Oracle® Database

Oracle Database API for MongoDB

F44905-10
November 2023

Oracle Database Oracle Database API for MongoDB,

F44905-10

Primary Author: Drew Adams

Contents

Index

List of Examples

List of Tables

Preface

This document provides a conceptual overview of Oracle Database API for MongoDB.

- Audience
- Documentation Accessibility
- Related Resources
- Conventions

Audience

This document is intended for users of Oracle Database API for MongoDB.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Resources

For more information, see these Oracle resources:

- Oracle Database API for MongoDB at Oracle Help Center for complete information about this product
- Autonomous JSON Database
- *Oracle Database JSON Developer's Guide*
- *Oracle as a Document Store* for general information about using JSON data in Oracle Database, including with Simple Oracle Document Access (SODA) and Oracle Database API for MongoDB

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Overview of Oracle Database API for MongoDB

Oracle Database API for MongoDB lets applications interact with collections of JSON documents in Oracle Database using MongoDB commands.

Oracle Database API for MongoDB is provided as part of Oracle Autonomous Database Serverless. You can enable it there using the Oracle Cloud Infrastructure Console. See Configure Access for MongoDB in *Using Oracle Autonomous Database Serverless*.

If you have release 22.3 or later of Oracle REST Data Services (ORDS), then you can use the MongoDB API with any Oracle database, release 21c or later, as well as with any Oracle Autonomous Database, release 19c (serverless, dedicated, and cloud@customer). See Oracle API for MongoDB Support in *Oracle REST Data Services Installation and Configuration Guide* for information about enabling the API.

- Purpose of Oracle Database API for MongoDB
  Oracle Database understands *Mongo-speak*. That's the purpose of Oracle Database API for MongoDB.
- Tools and Drivers for Oracle Database API for MongoDB
  Oracle Database API for MongoDB supports a variety of MongoDB tools and drivers.
- Terms and Concepts: MongoDB and Oracle Database
  Some application-user terms and concepts used by MongoDB are presented, together with description of their relation to Oracle Database..
- Default Naming of a Collection Table
  By default, the name of the database table that underlies a document collection is derived from the collection name.
- Mongo DB API Collections Supported by JSON-Relational Duality Views
  A JSON-relational duality view supports JSON documents that are *generated*, based on underlying table data. A JSON document collection that's compatible with Oracle Database API for MongoDB has a single document-identifier field, named **`_id`**. After creating a duality view, you use PL/SQL function `DBMS_SODA.create_dualv_collection` to create the collection it supports.

> **See Also:**
>
> Using the Oracle Database API for MongoDB in *Using Oracle Autonomous Database Serverless* for information about using an Autonomous Database (including an Autonomous JSON Database) with Oracle Database API for MongoDB. This covers configuring the database for use with the API, including for security and connection.

1.1 Purpose of Oracle Database API for MongoDB

Oracle Database understands *Mongo-speak*. That's the purpose of Oracle Database API for MongoDB.

You have one or more applications that interact with a MongoDB NoSQL database, and you want to migrate the data to Oracle Database. Or you have relatively simple collections of JSON documents and you prefer not to learn and use SQL (Structured Query Language). Or you're used to and prefer to use MongoDB commands, particularly for the business logic of your applications (query by example) but also for data definition (creating collections and indexes), data manipulation (CRUD operations), and some database administration (status information). You appreciate the flexibility of a JSON document store: no fixed data schemas, easy to use document-centric APIs.

If you have applications that use MongoDB, you'd like to make them more robust by providing advanced security; fully ACID transactions (atomicity, consistency, isolation, durability); standardized JOINs with all sorts of data; and analytics, machine-learning, and reporting capabilities.

Oracle Database API for MongoDB, or **Mongo API** for short, provides such advantages to developers who speak MongoDB. It translates the MongoDB wire protocol into SQL statements that are executed by Oracle Database. You can continue to use the drivers, frameworks, and tools you're used to, to develop your JSON document-store applications.

Oracle Database is a *converged* database. It's multi-model and polyglot — seemingly different kinds of databases rolled into one, providing synergy across very different features, supporting different workloads and data models.

Oracle Database is also *multitenant*, which means you can have both consolidation and isolation, for different teams and purposes. And it provides a single, common approach for security, upgrades, patching, and maintenance. But if you use an Autonomous Oracle Database, such as Autonomous JSON Database, then Oracle takes care of all such database administration responsibilities. And there's Always Free access to an autonomous database.

The standard, declarative language SQL (Structured Query Language) underlies processing on Oracle Database. You might develop applications using Mongo-speak or Simple Oracle Document Access (SODA) with a popular application development language, but SQL is behind it all, and it enables your app to play well with everything else on Oracle Database.

1.2 Tools and Drivers for Oracle Database API for MongoDB

Oracle Database API for MongoDB supports a variety of MongoDB tools and drivers.

Oracle recommends that you use the following tool and driver versions, or higher, with support for load-balanced connections.

- C 1.19.0
- C# 2.13.0
- Compass 1.28.1
- Database Tools 100.5.0 (includes `mongoexport`, `mongorestore`, and `mongodump`)

- Go 1.6.0
- Java 4.3.0
- MongoSH 0.15.6
- Node.js driver 4.1.0
- PyMongo 3.12.0 (for Python language)
- Ruby 2.16.0
- Rust 2.1.0

You can download these drivers from https://www.mongodb.com/docs/drivers/.

1.3 Terms and Concepts: MongoDB and Oracle Database

Some application-user terms and concepts used by MongoDB are presented, together with description of their relation to Oracle Database..

Some of the same terms are also used in Oracle Database API for MongoDB. In general, application developers need not be concerned with the Oracle Database concepts and technologies that underlie such terms.

Table 1-1 Application-User Terms

Term	Description
Database	A set of collections. On Oracle Database this corresponds to a database *schema*. Because of this possible confusion over use of the word *database*, in this documentation that word is used for Oracle Database, and the term **schema**, or **database schema**, is used for what MongoDB calls a "database".
User	For log-in purposes, a **user** of Oracle Database API for MongoDB is an Oracle Database user, which is also called a database schema (see previous). To use the collections in a given schema ("database") , you log in with the Oracle Database API for MongoDB using the MongoDB `PLAIN $external` mechanism and providing the credentials for that schema. A **root user**, that is, a user who has MongoDB role `root`, can create additional database schemas. And a root user can use the collections of any schema without needing to log in separately for that schema.
Collection	A collection contains a set of documents. A collection name is unique for a given database schema: Different collections can have the same name if they are in different schemas. On Oracle Database, a table or a view underlies a collection. The table name is derived from the collection name and is typically the same. (Exceptions include collection names that use words reserved by Oracle Database.) Typically all documents in a collection are JSON documents.
Document	The basic unit of storage for data in a collection. On Oracle Database a document corresponds roughly to a row in the table or view that underlies the collection. A document is typically a JSON document, that is, it contains only JSON data. On Oracle Autonomous Database a document is always a JSON document. On Oracle Autonomous Database the table **column** used to store documents is named **`data`**.

Table 1-1 (Cont.) Application-User Terms

Term	Description
Primary Key	On Oracle Database a primary key is used to uniquely identify a table or view row. MongoDB uses a unique **`_id`** field in a document to identify the document. On Oracle Database the primary key for a JSON document is stored in a column named **`id`**. Its value is automatically set to the value of the document's `_id` field. See Document Key: Differences and Conversion (Oracle Database Prior to 23c).
Query Expression	A JSON object that is sent by an application client to the server (Oracle Database), to query documents of a collection. The object can contain **query operator** fields, whose names start with **`$`**. The operators are interpreted, and their operations are invoked to act on the collection. The server returns the action results to the client. Query expressions are typically used to query a collection, but they can also be used to project or update data in documents. Oracle Database API for MongoDB translates query expressions into SQL (Structured Query Language) queries.
Index	Indexes enhance performance when acting on collections (querying, inserting, updating, and deleting documents). An index name is unique for a given database schema: Different indexes can have the same name if they are in different schemas. **Note:** If Oracle Database parameter `compatible` is less than `23` then MongoDB commands to create or drop indexes are ignored by Oracle Database API for MongoDB. You must instead create Oracle Database indexes that are relevant for your JSON data.
Pipeline	MongoDB aggregation operations chain multiple operations together, invoking them sequentially as a pipeline. If Oracle Database parameter `compatible` is less than `23` then MongoDB aggregation pipelines are not used; Oracle Database API for MongoDB carries out aggregation operations differently. See MongoDB Aggregation Pipeline Support.

Related Topics

- MongoDB Documents and Oracle Database
  Presented here is the relationship between a JSON document used by MongoDB and the same content as a JSON document stored in, and used by, Oracle Database.
- Migrate Application Data from MongoDB to Oracle Database
  Some ways to export your JSON data from MongoDB and then import it into Oracle Database are described. Migration considerations are presented.

See Also:

- Overview of SODA Document Collections in *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about collections
- Overview of SODA Documents in *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about documents
- Overview of SODA Filter Specifications (QBEs) in *Oracle Database Introduction to Simple Oracle Document Access (SODA)* for information about QBEs
- Query JSON Data in *Oracle Database JSON Developer's Guide* for information about querying JSON data using SQL

1.4 Default Naming of a Collection Table

By default, the name of the database table that underlies a document collection is derived from the collection name.

If you want a different table name from that provided by default then use custom collection metadata to explicitly provide the name.

The *default table name* is derived from the collection name you provide, as follows:

1. Each ASCII control character and double quotation mark character (`"`) in the collection name is replaced by an underscore character (`_`).
2. If *all* of the following conditions apply, then all letters in the name are converted to *uppercase*, to provide the table name. In this case, you need not quote the table name in SQL code; otherwise, you must quote it.
   - The letters in the name are either all lowercase or all uppercase.
   - The name begins with an ASCII letter.
   - Each character in the name is alphanumeric ASCII, an underscore (`_`), a dollar sign (`$`), or a number sign (`#`).

   Note:

   Oracle recommends that you do *not* use dollar-sign characters (`$`) or number-sign characters (`#`) in Oracle identifier names.

For example:

- Collection names "col" and "COL" both result in a table named "COL". When used in SQL, the table name is interpreted case-insensitively, so it need not be enclosed in double quotation marks (`"`).
- Collection name "myCol" results in a table named "myCol". When used in SQL, the table name is interpreted case-sensitively, so it must be enclosed in double quotation marks (`"`).

1.5 Mongo DB API Collections Supported by JSON-Relational Duality Views

A JSON-relational duality view supports JSON documents that are *generated*, based on underlying table data. A JSON document collection that's compatible with Oracle Database API for MongoDB has a single document-identifier field, named **`_id`**. After creating a duality view, you use PL/SQL function `DBMS_SODA.create_dualv_collection` to create the collection it supports.

JSON-relational duality views are supported only in Oracle Database Release 23c or later.

A **JSON-relational duality view** exposes data stored in relational database tables as JSON documents. The documents are materialized on demand, not stored as such. Duality views give data both a conceptual and an operational duality: it's organized both relationally and hierarchically. You can base different duality views on data stored in one or more of the same tables, providing different JSON hierarchies over the same, shared data.

This means that applications can access (create, query, modify) the same data as a collection of JSON documents or as a set of related database tables and columns, and both approaches can be employed at the same time.

You can manipulate the documents realized by duality views in the ways you're used to, using your usual drivers, frameworks, tools, and development methods. In particular, applications can use any programming languages. An application uses a collection that's supported by a duality view as if the documents were stored in a table column of `JSON` data type.

As one important use case, a MongoDB API application can easily make use of any *existing database data* — just create one or more duality views over that data, to support JSON collections.

An important aspect of the JSON-relational duality is that it lets different kinds of JSON document *share* common data (as well as share the same data in relational tables). How you define a duality view determines what data gets shared, and how (who can perform what kinds of updating operations on which document parts).

Creating MongoDB API-Compatible Duality Views

You cannot *create* a JSON-relational view using the MongoDB API. You use SQL statement `CREATE JSON RELATIONAL DUALITY VIEW` to do that.

A duality view that is compatible with the MongoDB API always has field `_id` as its document identifier. Its value specifies the document fields whose values are the primary-key columns of the root table that underlies the duality view.

- If there is only *one primary-key column* then you use that column as the value of field `_id` when you define the duality view. For example: `_id : race_id`, as in Example 1-1.
- If there are *multiple primary-key columns* then you use an *object* as the value of field `_id` when you define the view. The members of the object specify document fields whose values are the primary-key columns. For example, suppose you have a car-racing duality view with two primary-key columns, `race_id` and `race_year`,

which together uniquely identify a root-table row, but neither of which does so alone. This `_id` field in the duality view definition maps document fields `raceId` and `year` to primary-key columns `race_id` and `race_year`, respectively:

```
_id : {raceId : race_id, year : race_year}
```

If there is only one primary-key column, you can nevertheless use an object value for `_id`, if you like. Doing so lets you provide a meaningful field name. For example, here the single primary-key column, `race_id`, provides the value of field `raceId` as well as the value of field `_id`:

```
_id : {raceId : race_id}
```

The value(s) provided by field `_id` for the primary key column(s) it maps to must of course be insertable into those columns, which means that their data types must be compatible with the column types. For example, if field `_id` maps to a single primary-key column that is of SQL type `NUMBER`, then the `_id` value of a document you insert must be numeric. Otherwise, an error is raised for the insertion attempt.

If you don't explicitly include an `_id` field in a document that you insert, then it is added automatically, with an `ObjectId` value. (You can also explicitly use an `ObjectId` value in an `_id` field.) An `ObjectId` value can only be used for a field that the duality view maps to a column of SQL type `RAW`.

Example 1-1 Creating MongoDB API-Compatible Duality View RACE_DV Using GraphQL

This example creates a duality view, `race_dv`, that supports car-racing race documents.

```
CREATE JSON RELATIONAL DUALITY VIEW race_dv AS
  race @insert @update @delete
    {_id : race_id
     name   : name
     laps   : laps @noupdate
     date   : race_date
     podium : podium @nocheck
     result : driver_race_map @insert @update @delete
       [ {driverRaceMapId : driver_race_map_id
          position        : position
          driver @noinsert @update @nodelete
                 @unnest {driverId : driver_id}")
                          name     : name}} ]};
```

This definition is similar to the one in Creating Duality View RACE_DV Using GraphQL in *JSON-Relational Duality Developer's Guide*, but that view is *not* compatible with the MongoDB API because its document-identifier field is not `_id`. (See that documentation for similar duality view creations for driver and race documents.)

This duality view supports JSON documents where the race objects look like this — they contain a `result` field whose value is an array of objects that specify the drivers and their resulting positions in the given race:

```
{"_id" : 201,
"name" : "Bahrain Grand Prix",
```

```
"laps" : 57,
"date" : "2022-03-20T00:00:00",
"podium" : {...},
"result" : [ {"driverRaceMapId" : 3,
              "position" : 1,
              "driverId" : 103,
              "name" : "Charles Leclerc"},... ]}
```

The value of document identifier field `_id` is taken from the single primary-key column, `race_id` of the root table, `race`. For example, the document identified by the `_id` field whose value is `201` is generated from the row of data that has `201` in primary-key column `race_id` of the root table (`race`) underlying the duality view.

Generation of the documents supported by the view automatically joins data from columns `driver_race_map_id`, `position` and `driver_id` from table `driver_race_map`, and column `name` from table `driver`.

The annotations (GraphQL directives) `@insert`, `@update`, and `@delete` are used to specify that applications can insert, update, and delete documents supported by the view, respectively, but that they can only perform update operations on the `driver` field of the documents (a driver cannot be inserted or deleted when you modify a race document) and you cannot update the `laps` field (you cannot change the number of laps when you update a race document).

The `@nocheck` annotation applied to column `podium` specifies that updating field `podium` in a race document does not contribute to checking the state/version of the document (its ETAG value).

Example 1-2 Creating a MongoDB API-Compatible Document Collection Based On a Duality View

You use the MongoDB API to access and manipulate JSON document collections supported by a duality view. (You don't (cannot) access the view directly using the API.)

After creating a duality view you need to create a collection that's supported by it and that's compatible with the MongoDB API. You use PL/SQL function **`create_dualv_collection`** in package `DBMS_SODA` to do that. (You can't use the API itself to create such a collection.)

This example creates a collection of race documents, using duality view `race_dv`. The first argument to the function is the output collection name; the second argument is the input duality view name.

```
DECLARE col soda_collection_t;
BEGIN
  col := DBMS_SODA.create_dualv_collection('race_documents',
'RACE_DV');
END;
```

See Also:

- CREATE JSON RELATIONAL DUALITY VIEW in *Oracle Database SQL Language Reference*
- Document-Identifier Fields for Duality Views
- Annotations (NO)UPDATE, (NO)INSERT, (NO)DELETE, To Allow/Disallow Updating Operations
- Annotation (NO)CHECK, To Include/Exclude Fields for ETAG Calculation

2

Develop Applications with Oracle Database API for MongoDB

Considerations when developing or migrating applications — a combination of (1) how-to information and (2) descriptions of differences and possible adjustments.

- Indexing and Performance Tuning
  Oracle Database offers multiple technologies to accelerate queries over JSON data, including indexes, materialized views, in-memory column storage, and Exadata storage-cell pushdown. Which performance-tuning approaches you take depend on the needs of your application.
- Users, Authentication, and Authorization
  Oracle Database security differs significantly from that of MongoDB. The security model of Oracle Database API for MongoDB is described: the creation of users, their authentication, and their authorization to perform different operations.
- Migrate Application Data from MongoDB to Oracle Database
  Some ways to export your JSON data from MongoDB and then import it into Oracle Database are described. Migration considerations are presented.
- MongoDB Aggregation Pipeline Support
  Oracle Database API for MongoDB supports MongoDB aggregation pipelines, that is, MongoDB command `aggregate`. It lets you use pipeline code to execute a query as a sequence of operations. You can also use SQL as a declarative alternative to this procedural approach.
- MongoDB Documents and Oracle Database
  Presented here is the relationship between a JSON document used by MongoDB and the same content as a JSON document stored in, and used by, Oracle Database.
- Other Differences Between MongoDB and Oracle Database
  Various differences between MongoDB and Oracle Database are described. These differences are generally not covered in other topics. Consider these differences when you migrate an application to Oracle Database or you develop a new application for Oracle Database that uses MongoDB commands.
- Accessing Collections Owned By Other Users (Database Schemas)
  You can directly access a MongoDB API collection owned by another user (database schema) if you log into that schema. You can indirectly access a collection owned by another user, without logging into that schema, if that collection has been mapped to a collection in your schema.

2.1 Indexing and Performance Tuning

Oracle Database offers multiple technologies to accelerate queries over JSON data, including indexes, materialized views, in-memory column storage, and Exadata storage-cell pushdown. Which performance-tuning approaches you take depend on the needs of your application.

If your Oracle Database `compatible` parameter is `23` or greater, then you can use MongoDB index operations `createIndex` and `dropIndex` to automatically create and drop the relevant

Oracle indexes. If parameter `compatible` parameter is less than `23`, then such MongoDB index operations are not supported; they are *ignored*.

Regardless of your database release you can create whatever Oracle Database indexes you need directly, using (1) the JSON Page of *Using Oracle Database Actions* (see Creating Indexes for JSON Collections), (2) Simple Oracle Document Access (SODA), or (3) SQL — see Indexes for JSON Data in *Oracle Database JSON Developer's Guide*. Using the JSON page is perhaps the easiest approach to indexing JSON data.

> **Note:**
>
> MongoDB allows different collections in the same "database" to have indexes of the same name. This is not allowed in Oracle Database — the name of an index must be unique across all collections of a given database schema ("database").

Consider, for example, indexing a collection, named `orders`, of purchase-order documents such as this one:

```
{ "PONumber" : 1600,
  "User" : "ABULL",
  "LineItems" : [{ "Part"     : { "Description" : "One Magic Christmas",
                                  "UnitPrice"   : 19.95,
                                  "UPCCode"     : 13131092899 },
                   "Quantity" : 9.0 },
                 { "Part"     : { "Description" : "Lethal Weapon",
                                  "UnitPrice"   : 19.95,
                                  "UPCCode"     : 85391628927
                                },
                   "Quantity" : 5.0 } ]}
```

Two important use cases are (1) indexing a singleton scalar field, that is, a field that occurs only once in a document (2) indexing a scalar field in objects within the elements of an array. Indexing the value of field `PONumber` is an example of the first case. Indexing the value of field `UPCCode` is an example of the second case.

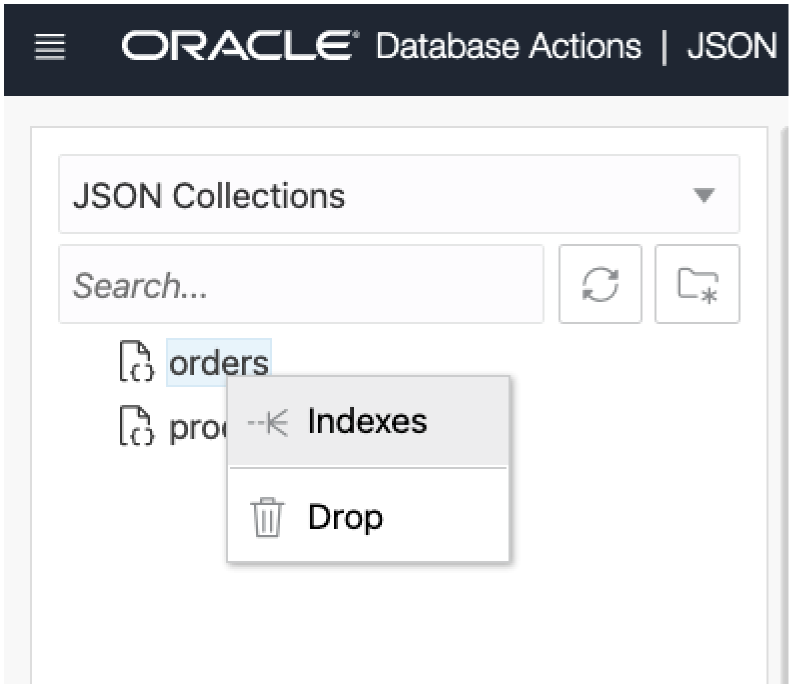
Example 2-1, Example 2-2, and Example 2-3 illustrate the first case. Example 2-5 illustrates the second case.

You can also index GeoJSON (spatial) data, using a function-based SQL index that returns `SDO_GEOMETRY` data. And for all JSON data you can create a JSON *search index*, and then perform full-text queries using SQL/JSON condition `json_textcontains`.

Example 2-1 Indexing a Singleton Scalar Field Using the JSON Page of Database Actions

To create an index for field `PONumber` using the JSON Page, do the following.

1. Right-click the collection name (`orders`) and select **Indexes** from the popup menu.



2. In the **New Index** page:

   - Type `*` in the **Properties** search box.

     This populates the **Properties** list with paths to all scalar fields in your collection. These paths are provided by sampling the collection data using a JSON data guide — see JSON_DATAGUIDE in *Oracle Database SQL Language Reference*.

     If you turn on option **Advanced**, by pushing its slider to the right, then the types of the listed scalar fields are also shown. The types shown are those picked up by sampling the collection. But you can change the type of a field for indexing purposes.

   - Select the paths of the fields to be indexed. In this case we want only a single scalar field indexed, `PONumber`, so select that.

     Note: This dialog box lets you select multiple paths. If you select more than one path then a composite index is created for the data at those paths.[1] But if you want to index two different fields separately then create two indexes, not one composite index (which indexes both fields together).

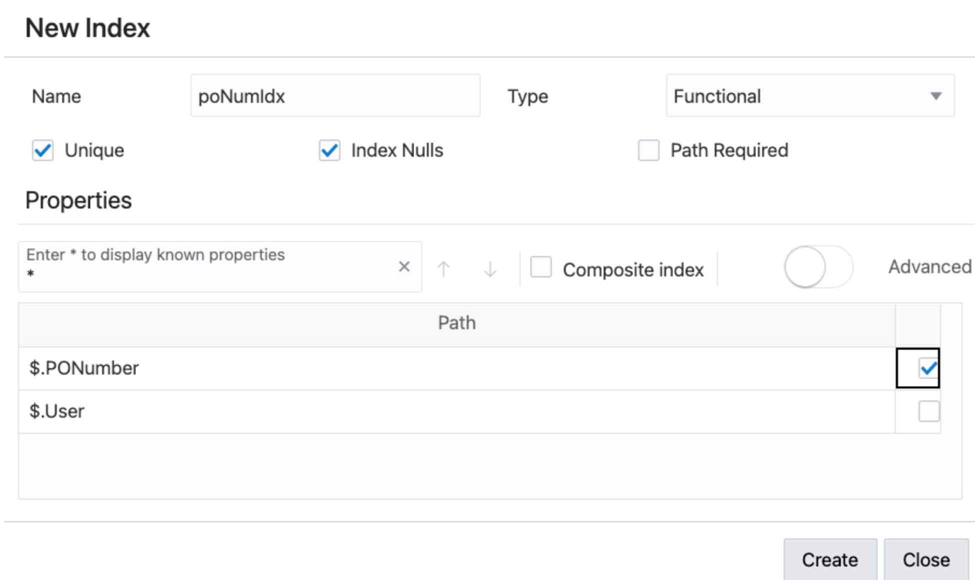
     The index data type is determined automatically by the types of the data at the selected paths, but you can control this by turning on **Automatic** and changing the data types. For example, JSON numbers in the collection data for a given field cause a type of `number` to be listed, but you can edit this to `VARCHAR2` to force indexing as a string value.

   The values of field `PONumber` are unique — the same numeric value is not used for the field more than once in the collection, so select **Unique** index.

   Select **Index Nulls** also. This is needed for queries that use `ORDER BY` to sort the results. It causes every document to have an entry in the index.

   The values in field `PONumber` are JSON numbers, which means the index can be used for numerical comparison.

[1] MongoDB calls a composite index a compound index. A composite index is also sometimes called a concatenated index.



Example 2-2 Indexing a Singleton Scalar Field Using SODA

Each SODA implementation (programming language or framework) that supports indexing provides a way to create an index. They all use a SODA *index specification* to define the index to be created. For example, with SODA for REST you use an HTTP POST request, passing URI argument `action=index`, and providing the index specification in the POST body.

This is a SODA index specification for a unique index named `poNumIdx` on field `PONumber`:

```
{ "name"  : "poNumIdx",
  "unique" : true,
  "fields" : [ { "path"     : "PONumber",
                 "dataType" : "NUMBER",
                 "order"    : "ASC" } ] }
```

Example 2-3 Indexing a Singleton Scalar Field Using SQL

You can use Database Actions to create an index for field `PONumber` in column `data` of table`orders` with this SQL code. This uses SQL/JSON function `json_value` to extract values of field `PONumber`.

The code uses `ERROR ON ERROR` handling, to raise an error if a document has no `PONumber` field or it has more than one.

Item method `numberOnly()` is used in the path expression that identifies the field to index, to ensure that the field value is numeric.

Method `numberOnly()` is used instead of method `number()`, because `number()` allows also for conversion of non-numeric fields to numbers. For example, `number()` converts a `PONumber` string value of `"42"` to the number `42`.

Other such "only" item methods, which similarly provide strict type checking, include `stringOnly()`, `dateTimeOnly()`, and `binaryOnly()`, for strings, dates, and binary values, respectively.

```
CREATE UNIQUE INDEX "poNumIdx" ON orders
  (json_value(data, '$.PONumber.numberOnly()' ERROR ON ERROR))
```

> **See Also:**
>
> SQL/JSON Path Expression Item Methods in *Oracle Database JSON Developer's Guide*

Example 2-4 Creating a Multivalue Index For Fields Within Elements of an Array

Starting with Oracle Database 21c you can create a multivalue index for the values of fields that can occur multiple times in a document because they are contained in objects within an array (objects as elements or at lower levels within elements).

This example creates a multivalue index on collection `orders` for values of field `UPCCode`. It example uses item method `numberOnly()`, so it applies only to numeric `UPCCode` fields.

```
CREATE MULTIVALUE INDEX mvi_UPCCode ON orders o
      (o.data.LineItems.Part.UPCCode.numberOnly());
```

> **See Also:**
>
> Creating Multivalue Function-Based Indexes for JSON_EXISTS in *Oracle Database JSON Developer's Guide*

Example 2-5 Creating a Materialized View And an Index For Fields Within Elements of an Array

Prior to Oracle Database 21c you cannot create a multivalue index for fields such as `UPCCode`, which can occur multiple times in a document because they are contained in objects within an array (objects as elements or at lower levels within elements).

You can instead, as in this example, create a materialized view that extracts the data you want to index, and then create a function-based index on that view data.

This example creates materialized view `mv_UPCCode` with column `upccode`, which is a projection of field `UPCCode` from within the `Part` object in array `LineItems` of column `data` of table `orders`. It then creates index `mv_UPCCode_idx` on column `upccode` of the materialized view (`mv_UPCCode`).

```
CREATE MATERIALIZED VIEW mv_UPCCode
  BUILD IMMEDIATE
  REFRESH FAST ON STATEMENT WITH PRIMARY KEY
  AS SELECT o.id, jt.upccode
       FROM orders o,
```

```
            json_table(data, '$.LineItems[*]'
                       ERROR ON ERROR NULL ON EMPTY
              COLUMNS (upccode NUMBER PATH '$.Part.UPCCode')) jt;

CREATE INDEX mv_UPCCode_idx ON mv_UPCCode(upccode);
```

The **query optimizer** is responsible for finding the most efficient method for a SQL statement to access requested data. In particular, it determines whether to use an index that applies to the queried data, and which index to use if more than one is relevant. In most cases the best guideline is to rely on the optimizer.

In some cases, however, you might prefer to specify that a particular index be picked up for a given query. You can do this with a MongoDB *hint* that names the index. (Oracle does not support the use of MongoDB index specifications — just provide the index name.)

For example, this query uses index `poNumIdx` on collection `orders`, created in Example 2-1.

```
db.orders.find({"PONumber":1600}).hint("poNumIdx")
```

Alternatively, you can specify an index to use by passing an Oracle SQL hint, using query-by-example (QBE) operator **`$native`**, which is an Oracle extension to the MongoDB hint syntax.

The argument for `$native` has the same syntax as a SQL hint string (that is, the actual hint text, without the enclosing SQL comment syntax `/*+...*/`). You can pass *any SQL hint* using `$native`. In particular, you can turn on *monitoring* for the current SQL statement using hint `MONITOR`. This code does that for a `find()` query:

```
db.orders.find().hint({"$native":"MONITOR"})
```

Related Topics

- MongoDB Aggregation Pipeline Support
  Oracle Database API for MongoDB supports MongoDB aggregation pipelines, that is, MongoDB command `aggregate`. It lets you use pipeline code to execute a query as a sequence of operations. You can also use SQL as a declarative alternative to this procedural approach.

See Also:

- The JSON Page in *Using Oracle Database Actions*
- Overview of SODA Indexing in *Oracle Database Introduction to Simple Oracle Document Access (SODA)*
- Creating Multivalue Function-Based Indexes for JSON_EXISTS in *Oracle Database JSON Developer's Guide*
- Performance Tuning for JSON in *Oracle Database JSON Developer's Guide* for detailed information about improving performance when using JSON data
- JSON Search Index for Ad Hoc Queries and Full-Text Search in *Oracle Database JSON Developer's Guide* for information about JSON search indexes
- Creating a Spatial Index For Scalar GeoJSON Data in *Oracle Database JSON Developer's Guide*
- Influencing the Optimizer with Hints in *Oracle Database SQL Tuning Guide*
- Monitoring Database Operations in *Oracle Database SQL Tuning Guide* for complete information about monitoring database operations
- MONITOR and NO_MONITOR Hints in *Oracle Database SQL Tuning Guide* for information about the syntax and behavior of SQL hints `MONITOR` and `NO_MONITOR`

2.2 Users, Authentication, and Authorization

Oracle Database security differs significantly from that of MongoDB. The security model of Oracle Database API for MongoDB is described: the creation of users, their authentication, and their authorization to perform different operations.

By default, MongoDB does not enable user authentication and authorization checks. Oracle Database always requires authentication, and it always verifies that a connected user is authorized to perform a requested operation. A valid username and password must be provided for authentication.

Oracle Database API for MongoDB supports only the following connection-option values for authentication:

- `PLAIN` value (plain-text authentication) for option `authMechanism`. In particular, the `SCRAM-SHA-*` authentication methods are not supported.
- `$external` value for option `authSource`. (This is anyway required for MongoDB whenever the authentication method is `PLAIN`.)

Oracle Database API for MongoDB relies on Oracle Database users, privileges, and roles. You cannot add or modify these users and roles using MongoDB clients or drivers. You can instead do this using SQL or the Oracle Autonomous Database console. The minimum Oracle Database roles required to use the API are `CONNECT`, `RESOURCE`, and `SODA_APP`.

For MongoDB, a "database" is a set of collections. For Oracle Database API for MongoDB, this corresponds to an Oracle Database **schema**.

> **Note:**
>
> Using Oracle API for MongoDB to drop a "database" does *not* drop the underlying database schema. Instead, it drops all collections within the schema.
>
> An administrative user can drop a schema using SQL (for example, using Database Actions with an Autonomous Oracle Database).

For the API, a username must be a database schema name. The name is case-insensitive, it cannot start with a nonalphabetic character (including a numeral), and it must be provided with a secure password.

Normally, a user of the API can only perform operations within its schema (the username is the schema name). Examples of such operations include creating new collections, reading and writing documents, and creating indexes.

When an administrative user tries to insert data into a database schema (user) that does not exist, that schema is created automatically as a schema-only account, which means that it does not have a password and it cannot be logged into. The new schema is granted these privileges: `SODA_APP`, `CREATE SESSION`, `CREATE TABLE`, `CREATE VIEW`, `CREATE SEQUENCE`, `CREATE PROCEDURE`, and `CREATE JOB`. The schema is also given an unlimited tablespace quota, and is enabled for using Oracle REST Data Services (ORDS).

For an ordinary user of the API, a MongoDB shell command (such as `use` *<database>*) that switches from the current MongoDB database to another one is typically not supported — switching to another database schema raises an error.

However, an **administrative user**, which is one that has all of the following privileges, can create new users (database schemas), and can access any schema as any user: `CREATE USER`, `ALTER USER`, `DROP USER`. User **`admin`** is a predefined administrative user.

An administrative user can do the following:

- Use the schemas of other users.

  Access to other schemas than that of the current user makes use of a proxied connection. For example, someone connected as an administrative user can perform operations in schema `other_user` using the same roles and privileges as if connected directly as `other_user`.
- Create new users (schemas).

  For example, if an administrative user tries to create a collection in a schema `toto` that does not already exist, that schema (user) is automatically created.

Oracle recommends that you *do not allow production applications* to make use of an administrative user. Applications should instead connect as ordinary users, with a minimum of privileges. In particular, connect an application to the database using a MongoClient that is specific to a particular schema (user).

Related Topics

- Terms and Concepts: MongoDB and Oracle Database
  Some application-user terms and concepts used by MongoDB are presented, together with description of their relation to Oracle Database..

- Migrate Application Data from MongoDB to Oracle Database
  Some ways to export your JSON data from MongoDB and then import it into Oracle Database are described. Migration considerations are presented.

Related Topics

- MongoDB Documents and Oracle Database
  Presented here is the relationship between a JSON document used by MongoDB and the same content as a JSON document stored in, and used by, Oracle Database.
- Users, Authentication, and Authorization
  Oracle Database security differs significantly from that of MongoDB. The security model of Oracle Database API for MongoDB is described: the creation of users, their authentication, and their authorization to perform different operations.

> **See Also:**
>
> - Create Users on Autonomous Database in *Using Oracle Autonomous Database Serverless*
> - Manage User Roles and Privileges on Autonomous Database in *Using Oracle Autonomous Database Serverless*
> - CREATE USER in *Oracle Database SQL Language Reference* for information about using SQL to create database schemas (also called database users)
> - GRANT in *Oracle Database SQL Language Reference* for information about using SQL to grant roles to database schemas
> - Using the Oracle Database API for MongoDB in *Using Oracle Autonomous Database Serverless* for information about using an Autonomous Database (including an Autonomous JSON Database) with Oracle Database API for MongoDB. This covers configuring the database for use with the API, including for security and connection.
> - ORDS.ENABLE_SCHEMA in *Oracle REST Data Services Developer's Guide* for information about enabling a database schema for ORDS

2.3 Migrate Application Data from MongoDB to Oracle Database

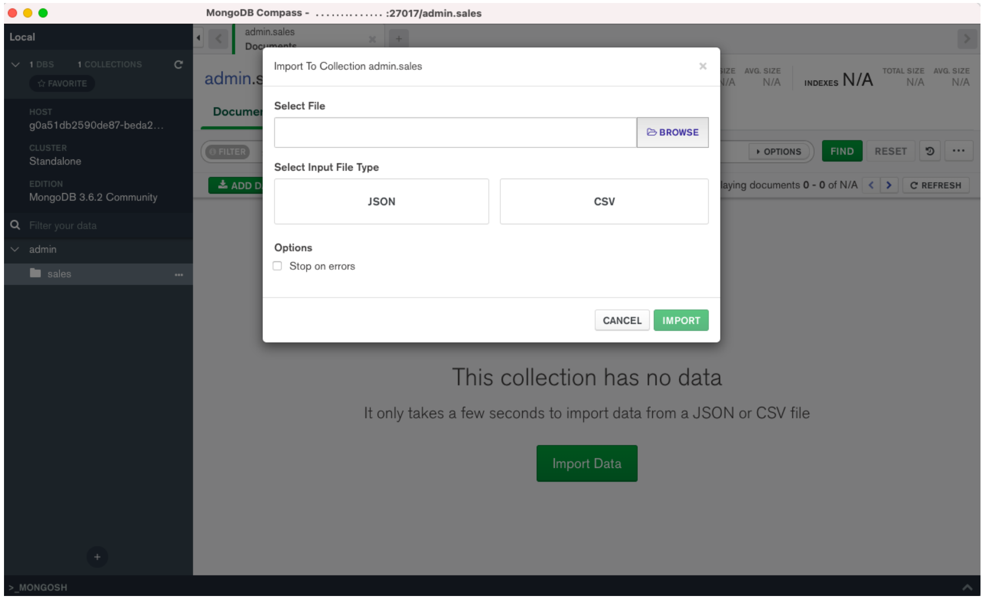
Some ways to export your JSON data from MongoDB and then import it into Oracle Database are described. Migration considerations are presented.

You can migrate your application data in any of these ways:

- Use the MongoDB command-line tools `mongoexport` and `mongoimport`.

  `mongoexport` exports data from a MongoDB instance to your file system, and `mongoimport` imports the exported data from your file system to Oracle Database. Provide your database connection information when using `mongoimport`. Example 2-6 illustrates this.
- Use a MongoDB tool such as Compass to import data into Oracle Database after connecting that tool to the database. Select the name of your JSON collection, then select **ADD DATA**.

This displays a popup dialog box where you browse to and import the JSON file containing your collection data. See MongoDB Compass.



- After exporting JSON data to your file system, import it to the Oracle Cloud Object Store, then load it from there into a collection using PL/SQL procedure `DBMS_CLOUD.copy_collection`. Example 2-7 illustrates this.

  This processes the data in parallel, so it is typically faster than `mongoimport`.

- Write a program that reads JSON documents from a connection to MongoDB and writes them to a connection to Oracle Database.

Example 2-6 Migrate JSON Data to Oracle Database Using mongoexport and mongoimport

This example exports collection `sales` from MongoDB to file-system file `sales.json`. It then imports the data from that file to Oracle Database as collection `sales`. The user is connected to host *<host>* as database schema *<user>* with password *<password>*.

```
mongoexport --collection=sales --out sales.json

mongoimport 'mongodb://<user>:<password>@<host>:27017/<user>?
authMechanism=PLAIN&authSource=$external&ssl=true' --collection=sales --
file=sales.json
```

Note:

Use URI percent-encoding to replace any reserved characters in your connection-string URI — in particular, characters in your username and password. These are the reserved characters and their percent encodings:

!	#	$	%	&	'	(	)	*	+
%21	%23	%24	%25	%26	%27	%28	%29	%2A	%2B

,	/	:	;	=	?	@	[	]
%2C	%2F	%3A	%3B	%3D	%3F	%40	%5B	%5D

For example, if your username is `RUTH` and your password is `@least1/2#?` then your MongoDB connection string to server *<server>* might look like this:

```
'mongodb://RUTH:%40least1%2F2%23%3F@<server>:27017/ruth/ ...'
```

Depending on the tools or drivers you use, you might be able to provide a username and password as separate parameters, instead of as part of a URI connection string. In that case you likely won't need to encode any reserved characters they contain.

See also:

- Percent Encoding - Reserved Characters
- Uniform Resource Identifier (URI): Generic Syntax

See Also:

Using the Oracle Database API for MongoDB in *Using Oracle Autonomous Database Serverless* for information about using an Autonomous Database (including an Autonomous JSON Database) with Oracle Database API for MongoDB. This covers configuring the database for use with the API, including for security and connection.

Example 2-7 Loading JSON Data Into a Collection Using DBMS_CLOUD.COPY_COLLECTION

This example loads data from the Oracle Cloud Object Store into a new collection, `newCollection`, using PL/SQL procedure `DBMS_CLOUD.copy_collection`. It assumes that the data was exported from MongoDB to your file system and then imported from there to the object-store location that's passed as the value of parameter `file_uri_list`.

The value passed as `copy_collection` parameter `FORMAT` is a JSON object with fields `recorddelimiter` and `type`:

- Field `recorddelimiter` specifies that records in the input data are separated by newline characters. A JSON document is created for each record, that is, for each line in the newline-delimited input data.
- Field `type` specifies that the input JSON data can contain EJSON extended objects, and that these should be interpreted.

See DBMS_CLOUD Package Format Options in *Using Oracle Autonomous Database Serverless* for information about parameter `FORMAT`.

```
BEGIN
  DBMS_CLOUD.copy_collection(
    collection_name => 'newCollection',
    file_uri_list   => 'https://objectstorage.../data.json',
    format          => json_object(
                         'recorddelimiter' : '''\n''',
                         'type'            : 'ejson'));
END;
/
```

Related Topics

- Users, Authentication, and Authorization
  Oracle Database security differs significantly from that of MongoDB. The security model of Oracle Database API for MongoDB is described: the creation of users, their authentication, and their authorization to perform different operations.
- Terms and Concepts: MongoDB and Oracle Database
  Some application-user terms and concepts used by MongoDB are presented, together with description of their relation to Oracle Database..

> **See Also:**
>
> - mongoexport and mongoimport
> - Load an Array of JSON Documents into a Collection in *Using Oracle Autonomous JSON Database* for information about using PL/SQL procedure `DBMS_CLOUD.COPY_COLLECTION`

2.4 MongoDB Aggregation Pipeline Support

Oracle Database API for MongoDB supports MongoDB aggregation pipelines, that is, MongoDB command `aggregate`. It lets you use pipeline code to execute a query as a sequence of operations. You can also use SQL as a declarative alternative to this procedural approach.

MongoDB's aggregation pipeline is essentially a weak emulation of SQL capabilities. With MongoDB you express operations such as sorting, grouping, and ordering as separate steps in a pipeline. This approach is *procedural*: you specify *how* to execute a query as a sequence of operations.

SQL on the other hand is *declarative*. You specify the query result you want, and the *optimizer* picks an optimal execution plan based on available indexes, data statistics, cost estimate, and so on. In other words, you specify *what* you want done, and the optimizer, not you, determines *how* it should be done.

Oracle Database SQL support of JSON data includes operating on documents and collections, as well as joining JSON and non-JSON data (relational, spatial, graph, …). As a user of Oracle Database API for MongoDB you can apply SQL directly to JSON data without worrying about manually specifying and sequencing any specific operations.

But if you do use MongoDB aggregation pipeline code then the MongoDB API automatically translates the pipeline stages and operations into equivalent SQL code, and the optimizer picks the best execution plan possible. The API supports a subset of the MongoDB aggregation pipeline stages and operations — see Aggregation Pipeline Operators for details.

Unlike MongoDB, Oracle Database does not limit the size of the data to be sorted, joined, or grouped. You can use it for reporting or analytical work that spans millions of documents across any number of collections.

You can use Oracle Database simplified dot notation for JSON data, or standard SQL/JSON functions `json_value`, `json_query`, and `json_table`, to extract values from your JSON data for reporting or analytic purposes. You can convert relational and other kinds of data (including spatial and graph data) to JSON data using the SQL/JSON generation functions. You can join JSON data from multiple tables and collections with a single SQL `FROM` clause.

A MongoDB aggregation pipeline performs operations on JSON documents from one or more collections. It's composed of successive *stages*, each of which performs document operations and passes the resulting documents to the next stage for further processing. The operations for any stage can *filter* the documents passed from the previous stage, *transform* (update) them, or even *create new documents*, for the next stage. Transformation can involve the use of aggregate operators, also called accumulators, such as `$avg` (average), which can combine field values from multiple documents.

Each stage in a pipeline is represented by an aggregation expression, which is a JSON value. See the MongoDB Aggregation Pipeline documentation for more background.

You can use declarative SQL code to accomplish what you would otherwise use an aggregation pipeline for. This is particularly relevant if your Oracle Database parameter `compatible` is less than `23`, in which case most MongoDB aggregation pipelines are not supported. Example 2-8 illustrates this.

Example 2-8   Using SQL Code Instead of MongoDB Aggregation Pipeline Code

This example calculates average revenues by zip code. It first shows a MongoDB aggregation pipeline expression to do this; then it shows equivalent SQL code.

MongoDB aggregation pipeline:

This code tells MongoDB how to calculate the result; it specifies the order of execution.

```
db.sales.aggregate(
  [{"$group" : {"_id"    : "$address.zip",
                "avgRev" : {"$avg" : "$revenue"}}},
   {"$sort"  : {"avgRev" : -1}}])
```

SQL:

This code specifies the grouping and order of the output presentation *declaratively*. It does not specify *how* the computation is to be carried out, including the order of execution. It simply says that the results are to be grouped by zipcode and presented in descending order of the average revenue figures. The query returns rows of two columns with scalar values for zipcode (a string) and average revenue (a number).

```
SELECT s.data.address.zip.string(),
       avg(s.data.revenue.number())
  FROM sales s
  GROUP BY s.data.address.zip.string()
  ORDER BY 2 DESC;
```

The following query is similar, but it provides the result as *rows of JSON objects*, each with a string field `zip`, for the zipcode, and a numeric field `avgRev`, for the average revenue. SQL/JSON generation function `json_object` constructs JSON objects from the results of evaluating its argument SQL expressions.

```
SELECT json_object('zip'    : s.data.address.zip.string(),
                   'avgRev' : avg(s.data.revenue.number()))
  FROM sales s
  GROUP BY s.data.address.zip.string()
  ORDER BY avg(s.data.revenue.number()) DESC;
```

Related Topics

- Aggregation Pipeline Operators
  Support of MongoDB query and projection operators is described.

2.5 MongoDB Documents and Oracle Database

Presented here is the relationship between a JSON document used by MongoDB and the same content as a JSON document stored in, and used by, Oracle Database.

> **Note:**
>
> This topic applies to JSON documents that you migrate from MongoDB and store in Oracle Database. It does not apply to JSON documents that are generated/supported by JSON-relational duality views. For information about MongoDB-compatible duality views see Mongo DB API Collections Supported by JSON-Relational Duality Views.

You can migrate an existing application and its data from MongoDB to Oracle Database, or you can develop new applications on Oracle Database, which use the same or similar data as applications on MongoDB. JSON data in both cases is stored in **documents**.

It's helpful to have a general understanding of the differences between the documents used by MongoDB and those used by Oracle Database. In particular, it helps to understand what happens to a MongoDB document that you import, to make it usable with Oracle Database.

Some of the information here presents details that you can ignore if you read this topic just to get a high-level view. But it's good to be aware of what's involved; you may want to revisit this at some point.

When you import a collection of MongoDB documents, the *key* and the *content* of each document are converted to forms appropriate for Oracle Database.

A MongoDB document has a native binary JSON format called BSON. An Oracle Database document has a native binary JSON format called OSON. So one change that's made to your MongoDB document is to translate its binary format from BSON to OSON. This translation applies to both the key and the content of a document

Document Key: Differences and Conversion (Oracle Database Prior to 23c)

This section applies only to Oracle Database releases *prior to 23c*.

For MongoDB, the unique key of a document, which identifies it, is the value of mandatory field `_id`, *in the document itself*. For Orace Database releases prior to 23c, the unique key that identifies a document is separate from the document; the key is stored in a separate database column from the column that stores the document. The key column has is named `id`, and it is the *primary key* column for the table that stores your collection data.

When you import a collection into Oracle Database prior to 23c, Oracle Database API for MongoDB creates `id` column values from the values of field `_id` in your MongoDB documents. MongoDB field `_id` can have values of several different data types. The Oracle Database `id` column that corresponds to that field is always of SQL data type `VARCHAR2` (character data; in other words, a string).

The `_id` field in your imported documents is untouched during import or thereafter. Oracle Database doesn't use it — it uses column `id` instead. But it also doesn't change it, so any use your application might make of that field is still valid. Field `_id` in your documents is never changed; even applications cannot change (delete or update) it.

If you need to work with your documents using SQL or Simplified Oracle Document Access (SODA) then you can directly use column `id`. You can easily use that primary-key column to join JSON data with other database data, for instance. The documents that result from importing from MongoDB are SODA documents (with native binary OSON data).

Be aware of these considerations that result from the separation of document key from document:

- Though all documents imported from MongoDB will continue to have their `_id` fields, for Oracle Database prior to 23c the documents in a JSON collection *need not* have an `_id` field. And because, for Oracle Database prior to 23c, a document and its key are separate, a document other than one imported from MongoDB could have an `_id` field that has no relation whatsoever with the document key.
- Because MongoDB allows `_id` values of different types, and these are all converted to string values (`VARCHAR2`), if for some reason your collection has documents with _id values `"123"` (JSON string) and `123` (JSON number) then importing the collection will raise a duplicate-key error, because those values would each be translated as the same string value for column `id`.

BSON values of field `_id` are converted to `VARCHAR2` column `id` values according to Table 2-1. If an `_id` field value is any type not listed in the table then it is replaced by a generated `ObjectId` value, which is then converted to the `id` column value.

Table 2-1 Conversion of BSON Field _id Value To Column ID VARCHAR2 Value

_id Field Type	ID Column VARCHAR2 Value
`Double`	Canonical numeric format string
`32-bit integer`	Canonical numeric format string
`64-bit integer`	Canonical numeric format string
`Decimal128`	Canonical numeric format string
`String`	No conversion, including no character escaping
`ObjectId`	Lowercase hexadecimal string
`Binary data` (UUID)	Lowercase hexadecimal string
`Binary data` (non-UUID)	Uppercase hexadecimal string

The canonical numeric format for a `VARCHAR2` value is as follows:

- If the input number has no fractional part (it is integral), and if it can be rendered in 40 digits or less, then it is rendered as an integer. If necessary, trailing zeros are used, to avoid notation with an exponent. For example, `1000000000` is used instead of `1E+9`.
- If the input number has a fractional part, the number is rendered in 40 digits or less with a decimal point separator. If necessary, zeros are used to avoid notation with an exponent. For example, `0.00001` is used instead of `1E-5`.
- If conversion of the input number would result in a loss of digit precision in the 40-digit format, the number is instead rendered with an exponent. This can happen for a number whose absolute value is extremely small or extremely large, even if the number is integral. For example, `1E100` is used, to avoid a 1 followed by 100 zeros.

In practice, this canonical numeric format means that in most cases the numeric `_id` field value results in an obvious, or "pretty" `VARCHAR2` value for column `id`. A format that uses an exponent is used only when necessary, which generally means infrequently.

Document Content Conversion

Two general considerations:

- BSON format allows duplicate field values in the same object. OSON format does not. When converting to OSON, detection of duplicate fields in BSON data raises an error.
- OSON format has no notion of the order of fields in an object; applications cannot depend on or expect any particular order (in keeping with the JSON standard). BSON format maintains the order of object fields; applications can depend on the order not changing.

Table 2-2 specifies the type mappings that are applied when converting scalar BSON data to scalar OSON data. The OSON scalar types used are SQL data types, except as noted. Any BSON types not listed are not converted; instead, an error is raised when they are encountered. This includes BSON types `regex`, and `JavaScript`.

Table 2-2 JSON Scalar Type Conversions: BSON to OSON Format

BSON Type	OSON Type[1]	Notes
`Double`	`BINARY_DOUBLE`	NA
`32-bit integer`	`NUMBER` (Oracle number)	Flagged as `int`.
`64-bit integer`	`NUMBER` (Oracle number)	Flagged as `long`.
`Decimal128`	`NUMBER` (Oracle number)	Flagged as `decimal`. ***Note:*** *This conversion can be lossy.*
`Date`	`TIMESTAMP WITH TIME ZONE`	Always UTC time zone.
`String`	`VARCHAR2`	Always in character set AL32UTF8 (Unicode UTF-8).
`Boolean`	`BOOLEAN`	Supported only if initialization parameter `compatible` has value `23` or larger. (There is no Oracle SQL `BOOLEAN` type in releases prior to 23c.)
`ObjectId`	`ID` (`RAW(12)`)	NA
`Binary data` (UUID)	`ID` (`RAW(16)`)	NA
`Binary data` (non-UUID)	`RAW`	NA
`Null`	`NULL`	Used for JSON `null`.

1 These are SQL data types, except as noted.

Related Topics

- Other Differences Between MongoDB and Oracle Database
  Various differences between MongoDB and Oracle Database are described. These differences are generally not covered in other topics. Consider these differences when you migrate an application to Oracle Database or you develop a new application for Oracle Database that uses MongoDB commands.
- Users, Authentication, and Authorization
  Oracle Database security differs significantly from that of MongoDB. The security model of Oracle Database API for MongoDB is described: the creation of users, their authentication, and their authorization to perform different operations.

> **See Also:**
>
> Overview of SODA Documents in *Oracle Database Introduction to Simple Oracle Document Access (SODA)*

2.6 Other Differences Between MongoDB and Oracle Database

Various differences between MongoDB and Oracle Database are described. These differences are generally not covered in other topics. Consider these differences when you migrate an application to Oracle Database or you develop a new application for Oracle Database that uses MongoDB commands.

- With MongoDB, fields in a JSON object are ordered. With Oracle Database, they are not ordered. For example, field `_id` is not necessarily the first field in an object. Applications must not expect or rely on any particular field order. According to the JSON language standard, object fields are not ordered; only array elements are ordered. See JSON Syntax and the Data It Represents in *Oracle Database JSON Developer's Guide*.
- With MongoDB, the value of field `_id` can be a JSON object. Oracle Database API for MongoDB supports only BSON types `ObjectId`, `String`, `Double`, `32-bit integer`, `64-bit integer`, `Decimal128`, and `Binary data` (subtype for UUID) for field `_id`; an error is raised for any other type. See BSON Types.

  If you are migrating an existing application that expects object values for `_id` then consider copying the values of field `_id` in your data to some new field and using a string value for `_id`.
- Read and write concerns regarding MongoDB transactions do not apply to Oracle Database. Oracle Database transactions are fully ACID-compliant, and thus reliable — atomicity, consistency, isolation, and durability. ACID compliance ensures that your data remains accurate and consistent despite any failure that might occur while processing a transaction.
- Oracle API for MongoDB does not support the following MongoDB transaction capabilities:
  - Inclusion of DDL operations, such as `createCollection`, within a transaction. Attempts to create a collection or an index within a transaction raise an error.
  - Inclusion of operations across multiple databases. All operations within a transaction must be confined to a single database (schema). Otherwise, an error is raised.
- Retryable writes or commits when an error is raised.

  MongoDB `retryWrite` operations raise an error. If you use a driver that has `retryWrite` turned on by default, then set `retryWrites=false` in your connection string to turn this off.
- Oracle Database and MongoDB have different read isolation and consistency levels. Oracle Database API for MongoDB uses read-committed consistency as described in Data Concurrency and Consistency of *Oracle Database Concepts*.
- Oracle Database API for MongoDB supports only the PLAIN (LDAP SASL) authentication mechanism, and it relies on Oracle Database authentication and authorization.
- Oracle Database does not support the MongoDB `collation` field for any command (such as `find`). An error is raised if you use field `collation`. Oracle collates values using the Unicode binary collation order.
- MongoDB allows different collections in the same "database" to have indexes of the same name. This is not allowed in Oracle Database — the name of an index must be unique across all collections of a given database schema ("database").
- The maximum size of a document for MongoDB is 16 MB. The maximum size for Oracle Database (and thus for the MongoDB API) is 32 MB.

Related Topics

- MongoDB Documents and Oracle Database
  Presented here is the relationship between a JSON document used by MongoDB and the same content as a JSON document stored in, and used by, Oracle Database.
- Users, Authentication, and Authorization
  Oracle Database security differs significantly from that of MongoDB. The security model of Oracle Database API for MongoDB is described: the creation of users, their authentication, and their authorization to perform different operations.

> **See Also:**
>
> Unicode Collation Algorithm, Unicode® Technical Standard #10

2.7 Accessing Collections Owned By Other Users (Database Schemas)

You can directly access a MongoDB API collection owned by another user (database schema) if you log into that schema. You can indirectly access a collection owned by another user, without logging into that schema, if that collection has been mapped to a collection in your schema.

A MongoDB API **collection** of JSON documents consists of (1) a **collection backing table**, which contains the JSON documents in the collection, and (2) some JSON-format **collection metadata**, which is stored in the data dictionary and specifies various collection-configuration properties. The backing table belongs to a given database user/schema. The metadata is stored in the database data dictionary.

A **mapped collection** is a collection that is defined (mapped) on top of an *existing* table, which can belong to any database schema and which could also back one or more other collections.

You can control which operations on a collection — including a mapped collection — are allowed for various users (schemas), by granting those users different privileges or roles on the backing table.

Example 2-9 illustrates this.

Example 2-9 Creating a Collection in One Schema and Mapping a Collection To It in Another Schema

In this example user `john` creates collection `john_coll` (in database schema `john`), and adds a document to it. User `john` then grants user `janet` some access privileges to the backing table of collection `john_coll`.

User `janet` then maps a new collection, `janet_coll` (in schema `janet`) to collection `john_coll` in schema `john`. (The original and mapped collections need not have different names, such as `john_coll` and `janet_coll`; they could both have the same name.)

User `janet` then lists the collections available to schema `janet`, and reads the content of mapped collection `janet_coll`, which is the same as the content of collection `john_coll`.

(The commands submitted to `mongosh` are each a single line (string), but they are shown here continued across multiple lines for clarity.)

1. When connected to the database as user **john**, run PL/SQL code to create collection **john_coll** backed by table john_coll. The second argument to create_collection is the metadata needed for a MongoDB-compatible collection. (The backing table name is derived from the collection name — see Default Naming of a Collection Table.)

```
DECLARE
  col SODA_COLLECTION_T;
BEGIN
  col := DBMS_SODA.create_collection(
           'john_coll',
           '{"contentColumn"      : {"name" : "DATA", "sqlType" : "BLOB",
"jsonFormat" : "OSON"},
             "keyColumn"          : {"name"              : "ID",
                                     "assignmentMethod" : "EMBEDDED_OID",
                                     "sqlType"          : "VARCHAR2"},
             "versionColumn"      : {"name" : "VERSION", "method" : "UUID"},
             "lastModifiedColumn" : {"name" : "LAST_MODIFIED"},
             "creationTimeColumn" : {"name" : "CREATED_ON"}}');
END;
```

2. Connect to the database using shell mongosh as user **john**, list the collections in that schema (John's collections), insert a document into collection john_coll, and show the result of the insertion.

```
mongosh 'mongodb://john:...
@MQSSYOWMQVGAC1Y-CTEST.adb.us-ashburn-1.oraclecloudapps.com:27017/john
?
authMechanism=PLAIN&authSource=$external&ssl=true&retryWrites=false&loa
dBalanced=true'
```

```
john> show collections;
```

Output:

```
john_coll
```

```
john> db.john_coll.insert({"hello" : "world"});
john> db.john_coll.find()
```

Output:

```
[ { _id: ObjectId("6318b0060a51240e4bf3b001"), hello: 'world' } ]
```

3. In schema john, grant user janet access privileges to collection john_coll and its backing table of the same name, john_coll.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON john.john_coll TO janet;
```

4. When connected to the database as user (schema) **janet**, Create a new collection **janet_coll** in schema janet that's *mapped* to collection john_coll in schema john.

The second argument to method create_collection() is the collection metadata. Among the things it specifies here are the schema and backing-table names of the collection to be mapped to. The last argument, **CREATE_MODE_MAP**, specifies that the new collection is to be mapped on top of the table that backs the original collection.

```
DECLARE
  col SODA_COLLECTION_T;
BEGIN
  col := DBMS_SODA.create_collection(
           'janet_coll',
           '{"schemaName"          : "JOHN",
             "tableName"           : "JOHN_COLL",
             "contentColumn"       : {"name" : "DATA", "sqlType" : "BLOB", "jsonFormat" :
"OSON"},
             "keyColumn"           : {"name"             : "ID",
                                      "assignmentMethod" : "EMBEDDED_OID",
                                      "sqlType"          : "VARCHAR2"},
             "versionColumn"       : {"name" : "VERSION", "method" : "UUID"},
             "lastModifiedColumn" : {"name" : "LAST_MODIFIED"},
             "creationTimeColumn" : {"name" : "CREATED_ON"}}',
           DBMS_SODA.CREATE_MODE_MAP);
END;
```

Note:

The schema and table names used in the collection metadata argument must be as they appear in the data dictionary, which in this case means they must be uppercase. You can use these queries to obtain the correct schema and table names for collection *<collection>* (when connected as the owner of *<collection>*):

```
SELECT c.json_descriptor.schemaName FROM USER_SODA_COLLECTIONS c
  WHERE uri_name = '<collection>';

SELECT c.json_descriptor.tableName FROM USER_SODA_COLLECTIONS c
  WHERE uri_name = '<collection>';
```

5. Connect to the database using shell mongosh as user **janet**, list the available collections, and show the content of collection janet_coll (which is the same as the content of John's collection john_coll).

```
mongosh 'mongodb://janet:...
@MQSSYOWMQVGAC1Y-CTEST.adb.us-ashburn-1.oraclecloudapps.com:27017/janet
?authMechanism=PLAIN&authSource=$external&ssl=true&retryWrites=false&loadBalanced=true'
```

```
janet> show collections;

            janet_coll

            janet> db.janet_coll.find()


            [ { _id: ObjectId("6318b0060a51240e4bf3b001"), hello: 'world' } ]
```

3

Support for MongoDB APIs, Operations, and Data Types — Reference

MongoDB APIs, operations, and data types supported by Oracle Database are listed, together with information about their support.

Unsupported MongoDB constructs raise an error. A construct that is *ignored* is listed in this documentation as a **no-op** (it does not raise an error). A construct can be ignored because it makes no sense or is not needed on Oracle architecture.

> **Note:**
>
> Only server commands are covered, not client-side wrapper functions. Client-side wrapper functions such as `deleteMany()` and `updateMany()` use server commands `delete()` and `update()` internally.

- Database Commands
  Support of MongoDB database commands is described. This includes commands for administration, aggregation, authentication, diagnostic, query and write operations, role management, replication, sessions, user management, and sharding.
- Query and Projection Operators
  Support of MongoDB query and projection operators is described. This includes array, bitwise, comment, comparison, element, evaluation, geospatial, and logical query operators, as well as projection operators.
- Update Operators
  Support of MongoDB update operators is described. This includes array, bitwise, field, and modifier update operators.
- Cursor Methods
  Support of MongoDB cursor methods is described.
- Aggregation Pipeline Operators
  Support of MongoDB query and projection operators is described.
- Data Types
  Support of MongoDB data types is described.
- Indexes and Index Properties
  Support of MongoDB indexes and index properties is described.

3.1 Database Commands

Support of MongoDB database commands is described. This includes commands for administration, aggregation, authentication, diagnostic, query and write operations, role management, replication, sessions, user management, and sharding.

See Also:

Database Commands in the MongoDB Reference manual

Table 3-1 Administration Commands

Command	Support (Since)	Notes
Capped Collections	No	None.
`cloneCollectionAsCapped`	No	None.
`collMod`	No	None.
`collMod`, `expireAfterSeconds`	No	None.
`convertToCapped`	No	None.
`create`	19c	Creates a collection in the current Oracle Database schema. If the specified collection already exists then this is a no-op.
`createView`	No	None.
`createIndexes`	19c	None.
`currentOp`	No	None.
`drop`	19c	None.
`dropDatabase`	19c	Deletes all collections in the current Oracle Database schema. Does *not* delete (drop) the schema itself. The command is available only to a user who is logged in with role `root`.
`dropIndexes`	19c	None.
`filemd5`	No	None.
`getParameter`	19c	Parameter supported: `authenticationMechanisms`
`killCursors`	19c	Supported field: `cursors`.
`killOp`	No	None.
`listCollections`	19c	Lists collections in the current Oracle Database schema.
`listDatabases`	19c	Lists Oracle Database schemas enabled for access by Oracle Database API for MongoDB and for Simple Oracle Document Access (SODA).
`listIndexes`	19c	Lists Oracle Database indexes relevant for the specified collection.
`reIndex`	19c	None.
`renameCollection`	No	None.
`setParameter`	No-op	Ignored (no error).
`validate`	19c	None.
`repairDatabase`	No-op	Ignored (no error).

Note:

Besides creating a collection with explicit use of command `create`, a collection is automatically created upon its first insertion of a document. That is, to create a collection it is sufficient to refer to it by name when inserting a document into it.

See Also:

Administration Commands in the MongoDB Reference manual

Table 3-2 Aggregation Commands

Command	Support (Since)	Notes
`aggregate`	19c	None.
`count`	19c	Supported field: `query`.
`distinct`	19c	Supported fields: `key`, `query`. Returns the distinct *scalar* values targeted by the path specified by `key`, as an array. Unlike MongoDB, nonscalar values targeted by the path are not included.
`mapReduce`	No	None.

See Also:

Aggregation Commands in the MongoDB Reference manual

Table 3-3 Authentication Commands

Command	Support (Since)	Notes
`logout`	19c	Logs out the *current user* of an Oracle Database schema on a *specific port*.

See Also:

Authentication Commands in the MongoDB Reference manual

Table 3-4 Diagnostic Commands

Command	Support (Since)	Notes
`buildInfo`	19c	Returns information about current build of Oracle Database API for MongoDB.
`collStats`	19c	None.
`compact`	No-op	Ignored (no error).
`connPoolStats`	No	None.
`connectionStatus`	19c	None.
`dataSize`	23c	Supported fields: `estimate`, `keyPattern`, `min`, `max`.
`dbHash`	No	None.
`dbStats`	19c	Supported field: `scale`. Lists statistics about an Oracle Database *schema*: its collections and relevant indexes.
`explain`	19c	None.
`explain`, **`executionStats`**	19c	None.
`features`	No	None.
`getLog`	No-op	Ignored (no error).
`hostInfo`	19c	None.
`listCommands`	19c	None.
`ping`	19c	None.
`profiler`	No	None.
`serverStatus`	19c	None.
`top`	No	None.
`whatsmyuri`	19c	None.

See Also:

Diagnostic Commands in the MongoDB Reference manual

Table 3-5 Query and Write Operation Commands

Command	Support (Since)	Notes
Change Streams	No	None.
delete	19c	• Supported fields: deletes, ordered. • Supported deletes array operators: q, limit. See Supported query operators for commands delete, find, findAndModify, and update.
find	19c	See Support for command find.
findAndModify	19c	• Supported fields: arrayFilters, fields, new, query, remove, sort, update, upsert. • Supported field update operators: $bit, $currentDate, $inc, $min, $max, $mul, $rename, $set, $setOnInsert, $unset. • Supported array update operators: $, $[], $[*<identifier>*], $addToOffset, $pop, $pull, $pullAll, $push. • Supported array update-operator modifiers supported: $each, $position, $slice, $sort. See Supported query operators for commands delete, find, findAndModify, and update.
getLastError	19c	None.
getMore	19c	Supported fields: batchSize, collection.
getPrevError	No	None.
GridFS	19c	None.
insert	19c	Supported field: documents.
parallelCollectionScan	No	None.
ReplaceOne	No	None.
resetError	19c	None.
update	19c	• Supported fields: ordered, updates. • Supported fields in elements of array updates: arrayFilters, multi, q, u, upsert. Returned response contains fields n, nModified, upserted, and writeErrors. Array upserted contains only the document _id values, no index.

Note:

Support for command `find`.

- Supported operators: see Supported query operators for commands `delete`, `find`, `findAndModify`, and `update`.
- Supported fields: `batchSize`, `filter`, `limit`, `projection`, `returnKey`, `singleBatch`, `skip`, `sort`.

  Field `returnKey` can only return the primary key (e.g. the ObjectID) associated with the documents found. You cannot use it to return only the index key if an index is used to support the query.
- `$` cannot be used in a `projection` specification. Only simple field selections or omissions can be performed.
- The JSON scalar types you can specify with `$type` are as follows:
  - `string` (default)
  - `number`
  - `date` — A date with no time component.
  - `dateTime` — A timestamp: a date with a time component.

Sorting JSON values:

- Oracle Database 23c or later: JSON values are sorted using a canonical sort order — see Comparison and Sorting of JSON Data Type Values.
- Oracle Database 19c: By default, sorting is lexicographical: JSON values are serialized to obtain strings, which are then compared.

  To request a numeric ordering, date ordering, or timestamp ordering, you use a `hint`, providing the relevant JSON scalar type with `$type`.

  For example, the following code requests an ascending lexicographical sort on field `name`, then an ascending numeric sort on field `age`, and then a descending date-time (that is, reverse chronological) sort on field `birthday`. (A positive number, such as `1`, means ascending; a negative number, such as `-1`, means descending.)

  ```
  find().sort({"name":1, "age":1,
  "birthday":-1}).hint({"$type":{"age":"number",
  "birthday":"dateTime"}})
  ```

Note:

Supported query operators for commands `delete`, `find`, `findAndModify`, and `update`.

- Comparison and logical: `$eq`, `$gt`, `$gte`, `$in`, `$lt`, `$lte`, `$ne`, `$nin`, `$and`, `$not`, `$nor`, and `$or`.
- Element and evaluation: `$type`, `$regex`, and `$text`.
- Geospatial: `$geoIntersects`, `$geoWithin`, `$near`, `$nearSphere`.
- Array: `$all`, `$elemMatch`.

See Also:

Query and Write Operation Commands in the MongoDB Reference manual

Table 3-6 Role Management Commands

Command	Support (Since)	Notes
`createRole`	No	None.
`dropRole`	No	None.
`dropAllRolesFromDatabase`	No	None.
`grantRolesToRole`	No	None.
`revokePrivilegesFromRole`	No	None.
`updateRole`	No	None.
`rolesInfo`	No	None.

See Also:

Role Management Commands in the MongoDB Reference manual

Table 3-7 Replication Commands

Command	Support (Since)	Notes
`hello`	19c	None.
`isMaster`	19c	None.
`replSetGetStatus`	No-op	Ignored (no error).

See Also:

Replication Commands in the MongoDB Reference manual

Table 3-8 Sessions Commands

Command	Support (Since)	Notes
`abortTransaction`	19c	None.
`commitTransaction`	19c	None.
`endSessions`	19c	None.
`killAllSessions`	19c	None.
`killAllSessionsByPattern`	19c	None.
`killSessions`	19c	None.
`refreshSessions`	19c	None.
`startSession`	19c	Starts a server-side session. Uses a UUID created by the client, if provided, or a secure random UUID. Returns the UUID used.

See Also:

Sessions Commands in the MongoDB Reference manual

Table 3-9 User Management Commands

Command	Support (Since)	Notes
`createUser`	No	None.
`dropAllUsersFromDatabase`	No	None.
`dropUser`	No	None.
`grantRolesToUser`	No	None.
`revokeRolesFromUser`	No	None.
`updateUser`	No	None.
`userInfo`	No	None.

See Also:

User Management Commands in the MongoDB Reference manual

Table 3-10 Sharding Commands

Command	Support (Since)	Notes
abortReshardCollection	No	None.
addShard	No	None.
addShardZone	No	None.
balancerCollectionStatus	No	None.
balancerStart	No	None.
balancerStatus	No	None.
balancerStop	No	None.
checkShardingIndex	No	None.
clearJumboFlag	No	None.
cleanupOrphaned	No	None.
cleanupReshardCollection	No	None.
commitReshardCollection	No	None.
enableSharding	No	None.
flushRouterConfig	No	None.
getShardMap	No	None.
getShardVersion	No	None.
isdbGrid	No	None.
listShards	No	None.
medianKey	No	None.
moveChunk	No	None.
movePrimary	No	None.
mergeChunks	No	None.
refineCollectionShardKey	No	None.
removeShard	No	None.
removeShardFromZone	No	None.
reshardCollection	No	None.
setAllowMigrations	No	None.
setShardVersion	No	None.
shardCollection	No	None.
shardingState	No	None.
split	No	None.
splitVector	No	None.
unsetSharding	No	None.
updateZoneKeyRange	No	None.

See Also:

Sharding Commands in the MongoDB Reference manual

3.2 Query and Projection Operators

Support of MongoDB query and projection operators is described. This includes array, bitwise, comment, comparison, element, evaluation, geospatial, and logical query operators, as well as projection operators.

See Also:

Query and Projection Operators in the MongoDB Reference manual

Table 3-11 Array Query Operators

Operator	Support (Since)	Notes
`$all`	19c	None.
`$elemMatch`	19c	None.
`$size`	19c	None.

See Also:

Array Query Operators in the MongoDB Reference manual

Table 3-12 Bitwise Query Operators

Operator	Support (Since)	Notes
`$bitsAllSet`	No	None.
`$bitsAnySet`	No	None.
`$bitsAllClear`	No	None.
`$bitsAnyClear`	No	None.

Note:

Bitwise Query Operators in the MongoDB Reference manual

Table 3-13 Comment Query Operator

Operator	Support (Since)	Notes
`$comment`	No	None.

> **See Also:**
>
> $comment in the MongoDB Reference manual

Table 3-14 Comparison Query Operators

Operator	Support (Since)	Notes
`$eq`	19c	None.
`$gt`	19c	None.
`$gte`	19c	None.
`$lt`	19c	None.
`$lte`	19c	None.
`$ne`	19c	None.
`$in`	19c	None.
`$nin`	19c	None.

> **See Also:**
>
> Comparison Query Operators in the MongoDB Reference manual

Table 3-15 Element Query Operators

Operator	Support (Since)	Notes
`$exists`	19c	None.
`$type`	19c	None.

> **See Also:**
>
> Element Query Operators in the MongoDB Reference manual

Table 3-16 Evaluation Query Operators

Operator	Support (Since)	Notes
`$expr`	No	None.
`$jsonSchema`	No	None.
`$mod`	No	None.
`$regex`	19c	None.
`$text`	19c	None.
`$where`	No	None.

> **See Also:**
>
> Evaluation Query Operators in the MongoDB Reference manual

Table 3-17 Geospatial Query Operators

Operator	Support (Since)	Notes
`$box`	No	None.
`$center`	No	None.
`$centerSphere`	No	None.
`$geoIntersects`	19c	None.
`$geometry`	No	None.
`$geoWithin`	19c	None.
`$maxDistance`	No	None.
`$near`	19c	None.
`$nearSphere`	19c	None.
`$polygon`	No	None.
`$uniqueDocs`	No	None.

Table 3-18 Logical Query Operators

Operator	Support (Since)	Notes
`$and`	19c	None.
`$nor`	19c	None.
`$not`	19c	None.
`$or`	19c	None.

See Also:

Logical Query Operators in the MongoDB Reference manual

Table 3-19 Projection Operators

Operator	Support (Since)	Notes
$elemMatch	19c	None.
$meta	No	None.
$slice	No	None.

See Also:

Projection Operators in the MongoDB Reference manual

3.3 Update Operators

Support of MongoDB update operators is described. This includes array, bitwise, field, and modifier update operators.

Table 3-20 Array Update Operators

Operator	Support (Since)	Notes
$	19c	None.
$[]	19c	None.
$[*<identifier>***]**	19c	None.
$addToSet	19c	None.
$pop	19c	None.
$pull	19c	None.
$pullAll	19c	None.
$push	19c	None.

See Also:

Update Array

Table 3-21 Bitwise Update Operator

Operator	Support (Since)	Notes
`$bit`	19c	None.

> **Note:**
>
> Update Bitwise in the MongoDB Reference manual

Table 3-22 Field Update Operators

Operator	Support (Since)	Notes
`$currentDate`	19c	None.
`$inc`	19c	None.
`$max`	19c	None.
`$min`	19c	None.
`$mul`	19c	None.
`$rename`	19c	None.
`$set`	19c	None.
`$setOnInsert`	19c	None.
`$unset`	19c	None.

> **See Also:**
>
> Update Field in the MongoDB Reference manual

Table 3-23 Modifier Update Operators

Operator	Support (Since)	Notes
`$each`	19c	None.
`$position`	19c	None.
`$slice`	19c	None.
`$sort`	19c	None.

> **See Also:**
>
> Update Operators in the MongoDB Reference manual

3.4 Cursor Methods

Support of MongoDB cursor methods is described.

Table 3-24 Cursor Methods

Method	Support (Since)	Notes
$cursor.batchSize()	19c	None.
$cursor.close()	19c	None.
$cursor.collation()	No	None.
$cursor.comment()	19c	None.
$cursor.count()	19c	None.
$cursor.explain()	19c	None.
$cursor.forEach()	19c	None.
$cursor.hasNext()	19c	None.
$cursor.hint()	19c	None.
$cursor.isClosed()	19c	None.
$cursor.isExhausted()	19c	None.
$cursor.itcount()	19c	None.
$cursor.limit()	19c	None.
$cursor.map()	19c	None.
$cursor.max()	19c	None.
$cursor.maxScan()	No	None.
$cursor.maxTimeMS()	19c	None.
$cursor.min()	19c	None.
$cursor.next()	19c	None.
$cursor.noCursorTimeout()	19c	None.
$cursor.objsLeftInBatch()	19c	None.
$cursor.pretty()	19c	None.
$cursor.readConcern()	19c	None.
$cursor.readPref()	19c	None.
$cursor.returnKey()	19c	None.
$cursor.showRecordId()	19c	None.
$cursor.size()	19c	None.
$cursor.skip()	19c	None.
$cursor.sort()	19c	None.
$cursor.tailable()	19c	None.
$cursor.toArray()	19c	None.

See Also:

Cursor Methods in the MongoDB Reference manual

3.5 Aggregation Pipeline Operators

Support of MongoDB query and projection operators is described.

See Also:

Aggregation Pipeline Operators in the MongoDB Reference manual

Table 3-25 Arithmetic Expression Operators

Operator	Support (Since)	Notes
`$abs`	23c	None.
`$add`	23c	None.
`$ceil`	23c	None.
`$divide`	23c	None.
`$exp`	23c	None.
`$floor`	23c	None.
`$ln`	23c	None.
`$log`	No	None.
`$log10`	No	None.
`$mod`	23c	None.
`$multiply`	23c	None.
`$pow`	23c	None.
`$round`	23c	None.
`$sqrt`	23c	None.
`$subtract`	23c	None.
`$trunc`	23c	None.

See Also:

Arithmetic Expression Operators in the MongoDB Reference manual

Table 3-26 Array Expression Operators

Operator	Support (Since)	Notes
`$arrayElemAt`	23c	None.

Table 3-26 (Cont.) Array Expression Operators

Operator	Support (Since)	Notes
`$arrayToObject`	23c	None.
`$concatArrays`	23c	None.
`$filter`	23c	None.
`$first`	23c	None.
`$firstN`	23c	None.
`$in`	23c	None.
`$indexOfArray`	No	None.
`$isArray`	23c	None.
`$last`	23c	None.
`$lastN`	23c	None.
`$objectToArray`	23c	None.
`$range`	No	None.
`$reduce`	23c	None.
`$reverseArray`	23c	None.
`$size`	23c	None.
`$slice`	23c	None.
`$sortArray`	23c	None.
`$zip`	23c	None.

> **See Also:**
>
> Array Expression Operators in the MongoDB Reference manual

Table 3-27 Boolean Expression Operators

Operator	Support (Since)	Notes
`$and`	23c	None.
`$not`	23c	None.
`$or`	23c	None.

> **See Also:**
>
> Boolean Expression Operators in the MongoDB Reference manual

Table 3-28 Comparison Expression Operators

Operator	Support (Since)	Notes
`$cmp`	23c	None.
`$eq`	23c	None.
`$gt`	23c	None.
`$gte`	23c	None.
`$lt`	23c	None.
`$lte`	23c	None.
`$ne`	23c	None.

> **See Also:**
>
> Comparison Expression Operators in the MongoDB Reference manual

Table 3-29 Conditional Expression Operators

Operator	Support (Since)	Notes
`$cond`	23c	None.
`$ifNull`	23c	None.
`$switch`	No	None.

> **See Also:**
>
> Conditional Expression Operators in the MongoDB Reference manual

Table 3-30 Date Expression Operators

Operator	Support (Since)	Notes
`$dateAdd`	No	None.
`$dateDiff`	No	None.
`$dateFromParts`	No	None.
`$dateFromString`	23c	None.
`$dateSubtract`	No	None.
`$dateToParts`	No	None.
`$dateToString`	23c	None.
`$dateTrunc`	No	None.
`$dayOfMonth`	No	None.
`$dayOfWeek`	No	None.
`$dayOfYear`	No	None.

Table 3-30 (Cont.) Date Expression Operators

Operator	Support (Since)	Notes
`$hour`	No	None.
`$isoDayOfWeek`	No	None.
`$isoWeek`	No	None.
`$isoWeekYear`	No	None.
`$millisecond`	No	None.
`$minute`	No	None.
`$month`	No	None.
`$second`	No	None.
`$week`	No	None.
`$year`	No	None.

See Also:

Date Expression Operators in the MongoDB Reference manual

Table 3-31 Literal Expression Operator ($literal)

Operator	Support (Since)	Notes
`$literal`	23c	None.

See Also:

Literal Expression Operator in the MongoDB Reference manual

Table 3-32 Object Expression Operators

Operator	Support (Since)	Notes
`$mergeObjects`	23c	None.
`$objectToArray`	No	None.
`$setField`	No	None.

See Also:

Object Expression Operators in the MongoDB Reference manual

Table 3-33 Set Expression Operators

Operator	Support (Since)	Notes
`$anyElementFalse`	No	None.
`$anyElementTrue`	No	None.
`$setDifference`	No	None.
`$setEquals`	No	None.
`$setIntersection`	No	None.
`$setIsSubset`	No	None.
`$setUnion`	23c	None.

> **See Also:**
>
> Set Expression Operators in the MongoDB Reference manual

Table 3-34 String Expression Operators

Operator	Support (Since)	Notes
`$concat`	23c	None.
`$indexOfBytes`	No	None.
`$indexOfCP`	No	None.
`$ltrim`	23c	None.
`$regexFind`	No	None.
`$regexFindAll`	No	None.
`$regexMatch`	No	None.
`$replaceAll`	No	None.
`$replaceOne`	No	None.
`$rtrim`	23c	None.
`$split`	No	None.
`$strcasecmp`	23c	None.
`$strLenBytes`	No	None.
`$strLenCP`	No	None.
`$substr`	No	None.
`$substrBytes`	No	None.
`$substrCP`	No	None.
`$toLower`	23c	None.
`$toUpper`	23c	None.
`$trim`	19c	None.

See Also:

String Expression Operators in the MongoDB Reference manual

Table 3-35 Text Expression Operator ($meta)

Operator	Support (Since)	Notes
`$meta`	No	None.

See Also:

Text Expression Operator in the MongoDB Reference manual

Table 3-36 Type Expression Operators

Operator	Support (Since)	Notes
`$convert`	No	None.
`$isNumber`	23c	None.
`$toBool`	No	None.
`$toDate`	No	None.
`$toDecimal`	No	None.
`$toDouble`	No	None.
`$toInt`	No	None.
`$toLong`	No	None.
`$toObjectId`	No	None.
`$toString`	23c	None.
`$type`	19c	None.

See Also:

Type Expression Operators in the MongoDB Reference manual

Table 3-37 Stage Operators

Operator	Support (Since)	Notes
`$addFields`	23c	None.
`$bucket`	No	None.
`$bucketAuto`	No	None.

Table 3-37 (Cont.) Stage Operators

Operator	Support (Since)	Notes
`$collStats`	19c	Lists statistics about the specified collection and the Oracle Database indexes relevant for it. Supported fields: `scale`.
`$count`	23c	None.
`$currentOp`	No	None.
`$facet`	No	None.
`$geoNear`	No	None.
`$graphLookup`	No	None.
`$group`	23c	None.
`$indexStats`	No	None.
`$limit`	23c	None.
`$listLocalSessions`	No	None.
`$listSessions`	No	None.
`$lookup`	No	None.
`$match`	23c	None.
`$merge`	No	None.
`$out`	23c	None.
`$planCacheStats`	No	None.
`$project`	23c	None.
`$redact`	No	None.
`$replaceRoot`	23c	None.
`$sample`	No	None.
`$setWindowFields`	No	None.
`$skip`	23c	None.
`$sort`	23c	None.
`$sortByCount`	No	None.
`$sql`	No	None.
`$unionWith`	No	None.
`$unset`	19c	None.
`$unwind`	23c	None.

See Also:

Aggregation Pipeline Stages in the MongoDB Reference manual

Table 3-38 Accumulator Expression Operators

Operator	Support (Since)	Notes
`$accumulator`	No	None.
`$addToSet`	No	None.
`$avg`	No	None.
`$bottom`	No	None.
`$bottomN`	No	None.
`$count`	23c	None.
`$first`	No	None.
`$firstN`	No	None.
`$last`	No	None.
`$lastN`	No	None.
`$max`	No	None.
`$maxN`	No	None.
`$min`	No	None.
`$push`	No	None.
`$stdDevPop`	No	None.
`$sum`	No	None.
`$top`	No	None.
`$topN`	No	None.

See Also:

Accumulators ($group) and Accumulators ($project)in the MongoDB Reference manual

Table 3-39 Variable Expression Operator

Operator	Support (Since)	Notes
`$let`	23c	None.

See Also:

Variable Expression Operators in the MongoDB Reference manual

Table 3-40 System Variables

Variable	Support (Since)	Notes
`$$CURRENT`	23c	None.
`$$DESCEND`	No	None.

Table 3-40 (Cont.) System Variables

Variable	Support (Since)	Notes
`$$KEEP`	No	None.
`$$PRUNE`	No	None.
`$$REMOVE`	No	None.
`$$ROOT`	23c	None.

See Also:

Variables in Aggregation Expressions in the MongoDB Reference manual

Table 3-41 Miscellaneous Operators

Operator	Support (Since)	Notes
`$getField`	No	None.
`$rand`	23c	None.
`$sampleRate`	No	None.
`$map`	No	None.

Hint $service: Application-Connection Service (Consumer Group)

You can use any of the following application-connection services (consumer groups) with any aggregation pipeline expression, by adding a **`$service`** hint to the expression. Service `LOW` is used by default. `LOW`, `MEDIUM`, and `HIGH` are typically used for reporting and batch processing; `TP` and `TPURGENT` are typically used for transaction processing.

- `LOW` — Low-priority service for reporting and batch processing. Operations are *not* run in parallel.
- `MEDIUM` — Medium-priority service for reporting and batch operations. All operations run in *parallel* and are subject to *queuing*.
- `HIGH` — High-priority service for reporting and batch operations. All operations run in *parallel* and are subject to *queuing*.
- `TP` — Typical service for transaction processing. Operations are *not* run in parallel.
- `TPURGENT` — Highest-priority service, for *time-critical* transaction processing. Supports *manual parallelism*.

For example, the hint here specifies that operator `$count` should use service `HIGH`.

```
db.foo.aggregate([ {"$count":"cnt"} ], {"hint":{"$service":"HIGH"}});
```

Related Topics

- MongoDB Aggregation Pipeline Support
  Oracle Database API for MongoDB supports MongoDB aggregation pipelines, that is, MongoDB command `aggregate`. It lets you use pipeline code to execute a

query as a sequence of operations. You can also use SQL as a declarative alternative to this procedural approach.

3.6 Data Types

Support of MongoDB data types is described.

Table 3-42 Data Types

Data Type and Alias	Support (Since)	Notes
32-Bit Integer (`int`)	19c	None.
64-Bit Integer (`long`)	19c	None.
Array (`array`)	19c	None.
Binary Data (`binData`)	19c	None.
Boolean (`bool`)	19c	None.
Date (`date`)	19c	None.
DBPointer (`dbPointer`)	No	None.
Decimal128 (`decimal`)	19c	None.
Double (`double`)	19c	None.
JavaScript (`javascript`)	No	None.
MaxKey (`maxKey`)	No	None.
MinKey (`minKey`)	No	None.
Null (`null`)	19c	None.
Object (`object`)	19c	None.
ObjectId (`objectId`)	19c	None.
Regular Expression (`regex`)	No	None.
String (`string`)	19c	None.
Symbol (`symbol`)	No	None.
Timestamp (`timestamp`)	No	None.
Undefined (`undefined`)	No	None.

See Also:

$type in the MongoDB Reference manual

3.7 Indexes and Index Properties

Support of MongoDB indexes and index properties is described.

Table 3-43 Indexes

Index Type	Support (Since)	Notes
2d Index	No (23c). No-op (19c)	None.

Table 3-43 (Cont.) Indexes

Index Type	Support (Since)	Notes
2dsphere Index	No (23c). No-op (19c)	You can create an Oracle Database spatial index using SQL `CREATE INDEX` on the backing table of the collection.
Compound Multikey Index	No (23c). No-op (19c)	See Note, below.
Hashed Index	No (23c). No-op (19c)	None.
Single Field Multikey Index	23c. No-op (19c)	See Note, below.
Text Index	19c	None.

> **Note:**
>
> You can create a suitable Oracle Database index using SQL `CREATE INDEX` on the backing table of the collection. See Indexes for JSON Data.
>
> If the field cannot ever have an array value then create a `json_value` function-based index. Otherwise, use an index over a materialized view. See JSON Query Rewrite To Use a Materialized View Over JSON_TABLE.

> **See Also:**
>
> Index Types in the MongoDB Reference manual

Table 3-44 Index Properties

Index Property	Support (Since)	Notes
Background	No (23c); No-op (19c)	None.
Case Insensitive	No (23c); No-op (19c)	None.
Partial	No (23c); No-op (19c)	None.
Sparse	No (23c); No-op (19c)	None.
TTL	No (23c); No-op (19c)	When creating the equivalent of a MongoDB compound or single field index using SQL, the index can have property TTL.
Unique	23c (No-op in 19c)	When creating the equivalent of a MongoDB compound or single field index using SQL, the index can be unique.

See Also:

Index Properties in the MongoDB Reference manual

Index

U

W