defined `_applications` resource, which represents the database "catalog". Common application management commands are summarized in the next sections. Because each storage service extends the Doradus data model and provides unique features, REST command details are described in the **Doradus OLAP Database** and **Doradus Spider Database** documents.

3.11.1 Create Application

An application is created by sending its schema in the following REST command:

```
POST /_applications
```

The schema's format (JSON or XML) must match the request's `content-type` header. Because Doradus supports *idempotent* updates, if the application defined by the schema already exists, the command is treated as a Modify Application command.

3.11.2 Modify Application

An existing application's schema is modified by sending its new schema in the following REST command:

```
PUT /_applications/{application}
```

where `{application}` is the application's name. An error is returned if the application does not exist or if the application key in the modified schema does not match.

3.11.3 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.

3.11.4 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 and `{key}` is the application key value. An error is returned if the application doesn't exist or the `{key}` doesn't match.

4. 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, but has many extensions to leverage link fields such as path expressions, quantifiers, and transitive searches.

The Object and OLAP storage services both support DQL, but each provides service-specific extensions. This section describes the DQL concepts common to both services.

4.1 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.

4.2 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.

4.3 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.

4.4 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
```

4.5 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.

4.5.1 Booleans

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

4.5.2 Numbers

A numeric literal is all digits, preceded with a minus sign (-) for negative values. All Doradus numeric values are considered long (64-bit). (Doradus does not currently support floating-point numbers.)

4.5.3 Timestamps

Timestamp literals have the general form:

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

Notes:

- In most cases, 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 date components only, the enclosing quotes can be omitted because the value conforms to the syntax of a term (see next section).
- Omitted time components default to 0; omitted date components default to 1.

4.5.4 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.

4.5.5 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"`.

4.6 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
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.

4.7 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
```

4.8 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.

4.8.1 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
```

In Spider applications, matching objects will have `John` in the `Name` field and `Smith` in any field. In OLAP applications, this query isn't allowed!

4.8.2 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.

4.9 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]
```

4.10 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
```

4.11 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)
```

4.12 Timestamp Clauses

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

4.12.1 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. 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.

4.12.2 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.

4.12.3 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.

4.13 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.

4.13.1 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`.

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

4.14 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.

4.14.1 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.

4.14.2 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.

4.14.3 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
```

4.14.4 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 a `MessageAddresses` value of `XYZ`.

4.14.5 Quantifiers with IS NULL

Because links are implicitly multi-valued, a link path creates a set of one or more values. Consider the following link path:

```
InternalRecipients.Person.Manager
```

`InternalRecipients`, `Person`, and `Manager` are all link fields. For a given perspective object, this path may produce any of the following value sets:

1) **`{}`**: If `InternalRecipients` is null for the perspective object, the value set is null. The set is also null if `InternalRecipients` is non-null but `Person` is null for every linked object.

2) **`{ID1, ID2, …}`**: If at least one `InternalRecipients.Person` exists, and every `InternalRecipients.Person.Manager` is non-null, the value set will consist entirely of non-null values. (Since `Manager` is a link, the set will consist of object IDs.)

3) **`{null, null, …}`**: If at least one `InternalRecipients.Person` exists, but every `InternalRecipients.Person.Manager` is null, the value set will consist entirely of null values.

4) **`{ID1, null, ID2, null, …}`**: If some `InternalRecipients.Person.Manager` values are null while some are non-null, the set will contain a mix of null and non-null values.

In other words, the value set produced by the link path has a null or non-null value for each instance of the terminal (right-most) field. However, when an intermediate link is null, it does not produce a path to the terminal field, so it does not contribute to the value set.

When a link path is used with the `IS NULL` clause, how the `IS NULL` clause behaves depends on how the link path is quantified. For example:

```
InternalRecipients.Person.Manager IS NULL
```

Each link field is implicitly quantified with `ANY`. For this clause to be true, the value set produced by the link path must be non-empty and contain at least one null value. In other words, the clause is true for an object that has at least one `InternalRecipients.Person` whose `Manager` is null.

Now consider the effect of `ALL` quantification, first with this example:

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

This clause is true for a perspective that has at least one `InternalRecipients.Person` value, but no `InternalRecipients.Person` has a `Manager`. That is, the quantified set cannot be empty, and all values in the set must be null. This means that if an object has no `InternalRecipients` or no `InternalRecipients` have an `Person` values, the set is empty therefore the clause is false.

However, if each link is quantified with `ALL` separately, the semantics are different. For example:

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

This clause returns true for a message that (a) has at least one `InternalRecipients` value, (b) every `InternalRecipients` object has at least one `Person` value, and (c) every `Person` object's `Manager` is null. In other words, if a message has two `InternalRecipients`, one of them has no `Person` but the other one has a `Person` whose `Manager` is null, the clause returns false. This is because separate quantification produces multiple value sets and `IS NULL` must select a value from each set in order for the overall clause to be true.

4.15 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 bound 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`.

4.16 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.

5. Object Queries

An *object query* is a DQL query that selects and returns selected fields for objects in the perspective table. Core object query features are described in this section. Extensions by the OLAP and Spider storage services are described in their respective documents.

5.1 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.

Additional object query parameters are supported by each storage service. How parameters are passed depends on the REST command.

5.2 Query Commands

Each storage service provides REST commands to perform object queries in two different ways.

5.2.1 URI Command

With this command, all parameters are passed as URI query parameters after a question mark (?). Parameters must be separated by ampersands (&) and escaped as required by URI rules. Below is an example REST command for the Spider Service:

```
GET /Email/Person/_query?q=FirstName:Daniel&o=LastName+DESC&f=FirstName,LastName&s=10&k=20
```

This query retrieves `Person` objects from the `Email` application. The parameters are:

- `?q`: This **query** parameter selects objects whose `FirstName` field contains the term `Daniel`.
- `&o`: This **order** parameter sorts the results by `LastName` in descending order.

- `&f`: This **fields** parameter returns the `FirstName` and `LastName` of each object. Each object's `_ID` is also returned.
- `&s`: This **page size** parameter limits the results to a maximum of 10 objects.
- `&k`: This **skip** parameter skips the first 20 objects and returns the subsequent page.

5.2.2 Entity Command

With this command, all parameters are passed in the request as an entity (message body). The entity can be formatted as a XML or JSON document. Below is the same query as above for the Spider Service with parameters passed as an XML message:

```
PUT /Email/Person/_query
```

Message body:

```
<search>
   <query>FirstName:Daniel</query>
   <order>LastName DESC</order>
   <fields>FirstName,LastName</fields>
   <size>10</size>
   <skip>20</skip>
</search>
```

As shown, the document uses element names that are analogs to URI parameters. Because some HTTP clients don't allow GET requests with message bodies, the storage services recognize the PUT method for this query, even though it is a query.

5.3 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.

5.3.1 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.

5.3.2 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".

5.3.3 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.

5.4 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.

5.4.1 Empty Results

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

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

In JSON:

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

5.4.2 SV Scalar Fields

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

```
GET /Email/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
```

5.4.3 MV Scalar Fields

The following Spider Service object query requests the MV scalar field Tags:

```
GET /Email/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.

5.4.4 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 Spider Service object query:

```
GET /Email/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.

6. 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.

6.1 Aggregate Parameters

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**.

Additional aggregate query parameters are supported by each storage service. How parameters are passed depends on the REST command.

6.2 Aggregate Commands

Each storage service provides REST commands to perform aggregate queries in two different ways.

6.2.1 URI Command

With this command, all parameters are passed as URI query parameters after a question mark (?). Parameters must be separated by ampersands (&) and escaped as required by URI rules. Below is an example REST command for the Spider Service:

```
GET /Email/Message/_aggregate?m=SUM(Size)&f=TOP(3,Sender),TOP(3,TRUNCATE(SendDate,HOUR))
   &q=SendDate >= 2010-01-01
```

This aggregate query selects `Message` objects in the `Email` application. The parameters are:

- `?m`: This **metric** parameter computes the sum of `Size` field values.
- `&f`: This **grouping** parameter groups results first by the top three `Sender` values and then by the top three `SendDate.HOUR` values.

- `&q`: This **query** parameter selects objects whose `SendDate` is greater than or equal to `2010-01-01`.

6.2.2 Entity Command

With this command, all parameters are passed in the request as an entity (message body). The entity can be formatted as XML or JSON document. Below is the same query as above for the Spider Service with parameters passed in an XML message:

```
PUT /Email/Message/_aggregate
```

With a message body such as this:

```
<aggregate-search>
   <metric>SUM(Size)</metric>
   <grouping-fields>TOP(3,Sender),TOP(3,TRUNCATE(SendDate,HOUR))</grouping-fields>
   <query>SendDate &gt;= 2010-01-01</query>
</aggregate-search>
```

As shown, the document uses element names that are analogs to URI parameters. Because some HTTP clients don't allow GET requests with message bodies, the storage services recognize the PUT method for this query, even though it is a query.

6.3 Metric Parameter

6.3.1 Metric Functions

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 *field*, which must be `integer` (or the synonymous `long`).

- **AVERAGE(*field*)**: Computes the average value for the given *field*, which must be `integer`, `long`, 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)
```

6.3.2 Multiple Metric Functions

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.

6.4 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. A wide range of grouping expressions are allowed as described in the following sections.

6.4.1 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 Spider Service REST command:

```
GET /Email/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 /Email/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 /Email/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.

6.4.2 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 Spider Service aggregate query uses a single-valued scalar as the grouping field:

```
GET /Email/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*.

6.4.3 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 /Email/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 /Email/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.

6.4.4 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 Spider Service REST command of a multi-level aggregate query:

```
GET /Email/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.

6.4.5 Special Grouping Functions

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

6.4.5.1 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. Spider Service example:

```
GET /Email/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.

6.4.5.2 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 values—not terms—and must be enclosed in quotes. Groups are generated only for scalar values other that those provided in the list.

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

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

Groups are generated only for the given values; all other values are skipped.

6.4.5.3 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 /Email/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 clause are found, the field passed to `TERMS` is parsed into alphanumeric terms, and a group is created for each unique term. Each contributes 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 /Email/Message/_aggregate?m=COUNT(*)&q=Sender.MessageAddress.Person.Name='Quest Support'
    &f=TOP(5,TERMS(Subject,(sundae tillage)))
```

6.4.5.4 TOP and BOTTOM Functions

By default, all group values are returned at each grouping level. In some cases, this may be a large number of groups. The groups at any level can be limited to those with the *highest* or *lowest* metric values by using the `TOP` and `BOTTOM` functions. These functions *wrap* a grouping field expression, adding a result limit function. For example:

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

This sums the `Size` field of message objects, grouped by recipient name, but it only returns the groups with the three highest `COUNT` values. Note that with `COUNT(*)`, a group can be the "(null)" group. Typical results for the example above in XML:

```
<results>
   <aggregate metric="COUNT(*)" group="TOP(3,Recipients.Person.Name)"/>
   <groups>
      <group>
         <field name="Recipients.Person.Name">(null)</field>
         <metric>3853</metric>
      </group>
      <group>
         <field name="Recipients.Person.Name">Marc Bourauel</field>
         <metric>342</metric>
      </group>
      <group>
         <field name="Recipients.Person.Name">Sergey Ermakov</field>
         <metric>151</metric>
      </group>
   </groups>
   <summary>6030</summary>
   <totalgroups>836</totalgroups>
</results>
```

In JSON:

```
{"results": {
   "aggregate": {"group": "TOP(3,Recipients.Person.Name)", "metric":"COUNT(*)"},
   "groups": [
      {"group": {
         "field": {"Recipients.Person.Name": "(null)"},
         "metric": "3853"
      }},
      {"group": {
         "field": {"Recipients.Person.Name": "Marc Bourauel"},
         "metric": "342"
      }},
      {"group": {
         "field": {"Recipients.Person.Name": "Sergey Ermakov"},
         "metric": "151"
      }}
   ],
   "summary": "6030",
   "totalgroups": "836"
}}
```

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

6.4.5.5 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 format to call the function is:

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

For example:

```
GET /Email/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"
        }},
        ...
    ]
}}
```

6.4.5.6 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.

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

6.5 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"
        }}
    ]
}}
```