

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 Data Model and Query Language:** Provides a detailed description of the core Doradus data model, query language (DQL), and object and aggregate query commands.
- **Doradus 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 API provided by the Doradus Server.

This document is organized into the following sections:

- **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:** Summarizes core Doradus data model concepts and describes extensions specific to Doradus OLAP such as xlinks.
- **Doradus Query Language Overview:** Provides a summary reference of DQL concepts: perspectives, clauses, link paths, etc.
- **OLAP REST Commands:** Describes the REST commands supported by OLAP databases for schema definition, queries, and shard management.

2. OLAP Database Overview

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

2.1 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"` set the REST API port to 5711.

- **Windows service:** Doradus can be started as a Windows service by using the `procrun` package from Apache Commons. See the Doradus Administration document for details.
- **Embedded app:** Doradus can be embedded in another JVM process. It is started by calling the following method:

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

where:

- `args` is the same arguments parameter passed to `main()` when started as a console application. For example `{"-restport", "5711"}` sets the REST port to 5711.
- `services` is the list of Doradus services to initialize. Each string must be the full package name of a service. The current set of available services is listed below:

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

Required services are automatically included. An embedded application must 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" },  
  ...  
}
```

2.2 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.

2.3 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.

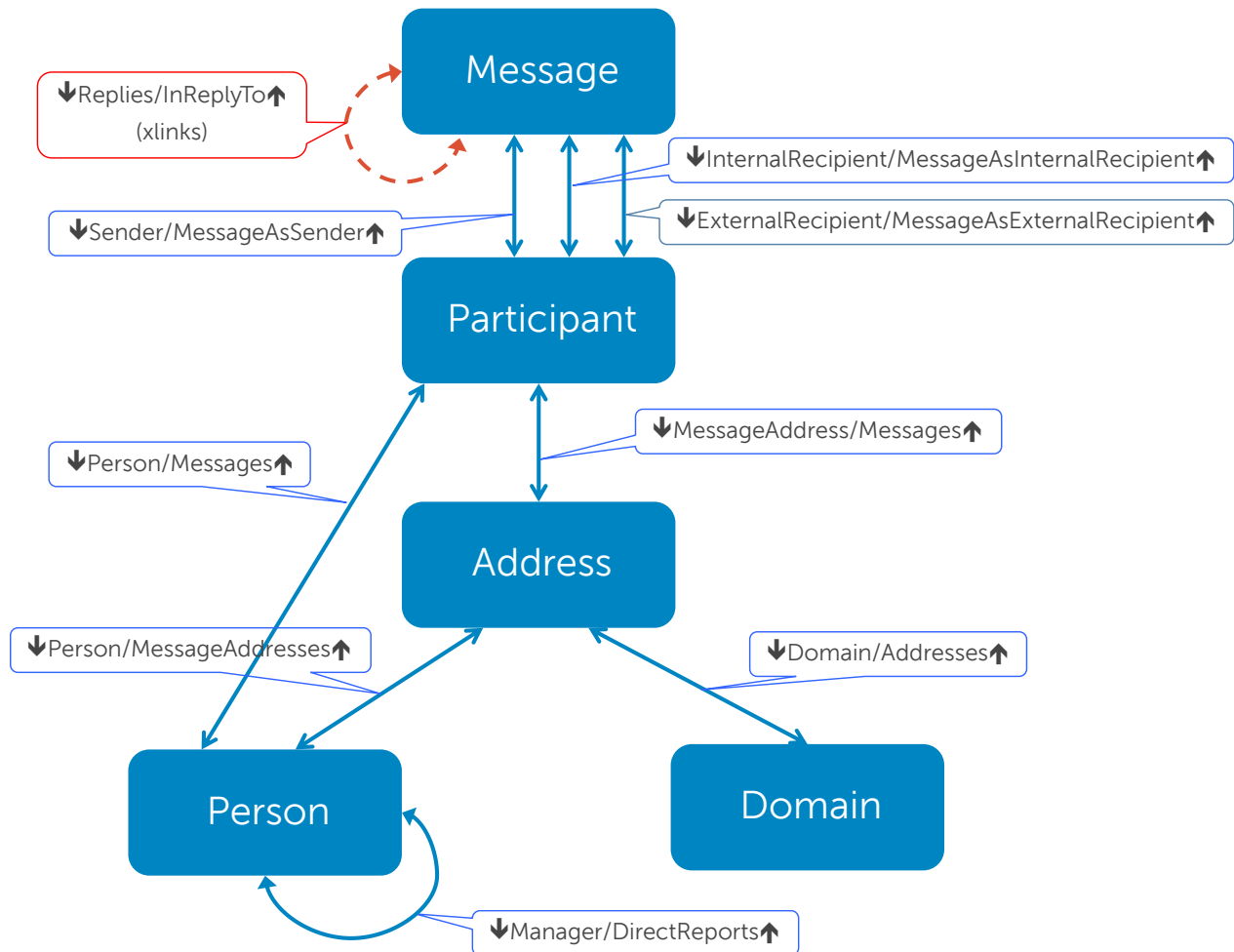
OLAP is not intended for applications with these requirements:

- **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.

2.4 The Email Sample Application

To illustrate features, a sample 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, \Downarrow **Sender** is a link owned by **Message**, pointing to **Participant**, and **MessageAsSender** \Uparrow 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:

```
<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>
    </table>
  </tables>
</application>
```

```

    <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".

3. OLAP Data Model

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

3.1 Core Data Model

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

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

Doradus OLAP places the following restrictions on the core Doradus data model:

- All stored fields must be defined in the schema. If a batch update assigns a value to an undefined field, that assignment is ignored.
- Only `text` fields can be multi-valued.
- A scalar field cannot be set to null once it has been assigned a value. (The entire object can be deleted, however.)
- The only schedule supported by OLAP applications is the `data-aging` schedule, which controls how often the background shard aging task is performed.

3.2 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.

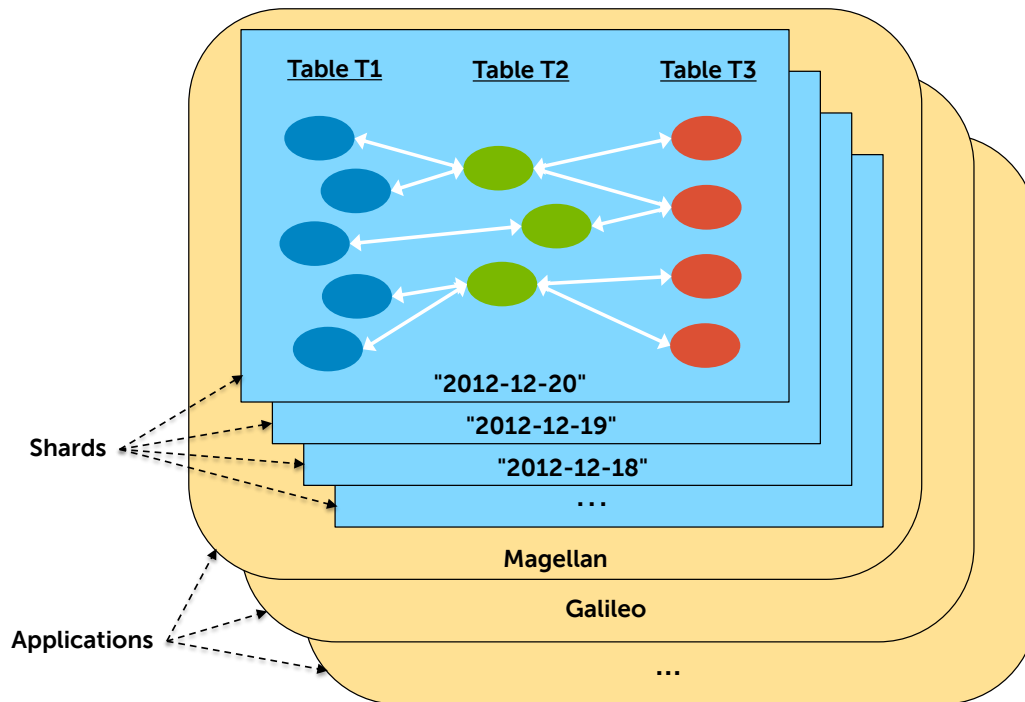
A shard holds objects that are related by user-defined criteria. The most common criteria is time: for example, objects that were created in the same hour or day. But other criteria can be used such as geography or department. The goal of each shard is to hold “a few million” objects that balances load/merge 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 or 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.

3.3 Cross-shard Relationships: Xlinks

With Doradus OLAP, links can only reference objects in the same 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 not 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 new 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
```

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 **Query Commands** 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.

3.4 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. This is done in the `schedules` section of the schema, as in the following XML example:

```
<application name="Email">
  <tables>
    ...
  </tables>
  <schedules>
    <schedule type="data-aging" value="0 5 * * *"/>
  </schedules>
</application>
```

In JSON:

```
{ "Email": {
  "tables": {
    ...
  },
  "schedules": [
    { "type": "data-aging", "value": "0 5 * * *" }
  ]
}}
```

The `type` property must be `data-aging` since this is the only task type supported by OLAP. The `value` property is a cron expression: in this example, the expression means "once per day at 05:00". A cron expression can use the following general format:

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

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

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

Rules for specific patterns are defined below:

- `<minute pattern>`: Values must be in the range 0 to 59.
- `<hour pattern>`: Values must be in the range 0 to 23.
- `<month day pattern>`: Values must be in the range 1 to 31. The special value `L` (uppercase "l") denotes the last day of the month.

- <month pattern>: Values must be in the range 1 (January) to 12 (December). Alternatively, 3-letter aliases can be used: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec.
- <week day pattern>: Values must be in the range 0 (Sunday) to 6 (Monday). Alternatively, 3-letter aliases: can be used: Mon, Tue, Wed, Thu, Fri, Sat, and Sun.

Below are some example cron expressions and their meaning:

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

4. Doradus Query Language (DQL)

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

4.1 Core DQL Overview

4.1.1 Query Perspective

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

4.1.2 Clauses

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

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

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

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

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

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

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

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

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

```
Name="John Smith"
Name="J* Sm?th"
HasBeenSent=false
Size=1024
ModifiedDate="2012-11-16 17:19:12.134"
Manager=sqs284 // Manager is a link; sqs284 is an object ID
_ID='kUNaqNJ2ymbb07jHY90POw==' // Enclose strings in single or double quotes
```

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

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

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

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

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

```
NOT Size=1024
```

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

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

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

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

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

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

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

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

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

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

4.1.3 Link Paths

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

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

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

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

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

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

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

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

4.1.4 Quantifiers

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

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

Is the same as each of these:

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

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

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

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

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

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

- Quantifiers can be used on scalar fields. Examples:

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

4.1.5 Transitive Function

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

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

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

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

4.1.6 Aliases in Queries

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

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

Is expanded and parsed as this DQL clause:

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

4.2 OLAP Aggregate Query Extensions

Doradus OLAP supports extensions to Doradus aggregate queries, described in this section.

4.2.1 Additional Metric Functions

Doradus OLAP supports the core metric functions **AVERAGE**, **COUNT**, **MAX**, **MIN**, and **SUM**. Additionally, OLAP supports two additional functions described below:

- `MAXCOUNT(field)`: This function computes the maximum cardinality of the given *field*. The given field must be a link or text 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*. The given field must be a link or text field. If any selected objects have no value for the given field, the result will be 0.

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

Doradus OLAP also allows the `AVERAGE` function to be applied to link fields, in which case it computes the average number of link values (aka, “fan out”) of each object. That is, `AVERAGE` on a link field behaves like an *average count* function. For example:

```
GET /Emails/Message/_aggregate?m=AVERAGE(InternalRecipients)&range=0
```

This computes the average number of `InternalRecipients` values for each `Message` object. Note that *null elimination* semantics are not used for link fields: that is, objects with no link values are counted while computing the average. Objects with a null link value can be excluded by adding a suitable query parameter:

```
// Don't count objects with no InternalRecipients
GET /Emails/Message/_aggregate?m=AVERAGE(InternalRecipients)&range=0
&q=COUNT(InternalRecipients) > 0
```

4.2.2 Metric Expressions

In core DQL aggregate queries, the metric parameter (`&m`) can be a list of metric functions: `COUNT`, `SUM`, etc. Doradus OLAP extends this functionality, allowing 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 64-bit integers or 64-bit 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.

4.2.3 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`, `HOURL`, `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 date components.

The two timestamp values are truncated to their `<units>` precision, similar to the `TRUNCATE` function. The difference between the two timestamps is 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.

4.2.4 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.

5. OLAP REST Commands

5.1 REST API Overview

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

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

Some example REST commands supported by OLAP are shown below:

- List all application schemas:

```
GET /_applications
```

- Query all shards of the `Email` application's `Person` table for objects whose `LastName` is `Smith`:

```
GET /Email/Person/_query?LastName=Smith&range=0
```

- Submit a batch of objects to the shard titled "2014-05-05":

```
POST /Email/2014-05-05
{"batch": {
  "docs": [
    "doc": {
      "_ID": "58275782",
      "_table": "Person",
      "FirstName": "John",
      "LastName": "Smith",
      ...
    },
    ...
    "doc": {
      ...
    }
  ]
}}
```

- Merge all batches in the shard titled "2014-05-05":

```
POST /Email/_shards/2014-05-05
```

- Perform an aggregate query that returns the top 5 departments with the most people, sub-grouped by the top 3 offices, in the shard "2014-05-05":

```
GET /Email/Person/_aggregate?m=COUNT(*)&f=TOP(5,Department),TOP(3,Office)&shards=2014-05-05
```

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

5.1.1 Common REST Headers

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

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

Header names and values are case-insensitive.

5.1.2 Common REST URI Parameters

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

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

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

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

5.1.3 Common JSON Rules

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

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

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

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

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

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

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

5.1.4 Common REST Responses

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

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

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

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

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

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

5.2 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 <code>/_applications</code>
Modify Application	PUT <code>/_applications/{application}</code>
List Application	GET <code>/_applications/{application}</code>

REST Command	Method and URI
List All Applications	GET /_applications
Delete 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}
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 Management Commands	
Get All Tasks Status	GET /_tasks
Get Application Task Status	GET /_tasks/{application}
Modify Application Task Status	PUT /_tasks/{application}?{command}
OLAP Browser Interface	
OLAP Browser Home	GET /_olapp

Details on each command are described in the following sections.

5.3 Application Management Commands

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

5.3.1 Create Application

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

```
POST /_applications
```

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

Because Doradus 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.

OLAP supports all core Doradus data model concepts with the following restrictions:

- The only application-level option supported is `StorageService`, which can be explicitly set to `OLAPService` if there are multiple storage services supported by the server.
- Only text fields can be declared as multi-valued (`collection=true`).
- The only task schedule type supported by OLAP is `data-aging`. This task schedule defines how often the data aging task looks for and deletes expired shards.

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

5.3.2 Modify Application

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

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

where `{application}` is the application's name. The request must include the 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. 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.

5.3.3 List Application

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

```
GET /_applications
```

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

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

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

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

5.3.4 Delete Application

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

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

where `{application}` is the application's name. The `{key}` must match the application's defined key as a safety mechanism.

5.4 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.

5.4.1 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}
```

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 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:

```
<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">95lfrCiljbiKQK9UH7LYg==</field>
    </doc>
    <doc _table="Person">
      <field name="_ID">x30KbjCmKw47wEHaqV0nLQ==</field>
      <field name="FirstName">John</field>
      <field name="Manager">
        <add>
          <value>LjJEtDcwp11tqJWJ980+HQ==</value>
        </add>
      </field>
      ...
    </doc>
    ...
  </docs>
</batch>
```

In JSON:

```
{"batch": {
```

```
"docs": [
  {"doc": {
    "_table": "Message",
    "_ID": "92XJedWQ81D3/RS4yM5gTg==",
    "Size": "10334",
    "Tags": {
      "add": ["Confidential", "Sensitive"]
    },
    ...
  }},
  {"doc": {
    "_table": "Message",
    "_deleted": "true",
    "_ID": "951frCiljbiK0QK9UH7LYg=="
  }},
  {"doc": {
    "_table": "Person",
    "_ID": "x30KbjCmKw47wEHaqV0nLQ==",
    "FirstName": "John",
    "Manager": {
      "add": ["LjJEtDcwp11tqJWJ980+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`.

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 as 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 compression helps with server memory because object batches are parsed and loaded from the compressed message entity.

5.4.2 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">95lfrCiljbiK0QK9UH7LYg==</field>
    </doc>
    <doc _table="Person">
      <field name="_ID">x30KbjCmKw47wEHaqV0nLQ==</field>
    </doc>
    ...
  </docs>
</batch>
```

In JSON:

```
{ "batch": {
  "docs": [
    { "doc": {
      "_table": "Message",
      "_ID": "92XJeDwQ8lD3/RS4yM5gTg=="
    } },
    { "doc": {
      "_table": "Message",
      "_ID": "95lfrCiljbiK0QK9UH7LYg=="
    } },
    { "doc": {
      "_table": "Person",
      "_ID": "x30KbjCmKw47wEHaqV0nLQ=="
    } },
    ...
  ]
}
```



```
]
}}
```

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.

5.5 Shard Management Commands

Doradus OLAP provides the following commands to list, merge, and delete shards.

5.5.1 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"
    ]
  }
}}
```

5.5.2 Get Shard Statistics

Statistics for a specific shard can be retrieved with the following REST command:

```
GET /{application}/_shards/{shard}
```

Statistics will be available for a shard only if at least one batch has been loaded for it and the shard has been merged. 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.

5.5.3 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.

5.5.4 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.

5.6 Object Query Commands

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

5.6.1 Object Query via URI

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

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

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

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

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

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

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

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

- **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
```

- **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=2014-01-01           // "2013-01-01" <= shard name
range=13                   // "13" <= shard name
```

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.

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">gfNqhYF7LgBAAtKTdIx3BKw==</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"/>
    </doc>
  </docs>
</results>
```

```
<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": "gfNqhYF7LgBAAtKTdIx3BKw==",
          "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.

5.6.2 Object Query via Entity

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

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

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

URI Parameter	Element name
f	fields
k	skip
o	order
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"
}}
```

5.7 Aggregate Query Commands

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

5.7.1 Aggregate Query via URI

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

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

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

- **m=metric 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=2014-01-01           // "2013-01-01" <= shard name
```



```
range=13                // "13" <= 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 or 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 or 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"}}
  }
]
}
}

```

5.7.2 Aggregate Query via Entity

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

```

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

```

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

URI Parameter	Element name
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>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"
}}
```

5.7.3 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.

5.8 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">kUNaQNJ2ymmb07jHY90POw==</field>
      <field name="shards">2014-01-01,2014-01-02</field>
    </doc>
  </docs>
</results>
```

5.9 Task Management Commands

Doradus OLAP performs automatic data aging by assigning an `expire-date` to shards via the Merge Shard command. When an application has at least one shard with an `expire-date`, a background data-aging task is automatically created, and its default schedule is to look for expired shards once per day at midnight. When the data-aging task executes, it looks for and deletes shards older than their assigned `expire-date`. (Deleting expired shards is efficient and does not require many resources.)

For most applications, the default data aging policy is sufficient. However, an application can change the schedule at which its data-aging task is performed by including a schedule declaration in its schema. In addition, the Doradus Task Manager service provides REST commands that allow dynamic interrogation and modification of tasks. These commands are described in this section.

5.9.1 Get Task Status

The following commands get the current status of all tasks currently known to the Task Manager:

```
GET /_tasks
```

Alternatively, the tasks for a specific application or task can be obtained. Since OLAP applications use a single data-aging task, the following commands are all equivalent:

```
GET /_tasks/{application}
GET /_tasks/{application}/*
GET /_tasks/{application}/*/data-aging
```

where `{application}` is an application name. The Get Task Status commands return a `tasks` document with one `task` group for each task containing its name, schedule, and most recent execution information. An example in XML:

```
<tasks>
  <task name="Email/*/data-aging">
    <schedule>0 0 3 * *</schedule>
    <state>Succeeded</state>
    <last-scheduled-time>Tue Feb 18 13:48:00 PST 2014</last-scheduled-time>
    <last-started-time>Tue Feb 18 13:49:02 PST 2014</last-started-time>
    <last-finished-time>Tue Feb 18 13:49:03 PST 2014</last-finished-time>
  </task>
  <task name="OLAPEvents/*/data-aging">
    <schedule>0 0 * * *</schedule>
    <state>Undefined</state>
  </task>
</tasks>
```

In JSON:

```
{"tasks": [
  {"task": {
    "name": "Email/*/data-aging",
    "schedule": "0 0 3 * *",
```

```
    "state": "Succeeded",
    "last-scheduled-time": "Tue Feb 18 13:48:00 PST 2014",
    "last-started-time": "Tue Feb 18 13:49:02 PST 2014",
    "last-finished-time": "Tue Feb 18 13:49:03 PST 2014"
  }},
  {"task": {
    "name": "OLAPEvents/*/data-aging",
    "schedule": "0 0 * * *",
    "state": "Undefined"
  }}
]}
```

This example shows the status of data-aging tasks for two OLAP applications:

- `Email/*/data-aging`: This is the data-aging task for the `Email` application. Its schedule is `"0 0 3 * *"` (every day at 03:00). Its last execution was successful with the timestamps shown.
- `OLAPEvents/*/data-aging`: This is the data-aging task for the `OLAPEvents` application. It has the default schedule `"0 0 0 * *"` (every day at midnight) and has not yet executed.

5.9.2 Modify Task

An OLAP application's data-aging task can be modified with a PUT command that has no input entity. Because each OLAP application has a single data-aging task, the following commands are all equivalent:

```
PUT /_tasks/{application}?{command}
PUT /_tasks/{application}/*?{command}
PUT /_tasks/{application}/*/data-aging?{command}
```

The possible values for `{command}` and its effect are:

- `start`: Immediately starts the data-aging task (if not already running).
- `interrupt` (synonym: `stop`): Sends a signal to stop the data-aging task if it is running. The time it takes for the task to respond to an `interrupt` command is variable.
- `suspend`: Stops the data-aging task from being scheduled. If it is already running, the task keeps running until it is finished or interrupted.
- `resume`: Resumes scheduling of the data-aging task according to its schedule.

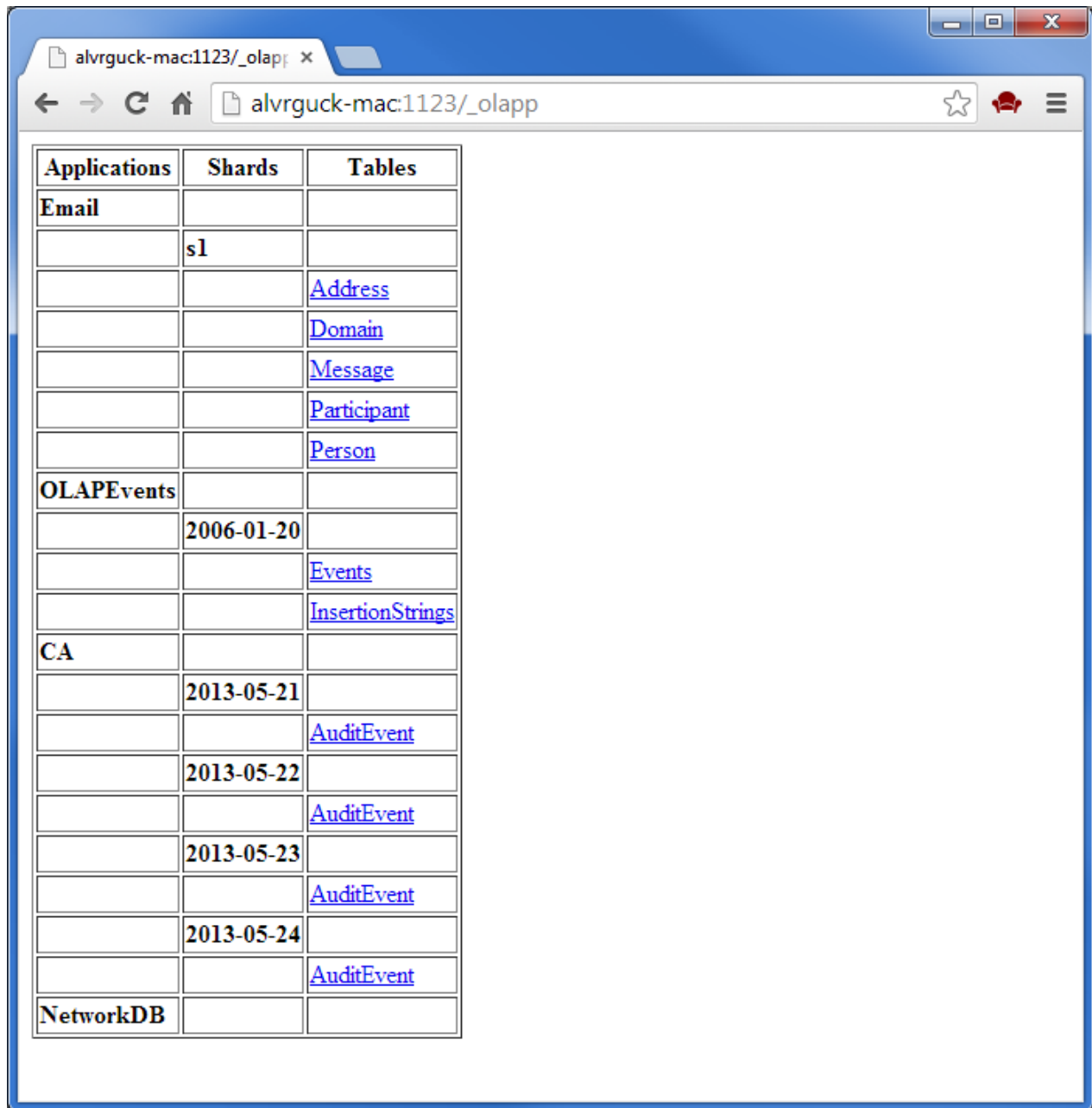
If successful, the command returns a `200 OK` response.

5.10 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 that summarizes the currently-defined applications, their shards, and the tables present in each shard. Example:



The screenshot shows a web browser window with the address bar displaying 'alvrguck-mac:1123/_olapp'. The main content area contains a table with three columns: Applications, Shards, and Tables. The table lists several applications and their associated shards and tables.

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.