Doradus OLAP Database

1. Introduction

This document describes the Doradus OLAP database, which is a Doradus server configured to use the OLAP storage service to manage data. A Doradus OLAP database offers unique performance and storage advantages that benefit specific types of applications. This document describes the unique features of Doradus OLAP, its data model, and its REST commands.

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

- **Doradus Spider Database**: Describes the features, data model extensions, and REST commands specific to the Doradus Spider Database.

- **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.

- **OLAP Database Overview**: An overview of an OLAP database including its architecture, unique features, the types of applications it is best suited for, and an example application schema.

- **OLAP Data Model**: Describes core Doradus data model concepts and extensions specific to Doradus OLAP such as xlinks.

- **Doradus Query Language (DQL)**: A detailed description of DQL: perspectives, clauses, link paths, etc.

- **OLAP Object Queries**: Describes the Object Query type including parameters and output results in XML and JSON.

- **OLAP Aggregate Queries**: Describes the Aggregate Query type including parameters and output results in XML and JSON.

- **OLAP REST Commands**: Describes the REST commands supported by OLAP databases for schema definition, updates, queries, and shard management.

1.1 Recent Changes

The following changes were made to reflect new features for the v2.3 release:

- In object queries, the order parameter (&o) now supports multiple sort fields. Example: `&o=LastName,FirstName`.

- The application schema `key` property is now optional. A key is only needed in a schema change or delete request when the applications was originally defined with a key.

- The transitive function (^) now allows postfix `WHERE` filters as in `Manager^.WHERE(LastName=Smith)`.

- Doradus OLAP provides new REST commands that allow deeper inspection of shard statistics.

- Aggregate queries support new `FIRST` and `LAST` grouping functions, which limit group results based on group names instead of metric values.

- Aggregate queries that use the `TOP`, `BOTTOM`, `FIRST`, or `LAST` functions can be "flattened" by adding the REST parameter `&flat=true`. This allows a more predicable output size for multi-level queries.

- Object queries now support paged queries using the continuation parameters "continue at" (&e) and "continue greater than" (&g).

- When the TRUNCATE function uses precisions coarser than HOUR, the timestamp values returned in groups use a date format (`YYYY-MM-DD`). That is, the time components are omitted.

- In input messages, an XML element can contain characters that are considered illegal in XML by base64-encoding the element content and adding the attribute `encoding="base64"`. In output messages, Doradus performs this encoding automatically.

- The `<schedules>` section of a schema has been eliminated in favor of the application-level option `data-check-frequency`.

- All task types other than `data-aging` have been eliminated, and a single `GET /_tasks` command is now used to query task status.

- Objects can now be added with an explicit `_ID` assigned, in which case Doradus assigns a network-unique ID value.

- Aggregate query grouping expressions can now use the `SETS` function to create groups with arbitrary query expressions. Example: `&f=SETS(Size<10, Tags=Customer, Sender.Person.Department:support)`.

- Object queries can now use the metric parameter (&m) to request one or more metric expressions for each selected object. The metric expressions allowed are the same as used by aggregate queries, including the `AS` clause for relabeling output fields in the query result.

2. Architecture

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

An overview of Doradus architecture is depicted below:
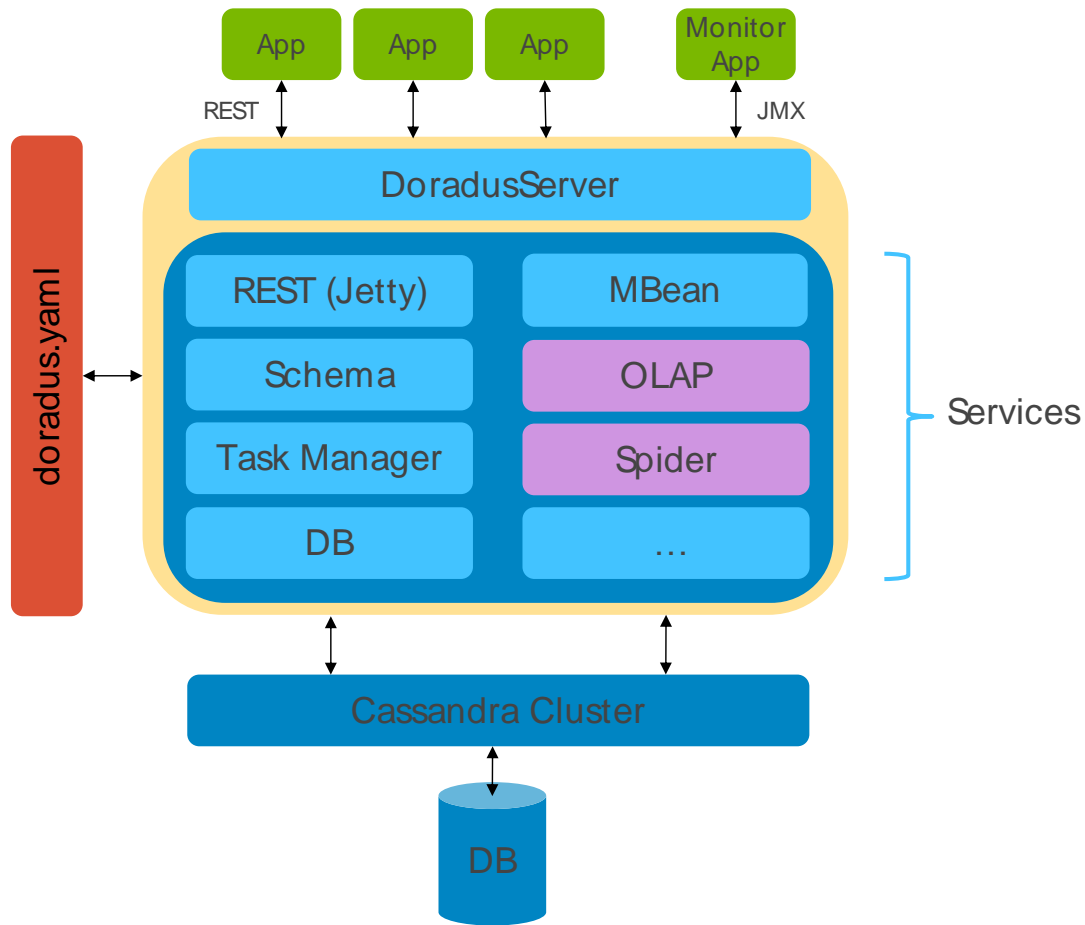
Key components of this architecture are summarized below:

- **Apps**: One or more applications access a Doradus server instance using a simple **REST** API. A **JMX** API is available to monitor Doradus and perform administrative functions.

- **DoradusServer**: This core component controls server startup, shutdown, and services. Entry points are provided to run the server as a stand-alone application, as a Windows service (via procrun), or embedded within another application.

- **Services**: Doradus' architecture encapsulates functions within service modules. Services are initialized based on the server's **doradus.yaml** configuration. Services provide functions such as the **REST** API (an embedded Jetty

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

   o **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.

   o **Spider Service**: A Doradus database configured to use the Spider storage service is termed a *Doradus Spider Database*. The Spider service supports schemaless applications, fully inverted indexing, fine-grained updates, table-level sharding, and other features that support applications that use highly mutable and/or variable data.

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

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

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

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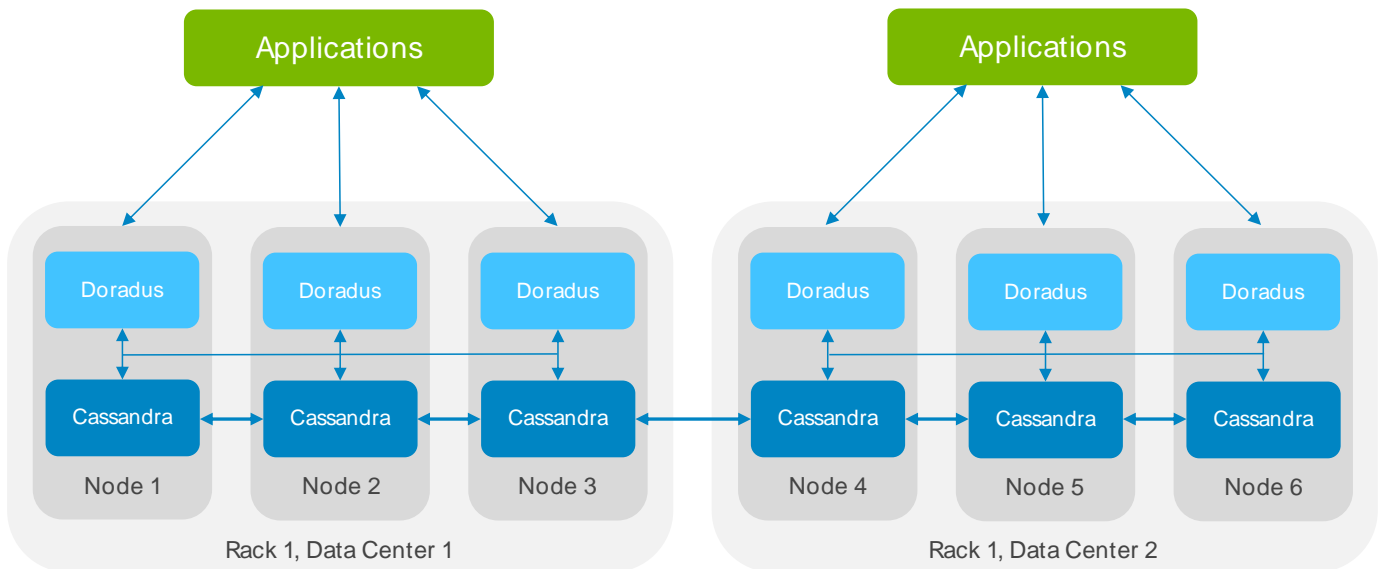
This example demonstrates several deployment features:

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

- Doradus instances are *peers*, hence an application can submit requests to any Doradus instance in the cluster.

- Each Doradus instance is typically configured to use all *network near* Cassandra instances. This allows it to distribute requests to local Cassandra instances, providing automatic failover should a Cassandra instance fail.

- Cassandra can be configured to know which nodes are in the same *rack* and which racks are in the same *data center*. With this knowledge, Cassandra uses replication strategies to balance network bandwidth and recoverability from node-, rack-, and data center-level failures.

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

3. OLAP Database Overview

This section describes the usage and unique features of Doradus OLAP.

3.1 Starting an OLAP Database

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

```
storage_services:
    - com.dell.doradus.service.olap.OLAPService
```

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

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

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

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

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

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

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

  where:

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

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

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

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

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

```
{"Email": {
    "options": {"StorageService": "OLAPService"},
    ...
}
```

3.2 OLAP Overview

Online Analytical Processing is a decision support technology that allows large amounts of data to be analyzed. In traditional OLAP, data from OLTP databases and other sources typically undergoes an *extract/transform/load* (ETL) process, placing it in a *data warehouse* or *data mart* database. This process organizes data into time-oriented *dimension tables* that facilitate subject-based analytical queries. This structure allows a wide range of statistical queries that can compute aggregate results, detect trends, find data anomalies, and perform other analyses.

However, traditional OLAP has numerous drawbacks, including long ETL times, large disk space consumption, and complex, specialized schemas.

Doradus OLAP supports complex analytical queries but employs unique storage and access techniques that overcome drawbacks of traditional OLAP. Some advantages of Doradus OLAP are:

- **Data model**: Applications can use the full Doradus data model, including bi-directional relationships via *link* fields. Doradus provides full referential integrity and bi-directional navigation of link fields.

- **Doradus Query Language**: DQL is used for *object queries*, which retrieve specific objects and their values, and for *aggregate queries*, which perform statistical computations across large object sets. DQL features include *full text searching*, *path expressions*, *quantifiers*, *transitive* relationship searches, *multi-level grouping*, and other advanced search features.

- **Query speed**: Most single-shard object and aggregate queries complete within a few seconds. Multi-shard queries scale linearly to the amount of data being accessed.

- **Space usage**: Doradus stores data in a columnar format that compress very well. In one test, a ~1 billion object OLAP *event* database required only 2GB of disk space.

- **Schema evolution**: An application's schema can be changed at any time, allowing new tables and fields to be added. Automatic data aging is available to expire old data.

- **Load time**: Data is loaded in batches and then *merged* to become visible to the corresponding shard. Load times of 250,000 objects/second or higher per node are typical depending on object complexity.

- **Lag time**: The time required to merge new batches into the live shard is typically between 1 and 30 seconds. This means data is visible to queries is near real time with little time lag.

3.3 When OLAP Works Best

Doradus OLAP works best for applications that fit the following criteria:

- **Partitionable data**: For smaller databases (a few million objects), all data may fit in a single shard. Otherwise, an application will need some criteria on which to divide data into *shards*. Time-based data (events, log entries,

transactions, etc.) is the easiest to partition: for example, each shard holds data from the same hour or day. But other criteria for partitioning will also work.

- **Immutable/semi-mutable data**: Objects can be modified and deleted after they are added to a shard. However, since updates are performed in batches, OLAP is not intended for frequent, fine-grained updates. Ideally, objects are write-once or only occasionally updated.

- **Batchable data**: Data must be added and updated in batches, typically thousands of objects per batch. Load performance degrades with frequent, small-batch updates.

- **Not absolute real time**: Batch updates do not become visible to queries until the containing shard is merged. (Shards can be merged repeatedly, after every batch or after several batches.) Merge time is typically a few seconds to a few minutes, but this means there is a lag between the time data is added and when it is queryable.

- **Emphasis on statistical queries**: The fastest Doradus OLAP queries are single-shard aggregate queries. Multi-shard queries perform proportionally to the number of shards queried. Object queries are similar to aggregate queries but are affected by the number of fields are returned for each object. Full text searching is supported, but it works best for short text fields, not large document bodies. In other words, the primary focus of OLAP is analytics via aggregate queries.

OLAP is not intended for applications with these requirements:

- **Unstructured data**: All tables and fields used in a Doradus OLAP application must be defined in the schema. The schema can evolve over time, but queries are evaluated in context of the most recent schema. Variable fields can be supported with techniques such as a link to a name/value object, but OLAP does not support schemaless applications (like Doradus Spider).

- **OLTP transactions**: Because it requires batch loading and shard merging, OLAP does not work for application that need frequent, fine-grained updates to data.

- **Real time applications**: Because of the lag between the time data is loaded and visible to queries, OLAP does not work for applications that require data to be visible immediately after it is added.

- **Document management**: OLAP doesn't work well for applications that need to store "documents" with large text bodies that are subsequently searched with full text expressions. OLAP supports large text (and binary) fields, but text fields are not pre-indexed with *term vectors* like Doradus Spider. Instead, full text searches dynamically tokenize each text field, which is slower for numerous, large text fields.

3.4 The `Email` Sample Application

To illustrate features, a sample OLAP application called `Email` is used. The `Email` application schema is depicted below:
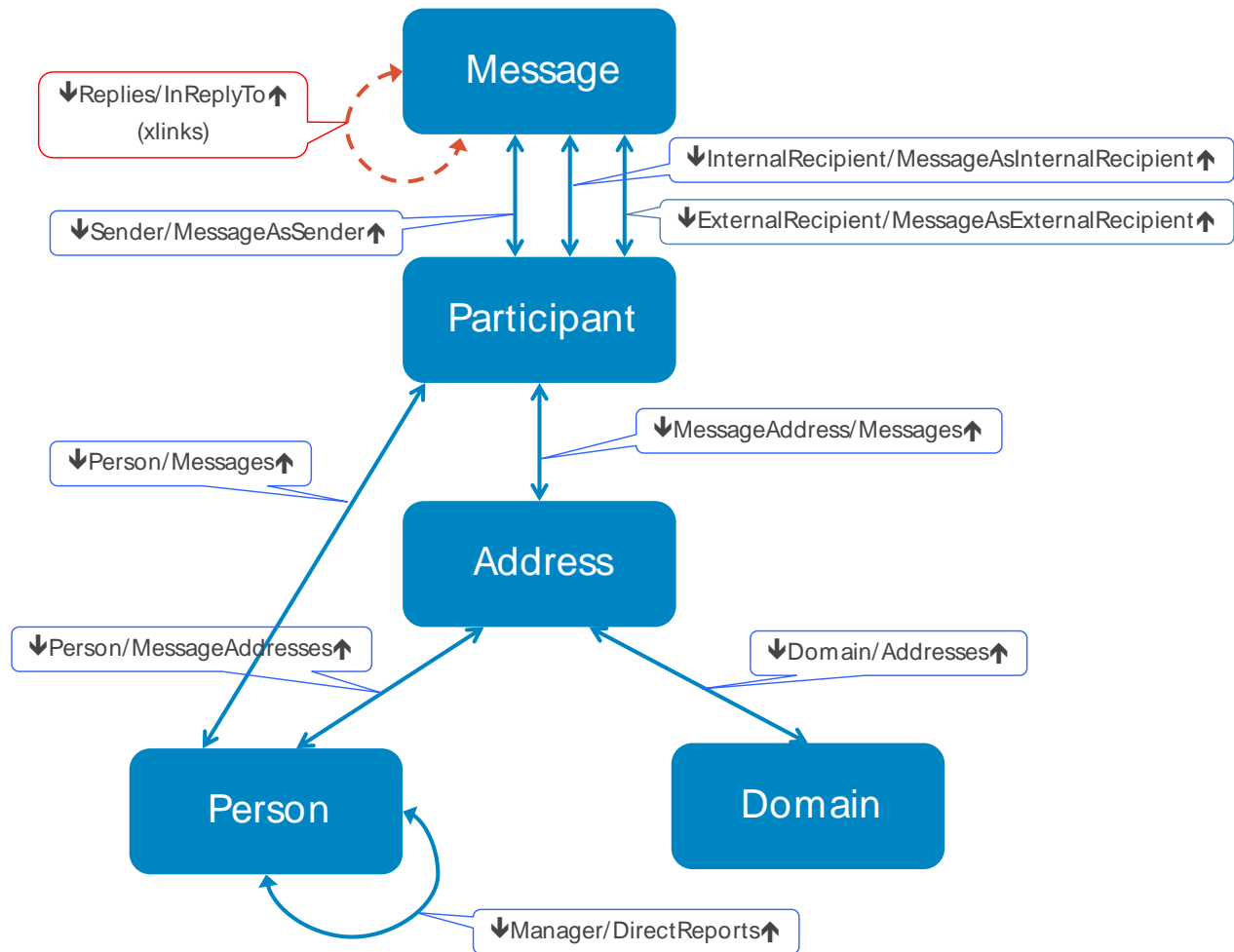
The tables in the `Email` application are used as follows:

- `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.

`Message` objects are stored in a shard named `YYYY-MM-DD` based on their `SendDate`. The related `Participant`, `Address`, `Person`, and `Domain` objects are stored in the same shard. This means, for example, that `Person` and `Domain` objects are replicated to every shard in which they are referenced.

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

```xml
<application name="Email">
    <key>EmailKey</key>
    <options>
        <option name="StorageService">OLAPService</option>
    </options>
    <tables>
        <table name="Message">
            <fields>
                <field name="InReplyTo" type="XLINK" table="Message" inverse="Responses"
                    junction="ThreadID"/>
                <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="Responses" type="XLINK" table="Message" inverse="InReplyTo"
                    junction="_ID"/>
                <field name="SendDate" type="TIMESTAMP"/>
                <field name="Size" type="INTEGER"/>
                <field name="Subject" type="TEXT"/>
                <field name="Tags" collection="true" type="TEXT"/>
                <field name="ThreadID" 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>
    <schedules>
        <schedule type="data-aging" value="0 0 3 * *"/>
    </schedules>
</application>
```

The same schema in JSON is shown below:

```
{"Email": {
    "key": "EmailKey",
    "options": {"StorageService": "OLAPService"},
    "tables": {
        "Message": {
            "fields": {
                "InReplyTo": {"type": "XLINK", "table": "Message", "inverse": "Responses", "junction":
                    "ThreadID"},
```

```
        "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"}
                    }
                }
            }
        },
        "Responses": {"type": "XLINK", "table": "Message", "inverse": "InReplyTo", "junction":
            "_ID"},
        "SendDate": {"type": "TIMESTAMP"},
        "Size": {"type": "INTEGER"},
        "Subject": {"type": "TEXT"},
        "Tags": {"collection": "true", "type": "TEXT"},
        "ThreadID": {"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"}
            }
        }
    },
    "schedules": [
        {"schedule": {"type": "data-aging", "value": "0 0 3 * *"}}
    ]
}}
```

Some highlights of this schema:

- The application-level option `StorageService` explicitly assigns the application to the `OLAPService`.

- The `Message` table contains a group field called `Participants`, which contains the link field `Sender` and a second-level group called `Recipients`, which contains the links `InternalRecipients` and `ExternalRecipients`.

- The `Message` table contains a reflexive, cross-shard relationship formed by the xlinks `InReplyTo` and `Responses`.

- The `Message` table defines an alias called `$SalesEmails`, which is assigned the expression `Sender.Person.WHERE(Department:Sales)`. `$SalesEmails` is dynamically expanded when used in DQL queries.

- The application defines a `data-aging` task, assigning its schedule the cron expression "`0 3 * * *`", which means "every day at 03:00".

4. OLAP Data Model

Doradus OLAP extends the core Doradus data model with unique features advantageous to OLAP applications. This section describes the core data model and the extended OLAP features.

4.1 Motivation

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

4.2 Identifiers

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

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

4.3 Application

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

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

- **Key**: A user-defined string that acts as a secondary identifier. The key is optional and acts as an extra safety mechanism. In application's schema is defined with a key, the same key must be provided to modify the schema or delete the application.

- **Options**: The following application-level options can be defined:

  o **StorageService**: defines the storage service that will manage the application's data.

  o **aging-check-frequency**: Specifies the data aging check frequency for the application. The value of the `aging-check-frequency` must be in the form "<number> <units>" where <number> is a positive integer and <units> is `MINUTES`, `HOURS`, or `DAYS`. (Singular forms of these mnemonics are also allowed.) At the specified frequency, a background task looks for and deletes expired shards.

- **Tables**: Tables and their fields that the application owns.

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>
</application>
```

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

```
{"Email": {
    "key": "EmailKey",
    "options": {
        // options
    },
    "tables": {
        // table definitions
    }
}}
```

4.4 Table

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

A table can include the following components:

- **Options**: 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
        }
    },
    ...
}
```

4.5 Objects

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

4.6 Object IDs

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

- IDs are variable length. Objects in the same table can have different ID lengths as long as every value is unique.

- User applications set an object's ID by assigning the `_ID` field when the object is created. It is the application's responsibility to generate unique ID values. If two objects are added with the same ID, they are *merged* into a single object. That is, one "add" is treated as an "update" of an existing object.

- To Doradus, IDs are Unicode text strings. To use binary (non-Unicode) ID values, user applications must convert them to a Unicode string, e.g., using hex or base64 encoding.

- When IDs are included in XML or JSON messages or in URIs, some characters may need to be escaped. For example, the base64 characters '+' and '/' must be converted to `%2B` and `%2F` respectively in URIs. IDs returned by Doradus in messages are always escaped when needed.

4.7 Fields

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

4.7.1 Scalar Fields

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

- **Text**: An arbitrary length string of Unicode characters.

- **Boolean**: A logical *true* or *false* value.

- **Integer** and **Long**: A signed 64-bit integer value. These two types are synonyms.

- **Float**: A 32-bit floating-point value.

- **Double**: A 64-bit floating-point value.

- **Timestamp**: A date/time value with millisecond precision. Doradus treats all timestamp values as occurring in the UTC time zone. Timestamp values must be given in the format:

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

  where:

  - `yyyy` is a 4-digit year between 0000 and 9999
  - `MM` is a 1- or 2-digit month between 1 and 12
  - `dd` is a 1- or 2-digit day-of-month between 1 and 31
  - `HH` is a 1- or 2-digit hour between 0 and 23
  - `mm` is a 1- or 2-digit minute between 0 and 59
  - `ss` is a 1- or 2-digit second between 0 and 59
  - `SSS` is a 1-to-3 digit millisecond between 0 and 999

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

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

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

When type names are used in schema definitions, they are case-insensitive (e.g., `integer` or `INTEGER`). Example field definitions in XML are shown below:

```xml
<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 setting its `collection` property to `true`. Example:

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

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

4.7.2 Link Fields

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

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

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

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

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

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

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

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

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

4.7.3 Group Fields

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

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

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

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

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

4.8 Aliases

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

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

Below is an example alias declaration:

```
<table name="Message">
```

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

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

```
GET /Email/Message/_query?q=$SalesEmails.LastName:powell&range=0
```

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&range=0
```

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

4.9 OLAP Sharding Model

OLAP employs a *sharding* model that partitions data into named *shards*. The sharding strategy controls how data is loaded and queried.

A shard is a data partition. It is analogous to an OLAP *cube*, containing data that is organized into queryable *dimensions*. A shard is an application-level partition: every object is assigned to a shard, and a shard holds objects from all tables. Data is stored in arrays on a per-shard/per-field basis. Queries define one or more shards as their query scope. When a field is accessed by a query for a specific shard, the corresponding array is loaded into memory, typically in one I/O. Accessing field values is very fast: typically millions of values/second. Arrays are cached on an LRU basis.

A shard holds objects that are related by user-defined criteria. The most common criteria is time: for example, objects that were created in the same hour or day. But other criteria can be used such as geography or department. The goal of each shard is to hold "a few million" objects that balances load/merge time with memory usage. (Experimentation is highly encouraged.) A shard's name should reflect its contents, e.g., "2014-01-03" for a shard that holds data from January 3rd, 2014. Shards should use alphabetically-ordered names since queries can operate on shard name ranges (e.g., "2014-01-01" to "2014-01-31").

Objects are loaded into a shard in *batches*. A batch contains new, modified, and deleted objects in any order for any/all tables. Batches are intended to be large: typically thousands of objects. When one of more batches are loaded, the shard is *merged* to apply all updates. Once merged, a shard's updates are visible to queries. After a merge, a shard can receive more batches and be merged again.

A given object ID can be inserted into the same table in multiple shards. The duplicates may be identical or they may reflect the state of a given person, account, etc. over time. Duplicating objects between shards is desirable so that shards are self-contained graphs.

A shard can be explicitly deleted, which removes all of its data. Alternatively, each shard can be assigned an *expire-date* and will be automatically deleted by a background data-aging task when it expires.

An example of the OLAP sharding model is shown below:
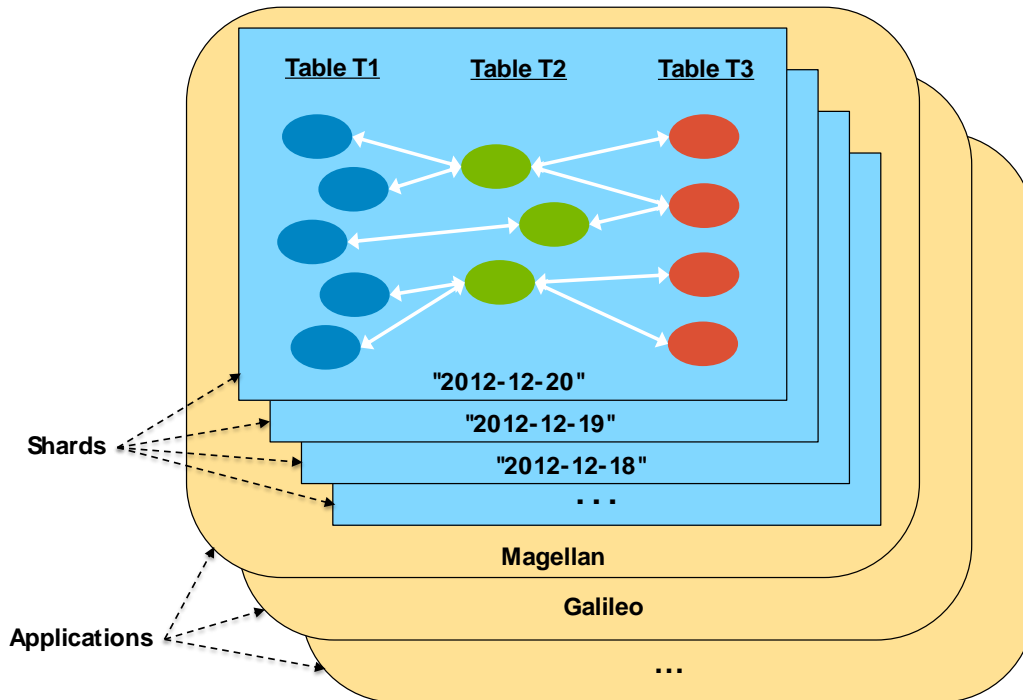
In this example, the database holds multiple applications (`Magellan`, `Galileo`.) Each application holds its own shards: The `Magellan` application's shards are named with dates ("`2012-12-20`", "`2012-12-19`", …). Each shard holds objects from all tables and are interrelated using links. Note that it is not necessary for an application to use multiple tables or links in order to take advantage of OLAP: a single table with only scalar fields is all some applications need.

Although data is organized into per-field/per-shard arrays, this structure is hidden from applications. Objects are queried using DQL object and aggregate queries. The only extra parameter required for queries is the shard list or range that defines the query's scope.

4.10 Cross-shard Relationships: Xlinks

With Doradus OLAP, links can only reference objects in the <u>same</u> shard. A link such as `Person.Manager` cannot reference an object in another shard. (In fact, setting `Person.Manager` to a given object ID implicitly creates the inverse object in the same shard if it does not already exist.) This means that some objects may need to be duplicated in multiple shards so that each shard is a complete graph, allowing queries to work efficiently. Because OLAP stores data compactly, the duplication is worthwhile since same-shard link path evaluation is extremely fast.

However, sometimes object duplication is not practical, and relationships must span shards. For example, suppose we want to track messages in the same conversation *thread*. Since replies and forwards can be sent at any future date, messages in the same thread may reside in any shard. We could add a scalar field `ThreadID` that identifies messages in the same thread and query for a given value across shards. For some scenarios, this may be sufficient.

But in some cases, we may want to treat the object relationships in a way that allows us to use link paths. For these cases, Doradus OLAP supports a cross-shard field type called an *xlink*.

Xlinks are similar to regular links: a pair of xlinks are defined in the schema as inverses of each other, forming a bi-directional relationship. However, xlinks are not explicitly assigned: relationships are *implicitly* formed via foreign keys called *junction* fields. In its definition, each xlink identifies its junction field, which is a text field whose values point to

related objects, which reside in the same and/or other shards. An example for connecting objects in a message thread is shown below:

```
<table name="Message">
    <fields>
        <field name="ThreadID" type="Text"/>
        <field name="InReplyTo" type="XLINK" table="Message" inverse="Responses"
            junction="ThreadID"/>
        <field name="Responses" type="XLINK" table="Message" inverse="InReplyTo"
            junction="_ID"/>
        ...
    </fields>
</table>
```

Here is how the xlinks `InReplyTo` and `Responses` work:

- The `_ID` of a *root* message that begins a new conversation thread is used as the thread ID.

- When a new message is created that is <u>not</u> part of another thread, we set its `ThreadID` to its own `_ID`. That is, every root message is the initially the only member of its own thread.

- When other messages are created (replies or forwards) in the same message thread, we set their `ThreadID` to the root message's `_ID`, even if the root message resides in another shard.

- We can then traverse `Message.Responses` to navigate from the root message to other messages in the same thread. To do this, Doradus takes the root message's `_ID` (because it is the junction field for `Responses`) and searches for messages in other shards whose `ThreadID` matches (because it is the junction field for the inverse link, `InReplyTo`).

- Similarly, we can traverse `Message.InReplyTo` to navigate from any message back to the root message. In this case, Doradus takes the message's `ThreadID` and searches for another message with a matching `_ID`.

One consideration used in this example is *shard merging*. In an OLAP database that uses time-oriented shards, we generally want to add data to <u>new</u> shards, which are then merged. We don't want to modify data in older shards if possible because this requires extra merging. In the example above, message threads are formed by simply setting the `ThreadID` of newer messages. Older messages in the thread, including the root message, are never modified, hence we don't need to merge older shards.

Although xlinks are similar to regular links, there are differences in how they are declared and used:

- The inverse of an xlink must also be an xlink. In the example above, `Responses` and `InReplyTo` are inverses. Although these xlinks both belong to `Message`, in general xlinks can relate objects between any tables.

- Each xlink identifies a *junction* field, which must be a text field belonging to the same table or the `_ID` field. The junction field is a *foreign key* to related objects. In a given relationship, at least one xlink must `_ID` field as its junction field. If the junction field is not explicitly defined, it defaults to the `_ID` field.

- One xlink can use a text field as its junction field. This is the normal practice for most use cases. In the example above:

- The xlink `InReplyTo` defines `ThreadID` as its junction field. This means an object is related via `InReplyTo` to the message(s) whose `_ID` matches its `ThreadID`.

- The xlink `Responses` uses `_ID` as its junction field. This means an object is related via `Responses` to the message(s) whose `ThreadID` matches its `_ID`.

- If both xlinks in a relationship use `_ID` as their junction field, each object is related to objects with the same object ID. This is allowed even if the xlinks are defined in different tables.

- A xlink's junction field can be an MV text field, thereby allowing the xlink to refer to multiple objects in each shard.

Xlinks form *soft* relationships, hence no referential integrity is assured. When a junction field is assigned a value, there may or may not exist any foreign objects with a matching value. Likewise, if two objects are related, the relationship may be broken by altering the junction field value, deleting one of the objects, or shard aging. Traversing an xlink whose junction field doesn't match any foreign objects acts as if the xlink is null.

In aggregate queries, xlinks can be used anywhere regular links are used: query expressions, aggregate grouping expressions, and metric expressions. Doradus OLAP searches the shards defined by the `shards` or `range` parameter for *perspective* objects, and it searches shards defined by the `xshards` or `xrange` parameter for objects related via xlinks. For example:

```
GET /Email/Message/_aggregate?m=COUNT(*)&q=_ID=XYZ&shards=2014-01-01&xrange=2014-01-01
    &f=Responses.Sender.Person.Department&range=0
```

This query counts the messages in the thread rooted by the message with `_ID=XYZ`, grouped by the `Sender.Person.Department` of each response. Only shard `2014-01-01` is searched for the root message; all shards named `2014-01-01` or greater are searched for objects related to the xlink `Responses`. See the section **Doradus Query Language (DQL)** for more details on query parameters.

Using xlinks in queries is slower than regular links. Consequently, they should be used only in those cases where normal links are not feasible.

4.11 Shard Aging

When at least one shard is assigned an expiration date, Doradus OLAP automatically creates a background `data-aging` task that checks for and deletes expired segments once per day. Optionally, the schedule of an application's `data-aging` task can be explicitly controlled in the schema via the `aging-check-frequency` application option.

5. Doradus Query Language (DQL)

The Doradus query language (DQL) is used in object and aggregate queries. DQL is analogous to full text languages used by search engines such as Lucene and from which DQL borrows concepts such as *terms*, *phrases*, and *ranges*. To these, DQL adds *link paths*, *quantifiers*, and a *transitive* function to support graph-based searches.

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

5.1 Query Perspective

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

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

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

5.2 Clauses

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

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

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

5.3 Clause Negation: NOT

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

5.4 Clause Connectors: AND, OR, and Parentheses

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

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

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

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

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

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

This is evaluated as if the following parentheses were used:

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

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

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

5.5 Literals

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

```
Size=Foo
```

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

The types of literals supported by Doradus are described below.

5.5.1 Booleans

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

5.5.2 Numbers

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

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

5.5.3 Timestamps

Timestamp literals have the general form:

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

Notes:

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

5.5.4 Terms

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

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

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

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

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

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

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

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

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

5.5.5 Strings

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

```
Name=Smith
```

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

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

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

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

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

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

- `\x`: Where `x` can be one of these characters: `t` (tab character), `b` (backspace), `n` (newline or LF), `r` (carriage return), `f` (form feed), `'` (single quote), `"` (double quote), or `\` (backslash).

- `\uNNNN`: This escape sequence can be used for any Unicode character. `NNNN` must consist of four hex characters 0-F.

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

5.6 Null Values

A field that has no value for a given object is *null*. Doradus treats null as a "value" that will not match any literal. For example, the clause `Size=0` will be false if the `Size` field is null. This means that `NOT(clause)` will be true if *clause* references a null field.

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

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

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

5.7 IS NULL Clause

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

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

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

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

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

5.8 Text *Contains* Clauses

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

5.8.1 Term Clauses

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

```
Name:Smith
```

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

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

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

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

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

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

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

Matching objects must have `John` in the `Name` field and `Smith` in <u>any</u> field. In OLAP applications, this query isn't allowed because the second "all fields" clause type is not supported.

5.8.2 Phrase Clauses

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

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

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

5.9 Range Clauses

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

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

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

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

This is shorthand for:

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

Text fields can also use bracketed range clauses:

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

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

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

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

5.10 Equality Clauses

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
NOT Size=1024
```

5.11 IN Clause

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

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

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

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

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

is the same as the following `OR` clauses:

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

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

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

5.12 Timestamp Clauses

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

5.12.1 Subfield Clauses

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

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

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

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

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

5.12.2 NOW Function

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

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

The basic formats supported by the NOW function are:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.12.3 PERIOD Function

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

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

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

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

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

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

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

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

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

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

```
LASTMINUTE
LASTHOUR
LASTDAY
LASTWEEK
LASTMONTH
LASTYEAR
```

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

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

Example timestamp clauses using the PERIOD function are shown below:

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

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

\<range\> Mnemonic	Timestamp Range	Comments
THISMINUTE	["2013-12-04 01:24:00" TO "2013-12-04 01:25:00"}	Same year, month, day, hour, and minute as now.
LASTMINUTE	["2013-12-04 01:23:35" TO "2013-12-04 01:24:35"]	Within the last minute (60 seconds).
THISHOUR	["2013-12-04 01:00:00" TO "2013-12-04 02:00:00"}	Same year, month, day, and hour as now.
LASTHOUR	["2013-12-04 00:24:35" TO "2013-12-04 01:24:35"]	Within the last hour (60 minutes).
TODAY	["2013-12-04 00:00:00" TO "2013-12-05 00:00:00"}	Same year, month, and day as now.
LASTDAY	["2013-12-03 01:24:35" TO "2013-12-04 01:24:35"]	Within the last day (24 hours).
THISWEEK	["2013-12-02 00:00:00" TO "2013-12-09 00:00:00"}	Same week as now (based on ISO 8601).
LASTWEEK	["2013-11-27 01:24:35" TO "2013-12-04 01:24:35"]	Within the last 7 days.
THISMONTH	["2013-12-00 00:00:00" TO "2014-01-01 00:00:00"}	Same year and month as now.
LASTMONTH	["2013-11-04 01:24:35" TO "2013-12-04 01:24:35"]	Within the last calendar month.
THISYEAR	["2013-01-01 00:00:00" TO "2014-01-01 00:00:00"}	Same year as now.
LASTYEAR	["2012-12-04 01:24:35" TO "2013-12-04 01:24:35"]	Within the last year.

5.13 Link Clauses

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

```
Manager=def413
```

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

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

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

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

5.13.1 Link Path Clauses

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

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

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

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

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

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

In order, these examples are interpreted as follows:

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

5.13.2 WHERE Filter

Sometimes we need multiple selection clauses for the objects in a link path, but we need the clauses to be "bound" to the same instances. To illustrate this concept, consider this query: Suppose we want to find messages where an internal recipient is within the R&D department <u>and</u> 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'&range=0
```

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'&range=0
```

Now the problem is that the query won't select messages that have one or more internal recipients who are <u>not</u> 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')
    &range=0
```

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')&range=0
```

5.13.3 Outer WHERE Filter

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

```
GET /Email/Person/_query?q=WHERE(Department:sales)&range=0
```

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

```
GET /Email/Person/_query?q=Department:sales&range=0
```

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

```
GET /Email/Person/_query?q=WHERE(Department:sales).WHERE(Office:aliso)&range=0
```

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

```
GET /Email/Person/_query?q=Department:sales AND Office:aliso&range=0
```

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

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

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

```
GET /Email/Person/_query?q=$SalesPeople&range=0
GET /Email/Person/_query?q=$SalesPeople.WHERE(Office:aliso)&range=0
GET /Email/Message/_query?q=Sender.Person.$SalesPeople&range=0
```

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

Outer `WHERE` filters can also be used in aggregate query metric functions and grouping parameters (described elsewhere).

5.14 Quantifier Functions

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

5.14.1 Overview: ANY, ALL, and NONE

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

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

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

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

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

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

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

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

values, `NONE(C)` is true for that B; if a given A has no B values, `ANY(B)` is false for that A; if a given object P has no A values, `ALL(A)` is false for that P.

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

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

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

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

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

5.14.2 Quantifiers on MV Scalar Fields

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

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

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

5.14.3 Quantifiers on SV Scalar Fields

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

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

5.14.4 Quantifiers on Link Fields

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

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

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

5.14.5 Quantifiers with IS NULL

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

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

This clause is true for a perspective object if:

- `InternalRecipients` is null, or
- `Person` is null for every `InternalRecipients` object, or
- `LastName` is null for ever `Person` object.

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

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

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

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

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

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

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

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

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

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

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

are selected by the clause:

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

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

5.15 Transitive Function

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

```
DirectReports.Name = Fred
```

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

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

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

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

- A cycle is detected: If an object is found that was already visited in evaluating the `DirectReports^` expression, that object is not evaluated again.

- A null is detected: The recursion stops at the "edges" of the graph, in this case someone with no direct reports.

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

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

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

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

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

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

The transitive function can also be used with WHERE filter clauses, as long as the filter appears after the transitive operator. For example:

```
Manager^.WHERE(LastName=Smith).FirstName : Do*
```

This query searches "up" the org chart for all `Manager` objects. Only managers whose `LastName` is `Smith` and whose `FirstName` begins with the letters `do` are selected.

5.16 COUNT(<link path>)

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

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

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

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

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

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

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

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

This clause returns true for objects that have at least one direct report whose last name is `Smith`. Note that the `WHERE` expression is specified immediately after the link name to be filtered. The parameter to the `WHERE` function is an expression relative to the link field. In this example, `LastName` must be a scalar field belong to `DirectReports`' extent table.

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

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

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

6. OLAP Object Queries

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

6.1 Query Parameters Overview

An OLAP 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.

- **Metrics** (optional): A comma-separated list of metric expressions to be computed for each object. This parameter essentially extends the fields parameter with additional computed values. See the section **Metric Parameter** for more information.

- **Order** (optional): Orders the objects returned by one or more fields. Without this parameter, objects are returned in an internally-defined order. When a single order field is specified, by default objects are sorted in ascending order 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. Multiple order fields can be specified to sort objects by primary values followed by secondary values. See the section **Order Parameter** for more information.

- **Skip** (optional): Causes the specified number of objects to be skipped. Without this parameter, the first page of objects is returned.

- **Continuation** (optional): Causes a secondary set of objects to be returned. There are two forms for this parameter, which can only be used with unsorted queries.

- **Shards** (this or **Range** is required): Specifies a list of shards to be searched by the query.

- **Range** (this or **Shards** is required): Specifies a range of shards to be searched by the query.

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

- **URI command**: All query parameters are passed in the URI of the command.

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

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

6.2 Fields Parameter

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

6.2.1 Basic Formats

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

- `*`: Using the value "*" (without quotes) is a synonym for omitting the fields parameter. All scalar fields are returned for each object.

- `_all`: This keyword requests all scalar fields of each perspective object and the scalar fields of each first-level object connected via a link value.

- `_local`: This keyword requests all scalar field values and the `_ID` value of all link fields for each object.

- `<scalar field>`: When a scalar field is requested, its value (SV) or values (MV), if any, are returned.

- `<link field>`: When a link field is requested, the _ID of each linked object is returned.

- `<group field>`: When a group field is requested, all *leaf* scalar and link fields within the group and its nested groups, recursively, are returned.

- `f1,f2,f3`: The fields parameter can be a comma-separated list of field names. This example is a simple list of names: only fields f1, f2, and f3 are returned. Each field can be a scalar, link, or group field.

- `f1,f2.f3,f2.f4.f5`: This example uses *dotted notation* to define link paths. This example requests three fields: `f1`, which belongs to the perspective object; `f3`, which is connected to the perspective object via link `f2`; and `f5`, which is related via links `f2` to `f4`. When dotted notation is used, each field to the left of a dot must be a link belonging to the object on its left.

- `f=f1,f2(f3,f4(f5))`: This is the same example as above using *parenthetical qualification* instead of dotted notation. Parenthetical and dotted notations can be mixed within a single fields parameter list. Example: `f=Name,Manager(Name,Manager.Name)`.

- `f1[s1],f2[s2].f3[s3],f2.f4[s4].f5`: This is a dotted notation example where *size limits* are placed on specific link fields. If `f1` is a link field, the number of `f1` values returned for each object will be no more than `s1`, which must be a number. Similarly, link `f2` is limited to `s2` values per object; `f3` is limited to `s3` values for each `f2` value; and `f4` is limited to `s4` values per `f2` value. Since it has no bracketed parameter, field `f5` is not limited in the number of values returned. Link size limits are explained further later.

- `f=f1[s1],f2[s2](f3[s3],f4[s4](f5))`: This is the same *size limits* example as above using parenthetical qualification instead of dotted notation. Link size limits are explained further later.

6.2.2 WHERE Filters

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

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

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

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

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

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

6.2.3 Link Field Size Limits

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 <u>after</u> 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 <u>after</u> a link that uses a WHERE filter. (Using purely dotted notation, the WHERE condition would have to be repeated.)

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

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

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

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

6.2.4 Link Field AS Aliases

By default, when an object query returns link fields, the link field name and filter, if present, are used in the query result. For example, in the following parameter:

```
.../_query?f=InternalRecipients.WHERE(Person.LastName:wright)&q=...
```

The query results identifies the values belonging to the InternalRecipients link as follows:

```
<doc>
    <field name="_ID">D+EGzQxdH1jWk9CCZqts1Q==</field>
    <field name="_shard">s1</field>
    <field name="InternalRecipients.WHERE(Person.LastName:wright)">
        <doc>
            <field name="_ID">w1snUDxw3pZHh5yr55oHBA==</field>
        </doc>
    </field>
</doc>
```

The expanded link field name can be simplified by using the AS function, as shown below:

```
.../_query?f=InternalRecipients.WHERE(Person.LastName:wright).AS(Wright)&q=...
```

The `AS` function must appear after the `WHERE` filter and/or link size limit, if any. The parameter to the `AS` function can be an unquoted term or a quoted string. The parameter value defines a field name *alias* that is used instead of the default link field name in the query results. For example:

```
<doc>
    <field name="_ID">D+EGzQxdH1jWk9CCZqts1Q==</field>
    <field name="_shard">s1</field>
    <field name="Wright">
    <doc>
        <field name="_ID">w1snUDxw3pZHh5yr55oHBA==</field>
    </doc>
    </field>
</doc>
```

Multiple `AS` functions can be used in the same object query. However, alias names used at the same level must be unique so that each field name in the output results is unique.

The AS function can be useful to identify query results when table-level alias is used. For example, assume the following alias definition:

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

The alias `$SalesPeople` can be used as in a fields parameter, potentially with additional filtering, as in the following example:

```
GET /Email/Message/_query?q=Size>100000&range=0
    &f=Sender.Person.$SalesPeople.WHERE(Office:aliso).LastName
```

By default, the query results for the `Person` link includes the filters in both the alias definition and the &f parameter. For example, it will appear within the `Sender` group element as follows:

```
<field name="Sender">
    <doc>
        <field name="_ID">Klk8VEzG2ZBQMLibi/t4Ig==</field>
        <field name="Person.WHERE((Department:sales) AND (Office:aliso))">
            <doc>
                <field name="LastName">McKeehan</field>
                <field name="_ID">tNI90WCArcnd76jEeEWmxg==</field>
            </doc>
        </field>
    </doc>
</field>
```

The AS function can apply an alias that makes the link element easier to identify. So, for example:

```
GET /Email/Message/_query?q=Size>100000&range=0
    &f=Sender.Person.$SalesPeople.WHERE(Office:aliso).AS("Sales Person").LastName
```

This causes the results to appear as follows:

```
<field name="Sender">
    <doc>
        <field name="_ID">Klk8VEzG2ZBQMLibi/t4Ig==</field>
        <field name="Sales Person">
            <doc>
                <field name="LastName">McKeehan</field>
                <field name="_ID">tNI90WCArcnd76jEeEWmxg==</field>
            </doc>
        </field>
    </doc>
</field>
```

6.3 Metric Parameter

In an object query, the metric parameter returns one or more metric computations for each selected object. These computed values are in addition to the field values specified by the **fields** parameter. The syntax for the metric parameter is the same as in aggregate queries: it is a comma-separated list of metric expressions. Each metric expression is an algebraic expression using metric functions, arithmetic operators, and parentheses. See the section **Metric Parameter** for aggregate queries for full details. Below is an example object query with a single metric function:

```
GET /Email/Person/_query?shards=s1&m=COUNT(Messages.MessageAsSender)&f=FirstName,LastName
```

This query selects all `Person` objects and returns their `FirstName` and `LastName` fields. Additionally, the metric parameter returns the `COUNT` of all `Messages.MessageAsSender` values for each object. The metric expression result is returned as an additional field element whose name equals the metric expression text. Example:

```
<results>
    <totalobjects>185</totalobjects>
    <docs>
        <doc>
            <field name="COUNT(Messages.MessageAsSender)">3</field>
            <field name="FirstName">Chris</field>
            <field name="LastName">Rudd</field>
            <field name="_ID">+IBGjRcg4dMXgu3l7aOZXg==</field>
            <field name="_shard">s1</field>
        </doc>
        <doc>
            <field name="COUNT(Messages.MessageAsSender)">9</field>
            <field name="FirstName">Scott</field>
            <field name="LastName">Reasoner</field>
            <field name="_ID">+eVQDWAtRBSjWc6DHYVtTg==</field>
            <field name="_shard">s1</field>
        </doc>
        ...
    </docs>
<results>
```

The field name used for the metric function can be customized for easier recognition using the AS syntax. Example:

```
GET /Email/Person/_query?shards=s1&m=COUNT(Messages.MessageAsSender) AS 'Messages Sent'
    &f=FirstName,LastName
```

The name following AS can be a simple term or a quoted string.

Multiple expressions can be computed by the metrics parameter by separating each expression by a comma. For example, suppose we want to know the total number of messages sent by each person and the total size of all sent messages. The following query could be used:

```
GET /Email/Person/_query?range=0&f=FirstName,LastName
    &m=COUNT(Messages.MessageAsSender) AS TotalMessages,
        SUM(Messages.MessageAsSender.Size) AS TotalSize
```

Below is an example where a metric computation uses an algebraic expression to compute a more complex result:

```
GET /Email/Person/_query?range=0&f=FirstName,LastName
    &m=COUNT(DirectReports^.WHERE(LastName:smith)) + COUNT(Manager^.WHERE(Department:support))
```

In addition to FirstName and LastName, this query returns for each person the sum of (1) the count of all DirectReports transitively ("down" the org chart) whose LastName contains the term "smith", and (2) the count of all Managers transitively ("up" the org chart) whose Department contains the term "support".

6.4 Order Parameter

The order parameter sorts the perspective objects by one or more fields. In the simplest case, a scalar field belonging to the perspective table is chosen. Example:

```
GET /Email/Message/_query?q=Tags=Customer&range=0&o=Size
```

This sorts selected messages by Size in ascending order. The suffix ASC can be added after the sort field to explicitly request ascending order. Alternatively, the suffice DESC can be added to request descending sort order:

```
GET /Email/Message/_query?q=Tags=Customer&range=0&o=Size+DESC
```

If a sort field is MV, the smallest (ascending) or largest (descending) value for each object is chosen to decide the object's sort position.

The order parameter can also use multiple sort fields, as in this example:

```
GET /Email/Person/_query?q=Department:support&range=0&o=LastName,FirstName+DESC
```

This query sorts selected people by ascending LastName and then by descending FirstName.

A sort field can also belong to an object linked to the perspective table. When an extended field is used for sorting, the same field should also be requested in the query's field list. Example:

```
GET /Email/Message/_query?q=NOT Sender.Person.Name IS NULL&range=0
    &o=Sender.Person.Name&f=Sender.Person.Name
```

This query returns all messages where `Sender.Person.Name` is not null, sorted by the sender's name, which is also returned.

6.5 Query Results

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

6.5.1 Empty Results

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

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

In JSON:

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

6.5.2 SV Scalar Fields

This object query requests all scalar fields of `Person` objects whose `LastName` is `Garn` in shard `2014-06-18`:

```
GET /Email/Person/_query?q=LastName:Garn&f=*&shards=2014-06-18
```

In XML, the result looks like this:

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

```
            <field name="Name">Doug Garn</field>
            <field name="Office">Aliso Viejo 5</field>
            <field name="_ID">m1yYabbtytmjw+e8OCz1dg==</field>
        </doc>
    </docs>
</results>
```

In JSON:

```
{"results": {
    "docs": [
        {"doc": {
            "Department": "Field Sales",
            "FirstName": "Chris",
            "LastName": "Garn",
            "Name": "Chris Garn",
            "Office": "Aliso Viejo 5",
            "_ID": "07Z094KNjmEsqMoV/yNI0g=="
        }},
        {"doc": {
            "Department": "Sales Operations",
            "FirstName": "Jim",
            "LastName": "Garn",
            "Name": "Jim Garn",
            "Office": "Aliso Viejo 5",
            "_ID": "kUNaqNJ2ymmb07jHY9OPOw=="
        }},
        {"doc": {
            "Department": "Admin",
            "FirstName": "Doug",
            "LastName": "Garn",
            "Name": "Doug Garn",
            "Office": "Aliso Viejo 5",
            "_ID": "m1yYabbtytmjw+e8OCz1dg=="
        }}
    ]
}}
```

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

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

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

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

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

6.5.3 MV Scalar Fields

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

```
GET /Email/Message/_query?q=*&f=Tags&shards=2014-06-18
```

A typical result is shown below:

```
<results>
    <docs>
        <doc>
            <field name="Tags">
                <value>AfterHours</value>
            </field>
            <field name="_ID">+/pz/q4Jf8Rc2HK9Cg08TA==</field>
        </doc>
        <doc>
            <field name="Tags">
                <value>Customer</value>
                <value>AfterHours</value>
            </field>
            <field name="_ID">+/wqUBY1WsGtb7zjpKYf7w==</field>
        </doc>
        <doc>
            <field name="Tags"/>
            <field name="_ID">+1ZQASSaJei0HoGz6GdINA==</field>
        </doc>
    </docs>
</results>
```

The same request in JSON is shown below:

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

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

6.5.4 Link Fields

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

```
GET /Email/Person/_query?q=LastName=Powell&f=Manager,DirectReports&shards=2014-06-18
```

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

```
<results>
    <docs>
        <doc>
            <field name="_ID">gfNqhYF7LgBAtKTdIx3BKw==</field>
            <field name="DirectReports">
                <doc>
                    <field name="_ID">mKjYJmmLPoTVxJu2xdFmUg==</field>
                </doc>
            </field>
            <field name="Manager">
                <doc>
                    <field name="_ID">nLOCpa7aH/Y3zDrnMqG6Fw==</field>
                </doc>
            </field>
        </doc>
        <doc>
            <field name="_ID">sHUm0PEKu3gQDDNIHHWv1g==</field>
            <field name="DirectReports"/>
            <field name="Manager">
                <doc>
                    <field name="_ID">tkSQlrRqaeHsGvRU65g9HQ==</field>
                </doc>
            </field>
        </doc>
    </docs>
</results>
```

In JSON:

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

```
            }}
        ]
    }},
    {"doc": {
        "_ID": "sHUm0PEKu3gQDDNIHHWv1g==",
        "DirectReports": [],
        "Manager": [
            {"doc": {
                "_ID": "tkSQlrRqaeHsGvRU65g9HQ=="
            }}
        ]
    }}
  ]
}}
```

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

7. OLAP Aggregate Queries

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

7.1 Aggregate Parameters Overview

Aggregate queries use the following parameters:

- **Metric** (required): Defines one or more functions to calculate for selected objects. Metric functions such as COUNT, SUM, and MAX are supported. Each function is computed across selected objects, optionally subdivided into groups as defined by the grouping parameter. See the section **Metric Parameter** for details.

- **Query** (optional): A DQL query expression that selects objects in the perspective table. When this parameter is omitted, all objects in the table are included in metric computations.

- **Grouping** (optional): A list of one or more grouping expressions, which divide computations into single- or multi-level groups. When this parameter is omitted, the corresponding *global query* computes a single value for each metric function. When provided, the corresponding *grouped query* computes a value for each group value/metric function combination. A wide range of grouping expressions are supported as described in the section **Grouping Parameter**.

- **Pair** (optional): A pair of field paths used to perform a special "pair query". See the section **Pair Parameter** for more details.

- **Shards** (this or **Range** is required): Specifies a list of shards to search for objects by the query.

- **Range** (this or **Shards** is required): Specifies a range of shards to search for object by the query.

- **XShards** (optional): Specifies a list of shards to search for objects linked via xlinks. Either XShards or XRange can be specified, but not both. If neither is specified, xlink search scope defaults to Shards or Range parameter.

- **XRange** (optional): Specifies a range of shards to search for objects linked via xlinks. Either XShards or XRange can be specified, but not both. If neither is specified, xlink search scope defaults to Shards or Range parameter.

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

- **URI command**: All parameters are passed in the URI of the command.

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

Details of each command are described in the section **OLAP Aggregate Queries**. In the examples used in this section, the URI command is used. The following sections describe the more complex parameters used by aggregate queries.

7.2 Metric Parameter

The aggregate query metric parameter has multiple formats, as described below.

7.2.1 Metric Functions

In it's basic form, 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).

If the metric function parameter is a field path with a reflexive link, the transitive operator can be used. Example:

```
COUNT=(DirectReports^)
```

This function counts all DirectReports link values for each perspective object (i.e., all employees in the org chart reporting to each person). Scalar fields of transitively-linked objects can also be used as the metric function's parameter. For example:

```
MAX(Manager^.FirstName)
```

This function finds the maximum FirstName (highest, alphabetically) of all managers "up" the org chart for each perspective object. For more information on the transitive operator, see the section Transitive Function.

The supported metric functions are summarized below:

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

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

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

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

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

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

- **MAXCOUNT(*field*)**: This function computes the maximum cardinality of the given scalar or link *field*. For example, for the `Message` table, `MAXCOUNT(ExternalRecipients)` returns the highest number of `ExternalRecipients` values found for selected objects.

- **MINCOUNT(*field*)**: This function computes the minimum cardinality of the given *field*. If any selected objects have no value for the given field, the result will be 0.

When applied to a single-valued field, the result of both `MINCOUNT` and `MAXCOUNT` will be either 0 or 1.

Example metric functions are shown below:

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

7.2.2 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.

7.2.3 Metric Expressions

Doradus OLAP allows the metric parameter to be a list of *metric expressions*. A metric expression is a set of one or more algebraic clauses containing metric functions, constants, and the `DATEDIFF` function. Multiple clauses are combined with basic math operators (+, -, *, and /) and parentheses. This allows arbitrary calculations to be performed for selected objects. Examples of metric expressions are shown below:

```
MAX(Size) + COUNT(InternalRecipients.Person.Domain)

COUNT(*) * 2

SUM(Size) / COUNT(Sender.Person.MessageAddresses) / (DATEDIFF(DAY, "2013-11-01", NOW(PST)) + 1)
```

As with metric functions, metric expressions can be used in global and grouped aggregate queries. Multiple metric expressions can be computed in a single aggregate query. Each metric expression compute a value for each group. Computations are performed as integers or floating-point values as necessary. When a metric expression attempts to divide by 0, the metric value is returned as "Infinity". Parentheses override default operator evaluation precedence in the usual manner. Calculations are performed across all objects or for each group when a grouping parameter is provided.

7.2.4 DATEDIFF Function

Metric expressions can use the `DATEDIFF` function to calculate the difference between two timestamp constants in a specific granularity. Its result is an integer that may be positive, negative, or zero. The `DATEDIFF` function is similar to that used by SQL Server and uses the following syntax:

```
DATEDIFF(<units>, <start date>, <end date>)
```

Where:

- `<units>` can be any of the following mnemonics: `SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH`, `QUARTER`, and `YEAR`. The units mnemonic must be uppercase.

- `<start date>` and `<end date>` are timestamp literals or the `NOW()` function. All date/time components are optional from right-to-left except for year. Example literal values are `"2013-11-13 11:03:02.571"`, `"2013-11-13 11:03"`, and `"2013-11-13"`. Quotes are optional if the value has only a year component.

The two timestamp values are truncated to their `<units>` precision, similar to the `TRUNCATE` function. The difference between the two timestamps in then computed in the requested units, producing a numeric result:

- If the truncated `<start date>` is less than the truncated `<end date>`, the result is a positive number.

- If the truncated `<start date>` is greater than the truncated `<end date>`, the result is a negative number.

- If the two truncated timestamps are the same, the result is zero.

Common practice is to use an `<end date>` >= `<start date>` to produce a positive value. Hence, when `NOW()` is used, it normally appears as the `<end date>` parameter. Since `DATEDIFF` produces a constant, it is most useful when combined with other computations, e.g., to divide an average by the number of days in a query date range. For example:

```
.../Message?m=COUNT(*)/DATEDIFF(DAY, "2013-12-01", NOW())&q=SendDate:["2013-12-01" TO NOW()]
```

This query counts the number of messages sent between 2013-12-01 and "now", inclusively, and divides by the number of days between these same dates.

7.2.5 WHERE Filter

When the parameter to a metric function is a link path, the `WHERE` function can be used to filter values passed to the metric calculation. When the `WHERE` function follows a link name, it filters objects connected via that link. For example:

```
GET /Email/Message/_aggregate?range=0&m=MIN(Sender.Person.WHERE(Department:sales).Office)
```

This query selects all `Message` objects, but the `WHERE` filter selects only those `Person` objects whose `Department` contains the term `sales`. Hence, the `MIN` metric finds the lowest (first alphabetic) `Office` value whose sender is in the sales department.

When the `WHERE` function appears at the beginning of the metric function, it filters *perspective* objects, acting in conjunction with the query selection expression, if present. When used in this way, the `WHERE` function must be followed by `.field` where *field* is the metric field name or link path. For example:

```
GET /Email/Person/_aggregate?range=0&m=COUNT(WHERE(Department:sales).*)&q=Office:aliso
```

Since the `COUNT` function's outer `WHERE` filter is followed by `.*`, the function counts objects. The `WHERE` expression is AND-ed with the query expression `Office:also`. Therefore, this query is similar (but not identical) to the following:

```
GET /Email/Person/_aggregate?range=0&m=COUNT(*)&q=Office:aliso AND Department:sales
```

Because the *scope* of an outer WHERE function remains at the perspective table, multiple WHERE functions can be chained together to filter perspective objects. Example:

```
GET /Email/Message/_aggregate?range=0
    &m=MIN(WHERE(Sender.Person.Department:sales).WHERE(InternalRecipients.Person.Office:aliso).Size)
    &q=Tags=AfterHours
```

This query finds the smallest `Message.Size` value of all messages where (1) the message is tagged with `AfterHours`, (2) at least one sender belongs to the `sales` department, and (3) at least one internal recipient resides in the `aliso` office.

Note that filtering objects within the metric function does not produce exactly the same results as filtering objects in the query expression. In the previous example, the query expression `Tags=AfterHours` determines which objects are initially selected. The number of these objects is reflected in the `<totalobjects>` element returned by the query. However, the objects actually passed to the metric function are filtered by the other `WHERE` functions. This number is probably less than `<totalobjects>` but otherwise unknown unless the metric function is `COUNT(*)`. A typical result is shown below:

```
<results>
    <aggregate metric="MIN(WHERE(Sender.Person.Department:sales).
        WHERE(InternalRecipients.Person.Office:aliso).Size)" query="Tags=AfterHours"/>
    <totalobjects>6029</totalobjects>
    <value>4802</value>
</results>
```

Suppose we moved the outer WHERE functions to the query parameter:

```
GET /Email/Message/_aggregate?range=0&m=MIN(Size)
    &q=Sender.Person.Department:sales AND InternalRecipients.Person.Office:aliso AND Tags=AfterHours
```

Here, the objects are filtered by the query expression, so only those selected are reflected in the `<totalobjects>` element returned by the query. A typical result is shown below:

```
<results>
    <aggregate metric="MIN(Size)" query="Sender.Person.Department:sales AND
        InternalRecipients.Person.Office:aliso AND Tags=AfterHours"/>
    <totalobjects>2</totalobjects>
    <value>4802</value>
</results>
```

Compared to the previous query, which selected 6029 objects, this query selected only 2 objects. This also shows that selecting objects in the query expression (&q) is more efficient, especially for large data sets.

However, sometimes metric-level filtering is the only way to get the results we need. For example, suppose we want to count messages sent by people in two different departments but grouped by the same office. We could use the following multi-metric query:

```
GET /Email/Message/_aggregate?range=0
    &m=COUNT(WHERE(Sender.Person.Department:sales).*),
        COUNT(WHERE(Sender.Person.Department:support).*)
    &f=TOP(3,Sender.Person.Office)
```

This query performs two `COUNT(*)` functions: the first selects messages whose sender belong to `sales`, the second those sender belong to `support`. By definition, when a TOP or BOTTOM grouping field is used with a multi-metric query, the groups are generated from the sorted values of the <u>first</u> metric function. Secondary metric functions follow the same grouping pattern so that the metric computations are correlated. A typical result for this query looks like this:

```
<results>
    <aggregate metric="COUNT(WHERE(Sender.Person.Department:sales).*),
        COUNT(WHERE(Sender.Person.Department:support).*)" query="*"
        group="TOP(3,Sender.Person.Office)"/>
    <totalobjects>6030</totalobjects>
    <groupsets>
        <groupset group="TOP(3,Sender.Person.Office)"
            metric="COUNT(WHERE(Sender.Person.Department:sales).*)">
            <summary>81</summary>
            <totalgroups>40</totalgroups>
            <groups>
                <group>
                    <metric>68</metric>
                    <field name="Sender.Person.Office">Maidenhead</field>
                </group>
                <group>
                    <metric>5</metric>
                    <field name="Sender.Person.Office">Madrid</field>
                </group>
                <group>
                    <metric>2</metric>
                    <field name="Sender.Person.Office">Aliso Viejo 5</field>
                </group>
            </groups>
        </groupset>
        <groupset group="TOP(3,Sender.Person.Office)"
            metric="COUNT(WHERE(Sender.Person.Department:support).*)">
            <summary>16</summary>
            <totalgroups>40</totalgroups>
            <groups>
                <group>
                    <metric>14</metric>
                    <field name="Sender.Person.Office">Maidenhead</field>
                </group>
                <group>
                    <metric>0</metric>
                    <field name="Sender.Person.Office">Madrid</field>
                </group>
                <group>
                    <metric>2</metric>
                    <field name="Sender.Person.Office">Aliso Viejo 5</field>
                </group>
            </groups>
        </groupset>
    </groupsets>
</results>
```

This allows the offices with the most email senders in `sales` to be directly compared to the number of email senders in `support` for the same offices. If we executed two separate aggregate queries—one for each function—but with the same grouping parameter, we would get different, uncorrelated groups.

7.3 Grouping Parameter

When a grouping parameter is provided, it causes the aggregate query to compute multiple values, one per *group value* as defined by the grouping expression. A wide range of grouping expressions are allowed as described in the following sections.

7.3.1 Global Aggregates: No Grouping Parameter

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

```
GET /Email/Message/_aggregate?m=MAX(Size)&range=0
```

This aggregate query returns the largest `Size` value among all messages in all shards. 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)&range=0
```

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)&shards=2014-06-01
```

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

7.3.2 Single-level Grouping

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

```
GET /Email/Message/_aggregate?m=MAX(Size)&f=Tags&range=0
```

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

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

In JSON:

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

For grouped aggregate queries, the `results` element contains a `groups` element, which contains one `group` for each group value. Each `group` contains the `field` name and value for that group and the corresponding `metric` value. The `totalobjects` value computes the number of objects selected in the computation. The `summary` value computes the metric value across all selected objects independent of groups. For the `AVERAGE` function, this provides the *true average*, not the *average of averages*.

7.3.3 Grouping Field Aliases

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

```
GET /Email/Message/_aggregate?m=COUNT(*)&f=TOP(3,Sender.Person.Name)&range=0
```

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* or `.AS(`*name*`)` to the grouping field. The two syntaxes are equivalent. The `AS` function's parameter can be an unquoted term or a quoted string. Example:

```
GET /Email/Message/_aggregate?m=COUNT(*)&f=TOP(3,Sender.Person.Name) AS Name&range=0
GET /Email/Message/_aggregate?m=COUNT(*)&f=TOP(3,Sender.Person.Name).AS(Name)&range=0
```

With either form, the alias `Name` is used in the output as shown below:

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

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

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

7.3.4 Multi-level Grouping

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

```
GET /Email/Message/_aggregate?m=COUNT(*)&f=TRUNCATE(SendDate,DAY),Tags&range=0
```

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

```
<results>
    <aggregate metric="COUNT(*)" group="TRUNCATE(SendDate,DAY),Tags"/>
    <totalobjects>6030</totalobjects>
    <summary>6030</summary>
    <groups>
        <group>
            <summary>4752</summary>
            <field name="SendDate">2010-07-17 00:00:00</field>
            <groups>
                <group>
                    <metric>1</metric>
                    <field name="Tags">(null)</field>
                </group>
                <group>
                    <metric>4751</metric>
                    <field name="Tags">AfterHours</field>
                </group>
                <group>
                    <metric>1524</metric>
                    <field name="Tags">Customer</field>
                </group>
            </groups>
        </group>
        <group>
            <summary>1278</summary>
            <field name="SendDate">2010-07-18 00:00:00</field>
```

```
            <groups>
                <group>
                    <metric>1278</metric>
                    <field name="Tags">AfterHours</field>
                </group>
                <group>
                    <metric>700</metric>
                    <field name="Tags">Customer</field>
                </group>
            </groups>
        </group>
    </groups>
</results>
```

The same response in JSON:

```
{"results": {
    "aggregate": {"metric": "COUNT(*)", "group": "TRUNCATE(SendDate,DAY),Tags"},
    "totalobjects": "6030",
    "summary": "6030",
    "groups": [
        {"group": {
            "summary": "4752",
            "field": {"SendDate":"2010-07-17 00:00:00"},
            "groups": [
                {"group": {
                    "metric": "1",
                    "field": {"Tags": "(null)"}
                }},
                {"group": {
                    "metric": "4751",
                    "field": {"Tags":"AfterHours"}
                }},
                {"group": {
                    "metric": "1524",
                    "field": {"Tags":"Customer"}
                }}
            ]
        }},
        {"group": {
            "summary": "1278",
            "field": {"SendDate": "2010-07-18 00:00:00"},
            "groups": [
                {"group": {
                    "metric": "1278",
                    "field": {"Tags":"AfterHours"}
                }},
                {"group": {
                    "metric": "700",
                    "field": {"Tags":"Customer"}
                }}
```

```
            ]
        }}
     ]
  }}
```

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

7.3.5 Group by Timestamp Subfield

Doradus OLAP supports grouping by timestamp subfields. For example:

```
.../Message/_aggregate?q=*&m=COUNT(*)&f=TOP(3,SendDate.HOUR)&range=0
```

`SendDate` is a timestamp field belonging to the `Message` table; the subfield `HOUR` extracts the hour-of-day component of the timestamp, which is a value between 0 and 23. Since `SendDate.HOUR` is wrapped in a `TOP` function, the grouping parameter produces the 3 top-most hours. In other words, this aggregate query finds the 3 hours of the day in which the most messages are sent.

Grouping by a timestamp subfield is different than using the `TRUNCATE` function. Grouping by a subfield extracts a timestamp component, so there will never be more values than the component allows (e.g., 12 months in a year, 24 hours in a day). In comparison, `TRUNCATE(SendDate,HOUR)` rounds down (truncates) each `SendDate` value to its hour precision, but it will produce a value for every year-month-day-hour combination found among selected objects.

7.3.6 Special Grouping Functions

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

7.3.6.1 BATCH Function

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

```
GET /Email/Message/_aggregate?m=COUNT(*)&f=BATCH(Size,100,1000,10000,100000)&shards=2014-06-01
```

This query counts messages grouped by specific ranges of the `Size` field from the shard named `2014-06-01`. The ranges are divided at the given literal values: `100`, `1000`, `10000`, and `100000`. The lowest value implicitly creates an extra *less than* group; the highest value is open-ended and creates a *greater than or equal to* group. The query in the example above defines the following 5 groups:

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

The example above returns a result such as the following:

```
<results>
```

```
        <aggregate metric="COUNT(Size)" group="BATCH(Size,100,1000,10000,100000)"/>
        <groups>
            <group>
                <field name="Size">&lt;100</field>
                <metric>0</metric>
            </group>
            <group>
                <field name="Size">100-1000</field>
                <metric>125</metric>
            </group>
            <group>
                <field name="Size">1000-10000</field>
                <metric>4651</metric>
            </group>
            <group>
                <field name="Size">10000-100000</field>
                <metric>1149</metric>
            </group>
            <group>
                <field name="Size">&gt;=100000</field>
                <metric>105</metric>
            </group>
        </groups>
        <summary>6030</summary>
    </results>
```

In JSON:

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

```
        ],
        "summary":"6030"
    }}
```

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

7.3.6.2 INCLUDE and EXCLUDE Functions

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

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

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

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

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

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

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

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

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

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

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

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

7.3.6.3 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))&shards=2014-06-01
```

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

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

In JSON:

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

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

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

```
GET /Email/Message/_aggregate?m=COUNT(*)&q=Sender.MessageAddress.Person.Name='Quest Support'
    &f=TOP(5,TERMS(Subject,(sundae tillage)))&shards=2014-06-01
```

7.3.6.4 TOP and BOTTOM Functions

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

```
GET /Email/Message/_aggregate?m=SUM(Size)&f=TOP(3,InternalRecipients.Person.Name)&range=0
```

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

```
<results>
    <aggregate metric="SUM(Size)" query="*" group="TOP(3,InternalRecipients.Person.Name)"/>
    <totalobjects>6030</totalobjects>
    <summary>190643320</summary>
    <totalgroups>836</totalgroups>
    <groups>
        <group>
            <metric>97744099</metric>
```

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

In JSON:

```
{"results": {
    "aggregate": {
        "metric": "SUM(Size)",
        "query": "*",
        "group": "TOP(3,InternalRecipients.Person.Name)"
    },
    "totalobjects": "6030",
    "summary": "190643320",
    "totalgroups": "836",
    "groups": [
        {"group": {
            "metric": "97744099",
            "field": {"InternalRecipients.Person.Name": "(null)"}
        }},
        {"group": {
            "metric": "20060808",
            "field": {"InternalRecipients.Person.Name": "Marc Bourauel"}
        }},
        {"group": {
            "metric": "16798901",
            "field": {"InternalRecipients.Person.Name": "Nina Cantauw"}
        }}
    ]
}}
```

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

Some numeric metric expressions can produce a result that is not a valid number, such as when division by zero occurs. Such non-numeric values include "not-a-number" (NaN), positive infinity, and negative infinity. All such invalid numeric results sort to the end (last group) for both TOP and BOTTOM.

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

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

```
GET /Email/Message/_aggregate?m=COUNT(*)&f=TOP(3,Sender.Person),BOTTOM(2,TRUNCATE(SendDate,DAY))
    &shards=2014-06-01
```

7.3.6.5 TRUNCATE Function

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

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

For example:

```
GET /Email/Message/_aggregate?m=COUNT(*)&f=TRUNCATE(SendDate,DAY,GMT-2)
    &q=SendDate >= "2010-07-17"&shards=2014-06-01
```

This query finds all messages whose `SendDate` is >= "2010-07-17" in the shard named `2014-06-01`. 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"
        }},
        ...
    ]
}}
```

When the precision parameter is DAY, WEEK, MONTH, QUARTER, or YEAR, the timestamp values in group names are returned as dates only. For example:

```
GET /Email/Message/_aggregate?m=COUNT(*)&f=TRUNCATE(SendDate,WEEK)&range=0
```

This returns a response such as the following:

```
<results>
    <aggregate metric="COUNT(*)" query="*" group="TRUNCATE(SendDate,WEEK)"/>
    <totalobjects>6030</totalobjects>
    <summary>6030</summary>
    <totalgroups>2</totalgroups>
    <groups>
        <group>
            <metric>5972</metric>
            <field name="TRUNCATE(SendDate,WEEK)">2010-07-19</field>
        </group>
        <group>
            <metric>58</metric>
            <field name="TRUNCATE(SendDate,WEEK)">2010-07-26</field>
        </group>
    </groups>
</results>
```

7.3.6.6 UPPER and LOWER Functions

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

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

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

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

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

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

7.3.6.7 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 <u>and</u> the recipient's address is considered external. As an example, this aggregate query <u>won't</u> 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
    &shards=...
```

This query doesn't work because it selects messages for which <u>at least one</u> recipient's `ReceiptDate` is > `"2014-01-01"`, and <u>at least one</u> recipient has an external domain. Every such message is then counted in <u>all</u> 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)
    &shards=...
```

This causes the correct objects to be selected, but it still counts them in <u>all</u> `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
    &shards=...
```

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"` <u>and</u> 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
    &shards=...
```

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

```
GET /Email/Person/_aggregate?range=0&m=COUNT(*)&f=WHERE(Department:sales).Office
```

In this query, the grouping field is `Office`. But because it is *prefixed* with a `WHERE` filter, the filter is applied to perspective objects (`Person`). Following an outer `WHERE` filter, the *scope* of the link path remains at the query perspective, consequently the subsequent link path or additional `WHERE` filters must be applicable to the query perspective.

When the grouping field is used with a grouping function such as `BATCH`, `TERMS`, or `TOP`, the outer `WHERE` field is always specified as a prefix to the grouping field. Examples:

```
GET /Email/Person/_aggregate?range=0&m=COUNT(*)&f=TOP(6,WHERE(Department:sales).Office)
GET /Email/Message/_aggregate?range=0&m=COUNT(*)
    &f=BATCH(WHERE(Tags=AfterHours).Size,1000,10000,100000)
GET /Email/Message/_aggregate?range=0&m=COUNT(*)&f=TOP(5,TERMS(WHERE(Size>100000).Subject))
GET /Email/Message/_aggregate?range=0&m=COUNT(*)
    &f=TERMS(WHERE(Size>100000).Subject),WHERE(Sender.Person.LastName:Cuthill).Tags
```

If present, the query parameter (&q) is used first to select objects, the total number of which is reflected in the `<totalobjects>` element in the query results. Each outer `WHERE` function is used to further filter objects passed to the calculations of the corresponding aggregate groups.

7.3.6.9 FIRST and LAST Functions
The `FIRST` and `LAST` functions are similar to the `TOP` and `BOTTOM` functions except that a limited set of groups are returned based on the <u>group names</u> instead of the <u>metric values</u>.

```
GET /Email/Person/_aggregate?q=*&m=COUNT(*)&f=FIRST(10,LastName)&range=0
```

This query returns the first 10 `LastName` groups alphabetically. `LAST` returns the last groups (descending sort). This should help with aggregate queries that must be "paged".

7.3.6.10 Flattened Multi-level Grouping
An aggregate query with multi-level grouping is returned as a nested document. When the `TOP`, `BOTTOM`, `FIRST`, or `LAST` functions are used, the limit value is normally applied to the corresponding grouping level only. For example:

```
GET /Email/Message/_aggregate?range=0&m=COUNT(*)
    &f=TOP(10,Sender.Person.LastName),Sender.Person.FirstName
```

This query returns the first 10 `Sender.Person.LastName` groups, but it returns <u>all</u> `FirstName` subgroups for each `LastName` group. The query can be flattened by adding the REST parameter `&flat=true`:

```
GET /Email/Message/_aggregate?range=0&m=COUNT(*)
    &f=TOP(10,Sender.Person.LastName),Sender.Person.FirstName&flat=true
```

This returns only the top 10 `LastName`/`FirstName` group pairs. This allows a more predictable result size, e.g., when the results will be displayed in a web page.

7.3.6.11 SETS Function

The `SETS` function allows aggregate query groups to be created from arbitrary query expressions. The general syntax is:

```
SETS(<query 1> [AS <name>], <query 2> [AS <name>], ..., <query n> [AS <name>]) [AS <name>])
```

Where each <query> is a DQL query expression that selects a set of objects from the perspective table. The queries can overlap such that multiple expressions can select the same objects. For example, the following query selects messages with overlapping `Size` values:

```
GET /Email/Message/_aggregate?range=0&m=COUNT(*)
    &f=SETS(Size < 1000, Size > 100 AND Size < 100000, Size > 10000)
```

This returns a result such as the following:

```
<results>
    <aggregate metric="COUNT(*)" query="*" group="SETS(Size < 1000, Size > 100 AND Size < 100000,
        Size > 10000)"/>
    <totalobjects>6030</totalobjects>
    <summary>6030</summary>
    <totalgroups>3</totalgroups>
    <groups>
        <group>
            <metric>125</metric>
            <field name="SETS(Size < 1000, Size > 100 AND Size < 100000, Size > 10000)">Size IN {null
                - 1000}</field>
        </group>
        <group>
            <metric>5925</metric>
            <field name="SETS(Size < 1000, Size > 100 AND Size < 100000, Size > 10000)">(Size IN {100
                - null}) AND (Size IN {null - 100000})</field>
        </group>
        <group>
            <metric>1254</metric>
            <field name="SETS(Size < 1000, Size > 100 AND Size < 100000, Size > 10000)">Size IN
                {10000 - null}</field>
        </group>
    </groups>
</results>
```

By default, each group uses the corresponding query expression in both the field `name` and value. The query expression text may not match the original query text exactly since expressions may be transformed internally. A custom field `name` can be used by using the `AS` syntax after the `SETS` function. Example:

```
GET /Email/Message/_aggregate?range=0&m=COUNT(*)
    &f=SETS(Size < 1000, Size > 100 AND Size < 100000, Size > 10000) AS Sizes
```

This creates the simpler output format:

```
<results>
    <aggregate metric="COUNT(*)" query="*" group="SETS(Size < 1000, Size > 100 AND Size < 100000,
        Size > 10000) AS Sizes"/>
    <totalobjects>6030</totalobjects>
    <summary>6030</summary>
    <totalgroups>3</totalgroups>
    <groups>
        <group>
            <metric>125</metric>
            <field name="Sizes">Size IN {null - 1000}</field>
        </group>
        <group>
            <metric>5925</metric>
            <field name="Sizes">(Size IN {100 - null}) AND (Size IN {null - 100000})</field>
        </group>
        <group>
            <metric>1254</metric>
            <field name="Sizes">Size IN {10000 - null}</field>
        </group>
    </groups>
</results>
```

The field values can also be customized by using the AS syntax after each query expression. For example:

```
GET /Email/Message/_aggregate?range=0&m=COUNT(*)
    &f=SETS(Size < 1000 AS Littles, Size > 100 AND Size < 100000 AS Middles,
        Size > 10000 AS Biggies) AS Sizes
```

This creates an even simpler format:

```
<results>
    <aggregate metric="COUNT(*)" query="*" group="SETS(Size < 1000 AS Littles, Size > 100 AND Size <
        100000 AS Middles, Size > 10000 AS Biggies) AS Sizes"/>
    <totalobjects>6030</totalobjects>
    <summary>6030</summary>
    <totalgroups>3</totalgroups>
    <groups>
        <group>
            <metric>125</metric>
            <field name="Sizes">Littles</field>
        </group>
        <group>
            <metric>5925</metric>
            <field name="Sizes">Middles</field>
        </group>
        <group>
            <metric>1254</metric>
            <field name="Sizes">Biggies</field>
        </group>
    </groups>
```

```
    </results>
```

Note that if objects are selected by the aggregate query but not included in any query expressions defined by the SETS function, those objects are automatically included in a null set. For example:

```
GET /Email/Message/_aggregate? range=0&m=COUNT(*)
    &f=SETS(Size < 10 AS BittyEmails, Size > 100000 AS MegaEmails) AS Sizes
```

Messages that don't fit either of the two query expressions are included in a group named `(null)` as shown below:

```
<results>
    <aggregate metric="COUNT(*)" query="*" group="SETS(Size < 10 AS BittyEmails, Size > 100000 AS
        MegaEmails) AS Sizes"/>
    <totalobjects>6030</totalobjects>
    <summary>6030</summary>
    <totalgroups>3</totalgroups>
    <groups>
        <group>
            <metric>5925</metric>
            <field name="Sizes">(null)</field>
        </group>
        <group>
            <metric>0</metric>
            <field name="Sizes">BittyEmails</field>
        </group>
        <group>
            <metric>105</metric>
            <field name="Sizes">MegaEmails</field>
        </group>
    </groups>
</results>
```

As shown, the majority of messages (5925) did not match either query expression and were therefore included in the (null) group. No messages matches the `BittyEmails` query expression, hence the metric value for that group is 0.

Note that the query expressions do not need to use the same fields. Each query expression can use any criteria to create its groups. Example:

```
GET /Email/Message/_aggregate?m=COUNT(*)&range=0
    &f=SETS(Size < 1000, Tags=Customer, Sender.Person.Department:support)
```

7.4 Multi-metric Aggregate Queries

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

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

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 <u>first</u> 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)&shards=...
```

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:

```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:

```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"
        }}
    ]
}}
```

7.5 Pair Parameter

The *pair* parameter supports a special kind of aggregate query called a *dual role* query. To illustrate when it is needed, let's first look at a dual role *object* query, which doesn't require the "pair" functionality.

Assume we divide the participants of a Message object into *senders* and *internal recipients*. Suppose we want to find all messages where either:

(1) The sender resides in the Kanata office and the internal recipient is a member of a Support department, or:
(2) The internal recipient resides in the Kanata office and the sender is a member of a Support department.

In other words, we want to find messages between the Kanata office and a Support department, but we don't care if the message is sent from Kanata to Support or the other way around. We also have to eliminate the case where one participant is both in Kanata and belongs to Support but the other participants are neither (otherwise it's not really a query "between" these two roles.)

As an object query, this is rather straightforward and can be specified as follows:

```
.../Message/_query?q=(Sender.Person.Office:Kanata AND InternalRecipients.Person.Department:Support)
    OR
    (InternalRecipients.Person.Office:Kanata AND Sender.Person.Department:Support)
```

This query uses two OR clauses: the first clause selects messages sent by someone in the Kanata office and received by anyone in the Support department; the second clause uses the same office/department criteria but reverses the sender/receiver roles. A message that satisfies either clause is selected.

Now suppose we want to execute this as an aggregate query. If all we want is a total count of messages, we can use the same query parameter as above.

But suppose we want to group the results based on a field belonging to one of the roles: for example, we want to group by the Department of the Kanata participant. When the Kanata participant is the sender, we want to group by

`Sender.MessageAddress.Person.Department`. When the Kanata participant is the receiver, we want to group by `InternalRecipients.MessageAddress.Person.Department`. But we can only group on one field at each level—we can't group on a participant's *role* ("the Kanata participant"), which is defined by two different link paths.

The aggregate "pair" feature allows such dual role queries. To demonstrate how it works, below is the aggregate query proposed above, grouping messages by the Kanata participants' department, regardless of whether those participants are senders or the internal recipients:

```
.../Message/_aggregate?pair=Sender,InternalRecipients
    &q=_pair.first.Person.Office:Kanata AND _pair.second.Person.Department:Support
    &f=_pair.first.Person.Department
```

The pair parameter works as follows:

- When used, the pair parameter must be a comma-separated list of exactly two link paths referring to the same table. These fields are referred to as the "pair fields". In this example, the pair fields are `Sender` and `InternalRecipients`, which both refer to the `Participant` table. Though not shown in this example, the pair field link paths can use `WHERE` filters if needed.

- When the pair parameter is used, the system field `_pair` can be used in the query and/or fields parameters. The `_pair` system field must be followed by the subfield `first` or `second`, and the remainder of the expression must be valid based on the pair fields' table. In this example, since the pair fields are links to the `Participant` table, the remainder of the expression must be valid for `Participant` objects.

- The subfield name (`first` or `second`) is used for *binding* purposes: the `_pair` field expression is actually applied to both pair fields. That is, the query is executed twice with the roles of the pair fields reversed. In the first execution, the query expression is first evaluated using `Sender` in the `first` role and `Recipients` in the `second` role:

  ```
  Sender.Person.Office:Kanata AND InternalRecipients.Person.Department:Support
  ```

  In this execution, the sender(s) are *bound* to `_pair.first` and the internal recipients are bound to `_pair.second`. Because the grouping parameter groups by `_pair.first.Person.Department`, this means that if the expression selects a message, the sender's department is used to choose the message's group. In other words:

  ```
  &f=_pair.first.Person.Department
  ```

  Is interpreted as:

  ```
  &f=Sender.Person.Department
  ```

- The roles of the pair fields are then swapped and the query expression is executed again. The second query executed is:

  ```
  InternalRecipients.Person.Office:Kanata AND Sender.Person.Department:Support
  ```

  Here, `_pair.first` is bound to `Recipients` and `_pair.second` is bound to `Sender`. Consequently, if the expression selects a message, it is recipient's department that is used to choose the message's group. In other words:

  ```
  &f=_pair.first.Person.Department
  ```

Is interpreted as:

```
&f=InternalRecipients.Person.Department
```

In this example, "location in Kanata" is one role and "department in Support" is the other role. Messages are found by looking for one role as sender and the other role as internal recipients and then vice versa. Regardless of which combination selects a message, the aggregate results are grouped by the Kanata office's department.

Note that when the pair parameter is used, the query and fields parameters can use WHERE filters and all other normal aggregate query features.

8. OLAP REST Commands

This section describes the REST commands supported by a Doradus OLAP database.

8.1 REST API Overview

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

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

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

8.1.1 Common REST Headers

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

- `Content-Type`: Describes the MIME type of the input entity, optionally with a `Charset` parameter. The MIME types supported by Doradus are `text/xml` (the default) and `application/json`.

- `Content-Length`: Identifies the length of the input entity in bytes. The input entity cannot be longer than `max_request_size`, defined in the doradus.yaml file.

- `Accept`: Indicates the desired MIME type of an output (response) entity. If no `Accept` header is provided, it defaults to input entity's MIME type, or `text/xml` if there is no input entity.

- `Content-Encoding`: Specifies that the input entity is compressed. Only `Content-Encoding: gzip` is supported.

- `Accept-Encoding`: Requests the output entity to be compressed. Only `Accept-Encoding: gzip` is supported. When Doradus compresses the output entity, the response includes the header `Content-Encoding: gzip`.

- `X-API-Version`: Requests a specific API version for the command. Currently, `X-API-Version: 2` is supported.

Header names and values are case-insensitive.

8.1.2 Common REST URI Parameters

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

- `api=N`: Requests a specific API version for the command. Currently, `api=2` is supported. This parameter overrides the `X-API-Version` header if present.

- `format=[json|xml]`: Requests the output message format in JSON or XML, overriding the `Accept` header if present.

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

```
GET /_applications?format=json
GET /Email/_shards?format=xml&api=2
GET /Email/Person/_query?q=LastName:Smith&shards=2014-01-01&s=5&api=2&format=json
```

8.1.3 Common JSON Rules

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

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

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

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

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

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

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

8.1.4 Illegal Characters in XML Content

Although Doradus allows text fields to use all Unicode characters, XML does not allow element content to contain certain characters. The only characters allowed within XML content are in the following:

```
#x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]
```

In an output message, when Doradus creates an XML element whose content would contain any characters outside of this set, it base64-encodes the entire element value and adds the attribute encoding="base64" to the element. For example, suppose a field value contains the Unicode value "ABC\u0000". The 4th character, hex 0x00, is not valid in XML, hence the field value would be returned as follows:

```
<field name="foo" encoding="base64">QUJDAA==</field>
```

Similarly, in an input XML message, element content that contains invalid XML characters must be base64-encoded, and the application must add the encoding="base64" attribute to the corresponding element.

8.1.5 Common REST Responses

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

```
HTTP/1.1 503 Service Unavailable
Content-length: 43
Content-type: Text/plain

Database is not reachable. Waiting to retry
```

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

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

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

8.2 OLAP REST Command Summary

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

REST Command	Method and URI
Application Management Commands	
Create Application	`POST    /_applications`
Modify Application	`PUT     /_applications/{application}`
List Application	`GET     /_applications/{application}`
List All Applications	`GET     /_applications`
Delete Application	`DELETE /_applications/{application}` `DELETE /_applications/{application}/{key}`
Object Update Commands	
Add Batch	`POST    /{application}/{shard}`
Delete Batch	`DELETE /{application}/{shard}`
Shard Management Commands	
List Shards	`GET     /{application}/_shards`

Get Shard Statistics	`GET     /{application}/_shards/{shard}`
	`GET     /{application}/_statistics/{shard}`
	`GET     /{application}/_statistics/{shard}?{params}`

Merge Shard	`POST    /{application}/_shards/{shard}`
Delete Shard	`DELETE /{application}/_shards/{shard}`
Query Commands	
Object Query via URI	`GET     /{application}/{table}/_query?{params}`
Object Query via Entity	`GET     /{application}/{table}/_query` `PUT     /{application}/{table}/_query`
Aggregate Query via URI	`GET     /{application}/{table}/_aggregate?{params}`
Aggregate Query via Entity	`GET     /{application}/{table}/_aggregate` `PUT     /{application}/{table}/_aggregate`
Find Duplicates	`GET     /{application}/{table}/_duplicates?{params}`
Task Status Command	
Get All Tasks Status	`GET     /_tasks`

REST Command	Method and URI
OLAP Browser Interface	
OLAP Browser Home	GET /_olapp

Details on each command are described in the following sections.

8.3 Application Management Commands

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

8.3.1 Create Application

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

```
POST /_applications
```

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

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

A new application is automatically created as an OLAP application if the Doradus server is appropriately configured. However, an application can be explicitly managed by the OLAP server by setting the application-level option StorageService to OLAPService.

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

8.3.2 Modify Application

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

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

where {application} is the application's name. The request must include the application's modified schema in XML or JSON as specified by the request's content-type header. Because an application's name cannot be changed, {application} must match the application name in the schema. An application's key cannot be changed either, hence if the application was defined with a key, the same key must be provided. If application was defined without a key, a key can be added by including it in the updated schema. If the request is successful, a 200 OK response is returned with no message body.

Modifying an application *replaces* its current schema. If any previously-defined tables or fields are omitted in the new schema, those tables/fields can no longer be referenced in queries, but existing data is not modified or deleted. Data is deleted only when an object is explicitly deleted, the owning shard is explicitly deleted, or shard is automatically deleted due to data aging.

8.3.3 List Application

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

```
GET /_applications
```

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

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

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

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

8.3.4 Delete Application

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

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

where `{application}` is the application's name. If the application's schema defines a key, the `{key}` must be provided in the command, and it must match

8.4 Object Update Commands

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

8.4.1 Add Batch

A batch of new, updated, and/or deleted objects is loaded into a specific shard of an application using the following REST command:

```
POST /{application}/{shard}[?Overwrite={true|false}]
```

where `{application}` is the application name and `{shard}` is the shard to which the batch is to be added. Shard names are text strings and are not predefined: a shard is started when the first batch is added to it.

The optional `Overwrite` parameter (case-insensitive) indicates whether or not the batch should replace existing field values. The default is `true`, which means any field that already has a value for the corresponding object replaces the existing value. If multiple batches are added to the same shard with `Overwrite=true`, the last value added when the shard is merged takes precedence. If `Overwrite` is set to `false`, values in the corresponding batch are only additive: they never replace existing field values when the shard is merged.

The Add Batch command must include an input entity that contains the objects to be added, updated, and/or deleted. The format of an example input message in XML as shown below:

```xml
<batch>
    <docs>
        <doc _table="Message">
            <field name="_ID">92XJeDwQ8lD3/RS4yM5gTg==</field>
            <field name="Size">10334</field>
            <field name="Tags">
                <add>
```

```
                    <value>Confidential</value>
                    <value>Sensitive</value>
                </add>
            </field>
            ...
        </doc>
        <doc _table="Message" _deleted="true">
            <field name="_ID">95lfrCiljbiKOQK9UH7LYg==</field>
        </doc>
        <doc _table="Person">
            <field name="_ID">x3OKbjCmKw47wEHaqV0nLQ==</field>
            <field name="FirstName">John</field>
            <field name="Manager">
                <add>
                    <value>LjJEtDcwp1ltqJWJ980+HQ==</value>
                </add>
            </field>
            ...
        </doc>
        ...
    </docs>
</batch>
```

In JSON:

```
{"batch": {
    "docs": [
        {"doc": {
            "_table": "Message",
            "_ID": "92XJeDwQ8lD3/RS4yM5gTg==",
            "Size": "10334",
            "Tags": {
                "add": ["Confidential", "Sensitive"]
            },
            ...
        }},
        {"doc": {
            "_table": "Message",
            "_deleted": "true",
            "_ID": "95lfrCiljbiKOQK9UH7LYg=="
        }},
        {"doc": {
            "_table": "Person",
            "_ID": "x3OKbjCmKw47wEHaqV0nLQ==",
            "FirstName": "John",
            "Manager": {
                "add": ["LjJEtDcwp1ltqJWJ980+HQ=="]
            },
            ...
        }},
        ...
```

```
      ]
  }}
```

As shown, messages from multiple tables can be mixed in the same batch. The `_table` property identifies the table that the object will be inserted to, updated in, or deleted from. An object is added the first time fields are assigned to its `_ID`. Assigning an SV scalar field for an existing object replaces its current value. MV scalar and link field values are added in `add` groups. An object is deleted by giving its `_ID` value and setting the system field `_deleted` to `true`.

When an object is added, the `_ID` field can be omitted, in which case Doradus assigns a unique ID. The ID is a base 64-encoded 120-bit value that is generated in a way to ensure uniqueness even in a multi-node cluster. However, automatic IDs remove the idempotency of updates: if the same object is added twice with the `_ID` field assigned, it will be inserted twice with different ID values.

The only restrictions on Doradus OLAP updates are:

- Once assigned a value, a field cannot be set to null.

- There is no way to remove values from an MV scalar or link field. (However, deleting an object automatically updates affected inverse links.)

Batches are persisted but not processed until the containing shard is merged. When the shard is merged, all adds, updates, and deletes are merged with existing objects in the shard. If the same object is updated in multiple batches, the updates are merged; conflicts, such setting the same SV scalar field to different values, are resolved by using the update in the most recently-added batch.

The ideal batch size is application-specific and depends on several factors:

- If the number of objects per batch is too large or too small, merging the batches into a single segment will take longer, slowing down the overall load time.

- Because an object batch temporarily resides in memory, both the client and the Doradus server require memory proportional to the size of the batch. Using REST API compression helps with server memory because object batches are parsed and loaded from the compressed message entity.

8.4.2 Delete Batch

Objects can be deleted in the Add Batch command, but they can also be deleted en masse with the following REST command:

```
DELETE /{application}/{shard}
```

where `{application}` is the application name and `{shard}` is the shard from which objects are to be deleted. The command must include an input entity that only contains the `_table` and `_ID` of each object to be deleted. Example:

```
<batch>
    <docs>
        <doc _table="Message">
            <field name="_ID">92XJeDwQ8lD3/RS4yM5gTg==</field>
        </doc>
        <doc _table="Message">
```

```
            <field name="_ID">95lfrCiljbiKOQK9UH7LYg==</field>
        </doc>
        <doc _table="Person">
            <field name="_ID">x3OKbjCmKw47wEHaqV0nLQ==</field>
        </doc>
        ...
    </docs>
</batch>
```

In JSON:

```
{"batch": {
    "docs": [
        {"doc": {
            "_table": "Message",
            "_ID": "92XJeDwQ8lD3/RS4yM5gTg=="
        }},
        {"doc": {
            "_table": "Message",
            "_ID": "95lfrCiljbiKOQK9UH7LYg=="
        }},
        {"doc": {
            "_table": "Person",
            "_ID": "x3OKbjCmKw47wEHaqV0nLQ=="
        }},
        ...
    ]
}}
```

Only the `_table` and `_ID` of each doc element is required. If any other fields are assigned values, they are ignored. As with the Add Batch command, the delete batch is stored but not processed until the corresponding shard is merged.

8.5 Shard Management Commands

Doradus OLAP provides the following commands to list, merge, and delete shards.

8.5.1 List Shards

The list of all known shards of a given application can be obtained with the following REST command:

```
GET /{application}/_shards
```

Each shard that has at least one batch posted are listed, even if the shard has not yet been merged. A typical response in XML:

```
<result>
    <application name="Email">
        <shards>
            <value>2004-11-16</value>
            <value>2005-08-03</value>
            <value>2005-10-10</value>
            <value>2005-11-01</value>
```

```
            <value>2005-12-20</value>
            <value>2006-01-17</value>
            <value>2006-02-01</value>
        </shards>
    </application>
</result>
```

In JSON:

```
{"result": {
    "Email": {
        "shards": [
            "2004-11-16",
            "2005-08-03",
            "2005-10-10",
            "2005-11-01",
            "2005-12-20",
            "2006-01-17",
            "2006-02-01"
        ]
    }
}}
```

8.5.2 Get Shard Statistics

Statistics for a specific shard can be retrieved with any of the following REST commands:

```
GET /{application}/_shards/{shard}
GET /{application}/_statistics/{shard}
GET /{application}/_statistics/{shard}?{params}
```

Statistics will be available for the given {shard} only if at least one batch has been loaded for it and the shard has been merged. The first command shown returns general information for each table and field. A typical response in XML is shown:

```
<stats memory="0.984 MB" storage="1.654 MB" documents="28455">
    <tables>
        <table documents="370" memory="0.010 MB" name="Domain">
            <numFields>
                <field type="BOOLEAN" min="0" max="1" min_pos="1" bits="1" memory="0.000 MB"
                    name="IsInternal"/>
            </numFields>
            <textFields>
                <field valuesCount="370" doclistSize="370" isSingleValued="true" memory="0.001 MB"
                    name="Name"/>
            </textFields>
            <linkFields>
                <field linkedTableName="Address" inverseLink="Domain" doclistSize="1884"
                    isSingleValued="false" memory="0.009 MB" name="Addresses"/>
            </linkFields>
        </table>
        <table documents="6030" memory="0.272 MB" name="Message">
```

```
            ...
        </table>
        ...
    </tables>
</stats>
```

In JSON:

```
{"stats": {
    "memory": "0.984 MB",
    "storage": "1.654 MB",
    "documents": "28455",
    "tables": {
        "Domain": {
            "documents": "370",
            "memory": "0.010 MB",
            "numFields": {
                "IsInternal": {"type": "BOOLEAN", "min": "0", "max": "1", "min_pos": "1", "bits": "1",
                    "memory": "0.000 MB"}
            },
            "textFields": {
                "Name": {"valuesCount": "370", "doclistSize": "370", "isSingleValued": "true",
                    "memory": "0.001 MB"}
            },
            "linkFields": {
                "Addresses": {"linkedTableName": "Address", "inverseLink": "Domain", "doclistSize":
                    "1884", "isSingleValued": "false", "memory":"0.009 MB"}
            }
        },
        "Message": {
            ...
        },
        ...
    }
}}
```

The command response provides statistics about data contained in the given shard such as the total number of objects, the number of objects per table, the name value range of scalar fields, and so forth.

The second shard statistics command returns more similar information as the first command plus additional details about the virtual files (segments) stored within the shard. The third statistics command dumps the contents of a specific virtual file. These commands are intended for debugging purposes.

8.5.3 Merge Shard

All batches that have been loaded for a given shard are merged with the following REST command:

```
POST /{application}/_shards/{shard}[?expire-date=<timestamp>]
```

where {application} is the owning application's name and {shard} is the shard name. The command requires no input entity. It merges all segments synchronously and returns when the merge is complete.

The Merge Shard command optionally assigns an expiration date to the shard via the `expire-date` parameter. The given `<timestamp>` must be in the format of a timestamp (`YYYY-MM-DD HH:MM:SS`); trailing elements can be omitted. Example:

```
POST /Email/_shards/2014-01-01?expire-date=2015-01-01
```

If the `expire-date` is not given, the shard will have no expiration date, even if it was previously assigned one. If the command is successful, a `200 OK` response is returned, and all updates will be visible to queries.

Note that only one process can submit a merge request for a given shard at a time. If a merge request is submitted for a shard that is already undergoing a merge request, the second request immediately receives an error.

8.5.4 Delete Shard

A shard can be explicitly deleted with the following command:

```
DELETE /{application}/_shards/{shard}
```

All of the shard's data is removed, even if it was previously assigned an `expire-date` that has not yet passed. If the request is successful, a `200 OK` response is returned.

8.6 Object Query Commands

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

8.6.1 Object Query via URI

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

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

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

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

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

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

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

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

  ```
  f=*
  ```

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

- **m=*metric expression list*** (optional): A comma-separated list of metric expressions to be computed and returned for each selected objects. The syntax for metric expressions is the same as for aggregate queries. Examples:

```
m=COUNT(DirectReports)
m=COUNT(Messages.MessageAsSender) AS 'Messages Sent'
m=COUNT(Messages.MessageAsSender),SUM(Messages.MessageAsSender.Size)
m=COUNT(DirectReports^.WHERE(LastName:smith))+COUNT(Manager^.WHERE(Department:support))
```

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

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

- **k=*count*** (optional): Causes the first *count* objects in the query results to be skipped. Without this parameter, the first page of objects is returned. Examples:

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

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

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

- **shards=*shards*** (required*): A comma-separated list of shard names. Only the specified shards are searched. Either this or the range parameter must be provided. Examples:

```
shards=2010-01-01
shards=13Q1,13Q2,13Q3
```

- **range=*shard-from*[,*shard-to*]** (required*): A starting shard name and optional ending shard name that defines the range of shards to search. If the ending shard-to name is omitted, all shards whose name is greater than or equal to the shard-from name are searched. Either this or the shards parameter must be provided, but not both. Examples:

```
range=2013-11-01,2013-12-31  // "2013-11-01" <= shard name <= "2013-12-31"
```

```
range=2013-01-01            // "2013-01-01" <= shard name
range=2013                  // "2013" <= shard name
```

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

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

- **Unsorted queries**: To retrieve a secondary page, the same query text should be submitted using either the &g or &e parameter to pass the continuation token from the previous page. This form of paging is more efficient since Doradus OLAP does not re-execute the entire query but instead *continues* the query from the continuation object.

The following object query selects people whose LastName is Powell and returns their full name, their manager's name, and their direct reports' name. The shard named 2014-01-01 is queried:

```
GET /Email/Person/_query?q=LastName=Powell&f=Name,Manager(Name),DirectReports(Name)
    &range=2014-01-01
```

A typical result in XML is shown below:

```
<results>
    <totalobjects>2</totalobjects>
    <docs>
        <doc>
            <field name="Name">Karen Powell</field>
            <field name="_ID">gfNqhYF7LgBAtKTdIx3BKw==</field>
            <field name="DirectReports">
                <doc>
                    <field name="Name">PartnerMarketing EMEA</field>
                    <field name="_ID">mKjYJmmLPoTVxJu2xdFmUg==</field>
                </doc>
            </field>
            <field name="Manager">
                <doc>
                    <field name="Name">David Cuss</field>
                    <field name="_ID">nLOCpa7aH/Y3zDrnMqG6Fw==</field>
                </doc>
            </field>
        </doc>
        <doc>
            <field name="Name">Rob Powell</field>
            <field name="_ID">sHUm0PEKu3gQDDNIHHWv1g==</field>
            <field name="DirectReports"/>
            <field name="Manager">
                <doc>
                    <field name="Name">Bill Stomiany</field>
                    <field name="_ID">tkSQlrRqaeHsGvRU65g9HQ==</field>
                </doc>
            </field>
```

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

The same result in JSON:

```
{"results": {
    "totalobjects": "2",
    "docs": [
        {"doc": {
            "Name": "Karen Powell",
            "_ID": "gfNqhYF7LgBAtKTdIx3BKw==",
            "DirectReports": [
                {"doc": {
                    "Name": "PartnerMarketing EMEA",
                    "_ID": "mKjYJmmLPoTVxJu2xdFmUg=="
                }}
            ],
            "Manager": [
                {"doc": {
                    "Name": "David Cuss",
                    "_ID": "nLOCpa7aH/Y3zDrnMqG6Fw=="
                }}
            ]
        }},
        {"doc": {
            "Name": "Rob Powell",
            "_ID": "sHUm0PEKu3gQDDNIHHWv1g==",
            "DirectReports" :[],
            "Manager": [
                {"doc": {
                    "Name": "Bill Stomiany",
                    "_ID": "tkSQlrRqaeHsGvRU65g9HQ=="
                }}
            ]
        }}
    ]
}}
```

As shown, requested link fields are returned even if when they are empty.

8.6.2 Object Query via Entity

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

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

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

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

Here is an example `search` document in XML:

```
<search>
    <query>LastName=Powell</query>
    <fields>Name,Manager(Name),DirectReports(Name)</fields>
    <shards>2014-01-01</shards>
</search>
```

The same example in JSON:

```
{"search": {
    "query": "LastName=Powell",
    "fields": "Name,Manager(Name),DirectReports(Name)",
    "shards": "2014-01-01"
}}
```

8.7 Aggregate Query Commands

Doradus OLAP supports two commands for submitting aggregate queries. Details of aggregate query parameters and the output formats returned by queries are described in the section **OLAP Aggregate Queries**.

8.7.1 Aggregate Query via URI

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

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

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

- `m=metric expression list` (required): A list of one or more metric expressions to calculate for selected objects. A metric expression is an algebraic expression consisting of functions, constants, simple math operators, and parentheses. Each expression is computed across selected objects, optionally subdivided into groups as defined by the grouping parameter. Example metric parameters:

    ```
    m=COUNT(*)
    m=DISTINCT(Name)
    m=SUM(Size)
    m=MAX(Sender.Person.LastName)
    ```

```
m=AVERAGE(SendDate)
m=COUNT(*) / COUNT(Sender.Person)
m=COUNT(*) / COUNT(Sender.Person) / (DATEDIFF(DAY,"2010-01-01","2011-12-31") + 1)
m=MAX(Size), MIN(Size), AVERAGE(Size), COUNT(*)     // 4 metric epressions
```

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

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

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

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

- **pair=*pair list***: Defines two fields that are used to compute special *dual role* queries. See the section **Pair Parameter** for a description. Example:

  ```
  pair=Sender,InternalRecipients
  ```

- **shards=*shards*** (required*): A comma-separated list of shard names. Only the specified shards are searched. Either this or the `range` parameter must be provided. Examples:

  ```
  shards=2010-01-01
  shards=13Q1,13Q2,13Q2
  ```

- **range=*shard-from*[,*shard-to*]** (required*): A starting shard name and optionally an ending shard name. Only shard names in the specified range are searched. If the ending `shard-to` name is omitted, all shards whose name is greater than or equal to the `shard-from` name are searched. Either this or the `shards` parameter must be provided. Examples:

  ```
  range=2013-11-01,2013-12-31  // "2013-11-01" <= shard name <= "2013-12-31"
  range=2013-01-01             // "2013-01-01" <= shard name
  range=2013                   // "2013" <= shard name
  ```

- **xshards=*shards*** (optional): A comma-separated list of shard names to search for objects referenced by xlinks. If the aggregate query has no xlinks, this parameter is ignored. If this and the `xrange` parameter are both omitted, the scope of xlink searching is the same as the `shards` or `range` parameter. Either this of the `xrange` parameter can be provided, but not both. See the `shards` parameter above for examples.

- **xrange=*shard-from*[,*shard-to*]** (optional): A starting shard name and optional ending shard name that defines the range of shards to search for objects referenced by xlinks. If the ending `shard-to` name is omitted, all shards whose name is greater than or equal to the `shard-from` name are searched. If the aggregate query has no xlinks, this parameter is ignored. If this and the `xshards` parameter are both omitted, the xlink search scope is the same as

defined by the `shards` or `range` parameter. Either this of the `xshards` parameter can be provided, but not both. See the `range` parameter above for examples.

Below is an example aggregate query using URI parameters:

```
GET /Email/Message/_aggregate?q=Size>10000&m=COUNT(*)&f=TOP(3,Sender.Person.Department)&range=2014
```

An example response in XML is:

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

In JSON:

```
{"results": {
    "aggregate": {
        "metric": "COUNT(*)",
        "query": "Size>10000",
        "group": "TOP(3,Sender.Person.Department)"
    },
    "totalobjects": "1254",
    "summary": "1254",
    "totalgroups": "37",
    "groups": [
        {"group": {
            "metric": "926",
            "field": {"Sender.Person.Department": "(null)"}
        }},
        {"group": {
            "metric": "82",
            "field": {"Sender.Person.Department": "HR"}
        }},
        {"group": {
            "metric": "45",
```

```
            "field": {"Sender.Person.Department": "Sales Technical Specialists"}
        }}
    ]
}}
```

8.7.2 Aggregate Query via Entity

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

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

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

URI Parameter	Element name
q	query
m	metric
f	grouping-fields
pair	pair
shards	shards
range	shards-range
xshards	x-shards
xrange	x-shards-range

Below is an example aggregate query request entity in XML:

```
<aggregate-search>
    <query>Size&gt;10000</query>
    <metric>COUNT(*)</metric>
    <grouping-fields>TOP(3,Sender.Person.Department)</grouping-fields>
    <shards-range>2014</shards-range>
</aggregate-search>
```

In JSON:

```
{"aggregate-search": {
    "query": "Size>10000",
    "metric": "COUNT(*)",
    "grouping-fields": "TOP(3,Sender.Person.Department)",
    "shards-range": "2014"
}}
```

8.8 Find Duplicates Command

An object with a given ID can be added to the same table in multiple shards. This is necessary for each shard to be a self-contained graph. But sometimes an application needs to determine which shards have objects with the same ID for a given table. The Find Duplicates command is optimized for this use case has the following form:

```
GET /{application}/{table}/_duplicates[?{params}]
```

where `{application}` is the application name and `{table}` is the name of the table to search. The optional `{params}` define which shards are searched:

- **shards=*shards***: A comma-separated list of shard names. Either this or the `range` parameter can be specified, but not both.

- **range=*shard-from*[,*shard-to*]**: A starting shard name and optional ending shard name. All shards whose name falls between the given shard names (inclusive) are searched. If an ending shard name is not given all shards whose name is greater than or equal to the starting shard name are searched.

Either the `shards` or `range` parameter can be specified, but not both. If neither parameter is specified, all shards are searched. The result of the query is a `results` element containing a `totalobjects` value and a `docs` group, which contains one `doc` element for each object that was found in 2 or more shards. Below is an example:

```
<results>
    <totalobjects>3</totalobjects>
    <docs>
        <doc>
            <field name="_ID">kUNaqNJ2ymmb07jHY9OPOw==</field>
            <field name="shards">2014-01-01,2014-01-02</field>
        </doc>
    </docs>
</results>
```

8.9 Task Status Command

Doradus OLAP applications use background tasks to perform data aging. These tasks execute automatically at their defined frequency. On multi-node clusters, Doradus ensures that each data-aging task is executed by only a single node at each scheduled occurrence. The following REST command is available to list the status of all known tasks:

```
GET /_tasks
```

This command returns a response such as the following:

```
<tasks>
    <task name="Msgs/Message/data-aging">
        <Executor>127.0.0.1</Executor>
        <FinishTime>2015-02-11 14:12:32</FinishTime>
        <StartTime>2015-02-11 14:12:32</StartTime>
        <Status>Completed</Status>
    </task>
    <task name="OLAPEvents/*/data-aging">
        <Executor>127.0.0.1</Executor>
```

```
                <FinishTime>2015-02-11 14:11:31</FinishTime>
                <StartTime>2015-02-11 14:11:31</StartTime>
                <Status>Completed</Status>
            </task>
            <task name="Email/*/data-aging">
                <Executor>127.0.0.1</Executor>
                <FinishTime>2015-02-11 14:12:33</FinishTime>
                <StartTime>2015-02-11 14:12:33</StartTime>
                <Status>Completed</Status>
            </task>
        </tasks>
```

In JSON:

```
{"tasks": {
    "Msgs/Message/data-aging": {
        "Executor": "127.0.0.1",
        "FinishTime": "2015-02-11 14:12:32",
        "StartTime": "2015-02-11 14:12:32",
        "Status": "Completed"
    },
    "OLAPEvents/*/data-aging": {
        "Executor": "127.0.0.1",
        "FinishTime": "2015-02-11 14:11:31",
        "StartTime": "2015-02-11 14:11:31",
        "Status": "Completed"
    },
    "Email/*/data-aging": {
        "Executor": "127.0.0.1",
        "FinishTime": "2015-02-11 14:12:33",
        "StartTime": "2015-02-11 14:12:33",
        "Status":"Completed"
    }
}}
```
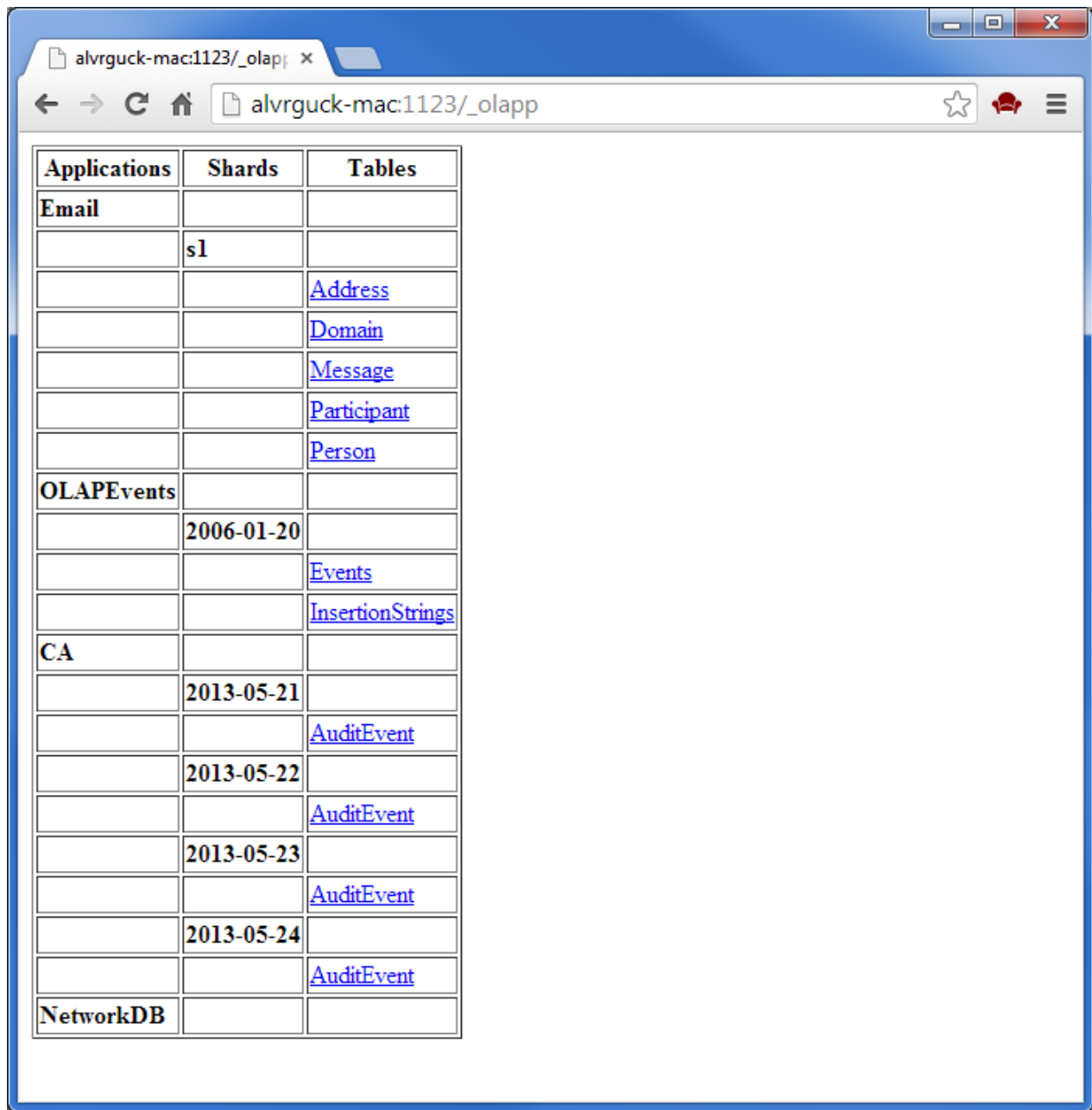
The status of each task is listed using the task name <application>/<table>/data-aging. For OLAP applications, the <table> name is "*". The status, start time, and finish time is shown. The "Executor" is the IP address if the Doradus node that performed the task.

8.10 OLAP Browser Interface

Primarily for testing and debugging, OLAP supports a data browsing URI which is intended to be used via a browser:

```
GET /_olapp
```

(Notice the double "p".) This URI returns a web page the summarizes the currently-defined applications, their shards, and the tables present in each shard. Example:

Applications	Shards	Tables
Email		
	s1	
		Address
		Domain
		Message
		Participant
		Person
OLAPEvents		
	2006-01-20	
		Events
		InsertionStrings
CA		
	2013-05-21	
		AuditEvent
	2013-05-22	
		AuditEvent
	2013-05-23	
		AuditEvent
	2013-05-24	
		AuditEvent
NetworkDB		

Links in this page can be followed to interactively query data within each shard and perform simple queries.