Oracle® Database
Introduction to Simple Oracle Document Access (SODA)
Contents

Preface

Audience viii
Documentation Accessibility viii
Diversity and Inclusion viii
Related Documents ix
Conventions ix

1 Overview of SODA

1.1 Overview of SODA Documents 1-4
1.2 Overview of SODA Document Collections 1-5
1.3 Default Naming of a Collection Table 1-10
1.4 A View of Your SODA Collections 1-11
1.5 Monitoring SODA Operation Performance 1-11

2 Overview of SODA Filter Specifications (QBEs)

2.1 Sample JSON Documents 2-4
2.2 Overview of Paths in SODA QBEs 2-5
2.3 Overview of QBE Comparison Operators 2-7
2.4 Overview of QBE Operator $not 2-9
2.5 Overview of QBE Item-Method Operators 2-9
2.6 Overview of QBE Logical Combining Operators 2-11
2.7 Overview of Nested Conditions in QBEs 2-13
2.8 Overview of QBE Operator $id 2-14
2.9 Overview of QBE Operator $orderby 2-15
2.10 Overview of QBE Spatial Operators 2-16
2.11 Overview of QBE Operator $contains 2-18
2.12 Overview of QBE Operator $textContains 2-19

3 Overview of SODA Indexing
### 5. SODA Filter Specifications (Reference)

**5.1. Composite Filters (Reference)**
- 5.1.1 Orderby Clause Sorts Selected Objects 5-3

**5.2. Filter Conditions (Reference)**
- 5.2.1 Scalar-Equality Clause (Reference) 5-8
- 5.2.2 Field-Condition Clause (Reference)
  - 5.2.2.1 Comparison Clause (Reference) 5-9
  - 5.2.2.2 Not Clause (Reference) 5-13
  - 5.2.2.3 Item-Method Clause (Reference) 5-14
  - 5.2.2.4 ISO 8601 Date, Time, and Duration Support 5-20
- 5.2.3 Logical Combining Clause (Reference)
  - 5.2.3.1 Omitting $and 5-22
- 5.2.4 Nested-Condition Clause (Reference) 5-24
- 5.2.5 ID Clause (Reference) 5-25
- 5.2.6 Text-Contains Clause (Reference) 5-26
- 5.2.7 Special-Criterion Clause (Reference)
  - 5.2.7.1 Spatial Clause (Reference) 5-28
  - 5.2.7.2 Contains Clause (Reference) 5-30

### 6. SODA Index Specifications (Reference)

### 7. SODA Collection Metadata Components (Reference)

**7.1. Default Collection Metadata** 7-3
**7.2. Schema** 7-5
**7.3. Table or View** 7-5
**7.4. Key Column Name** 7-6
**7.5. Key Column Type** 7-6
**7.6. Key Column Max Length** 7-7
**7.7. Key Column Assignment Method** 7-7
**7.8. Key Column Path** 7-8
**7.9. Key Column Sequence Name** 7-9
**7.10. Content Column Name** 7-10
**7.11. Content Column Type** 7-10
**7.12. Content Column Format** 7-11
**7.13. Content Column Max Length** 7-12
**7.14. Content Column JSON Validation** 7-12
### List of Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Default Collection Metadata</td>
<td>1-8</td>
</tr>
<tr>
<td>1-2</td>
<td>Selecting Collection Data From USER_SODA_COLLECTIONS</td>
<td>1-11</td>
</tr>
<tr>
<td>2-1</td>
<td>Sample JSON Document 1</td>
<td>2-5</td>
</tr>
<tr>
<td>2-2</td>
<td>Sample JSON Document 2</td>
<td>2-5</td>
</tr>
<tr>
<td>2-3</td>
<td>Sample JSON Document 3</td>
<td>2-5</td>
</tr>
<tr>
<td>2-4</td>
<td>Using $id To Find Documents That Have Given Keys</td>
<td>2-14</td>
</tr>
<tr>
<td>3-1</td>
<td>Specifying a B-Tree Index</td>
<td>3-2</td>
</tr>
<tr>
<td>3-2</td>
<td>Specifying a Spatial Index</td>
<td>3-2</td>
</tr>
<tr>
<td>3-3</td>
<td>Specifying a JSON Search Index</td>
<td>3-4</td>
</tr>
<tr>
<td>5-1</td>
<td>Filter Specification with Explicit $and Operator</td>
<td>5-23</td>
</tr>
<tr>
<td>5-2</td>
<td>Filter Specification with Implicit $and Operator</td>
<td>5-23</td>
</tr>
<tr>
<td>5-3</td>
<td>Use of Operator $id in the Outermost QBE Condition</td>
<td>5-26</td>
</tr>
<tr>
<td>5-4</td>
<td>QBE With a Spatial Clause</td>
<td>5-29</td>
</tr>
<tr>
<td>8-1</td>
<td>Workaround To Use BLOB Content With Oracle Database 21c Or Later</td>
<td>8-2</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1</td>
<td>Query-By-Example (QBE) Comparison Operators</td>
<td>5-10</td>
</tr>
<tr>
<td>5-2</td>
<td>Item-Method Operators</td>
<td>5-16</td>
</tr>
<tr>
<td>7-1</td>
<td>Key Assignment Methods</td>
<td>7-7</td>
</tr>
<tr>
<td>7-2</td>
<td>Version Generation Methods</td>
<td>7-16</td>
</tr>
<tr>
<td>8-1</td>
<td>SODA Driver Minimum Required Versions</td>
<td>8-1</td>
</tr>
<tr>
<td>9-1</td>
<td>Minimal Client and Server Versions for SODA Features</td>
<td>9-1</td>
</tr>
</tbody>
</table>
Preface

This document provides a conceptual overview of Simple Oracle Document Access (SODA).

- Audience
- Documentation Accessibility
- Diversity and Inclusion
- Related Documents
- Conventions

Audience

This document is intended for users of Simple Oracle Document Access (SODA).

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.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers’ existing technologies and the need to ensure continuity of service as Oracle’s offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.
Related Documents

For more information, see these Oracle resources:

- Simple Oracle Document Access (SODA) at Oracle Help Center for complete information about SODA and each of its implementations
- Oracle Database JSON Developer’s Guide
- Oracle as a Document Store for general information about using JSON data in Oracle Database, including with SODA

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at OTN Registration.

If you already have a user name and password for OTN then you can go directly to the documentation section of the OTN Web site at OTN Documentation.

Conventions

The following text conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td><code>monospace</code></td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
Overview of SODA

Simple Oracle Document Access (SODA) is a set of NoSQL-style APIs that let you create and store collections of documents (in particular JSON) in Oracle Database, retrieve them, and query them, without needing to know Structured Query Language (SQL) or how the documents are stored in the database.

There are separate SODA implementations for use with different languages and with the representational state transfer (REST) architectural style. SODA for REST can itself be accessed from almost any programming language. It maps SODA operations to Uniform Resource Locator (URL) patterns.

**Note:**

This book describes the features that are present in different SODA implementations. Some features described here may not be available for some implementations. In addition, different implementations can have different ways of providing some of the features. Please refer to the documentation for a particular implementation for detailed information about it.

SODA APIs are document-centric. You can use any SODA implementation to perform create, read, update, and delete (CRUD) operations on documents of nearly any kind (including video, image, sound, and other binary content). You can also use any SODA implementation to query the content of JavaScript Object Notation (JSON) documents using pattern-matching: query-by-example (QBE). CRUD operations can be driven by document keys or by QBEs.

Oracle Database supports storing and querying JSON data natively. SODA document collections are backed by ordinary database tables and views. Because of this, you can take advantage of database features for use with the content of SODA documents. SODA CRUD and query operations are automatically mapped to SQL operations on the underlying database tables or views, and these operations are optimized.

You do not need knowledge of SQL, or database administrator (DBA) assistance, to develop or deploy a SODA application. However, database role SODA_APP must be granted to the database schema (user account) that you use to store collections.

The SQL standard defines a set of SQL/JSON operators that allow direct querying of JSON data. Database views based on these operators provide schema-on-read behavior that is immune to changes in the structure of your documents. If needed, developers with SQL knowledge can use SQL/JSON to perform advanced operations on your SODA data that make full use of the database. For example, a SQL developer can apply database analytics and reporting to your JSON data, and can include it in aggregation and join operations that involve other data. In addition, your SODA applications can use database transactions.

These SODA abstractions hide the complexities of SQL and client programming:

- Collection
A document collection contains documents. Collections are persisted in an Oracle Database schema (also known as a database user). In some SODA implementations a database schema is referred to as a SODA database.

A SODA collection is analogous to an Oracle Database table or view.

SODA is designed primarily for working with JSON documents, but a document can be of any Multipurpose Internet Mail Extensions (MIME) type.

In addition to its content, a document has other document components, including a unique identifier, called its key, a version, a media type (type of content), and the date and time that it was created and last modified. The key is typically assigned by SODA when a document is created, but client-assigned keys can also be used. Besides the content and key (if client-assigned), you can set the media type of a document. The other components are generated and maintained by SODA. All components other than content and key are optional.

A SODA document is analogous to, and is in fact backed by, a row of a database table or view. The row has one column for each document component: key, content, version, and so on.

In addition to the documents it contains, a collection also has associated collection metadata. This specifies various details about the collection, such as its storage, whether it should track version and time-stamp document components, how such components are generated, and whether the collection can contain only JSON documents.

In some contexts collection metadata is represented as a JSON document. This metadata document is sometimes called a collection specification. You can supply a custom collection specification when you create a collection, to provide metadata that differs from that provided by default.

SODA provides CRUD operations on documents. JSON documents can additionally be queried, using query-by-example (QBE) patterns, also known as filter specifications. A filter specification is itself a JSON object.

SODA APIs provide operations for collection management (create, drop, list) and document management (CRUD).

These are some of the actions you can perform using SODA:

- Create collections
- Open existing collections
- Drop collections
- List all existing collections
- Create documents
- Insert documents into a collection
- Save documents into a collection (insert new or update existing)
- Truncate a collection (empty it, deleting all of its documents)
- Find a document in a collection, by key or by key and version
- Find all documents in a collection
- Find documents in a collection, by keys or by QBE
• Find documents in a collection whose content matches a text search
• Replace (update) a document in a collection, by key or by key and version (optimistic locking)
• Remove a document from a collection, by key or by key and version (optimistic locking)
• Remove documents from a collection, by keys or by QBE
• Index the documents in a collection (to improve query performance)
• Create a JSON data guide for a collection, which summarizes document structural and type information
• Create a relational view from a JSON data guide

Your applications use a database transaction when performing one or more such actions.¹

• Overview of SODA Documents
SODA is designed primarily to manipulate JavaScript Object Notation (JSON) documents, that is, documents whose content is JSON data, but other kinds of documents can also be used. A document has other components, besides its content.

• Overview of SODA Document Collections
A SODA collection is a set of documents that is backed by an Oracle Database table or view.

• 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.

• A View of Your SODA Collections
Oracle Database static data dictionary view USER_SODA_COLLECTIONS lists the basic features of all of your SODA collections, that is, all SODA collections created by the database user (database schema) that you are currently connected to the database as.

• Monitoring SODA Operation Performance
You can turn on performance monitoring of the SQL operations that underlie a SODA read or write operation, by adding a SQL hint to the SODA operation.

See Also:

• Simple Oracle Document Access (SODA) at Oracle Help Center for complete information about SODA and each of its implementations
• Oracle Database JSON Developer’s Guide for information about using SQL and PL/SQL with JSON data.
• Introducing JSON for information about JSON.
• Oracle as a Document Store for general information about using JSON data in Oracle Database, including with SODA

¹ SODA for REST is an exception in this regard; you cannot use database transactions for its actions.
1.1 Overview of SODA Documents

SODA is designed primarily to manipulate JavaScript Object Notation (JSON) documents, that is, documents whose content is JSON data, but other kinds of documents can also be used. A document has other components, besides its content.

Here is a textual representation of the content of a simple JSON document:

```json
{ "name"    : "Alexander",
  "address" : "1234 Main Street",
  "city"    : "Anytown",
  "state"   : "CA",
  "zip"     : "12345" }
```

You can set the following document components (with an application client, for example):

- Key
- Content
- Media type

In a collection, each document must have a document key, which is unique for the collection. By default, collections are configured to automatically generate document keys for inserted documents. If you want to instead use your own, custom, keys for a collection then you must provide the key for a document when you create it.

The media type specifies the type of content for a document. For JSON documents the media type is "application/json".

The following document components are set and maintained automatically by SODA itself:

- Version
- Creation time stamp
- Last-modified time stamp

A SODA document is an abstract object that encapsulates its components, including its content — it is a carrier of content. A SODA create-document operation creates such a programmatic document object, and a document object is returned by some SODA operations, such as find operations.

A document is stored in Oracle Database as a row in a table or view, with each component in its own column.

In a client application, a SODA document is represented in a way that is appropriate to the particular SODA implementation used. For example:

- In SODA for Java, a document is represented as a Java interface.
- In SODA for PL/SQL, a document is represented as a PL/SQL object type.

---

2 Because REST is not a programming language, SODA for REST has no programmatic "object" that represents a document. But SODA for REST operations involve the same concept of a document. For example, when you read a document you obtain a JSON representation of it, which includes all of the components (key, content, version, and so on).
In SODA for C, a document is represented as an Oracle Call Interface (OCI) handle. In all cases, there are methods or functions to create documents and access their components.

To write content to SODA collections and read content from them, you use create-document, write, and read operations:

- You use a SODA create-document operation to create a document object with content that you provide. (The content can be JSON data or something else.)
- You use a SODA write operation (such as insert), to store the document persistently in Oracle Database. (The document content is written to a database table.)
- You use a SODA read operation (such as find), to fetch the document back from Oracle Database. You use specific getter operations to read specific document components (such as key and content).

Related Topics
- **SODA Feature Support**
  Features that are supported in different SODA implementations (languages) are listed, along with their required client versions and database releases.

### See Also:
- Introducing JSON for general information about JSON

---

### 1.2 Overview of SODA Document Collections

A SODA collection is a set of documents that is backed by an Oracle Database table or view. 

*By default*, creating a SODA document collection creates the following in Oracle Database:

- Persistent default collection metadata.
- A table for storing the collection, in the database schema to which your SODA client is connected.

All SODA implementations provide a get-metadata operation, which returns the metadata for a collection, represented in JSON. Example 1-1 shows an example of the default collection metadata, which is returned for a default collection.

The default metadata specifies a collection that tracks five components for each document: key, content, version, last-modified time stamp, and created-on time stamp. These are specified in JSON by fields `keyColumn`, `contentcolumn`, `versionColumn`, `lastModifiedColumn`, and `creationTimeColumn`, respectively. Each of these components is stored in a separate column in the table or view that backs the collection in Oracle Database. The metadata further specifies various details about these components and the database columns that back them.

In Example 1-1, for the key component: the column name is "ID", the column type is "VARCHAR2", the maximum key length is 255, and the key generation method used is "UUID".

In a client application, a document collection is represented in a way that is appropriate to the particular SODA implementation used. For example:
• In SODA for Java, a collection is represented as a Java interface.
• In SODA for PL/SQL, a collection is represented as a PL/SQL object type.
• In SODA for C, a collection is represented as an Oracle Call Interface (OCI) handle.

When a collection is created, the create-collection operation returns a Java or PL/SQL object or an OCI handle, which you can use to perform various collection read and write operations.3

**Note:**

In the SODA for REST URI syntax, after the version component, you can use `custom-actions`, `metadata-catalog`, or a particular collection name. When you use `custom-actions` or `metadata-catalog`, the next segment in the URI, if there is one, is a collection name.

Because of this syntax flexibility, you cannot have a collection named either `custom-actions` or `metadata-catalog`. An error is raised if you try to create a collection with either of those names using SODA for REST.

In other SODA implementations, besides SODA for REST, nothing prevents you from creating and using a collection named `custom-actions` or `metadata-catalog`. But for possible interoperability, best practice calls for not using these names for collections.

When you create a collection you can specify things such as the following:

• Storage details, such as the name of the table that stores the collection and the names and data types of its columns.
• The presence or absence of columns for creation time stamp, last-modified time stamp, and version.
• Whether the collection can store only JSON documents (the media type for the column).
• Methods of document-key generation, and whether document keys are client-assigned or generated automatically.
• Methods of version generation.

This configurability also lets you map a new collection to an existing database table or view.

To configure a collection in a nondefault way, you must define custom collection metadata and pass it to the create-collection operation. This metadata is represented as JSON data.

---

3 This is the case only for language-based SODA implementations. In SODA for REST a collection is essentially represented by just a URL.
You can customize collection metadata to obtain different behavior from that provided by default. Just what you can change depends on the database you are using. If you are using an Oracle Autonomous Database — Autonomous JSON Database (AJD), Autonomous Transaction Processing (ATP), or Autonomous Data Warehouse (ADW) — then the only metadata you can customize are the document-key generation method and the content media type. In particular, you cannot change the SQL data type of the column used to store JSON content (the `content column`).

In addition, changing some components requires familiarity with Oracle Database concepts, such as SQL data types. Oracle recommends that you do not change such components unless you have a compelling reason. Because SODA collections are implemented on top of Oracle Database tables (or views), many collection configuration components are related to the underlying table configuration.

Reasons you might want to use custom metadata include:

- To configure SecureFiles LOB storage.
- To configure a collection to store documents other than JSON (a heterogeneous collection).
- To map an existing Oracle Database table or view to a new collection.
- To specify that a collection mapping to an existing table is read-only.
- To use a `VARCHAR2` column for JSON content, and to increase the default maximum length of data allowed in the column.

You might want to increase the maximum allowed data length if your database is configured with extended data types, which extends the maximum length of these data types to 32767 bytes. For more information about extended data types, see Oracle Database SQL Language Reference.

You can use online redefinition to change the metadata of an existing collection. An important use case for this arises when you upgrade your database so that initialization parameter `compatible` is at least 20. Then the default collection metadata specifies `JSON` type for the content storage and UUID for the method used to generate version values.

To take advantage of `JSON` type for the content storage of an existing collection, use online redefinition to change metadata field `contentColumn.sqlType` to "JSON". To use UUID as the version column generation method, use online redefinition to change metadata field `versionColumn.method` to "UUID".

See Redefining a SODA Collection in Oracle Database SODA for PL/SQL Developer's Guide for how to do this.
You can perform read and write operations on a collection only if it is open. *Opening* a collection amounts to obtaining an object (in Java and PL/SQL) or a handle (in C) that represents the collection. *Creating* a collection opens it automatically: the create-collection operation returns a collection object or handle. There is also an open operation, to open an existing collection. It too returns a collection object or handle. If you try to create a collection, and a collection with the same name already exists, then that existing collection is simply opened.

**Note:**

By default, the table name for a collection is derived from the collection name, but it can also be explicitly provided in the custom collection metadata that you pass to the create-collection operation. If this table name (derived or explicitly provided) matches an existing table in the currently connected database schema (user), the create-collection operation tries to use that existing table to back the collection.\(^4\)

You must therefore ensure that the existing table matches the collection metadata. For example, if the collection metadata specifies that the collection has three columns, for key, content, and version, then the underlying table must have these same columns, and the column types must match those specified in the collection metadata. The create-collection operation performs minimal validation of the existing table, to check that it matches collection metadata. If this check determines that the table and metadata do not match then the create-collection operation raises an error.

**Caution:**

Do not use SQL to drop the database table that underlies a collection. Dropping a collection involves more than just dropping its database table. In addition to the documents that are stored in its table, a collection has metadata, which is also persisted in Oracle Database. Dropping the table underlying a collection does not also drop the collection metadata.

### Example 1-1  Default Collection Metadata

This example shows the default metadata used for an Oracle Database that is *not* an Oracle Autonomous Database (Autonomous Transaction Processing or Autonomous Data Warehouse). The default metadata for an autonomous database is described in Default Collection Metadata.

If database initialization parameter `compatible` is at least 20, then SODA uses JSON data type by default for JSON content, and the version method is UUID. Here is the default metadata for this case:

```json
{
    "schemaName" : "mySchemaName",
    "tableName" : "myTableName",
```

\(^4\) SODA for REST is an exception here. For security reasons, in this context an error is raised for SODA for REST, to disallow access to existing tables using REST operations.
If initialization parameter compatible is less than 20, then SODA uses "BLOB" textual data by default for JSON content — the data is character data. Here is the default metadata for this case:

```json
{
    "schemaName" : "mySchemaName",
    "tableName" : "myTableName",
    "keyColumn" : {
        "name" : "ID",
        "sqlType" : "VARCHAR2",
        "maxLength" : 255,
        "assignmentMethod" : "UUID"
    },
    "contentColumn" : {
        "name" : "JSON_DOCUMENT",
        "sqlType" : "BLOB",
        "compress" : "NONE",
        "cache" : true,
        "encrypt" : "NONE",
        "validation" : "STANDARD"
    },
    "versionColumn" : {
        "name" : "VERSION",
    },
    "lastModifiedColumn" : {
        "name" : "LAST_MODIFIED"
    },
    "creationTimeColumn" : {
        "name" : "CREATED_ON"
    },
    "readOnly" : false
}
```
"method" : "SHA256"
},
"lastModifiedColumn" : {
  "name" : "LAST_MODIFIED"
},
"creationTimeColumn" : {
  "name" : "CREATED_ON"
},
"readOnly" : false
}

Related Topics

- SODA Collection Metadata Components (Reference)
  Collection metadata is composed of multiple components. A detailed definition of
  the components is presented.

- 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.

1.3 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 (").  

Related Topics  
• Table or View  
The collection metadata component that specifies the name of the table or view to which the collection is mapped.

1.4 A View of Your SODA Collections

Oracle Database static data dictionary view USER_SODA_COLLECTIONS lists the basic features of all of your SODA collections, that is, all SODA collections created by the database user (database schema) that you are currently connected to the database as.

The view includes, for each collection, its metadata and its underlying database information, in particular, the collection name and the name and database schema of the table or view that backs the collection.

Similarly, for database administrators there is view DBA_SODA_COLLECTIONS, which lists collection information for all users (database schemas). It has the additional column OWNER, which specifies the collection owner. You need the SELECT privilege on this view to select data from it. This privilege is granted to role SELECT_CATALOG_ROLE.

View DBA_SODA_COLLECTIONS

You can use Structured Query Language (SQL) to select data for one or more of your collections.

Note:  
Oracle Database Reference for complete information about data dictionary views USER_SODA_COLLECTIONS and DBA_SODA_COLLECTIONS. View DBA_SODA_COLLECTIONS is available only for Autonomous Oracle Database 19c.

Example 1-2 Selecting Collection Data From USER_SODA_COLLECTIONS

This example selects all columns from the row of view USER_SODA_COLLECTIONS that corresponds to collection name myCol.

```sql
SELECT * FROM USER_SODA_COLLECTIONS
WHERE URI_NAME = 'myCol';
```

1.5 Monitoring SODA Operation Performance

You can turn on performance monitoring of the SQL operations that underlie a SODA read or write operation, by adding a SQL hint to the SODA operation.
The hint you provide to the SODA operation uses the Oracle SQL hint syntax (that is, the actual hint text, without the enclosing SQL comment syntax /*+...*/).

Refer to the documentation for a particular SODA implementation (language) for how to provide a hint for a given operation.

Use only hint MONITOR (turn on monitoring) or NO_MONITOR (turn off monitoring). You can use this to pass any SQL hints, but MONITOR and NO_MONITOR are the useful ones for SODA, and an inappropriate hint can cause the optimizer to produce a suboptimal query plan.

**See Also:**

- 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
Overview of SODA Filter Specifications (QBEs)

A filter specification is a pattern expressed in JSON. You use it to select, from a collection, the JSON documents whose content matches it, meaning that the condition expressed by the pattern evaluates to true for the content of (only) those documents.

A filter specification is also called a query-by-example (QBE), or simply a filter.

Because a QBE selects documents from a collection, you can use it to drive read and write operations on those documents. For example, you can use a QBE to remove all matching documents from a collection.

Each SODA implementation that supports query-by-example provides its own way to query JSON documents. They all use a SODA filter specification to define the data to be queried. For example, with SODA for REST you use an HTTP POST request, passing URI argument action=query, and providing the filter specification in the POST body.

```
{ "age" : { "$gt" : 45 } }
```

There are different kinds of QBE operators. In particular, there are operators that do the following kinds of things. (This is not an exhaustive list of QBE operators.)

- Test whether a field value exists or how it compares with particular values.
This includes the comparison operators: $all, $between, $eq, $exists, $gt, $gte, $hasSubstring, $in, $instr, $like, $lt, $lte, $ne, $nin, $regex, and $startsWith.

For example, this QBE uses operator $gt to test whether the value of a field age is greater than 50.

{ "age" : { "$gt" : 50 } }

- Test spatial (geographic or geometric) properties of a GeoJSON field value.
  
  This includes operators $near, $intersects, and $within.
  
  For example, this QBE uses operator $near to test whether the value of field location is within 60 miles of the given $geometry value (elided here).

  { "location" : { "$near" : { "$geometry" : {...}, "$distance" : 60 } } }

- *Full-text search:* test whether a field value pattern-matches a given string or number.
  
  This uses QBE operator $contains.
  
  For example, this QBE tests whether the value of a field name contains the word "beth".

  { "name" : { "$contains" : "beth" } }

- Combine conditions logically.
  
  This includes operators $not, $and, $or, and $nor.
  
  For example, this QBE matches either (or both) a field age whose value is *not* greater than 50 or a field salary whose value is 10000.

  { "$or" : [ { "$not" : { "age" : { "$gt" : 50 } } }, { "salary" : { "$eq" : 10000 } } ] }

- Sort the objects selected by a QBE query.
  
  This uses QBE operator $orderby in conjunction with operator $query, which provides the QBE that selects the objects to sort.
  
  For example, this QBE first selects all objects in which the value of field salary is greater than 10,000. It then uses operator $orderby to sort those objects by ascending values of field name. The values to sort are interpreted as strings (data type VARCHAR2).

  { "$query" : { "salary" : { "$gt" : 10000 } }, "$orderby" : [ { "path" : "name", "datatype" : "varchar2" } ] }

- *Act on a matched value to produce a value that's tested in its place.*
  
  This includes the *item-method* operators: $abs, $boolean, $ceiling, $date, $double, $floor, $length, $lower, $number, $size, $string, $timestamp, $type, and $upper.
For example, in this QBE, item-method $date interprets the value of field birthday as a date value, which is then tested for being greater than (that is, later than) the date represented by ISO 8601 string "2000-01-01". If it is, then the field value is considered a match. The greater-than test uses the interpreted value that results from $date acting on the field value.

```
{ "birthday" : { "$date" : { "$gt":"2000-01-01" } } }
```

- **Sample JSON Documents**
  A few sample JSON documents are presented here. They are referenced in some query-by-example (QBE) examples, as well as in some reference descriptions.

- **Overview of Paths in SODA QBEs**
  A filter specification, or query-by-example (QBE), contains zero or more paths to fields in JSON documents. A path to a field can have multiple steps, and it can cross the boundaries of objects and arrays.

- **Overview of QBE Comparison Operators**
  A query-by-example (QBE) comparison operator tests whether a given JSON object field satisfies some conditions.

- **Overview of QBE Operator $not**
  Query-by-example (QBE) operator $not negates the behavior of its operand, which is a JSON object containing one or more comparison clauses, which are implicitly ANDed.

- **Overview of QBE Item-Method Operators**
  A query-by-example (QBE) item-method operator acts on a JSON-object field value to modify or transform it in some way (or simply to filter it from the query result set). Other QBE operators that would otherwise act on the field value then act on the transformed field value instead.

- **Overview of QBE Logical Combining Operators**
  You use the query-by-example (QBE) logical combining operators, $and, $or, and $nor, to combine conditions to form more complex QBEs. Each accepts an array of conditions as its argument.

- **Overview of Nested Conditions in QBEs**
  You can use a query-by-example (QBE) with a nested condition to match a document that has a field with an array value with object elements, where a given object in the array satisfies multiple conditions.

- **Overview of QBE Operator $id**
  Other query-by-example (QBE) operators generally look for particular JSON fields within documents and try to match their values. Operator $id is an exception in that it instead matches document keys. It thus matches document metadata, not document content. You use operator $id in the outermost condition of a QBE.

- **Overview of QBE Operator $orderby**
  Query-by-example (QBE) operator $orderby is described. It sorts query results in ascending or descending order.

- **Overview of QBE Spatial Operators**
  You can use query-by-example (QBE) operator $near, $intersects, or $within to select documents that have a field whose value is a GeoJSON geometry object that is near a specified position, intersects a specified geometric object, or is within another specified geometric object, respectively.
• **Overview of QBE Operator $contains**
  Query-by-example (QBE) operator $contains performs **full-text search** of JSON documents in a SODA collection.

• **Overview of QBE Operator $textContains**
  Query-by-example (QBE) operator $textContains performs **full-text search** of documents in a heterogeneous SODA collection, that is, a collection that has the media type column.

**Related Topics**

• **SODA Paths (Reference)**
  SODA filter specifications contain paths, each of which targets a value in a JavaScript Object Notation (JSON) document. A path is composed of a series of steps. A detailed definition of SODA paths is presented.

• **SODA Filter Specifications (Reference)**
  You can select JSON documents in a collection by pattern-matching. A detailed definition of SODA filter specifications (QBEs) is presented.

• **Overview of QBE Comparison Operators**
  A query-by-example (QBE) comparison operator tests whether a given JSON object field satisfies some conditions.

• **Overview of QBE Logical Combining Operators**
  You use the query-by-example (QBE) logical combining operators, $and, $or, and $nor, to combine conditions to form more complex QBEs. Each accepts an array of conditions as its argument.

• **Overview of QBE Operator $not**
  Query-by-example (QBE) operator $not negates the behavior of its operand, which is a JSON object containing one or more comparison clauses, which are implicitly ANDed.

• **Overview of QBE Item-Method Operators**
  A query-by-example (QBE) item-method operator acts on a JSON-object field value to modify or transform it in some way (or simply to filter it from the query result set). Other QBE operators that would otherwise act on the field value then act on the transformed field value instead.

• **Media Type Column Name**
  The collection metadata component that specifies the name of the column that stores the media type of the document. A media type column is needed if the collection is to be heterogeneous, that is, it can store documents other than JavaScript Object Notation (JSON).

---

**See Also:**

• Introducing JSON for information about JSON
• GeoJSON.org for information about GeoJSON geographic JSON data

---

### 2.1 Sample JSON Documents

A few sample JSON documents are presented here. They are referenced in some query-by-example (QBE) examples, as well as in some reference descriptions.
Example 2-1    Sample JSON Document 1

{ "name"    : "Jason",
"age"     : 45,
"address" : [ { "street" : "25 A street",
"city"   : "Mono Vista",
"zip"    : 94088,
"state"  : "CA" } ],
"drinks"  : "tea" }

Example 2-2    Sample JSON Document 2

{ "name"    : "Mary",
"age"     : 50,
"address" : [ { "street" : "15 C street",
"city"   : "Mono Vista",
"zip"    : 97090,
"state"  : "OR" },
{ "street" : "30 ABC avenue",
"city"   : "Markstown",
"zip"    : 90001,
"state"  : "CA" } ] }

Example 2-3    Sample JSON Document 3

{ "name"   : "Mark",
"age"    : 65,
"drinks" : ["soda", "tea"] }

Related Topics
• Field-Condition Clause (Reference)
  A field-condition clause specifies that a given object field must satisfy a given set of criteria. It constrains a field using one or more condition-operator clauses, each of which is a comparison clause, a not clause, or an item-method clause.

2.2 Overview of Paths in SODA QBEs

A filter specification, or query-by-example (QBE), contains zero or more paths to fields in JSON documents. A path to a field can have multiple steps, and it can cross the boundaries of objects and arrays.

(In the context of a QBE, the term "path to a field" is sometimes shortened informally to "field".)

For example, this QBE matches all JSON documents where a zip field exists under field address and has value 94088:

{ "address.zip" : 94088 }

The path in the preceding QBE is address.zip, which matches Example 2-1.
A SODA QBE is itself a JSON object. You must use strict JSON syntax in a QBE. In particular, you must enclose all field names in double quotation marks (" "). This includes field names, such as address.zip, that act as SODA paths. For example, you must write \{"address.zip" : 94088\}, not \{address.zip : 94088\}.

Paths can target a particular element of an array in a JSON document, by enclosing the array position of the element in square brackets ([ ]).

For example, path address[1].zip targets all zip fields in the second object of array addresses. (Array position numbers start at 0, not 1.) The following QBE matches Example 2-2 because the second object of its address array has a zip field with value 90001.

\{ "address[1].zip" : 90001 \}

Instead of specifying a single array position, you can specify a list of positions (for example, [1,2]) or a range of positions (for example, [1 to 3]). The following QBE matches Example 2-3 because it has "soda" as the first element (position 0) of array drinks.

\{ "drinks[0,1]" : "soda" \}

And this QBE does not match any of the sample documents because they do not have "soda" as the second or third array element (position 1 or 2).

\{ "drinks[1 to 2]" : "soda" \}

If you do not specify an array step then [*] is assumed, which matches any array element — the asterisk, *, acts as a wildcard. For example, if the value of field drinks is an array then the following QBE matches a document if the value of any array element is the string "tea":

\{"drinks" : "tea"\}

This QBE thus matches sample documents 1 and 2. An equivalent QBE that uses the wildcard explicitly is the following:

\{"drinks[*]" : "tea"\}

Related Topics

- SODA Paths (Reference)
  SODA filter specifications contain paths, each of which targets a value in a JavaScript Object Notation (JSON) document. A path is composed of a series of steps. A detailed definition of SODA paths is presented.
2.3 Overview of QBE Comparison Operators

A query-by-example (QBE) comparison operator tests whether a given JSON object field satisfies some conditions.

One of the simplest and most useful filter specifications tests a field for equality to a specific value. For example, this filter specification matches any document that has a field name whose value is "Jason". It uses the QBE operator $eq, which tests field-value equality.

```
{ "name" : { "$eq" : "Jason" } }
```

For convenience, for such a scalar-equality QBE you can generally omit operator $eq. This scalar-equality filter specification is thus equivalent to the preceding one, which uses $eq:

```
{ "name" : "Jason" }
```

Both of the preceding filter specifications match Example 2-1.

The comparison operators are the following:

- `$all` — whether an array field value contains all of a set of values
- `$between` — whether a field value is between two string or number values (inclusive)
- `$eq` — whether a field value is equal to a given scalar
- `$exists` — whether a given field exists
- `$gt` — whether a field value is greater than a given scalar value
- `$gte` — whether a field value is greater than or equal to a given scalar value
- `$hasSubstring` — whether a string field value has a given substring (same as $instr)
- `$in` — whether a field value is a member of a given set of scalar values
- `$instr` — whether a string field value has a given substring (same as $hasSubstring)
- `$like` — whether a field value matches a given SQL LIKE pattern
- `$lt` — whether a field value is less than a given scalar value
- `$lte` — whether a field value is less than or equal to a given scalar value
- `$ne` — whether a field value is different from a given scalar value
- `$nin` — whether a field value is not a member of a given set of scalar values
• $regex — whether a string field value matches a given regular expression
• $startsWith — whether a string field value starts with a given substring

You can combine multiple comparison operators in the object that is the value of a single QBE field. The operators are implicitly ANDed. For example, the following QBE uses comparison operators $gt and $lt. It matches Example 2-2, because that document contains an age field with a value (50), which is both greater than 45 and less than 55.

{ "age" : { "$gt" : 45, "$lt" : 55 } }

Note:
Both the operand of a SODA operator and the data matched in your documents by a QBE are JSON data. But a comparison operator can in some cases interpret such JSON values specially before comparing them. The use of item-method operators can specify that a comparison should first interpret JSON string data as, for example, uppercase or as a date or a time stamp (date with time). This is explained in the sections about item-method operators.

Related Topics
• Sample JSON Documents
A few sample JSON documents are presented here. They are referenced in some query-by-example (QBE) examples, as well as in some reference descriptions.
• Overview of QBE Operator $not
Query-by-example (QBE) operator $not negates the behavior of its operand, which is a JSON object containing one or more comparison clauses, which are implicitly ANDed.
• Field-Condition Clause (Reference)
A field-condition clause specifies that a given object field must satisfy a given set of criteria. It constrains a field using one or more condition-operator clauses, each of which is a comparison clause, a not clause, or an item-method clause.
• Comparison Clause (Reference)
A comparison clause is an object member whose field is a comparison operator. Example: "$gt" : 200.
• Overview of QBE Item-Method Operators
A query-by-example (QBE) item-method operator acts on a JSON-object field value to modify or transform it in some way (or simply to filter it from the query result set). Other QBE operators that would otherwise act on the field value then act on the transformed field value instead.
2.4 Overview of QBE Operator $not

Query-by-example (QBE) operator $not negates the behavior of its operand, which is a JSON object containing one or more comparison clauses, which are implicitly ANDed.

When any of those clauses evaluates to false, the application of $not to them evaluates to true. When all of them evaluate to true, it evaluates to false.

For example, this QBE matches Example 2-1 and Example 2-3: document 1 has a field matching path address.zip and whose value is not "90001", and document 3 has no field matching path address.zip.

```
{"address.zip" : { "$not" : { "$eq" : "90001"}}}
```

The $not operand in the following QBE has two comparison clauses. It too matches Example 2-1 and Example 2-3, because each of them has an age field whose value is not both greater than 46 and less than 65.

```
{"age" : { "$not" : { "$gt" : 46, "$lt" : 65}}}
```

Related Topics

- Logical Combining Clause (Reference)
  A logical combining clause combines the effects of multiple non-empty filter conditions.

- Sample JSON Documents
  A few sample JSON documents are presented here. They are referenced in some query-by-example (QBE) examples, as well as in some reference descriptions.

2.5 Overview of QBE Item-Method Operators

A query-by-example (QBE) item-method operator acts on a JSON-object field value to modify or transform it in some way (or simply to filter it from the query result set). Other QBE operators that would otherwise act on the field value then act on the transformed field value instead.

Suppose you want to select documents whose string-valued field name starts with “Jo”, irrespective of letter case, so that you find matches for name values "Joe", "joe", "JOE", "joE", "Joey", "joseph", "jose" , and so on. You might think of using operator $startsWith, but that matches string prefixes literally, considering J and j as different characters, for example.

This is where an item-method operator can come in. Your QBE can use item-method operator $upper to, in effect, transform the raw field data, whether it is "Joey" or "jose", to an uppercase string, before operator $startsWith is applied to test it.

The following QBE matches the prefix of the value of field name, but only after converting it to uppercase. The uppercase value is matched using the condition that it starts with JO.

```
{ "name" : { "$upperz" : { "$startsWith" : "JO" } }}
```
As another example, suppose that you have documents with a string-valued field
`deadline` that uses an ISO 8601 date-with-time format supported by SODA, and you
want to select those documents whose deadline is prior to 7:00 am, January 31, 2019,
UTC. You can use item-method operator `$timestamp` to convert the field string values
to UTC `time` values (not strings), and then perform a time comparison using an
operator such as `$lt`. This QBE does the job:

```json
{ "deadline" : {
  "$timestamp" : { "$lt" : "2019-01-31T07:00:00Z" } }
}
```

That matches each of the following `deadline` field values, because each of them
represents a time prior to the one specified in the QBE. (The last two represent the
exact same time, since 7 pm in a time zone that is 3 hours behind UTC is the same as
10 pm UTC.)

- { "deadline" : "2019-01-28T14:59:43Z" }
- { "deadline" : "2019-01-30T22:00:00Z" }
- { "deadline" : "2019-01-30T19:00:00–03:00" }

Not all item-method operators convert data to a given data type. Some perform other
kinds of conversion. Operator `$upper`, for instance, converts a string value to
uppercase — the result is still a string.

Some item-method operators even return data that is wholly different from the field
values they are applied to. Operator `$type`, for instance, returns a string value that
names the JSON-language data type of the field value.

So for example, this QBE selects only Example 2-3 of the three sample documents,
because it is the only one that has a `drinks` field whose value is an array ("soda",
"tea"). In particular, it does not match Example 2-1, even though that document has
a field `drinks`, because the value of that field is the string "tea" — a scalar, not an
array.

```json
{ "drinks" : { "$type" : "array" } }
```
An item-method field (operator) does not, itself, use or act on its associated value (its operand). Instead, it acts on the value of the JSON data that matches its parent field.

For example, in the QBE \{"birthday" : \{"$date" : \{"$gt" : "2000-01-01"\}\}\}, item-method operator $date acts on the JSON data that matches its parent field, birthday. It does not use or act on its operand, which is the JSON object (a comparison clause in this case) \{"$gt" : "2000-01-01"\}. The birthday data (a JSON string of format ISO 8601) in your JSON document is interpreted as a date, and that date is then matched against the condition that it be greater than the date represented by the (ISO date) string "2000-01-01" (later than January 1, 2000).

This can take some getting used to. The operand is used after the item-method operator does its job. It is matched against the result of the action of the operator on the value of its parent field. A item-method operator is a filter of sorts — it stands syntactically between the field (to its left) that matches the data it acts on and (to its right) some tests that are applied to the result of that action.

Note:
- To use item method operator $abs, $date, $size, $timestamp, or $type you need Oracle Database Release 18c or later.
- To use any other item method you need Oracle Database Release 12c (12.2.0.1) or later.

Related Topics
- **Item-Method Clause (Reference)**
  An item-method clause is an item-method equality clause or an item-method modifier clause. It applies an item method to the field of the field-condition clause in which it appears, typically to modify the field value. It then matches the result against the operand of the item-method.
- **ISO 8601 Date, Time, and Duration Support**
  International Standards Organization (ISO) standard 8601 describes an internationally accepted way to represent dates, times, and durations. Oracle Database supports the most common ISO 8601 formats as proper Oracle SQL date, time, and interval (duration) values. The formats that are supported are essentially those that are numeric-only, language-neutral, and unambiguous.

2.6 Overview of QBE Logical Combining Operators

You use the query-by-example (QBE) logical combining operators, $and, $or, and $nor, to combine conditions to form more complex QBEs. Each accepts an array of conditions as its argument.

QBE logical combining operator $and matches a document if each condition in its array argument matches it. For example, this QBE matches Example 2-1, because that document
contains a field name whose value starts with "Ja", and it contains a field drinks whose value is "tea".

{"$and" : [ {"name" : {"$startsWith" : "Ja"}}, {"drinks" : "tea"} ]}

Often you can omit operator $and — it is implicit. For example, the following query is equivalent to the previous one:

{"name" : {"$startsWith" : "Ja"}, "drinks" : "tea"}

QBE logical combining operator $or matches a document if at least one of the conditions in its array argument matches it. For example, the following QBE matches Example 2-2 and Example 2-3, because those documents contain a field drinks whose value is "soda" or they contain a field zip under a field address, where the value of address.zip is less than 94000, or they contain both:

{"$or" : [ {"drinks" : "soda"}, {"address.zip" : {"$le" : 94000}} ]}

QBE logical combining operator $nor matches a document if no condition in its array argument matches it. (Operators $nor and $or are logical complements.) The following query matches sample document 1, because in that document there is neither a field drinks whose value is "soda" nor a field zip under a field address, where the value of address.zip is less than 94000:

{"$nor" : [ {"drinks" : "soda"}, {"address.zip" : {"$le" : 94000}} ]}

Each element in the array argument of a logical combining operator is a condition. For example, the following condition has two conditions in the array argument of operator $or. The first of these has a single logical combining clause with $and, and it restricts the value of field age. The second condition has a single logical combining clause with $or, and it restricts either the value of field name or the value of field drinks.

{ "$and" : [ { "age" : {"$gte" : 60} }, { "$or" : [ {"name" : "Jason"}, {"drinks" : {"$in" : ["tea", "soda"]}} ] } ] }

• The condition with the comparison for field age matches sample document 3.
• The condition with logical combining operator $or matches sample documents 1 and 3.
• The overall condition matches only sample document 3, because that is the only document that satisfies both the condition on age and the condition that uses $or.

The following condition has two conditions in the array argument of operator $or. The first of these has a single logical combining clause with $and, and it restricts the values...
of fields name and drinks. The second has a single logical combining clause with $nor, and it restricts the values of fields age and name.

```json
{ "$or" : [ { "$and" : [ {"name" : "Jason"}, 
   {"drinks" : {"$in" : ["tea", "soda"]}} ] }, 
   { "$nor" : [ {"age" : {$lt : 65}}, 
   {"name" : "Jason"} ] } ] }
```

- The condition with operator $and matches sample document 1.
- The condition with operator $nor matches sample document 3.
- The overall condition matches both sample documents 1 and 3, because each of these documents satisfies at least one condition in the $or argument.

Related Topics

- Logical Combining Clause (Reference)
  A logical combining clause combines the effects of multiple non-empty filter conditions.

- Omitting $and
  Sometimes you can omit the use of $and.

- Sample JSON Documents
  A few sample JSON documents are presented here. They are referenced in some query-by-example (QBE) examples, as well as in some reference descriptions.

### 2.7 Overview of Nested Conditions in QBEs

You can use a query-by-example (QBE) with a nested condition to match a document that has a field with an array value with object elements, where a given object in the array satisfies multiple conditions.

The following nested-condition query matches documents that have both a city value of "Mono Vista" and a state value of "CA" in the same object element of array address.

```json
{ "address[*]" : { "city" : "Mono Vista", "state" : "CA" } }
```

It specifies that a matching document must have a field address, and if the value of that field is an array then it must have at least one object element that has a city field with value "Mono Vista" and a state field with value "CA".

Of the three sample JSON documents, this QBE matches only Example 2-1.

The following QBE also matches sample document 1, but it also matches Example 2-2, which has two addresses, one of which has city Mono Vista and the other of which has state CA.

```json
{ "address.city" : "Mono Vista", "address.state" : "CA" }
```

Unlike the preceding QBE, which uses a nested condition clause, nothing here constrains the city and state to belong to the same address. Instead, this QBE specifies only that matching documents must have a city field with value "Mono Vista" in some object child of an address field, and a state field with value "CA" in some object of the same address field. It does not specify that fields address.city and address.state must reside within the same object.
That last QBE is equivalent to the following one, which has the form of a nested-condition clause but without the [*].

{ "address" : { "city" : "Mono Vista", "state" : "CA" } }

*Do not forget to use [*], if your intention is to apply multiple conditions to an object in an array.*

**Related Topics**

- [Nested-Condition Clause (Reference)]
  You use a QBE nested-condition clause to apply *multiple conditions* at the same time, to array elements that are objects.

- [Sample JSON Documents]
  A few sample JSON documents are presented here. They are referenced in some query-by-example (QBE) examples, as well as in some reference descriptions.

---

### 2.8 Overview of QBE Operator $id

Other query-by-example (QBE) operators generally look for particular JSON fields within documents and try to match their values. Operator $id is an exception in that it instead matches document *keys*. It thus matches document *metadata*, not document content. You use operator $id in the outermost condition of a QBE.

**Example 2-4** shows three QBEs that use operator $id.

**Example 2-4  Using $id To Find Documents That Have Given Keys**

Find the unique document that has key key1.

{"$id" : "key1"}

Find the documents that have *any of the keys* key1, key2, and key3.

{"$id" : ["key1","key2","key3"]}

Find the documents that have *at least one of the keys* key1 and key2, and that have an object with a field address.zip whose value is at least 94000.

{"$and" : [{"$id" : ["key1", "key2"]},
   {"address.zip" : { "$gte" : 94000 }}]}

**Related Topics**

- [ID Clause (Reference)]
  Other query-by-example (QBE) operators generally look for particular JSON fields within the content of documents and try to match their values. An ID clause, which uses operator $id, instead matches document *keys*. It thus matches document *metadata*, not document content.
2.9 Overview of QBE Operator $orderby

Query-by-example (QBE) operator $orderby is described. It sorts query results in ascending or descending order.

You can specify the sort order for individual fields and the relative sort order among fields.

Operator $orderby can be used with two alternative syntaxes: array and abbreviated.

Regardless of the syntax choice, when you use $orderby in a filter specification together with one or more filter conditions, you must wrap those conditions with operator $query. In the queries shown here, the returned documents are restricted to those that satisfy a filter condition that specifies that field age must have a value greater than 40.

Using the Orderby Clause Array Syntax

The array syntax is the more straightforward of the two. You follow $orderby by an array of the fields to sort, in their relative sort order: the first array element specifies the first field to sort by, the second element specifies the second field to sort by, and so on.

The array syntax also lets you specify the SQL data type to use for sorting a given field, that is, how to interpret the field values, for sorting purposes.

For example, you can specify whether a field that has numeric codes (as a string or as a number) should be sorted lexicographically (as a string of digit characters) or numerically as a sequence of digits interpreted as a number). With "varchar2" as the sort data type, "100" sorts, in ascending order, before "9". With "number" as the sort type, "9" sorts, in ascending order, before "100", since the number 9 is smaller than the number 100.

Note:

To use a datatype value of date or timestamp you need Oracle Database Release 18c or later.

Finally, the array syntax also lets you specify, for a string-valued field, a maximum number of characters at the start of the string. An error is raised if a string value for the targeted field is too long.

The following QBE selects objects in which field salary is between 10,000 and 20,000, inclusive. It sorts the objects first by descending age, interpreted as a number, then by ascending name, interpreted as a string. An error is raised if the string value of field name is longer than 100 characters in any matching document. The default error handling also applies: raise an error if the value of any of the specified fields is not convertible to the specified datatype, but do not raise an error just because some of the specified fields are missing.

```json
{ "$query" : { "salary" : { "$gt" : 10000, "$lte" : 20000 } },
"$orderby" : [
{ "path" : "age",
  "datatype" : "number",
  "order" : "desc" },
{ "path" : "name",
  "datatype" : "varchar2"
},
```

```json
```
The following QBE is the same, except that it specifies `scalarRequired = true`, to require that field `name` be present in each matching document (as well as requiring that its value be convertible to a string). Otherwise, an error is raised at query time.

```
{ "$query" : { "salary" : { "$gt" : 10000, "$lte" : 20000 } },
"orderby" : { "$fields" :
  [ { "path" : "age",
    "datatype" : "number",
    "order" : "desc" },
    { "path" : "name",
    "datatype" : "varchar2",
    "order" : "asc",
    "maxLength" : 100 } ],
  "$scalarRequired" : true } }
```

### Using the Orderby Clause Abbreviated Syntax

The abbreviated syntax lets you list the fields to sort by and their relative sort order in a succinct way. You cannot use it to specify how to interpret the values of a given field for sorting purposes, that is, which data type to interpret the values as. And you cannot specify a maximum number of characters to take into account when sorting a string field.

The following QBE specifies the order of fields `age` and `name` when sorting documents where the `salary` is between 10,000 and 20,000. A value of `-1` specifies **descending** order for `age`. A value of `2` specifies **ascending** order for `name`. Sorting is done **first by age** and then by name, because the absolute value of `-1` is less than the absolute value of `2` — not because `-1` is less than `2`, and not because field `age` appears before field `name` in the `$orderby` object.

```
{ "$query" : { "salary" : { "$between" : [10000, 20000] } },
"orderby" : { "age" : -1, "name" : 2 } }
```

### Related Topics

- **Orderby Clause Sorts Selected Objects**
  A filter specification (query-by-example, or QBE) with an orderby clause returns the selected JSON documents in sorted order.

### 2.10 Overview of QBE Spatial Operators

You can use query-by-example (QBE) operator `$near`, `$intersects`, or `$within` to select documents that have a field whose value is a GeoJSON geometry object that is **near** a specified position, **intersects** a specified geometric object, or is **within** another specified geometric object, respectively.
Note:
To use QBE spatial operators you need Oracle Database Release 12c (12.2.0.1) or later.

The following QBE selects only documents that have a location field whose value is a Point GeoJSON geometry object that represents a position within 50 kilometers of the coordinates [34.0162, -118.2019].

```json
{ "location" : 
  { "$near" : 
    { "$geometry" : { "type" : "Point",
      "coordinates" : [34.0162, -118.2019] },
    "$distance" : 50,
    "$unit" : "KM" } } }
```

It can retrieve a document that has an object such as this one, for example:

```json
{ "location" : { "type" : "Point",
  "coordinates": [33.7243, -118.1579] } }
```

Any document that does not contain a location field is ignored (skipped) without error. But if the queried collection contains a document with a location field that does not have as value a (single) GeoJSON geometry object then an error is raised. A document with this object, for example, raises an error:

```json
{ "location" : "1600 Pennsylvania Ave NW, Washington, DC 20500" }
```

You can provide different (non-default) error-handling behavior for your QBE by including a true-valued $scalarRequired or $lax field (but not both together) in the object that is the value of spatial operator $near, $intersects, or $within.

- A true value for field $scalarRequired means raise an error if any document does not have a location field. (An error is still also raised for a location field whose value is not a geometry object.)
- A true value for field $lax means ignore not only a missing location field but also a location field whose value is not a GeoJSON geometry object.

For example, this QBE raises an error if any document has no location field or if any document has a location field whose value is not a geometry object:

```json
{ "location" : 
  { "$near" : 
    { "$geometry" : { "type" : "Point",
      "coordinates" : [34.0162, -118.2019] },
    "$distance" : 50,
    "$unit" : "KM",
    "$scalarRequired : true } } }
```
And this QBE does not raise an error for a document that has no location field or for a document that has a location field whose value is not a geometry object:

```json
{ "location" : 
{ "$near" : 
{ "$geometry" : { "type" : "Point", 
"coordinates" : [34.0162, -118.2019] }, 
"$distance" : 50, 
"$unit" : "KM", 
"$lax" : true } } }
```

**Note:**

If you have created a SODA spatial index for a field whose value is a GeoJSON geometry object, and if you use a QBE that targets that field, then the index can be picked up for the QBE only if both index and QBE specify the same error-handling behavior for that field. Both must specify the same one of these:

- scalarRequired : true
- lax : true
- Neither scalarRequired : true nor lax : true

**Related Topics**

- [Spatial Clause (Reference)](#)
  GeoJSON objects are JSON objects that represent geographic data. You can use a SODA QBE spatial clause to match GeoJSON geometry objects in your documents.

- [SODA Index Specifications (Reference)](#)
  You can index the data in JSON documents using index specifications. A detailed definition of SODA index specifications is presented.

### 2.11 Overview of QBE Operator $contains

Query-by-example (QBE) operator $contains performs full-text search of JSON documents in a SODA collection.

A QBE field whose value is an object with a $contains operator matches a JSON document if the document has that field with a full-word string value or a full-number value that matches the string operand of $contains somewhere, including in an array element. Matching is Oracle Text full-text.

The syntax of the search-pattern value for operator $contains is the same as the third parameter of SQL condition `json_textcontains`, and the resulting behavior is the same. This means, for instance, that you can query for text that is near some other text, or query use fuzzy pattern-matching. (If the search-pattern argument contains a character or a word that is reserved with respect to Oracle Text search then you must escape that character or word.)
To use full-text search with $contains you must create a JSON search index for the document collection. To do that you pass a search-index specification, such as in Example 3-3, to the index-creation function or method for your chosen SODA implementation (language).

Once you have a search index for your collection, this simple QBE searches the street fields of all documents, case-insensitively, for a value that contains the word "abc".

```json
{"street" : { "$contains" : "abc"}}
```

Example 2-2 is a match, for example, because it has a street field with value "30 ABC avenue".

Related Topics

- Contains Clause (Reference)
  A contains clause is a field followed by an object with one $contains operator, whose value is a string. It matches a JSON document only if a string or number in the field value matches the string operand somewhere, including in array elements. Matching is Oracle Text full-text.

- Overview of SODA Indexing
  The performance of SODA QBEs can sometimes be improved by using indexes. You define a SODA index with an index specification, which is a JSON object that specifies how particular QBE patterns are to be indexed for quicker matching.

- SODA Index Specifications (Reference)
  You can index the data in JSON documents using index specifications. A detailed definition of SODA index specifications is presented.

See Also:

- JSON_TEXTCONTAINS Condition in Oracle Database SQL Language Reference

2.12 Overview of QBE Operator $textContains

Query-by-example (QBE) operator $textContains performs full-text search of documents in a heterogeneous SODA collection, that is, a collection that has the media type column.

You can use it to find non-JSON documents that contain text that matches a full-text search pattern. For example, Microsoft Word, Portable Document Format (PDF), and plain-text documents can all be full-text searched using $textContains.

(You can also use $textContains with JSON documents, but for collection with only JSON documents operator $contains is preferable. Operator $contains is not for heterogeneous collections.)

The syntax of the search-pattern value for operator $textContains is the same as that used by SQL function contains, and the resulting behavior is the same. This means, for instance, that you can query for text that is near some other text, or query use fuzzy pattern-matching. (If the search-pattern argument contains a character or a word that is reserved with respect to Oracle Text search then you must escape that character or word.)
To be able to use operator $textContains you first must create an Oracle Text search index for the database table that underlies the heterogeneous collection to be searched. This SQL code creates search index mySearchIndex on content column myContentColumn of table myTextCollectionTable:

```
CREATE SEARCH INDEX mySearchIndex ON
myTextCollectionTable(myContentColumn)
```

Prior to Oracle Database Release 21c, this more verbose syntax is needed:

```
CREATE INDEX mySearchIndex ON
myTextCollectionTable(myContentColumn)
INDEXTYPE IS CTXSYS.CONTEXT
```

If you have a search index for your collection, this simple QBE searches for the text Is it about a bicycle?.

```
{"$textContains" : "Is it about a bicycle?"}
```

**Note:**

Operator $textContains is not supported for SODA for Java or SODA for REST.

To use $textContains with SODA for PL/SQL you need Oracle Database Release 21c (21.3) or later.

To use $textContains with SODA for C, SODA for Node.js, or SODA for Python you need the corresponding 21.3 client libraries.

**Related Topics**

- **Text-Contains Clause (Reference)**
  A text-contains clause is operator $textContains with a string value that is used as a full-text search pattern: $textcontains:pattern. It matches a non-JSON document only if some text in the document matches that pattern. Matching is Oracle Text full-text matching.
Overview of SODA Indexing

The performance of SODA QBEs can sometimes be improved by using indexes. You define a SODA index with an index specification, which is a JSON object that specifies how particular QBE patterns are to be indexed for quicker matching.

Suppose that you often use a query such as 

```
{"dateField" : {"$date" : DATE-STRING}}
```

where DATE-STRING is a string in an ISO 8601 format supported by SODA. Here, item method $date transforms DATE-STRING to a SQL value of data type DATE. You can typically improve the performance of queries on a field such as "dateField" by creating a **B-tree index** for it.

Or suppose that you want to query spatial data in a GeoJSON geometry object. You can improve the performance of such queries by creating a SODA spatial index for that data.

Or suppose that you want to be able to perform full-text queries using QBE operator $contains. You can enable such queries by creating a **JSON search index** for your data.

Or suppose that you want to perform metadata queries on a **JSON data guide**, which is a summary of the structural and type information about a set of JSON documents. You can create a JSON search index that holds and automatically updates such data-guide information.

In all such cases you specify the index you want by creating a SODA **index specification** and then using it to create the specified index.

Each SODA implementation 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.

---

**Note:**

- To create a B-tree index you need Oracle Database Release 12c (12.2.0.1) or later.
- To create a B-tree index that indexes a DATE or a TIMESTAMP value you need Oracle Database Release 18c (18.1) or later.
- To create a spatial index you need Oracle Database Release 12c (12.2.0.1) or later.
- To create a search index you need Oracle Database Release 12c (12.2.0.1) or later.
Example 3-1  Specifying a B-Tree Index

This example specifies a B-tree non-unique index for numeric field address.zip.

```json
{
  "name" : "ZIPCODE_IDXX",
  "fields" : [
    {
      "path" : "address.zip",
      "datatype" : "number",
      "order" : "asc"
    }
  ]
}
```

This indexes the field at path address.zip in Example 2-1 and Example 2-2.

Example 2-3 has no such field, so that document is skipped during indexing.

You can specify that the index requires that all indexed fields be scalar by including scalarRequired : true in the index specification:

```json
{
  "name" : "ZIPCODE_IDX",
  "fields" : [
    {
      "path" : "address.zip",
      "datatype" : "number",
      "order" : "asc"
    },
    {
      "scalarRequired" : true
    }
  ]
}
```

If a specification includes scalarRequired : true, and if the collection contains a document that is missing one or more of the specified fields (just address.zip in this case), or if any of them has a non-scalar value, then an error is raised when creating the index. In addition, if such an index exists when you try to write a document that lacks that one of the indexed fields then an error is raised for the write operation.

Regardless of the value of scalarRequired, an error is raised if you try to write a document that has the targeted field but with a value that is not convertible to the specified data type. For example, for the index defined in Example 3-1, if a document contains field address.zip, but the field value is not convertible to a number, then an error is raised. This would be the case, for instance, for a zip field whose value is an object.

Example 3-2  Specifying a Spatial Index

This example specifies an Oracle Spatial and Graph index named LOCATION_LONG_LAT_IDX, which indexes the GeoJSON geometry object that is the value of field location in your documents:

```json
{
  "name" : "LOCATION_LONG_LAT_IDX",
  "spatial" : "location"
}
```

This index specification applies to all documents that have a location field whose value is a GeoJSON geometry object, and only to such documents. Here’s an example of an object with such a location field, whose value is a geometry object of type Point:

```json
{
  "location" : {
    "type" : "Point",
    "coordinates" : [ 33.7243, 118.1579 ]
  }
}
```

That location value is indexed, because its value is a GeoJSON geometry object.
Because neither `ScalarRequired : true` nor `lax : true` is specified in the index specification, a document that has no `location` field is silently skipped (not indexed) during indexing.

And if the collection that is queried has a document with an object such as one of the following, whose `location` values are not GeoJSON geometry objects, then an error is raised during indexing.

```json
{ "location" : [ 33.7243, 118.1579 ] }
{ "location" : "1600 Pennsylvania Ave NW, Washington, DC 20500" }
```

In addition, if such an index exists, and you try to write a document that has `location` field with such a non-geometry value, then an error is raised for the write operation.

You can specify that the index requires that all indexed fields be scalar by including `ScalarRequired : true` in the index specification:

```json
{ "name" : "LOCATION_LONG_LAT_IDX",
  "spatial" : "location",
  "ScalarRequired" : true }
```

With `ScalarRequired : true`, if the collection contains a document that has no `location` field, then an error is raised when creating the index. In addition, if such an index exists, and you try to write a document that lacks the indexed field (`location`), then an error is raised for the write operation. (An error is still also raised, for index creation or a write operation, for a `location` field whose value is not a geometry object.)

You can specify that the index does not require indexed fields to be present and have GeoJSON geometry values by including `lax : true` in the index specification:

```json
{ "name" : "LOCATION_LONG_LAT_IDX",
  "spatial" : "location",
  "lax" : true }
```

With `lax : true`, no error is raised for a document that lacks a `location` field or for a document with a `location` field value (such as `{"location" : [33.7243, 118.1579]}`) that is not a GeoJSON geometry object. The index simply ignores such documents.

**Note:**

If you have created a SODA spatial index for a field whose value is a GeoJSON geometry object, and if you use a QBE that targets that field, then the index can be picked up for the QBE only if both index and QBE specify the same error-handling behavior for that field. Both must specify the same one of these:

- `ScalarRequired : true`
- `lax : true`
- Neither `ScalarRequired : true` nor `lax : true`
Example 3-3  Specifying a JSON Search Index

This example specifies a JSON search index. The index does both of these things:

- Enables you to perform ad hoc full-word and full-number queries on your JSON documents.
- Automatically accumulates and updates aggregate structural and type information about your JSON documents: a JSON data guide.

{ "name" : "SEARCH_AND_DATA_GUIDE_IDX" }

This index specification is equivalent. It just makes explicit the default values.

{ "name"      : "SEARCH_AND_DATA_GUIDE_IDX",
  "dataguide" : "on",
  "search_on" : "text_value" }

(To specify a search index without data-guide support, just set field dataguide to "off".)

Related Topics

- SODA Index Specifications (Reference)
  You can index the data in JSON documents using index specifications. A detailed definition of SODA index specifications is presented.

See Also:

- Oracle Database JSON Developer’s Guide for information about using SQL to create json_value B-tree indexes
- Oracle Spatial Developer’s Guide for information about Oracle Spatial and Graph indexes
- Oracle Database JSON Developer’s Guide for information about JSON search indexes
SODA Paths (Reference)

SODA filter specifications contain paths, each of which targets a value in a JavaScript Object Notation (JSON) document. A path is composed of a series of steps. A detailed definition of SODA paths is presented.

Note:
A SODA QBE is itself a JSON object. You must use strict JSON syntax in a QBE. In particular, you must enclose all field names in double quotation marks ("="). This includes field names, such as address.zip, that act as SODA paths. For example, you must write {"address.zip" : 94088}, not {address.zip : 94088}.

The following characters can have special syntactic meaning in some SODA path steps, in which case their use in that context is called syntactic (they are used syntactically):

- Brackets ([ and ]) delimit a JSON array
- Comma (,) separates array elements or array index components
- Wildcard (*) is a placeholder that matches any array index in an array step or any field name in a field step (defined below)
- Period (.) separates a parent-object field name (or *) from a child-object field name (or *)

In any other path-expression context than those just listed, these same characters have no special syntactic meaning. For example, outside of its use in array syntax a comma is not used syntactically.

A character that is not used syntactically in a given context is ordinary in that context. For example, a comma is ordinary outside of its use in array syntax, and the character d is always ordinary.

There are two kinds of steps in a path: field steps and array steps.

A field step is one of the following:

- The wildcard character * (by itself)
- A sequence of characters that are always ordinary — for example, cat
- A sequence of any characters that is enclosed in backquote characters (') — for example, 'dog' and 'cat.dog'

Characters within a field step that is enclosed in backquote characters are not used syntactically; they are treated literally. If you intend for a character not to be used syntactically where it normally would be then you must enclose its step in backquote characters.
All of the characters in field name `dog` are always ordinary, so backquote characters are optional in `dog`. But the following field steps must be enclosed in backquote characters because each contains one or more characters that would otherwise be used syntactically:

```
'cat.dog'
'cat[dog]'  
'\*'  
```

In the path `a.*.b`, the asterisk acts as a wildcard; it is a placeholder for a field name. But in the path `a.*.b` the asterisk does not act as a wildcard. Because it is escaped by backquotes it acts as an ordinary character — a field named `*`. (In both cases the unescaped periods are used syntactically.)

Besides using backquotes to inhibit special syntactic meaning, you can use them to escape a dollar-sign character (`$`) at the beginning of a field name, where it would otherwise be interpreted as introducing a SODA operator name. For example, because of the backquote characters, the field step `\$eq` does not represent SODA operator $eq; it represents an ordinary JSON field that has the same name. (Needing to query data that has field names that begin with $ is rare.)

If a step that you enclose in backquote characters contains a backquote character, then you must represent that character using two consecutive backquote characters. For example:

```
`Customer``s Comment`
```

An unescaped period (.) must be followed by a field step. After the first step in a path, each field step must be preceded by a period.

An **array step** is delimited by brackets ([ and ]). Inside the brackets can be either:

- The wildcard character `*` (by itself)
- One or more of these **array index (position) components**:
  - A single **array index**, which is an integer greater than or equal to zero
  - An **array index range**, which has this syntax:
    
    \[ x \text{ to } y \]

    \[ x \text{ and } y \text{ are integers greater than or equal to zero, and } x \text{ is less than or equal to } y. \text{ There must be at least one whitespace character between } x \text{ and } \text{to and between } \text{to and } y. \]

    Multiple components must be separated by commas (,). In a list of multiple components, array indexes must be in ascending order, and ranges cannot overlap.

For example, these are valid array steps:

```
[*]
[1]
[1,2,3]
[1 to 3]
[1, 3 to 5]
```
The following are not valid array steps:

[*, 6]
[3, 2, 1]
[3 to 1]
[1 to 3, 2 to 4]

Related Topics

- **Overview of Paths in SODA QBEs**
  A filter specification, or query-by-example (QBE), contains zero or more paths to fields in JSON documents. A path to a field can have multiple steps, and it can cross the boundaries of objects and arrays.

- **SODA Filter Specifications (Reference)**
  You can select JSON documents in a collection by pattern-matching. A detailed definition of SODA filter specifications (QBEs) is presented.

**See Also:**

- *Oracle Database JSON Developer’s Guide* for information about strict and lax JSON syntax
- Introducing JSON for information about JSON
You can select JSON documents in a collection by pattern-matching. A detailed definition of SODA filter specifications (QBEs) is presented.

A filter specification, also known as a query-by-example (QBE) or simply a filter, is a SODA query that uses a pattern expressed in JSON. A QBE is itself a JSON object. SODA query operations use a QBE to select all JSON documents in a collection that satisfy it, meaning that the filter evaluates to true for only those documents. A QBE thus specifies characteristics that the documents that satisfy it must possess.

A filter can use QBE operators, which are predefined JSON fields whose names start with a dollar-sign character ($). The JSON value associated with an operator field is called its operand or its argument.\(^1\)

Although a SODA operator is itself a JSON field, for ease of exposition in the context of filter specification descriptions, the term “field” generally refers here to a JSON field that is not a SODA operator. Also, in the context of a QBE, “field” is often used informally to mean “path to a field”.

---

\(\textbf{Note:}\)

You must use \textit{strict} JSON syntax in a SODA filter specification, enclosing each nonnumeric, non-Boolean, and non-\texttt{null} JSON value in double quotation marks ("'). In particular, the names of all JSON fields, including SODA operators, must be enclosed in double quotation marks.

A filter specification is a JSON object. There are two kinds of filter specification:

\begin{itemize}
  \item Composite filter.
  \item Filter-condition filter.
\end{itemize}

A filter specification (QBE) can appear only at the top (root) level of a query. However, a filter condition can be used either (a) on its own, as a filter-condition filter (a QBE), or (b) at a lower level, in the query clause of a composite filter.

---

\(^1\) A syntax error is raised if the argument to a QBE operator is not of the required type (for example, if operator \$gt\ is passed an argument that is not a string or a number).
Note:

In general, query-by-example (QBE) is for querying JSON documents. QBE operators $id and $textContains can exceptionally be used with a heterogeneous collection, that is, a collection that has the media type column. Operator $textContains can only be used with a heterogeneous collection. (A heterogeneous collection can, but it need not, contain JSON documents.)

SODA for Java and SODA for REST do not support operator $textContains, and they do not support operator $id for use with a heterogeneous collection.

- **Composite Filters (Reference)**
  A composite filter specification (query-by-example, or QBE) can appear only at the top level. That is, you cannot nest a composite filter inside another composite filter or inside a filter condition.

- **Filter Conditions (Reference)**
  A filter condition can be used either on its own, as a filter specification, or at a lower level, in the query clause of a composite filter specification.

**Related Topics**

- **Overview of SODA Filter Specifications (QBEs)**
  A filter specification is a pattern expressed in JSON. You use it to select, from a collection, the JSON documents whose content matches it, meaning that the condition expressed by the pattern evaluates to true for the content of (only) those documents.

- **Media Type Column Name**
  The collection metadata component that specifies the name of the column that stores the media type of the document. A media type column is needed if the collection is to be heterogeneous, that is, it can store documents other than JavaScript Object Notation (JSON).

**See Also:**

*Oracle Database JSON Developer’s Guide* for information about strict and lax JSON syntax.

### 5.1 Composite Filters (Reference)

A composite filter specification (query-by-example, or QBE) can appear only at the top level. That is, you cannot nest a composite filter inside another composite filter or inside a filter condition.

A composite filter is an object that has at most one member of each of these clauses:2

---

2 SODA for REST provides additional clauses for use in a composite filter.
• **Query clause**
  It has the form $query : filter\_condition$.

• **Orderby clause**
  It has the form $orderby : orderby\_spec$, where is an array or an object specifying the sort order for the selected documents.

The order of the clauses is not significant. Absence of a clause has the same effect as applying its operator to an operand that is an empty object: Absence of a query clause selects all documents; absence of an orderby clause imposes no order.

The following composite filter contains a query clause and an orderby clause. The query clause selects documents that have a salary field whose value is greater than 10,000. The orderby clause sorts the selected documents first by descending age and then by ascending zip code.

```json
{ "$query" : { "salary" : { "gt" : 10000 } }, "$orderby" : { "age" : -1, "zip" : 2 } }
```

**Orderby Clause Sorts Selected Objects**

A filter specification (query-by-example, or QBE) with an orderby clause returns the selected JSON documents in sorted order.

**Related Topics**

• **Filter Conditions (Reference)**
  A filter condition can be used either on its own, as a filter specification, or at a lower level, in the query clause of a composite filter specification.

## 5.1.1 Orderby Clause Sorts Selected Objects

A filter specification (query-by-example, or QBE) with an orderby clause returns the selected JSON documents in sorted order.

There are two ways of controlling the ordering behavior, with different orderby-clause syntaxes:

• **An array syntax** lets you specify the SQL data types used and provides simple control over the field order. Sorting is by the first field specified, then by the second, and so on.
  
  There are two variants of this syntax, depending on whether you need to change the default behavior for handling of errors or empty fields.

• **An abbreviated syntax** does not let you specify the SQL data types used. In its most abbreviated form it also does not provide control over the order of the fields used for sorting.

**Orderby Clause Array Syntax**

The simplest orderby array syntax is operator $orderby$ followed by an array of objects, each of which has a path field, which targets a particular field from the root of the candidate object, followed by at most one of each of these fields:

• **datatype**, which specifies the SQL data type to use — one of: "varchar2" (default), "number", "date", "datetime", "timestamp", "string" or "varchar". (Value datetime is a synonym for timestamp. Values "string" and "varchar" are synonyms for "varchar2".)
These values correspond to SQL data types VARCHAR2, NUMBER, DATE, and TIMESTAMP, respectively.

**Note:**

To use a datatype value of date or timestamp you need Oracle Database Release 18c or later.

- **order**, which specifies whether the field values are to be in ascending ("asc") or descending ("desc") order (default: "asc")
- **maxLength**, which is a positive integer that specifies the maximum length, in characters, of a targeted string value. If a string exceeds this limit then raise an error. The use of $lax (see below) inhibits raising the error and ignores the overlong string for sorting purposes. Field maxLength applies only when datatype is "varchar2".

For example, this filter specification selects objects in which field salary has a value greater than 10,000 and less than or equal to 20,000. It sorts the objects first by descending age, interpreted as a number, then by ascending name, interpreted as a string.

```
{ "$query" : { "salary" : { "$gt" : 10000, "$lte" : 20000 } },
  "$orderby" : [
    { "path" : "age", "datatype" : "number", "order" : "desc" },
    { "path" : "name", "datatype" : "varchar2", "order" : "asc" } ] }
```

The following SQL SELECT statement fragment is analogous:

```
WHERE (salary > 10000) AND (salary <= 20000) ORDER BY age DESC, name ASC
```

This syntax serves most purposes. No error is raised just because the targeted field is absent, and any other error encountered is raised.³

If you need to specify special handling of missing fields or errors then you need to use the more elaborate array syntax. This wraps the array in a $fields object, which lets you add another field, $scalarRequired or $lax, to the $orderby object. You cannot specify a true value for both $lax and $scalarRequired, or else a syntax error is raised at query time.

- **$scalarRequired** — Boolean. Optional. When set to true the targeted field must be present, and its value must be a JSON scalar that is convertible to data type datatype. Raise an error at query time if, for any matched document, that is not the case.⁴

---

³ The default error-handling behavior corresponds to the SQL/JSON semantics ERROR ON ERROR NULL ON EMPTY.

⁴ A true value of $scalarRequired corresponds to the use of SQL clause ERROR ON ERROR for a json_value expression.
• **$lax** — Boolean. *Optional.* When set to `true` the targeted field need not be present or have a value that is a JSON scalar convertible to data type `datatype`. Do not raise an error at query time if, for any matched document, that is the case.\(^5\)

If neither `scalarRequired` nor `lax` is specified as `true` then the default error-handling behavior applies (no error is raised just because the targeted field is absent, and any other error encountered is raised).

For example, this filter specification has the same behavior as the preceding one, except that it raises an error if any of the targeted fields is missing.

```
{ "$query" : { "salary" : { "$gt" : 10000, "$lte" : 20000 } },
"$orderby" : {
"$fields" : [ { "path" : "age",
   "datatype" : "number",
   "order" : "desc" },
   { "path" : "name",
   "datatype" : "varchar2",
   "order" : "asc",
   "maxLength" : 100 } ],
"$scalarRequired" : true 
}
```

### Note:

If you use Oracle Database Release 12c (12.1.0.2) then you must specify either `$scalarRequired` or `$lax`; otherwise a syntax error is raised.

### Note:

If you have defined a B-tree index for any of the fields targeted by a QBE that has an `orderby` clause then that index must be specified with a `true` value of `indexNulls` for it to be picked up for that query.

### See Also:

- *Oracle Database JSON Developer’s Guide* for information about SQL/JSON error-handling values `ERROR ON ERROR` and `NULL ON ERROR`
- *Oracle Database JSON Developer’s Guide* for information about SQL/JSON empty field-handling values `NULL ON EMPTY` and `ERROR ON EMPTY`

**Orderby Clause Abbreviated Syntax**

The abbreviated `$orderby` syntax specifies the fields to use for sorting, along with their individual directions and the order of sorting among the fields. It does not specify the SQL data types to use when interpreting field values for sorting, and it does not let you limit string sorting to the first `N` characters.

\(^5\) A true value of `$lax` corresponds to the use of SQL clause `NULL ON ERROR` for a functional index created on a `json_value` expression.
The orderby abbreviated syntax is `$orderby` followed by an object with one or more members, whose fields are used for sorting:

```
"$orderby" : { field1 : direction1, field2 : direction2, ... }
```

Each `field` is a string that is interpreted as a path from the root of the candidate object.

Each `direction` is a non-zero integer. The returned documents are sorted by the `field` value in ascending or descending order, depending on whether the value is positive or negative, respectively.

The fields in the `$orderby` operand are sorted in the order of their magnitudes (absolute values), smaller magnitudes before larger ones. For example, a field with value -1 sorts before a field with value 2, which sorts before a field with value 3. As usual, the order of the fields in the object value of `$orderby` is immaterial.

If the absolute values of two or more sort directions are equal then the order in which the fields are sorted is determined by the order in which they appear in the serialized JSON content that you use to create the JSON document.

Oracle recommends that you use sort directions that have unequal absolute values, to precisely govern the order in which the fields are used, especially if you use an external tool or library to create the JSON content and you are unsure of the order in which the resulting content is serialized.

This query acts like the one in Orderby Clause Array Syntax, except that interpretation of data types is not specified here, and (assuming that field name has string values) all characters in the name are used for sorting here. Note that the order of the object members is irrelevant here. In particular, it does not specify which field is sorted first — that is determined by the value magnitudes.

```
{ "$query" : { "salary" : { $between [10000, 20000] }},
  "$orderby" : { "age" : -1, "name" : 2 } }
```

The following SQL `SELECT` statement fragment is analogous:

```
WHERE (salary >= 10000) AND (salary <= 20000) ORDER BY age DESC, name ASC
```

**Related Topics**

- **Overview of QBE Operator `$orderby`**
  Query-by-example (QBE) operator `$orderby` is described. It sorts query results in ascending or descending order.

- **SODA Paths (Reference)**
  SODA filter specifications contain paths, each of which targets a value in a JavaScript Object Notation (JSON) document. A path is composed of a series of steps. A detailed definition of SODA paths is presented.

- **SODA Index Specifications (Reference)**
  You can index the data in JSON documents using index specifications. A detailed definition of SODA index specifications is presented.
5.2 Filter Conditions (Reference)

A filter condition can be used either on its own, as a filter specification, or at a lower level, in the query clause of a composite filter specification.

A filter condition is a JSON object whose members form one or more of these clauses:

- scalar-equality clause
- field-condition clause
- logical combining clause
- nested-condition clause
- ID clause
- text-contains clause
- special-criterion clause

A filter condition is true if and only if all of its clauses are true. A filter condition can be empty (the empty object, `{}`), in which case all of its (zero) clauses are vacuously true (the filter condition is satisfied).

For example, if a QBE involves only one filter condition and it is empty then all documents of the collection are selected. In this case, a find operation returns all of the documents, and a remove operation removes them all.

- Scalar-Equality Clause (Reference)
  A scalar-equality clause tests whether a given object field is equal to a given scalar value.

- Field-Condition Clause (Reference)
  A field-condition clause specifies that a given object field must satisfy a given set of criteria. It constrains a field using one or more condition-operator clauses, each of which is a comparison clause, a not clause, or an item-method clause.

- Logical Combining Clause (Reference)
  A logical combining clause combines the effects of multiple non-empty filter conditions.

- Nested-Condition Clause (Reference)
  You use a QBE nested-condition clause to apply multiple conditions at the same time, to array elements that are objects.

- ID Clause (Reference)
  Other query-by-example (QBE) operators generally look for particular JSON fields within the content of documents and try to match their values. An ID clause, which uses operator $id, instead matches document keys. It thus matches document metadata, not document content.

- Text-Contains Clause (Reference)
  A text-contains clause is operator $textContains with a string value that is used as a full-text search pattern: $textcontains:pattern. It matches a non-JSON document only if some text in the document matches that pattern. Matching is Oracle Text full-text matching.

- Special-Criterion Clause (Reference)
  A special criterion clause is a spatial clause (with operator $near, $intersects, or $within), or a contains clause (with operator $contains).
5.2.1 Scalar-Equality Clause (Reference)

A **scalar-equality clause** tests whether a given object field is equal to a given scalar value.

A scalar-equality clause is an object member with a scalar value. It tests whether the value of the field is equal to the scalar.

```
field : scalar
```

(Reminder: a JSON **scalar** is a value other than an object or an array; that is, it is a JSON number, string, `true`, `false`, or `null`.)

A scalar-equality clause is equivalent in behavior to a field-condition clause with a comparison clause that tests the same field value using operator `$eq`. That is, `field : scalar` is equivalent to `field : { "$eq" : scalar }`.

Though the behavior is equivalent, a scalar-equality clause cannot be used in some contexts where the corresponding "$eq" : field member can be used. For example, a scalar-equality clause cannot be used in a not clause. The array elements in the argument array of a not clause must be comparison clauses.

5.2.2 Field-Condition Clause (Reference)

A field-condition clause specifies that a given object field must satisfy a given set of criteria. It constrains a field using one or more condition-operator clauses, each of which is a comparison clause, a not clause, or an item-method clause.

A **field-condition clause** is JSON-object member whose field is not an operator and whose value is an object with one or more members, each of which is a condition-operator clause:

```
field : { condition-operator-clause ... }
```

A field-condition clause tests whether the field satisfies all of the condition-operator clauses, which are thus implicitly ANDed.

A **condition-operator clause** is any of these:

- A comparison clause
- A not clause
- An item-method clause
Note:

When a path that does not end in an array step uses a comparison clause or a not clause, and the path targets an array, the test applies to each element of the array.

For example, the QBE {"animal" : {"$eq" : "cat"}} matches the JSON data {"animal" : ["dog", "cat"]}, even though "cat" is an array element. The QBE {"animal" : { "$not : { "$eq" : "frog"} }} matches the same data, because each of the array elements is tested for equality with "frog" and this test fails.

• Comparison Clause (Reference)
  A comparison clause is an object member whose field is a comparison operator. Example: "$gt" : 200.

• Not Clause (Reference)
  A not clause logically negates the truth value of a set of comparison clauses. When any of the comparison clauses is true, the not clause evaluates to false; when all of them are false, the not clause evaluates to true.

• Item-Method Clause (Reference)
  An item-method clause is an item-method equality clause or an item-method modifier clause. It applies an item method to the field of the field-condition clause in which it appears, typically to modify the field value. It then matches the result against the operand of the item-method.

• ISO 8601 Date, Time, and Duration Support
  International Standards Organization (ISO) standard 8601 describes an internationally accepted way to represent dates, times, and durations. Oracle Database supports the most common ISO 8601 formats as proper Oracle SQL date, time, and interval (duration) values. The formats that are supported are essentially those that are numeric-only, language-neutral, and unambiguous.

Related Topics

• Nested-Condition Clause (Reference)
  You use a QBE nested-condition clause to apply multiple conditions at the same time, to array elements that are objects.

• Composite Filters (Reference)
  A composite filter specification (query-by-example, or QBE) can appear only at the top level. That is, you cannot nest a composite filter inside another composite filter or inside a filter condition.

• Sample JSON Documents
  A few sample JSON documents are presented here. They are referenced in some query-by-example (QBE) examples, as well as in some reference descriptions.

5.2.2.1 Comparison Clause (Reference)

A comparison clause is an object member whose field is a comparison operator. Example: "$gt" : 200.

Table 5-1 describes the comparison operators. See Sample JSON Documents for the documents used in the examples in column Description.
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
</table>
| $exists | Tests whether the field exists. Matches document if either:  
  - The field exists and the operand represents true, meaning that it is any scalar value except false, null, or 0.  
  - The field does not exist and the operand represents false, meaning that it is false, null, or 0.  

  **Operand**  
  JSON scalar.  

  **Example**  
  ```json
  {drinks : { "$exists" : true }}
  ```  
  matches sample document 3.  
  ```json
  {drinks : { "$exists" : false }}
  ```  
  matches sample documents 1 and 2. |
| $eq | Matches document only if field value equals operand value.  

  **Operand**  
  JSON scalar.  

  **Example**  
  ```json
  {"name" : { "$eq" : "Jason" }}
  ```  
  matches sample document 1. |
| $ne | Matches document only if field value does not equal operand value or there is no such field in the document.  

  **Operand**  
  JSON scalar.  

  **Example**  
  ```json
  {"name" : { "$ne" : "Jason" }}
  ```  
  matches sample documents 2 and 3. |
| $gt | Matches document only if field value is greater than operand value.  

  **Operand**  
  JSON number or string.  

  **Example**  
  ```json
  {"age" : { "$gt" : 50 }}
  ```  
  matches sample document 2. |
### Table 5-1  (Cont.) Query-By-Example (QBE) Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$lt</td>
<td>Matches document only if field value is less than operand value.</td>
</tr>
<tr>
<td></td>
<td><strong>Operand</strong></td>
</tr>
<tr>
<td></td>
<td>JSON number or string.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td></td>
<td>```json</td>
</tr>
<tr>
<td></td>
<td>{&quot;age&quot; : { &quot;$lt&quot; : 50 }}</td>
</tr>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>matches sample document 1.</td>
</tr>
<tr>
<td>$gte</td>
<td>Matches document only if field value is greater than or equal to operand value.</td>
</tr>
<tr>
<td></td>
<td><strong>Operand</strong></td>
</tr>
<tr>
<td></td>
<td>JSON number or string.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td></td>
<td>```json</td>
</tr>
<tr>
<td></td>
<td>{&quot;age&quot; : { &quot;$gte&quot; : 45 }}</td>
</tr>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>matches sample documents 1, 2, and 3.</td>
</tr>
<tr>
<td>$lte</td>
<td>Matches document only if field value is less than or equal to operand value.</td>
</tr>
<tr>
<td></td>
<td><strong>Operand</strong></td>
</tr>
<tr>
<td></td>
<td>JSON number or string.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td></td>
<td>```json</td>
</tr>
<tr>
<td></td>
<td>{&quot;age&quot; : { &quot;$lte&quot; : 45 }}</td>
</tr>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>matches sample document 1.</td>
</tr>
<tr>
<td>$between</td>
<td>Matches document only if string or number field value is between the two operand array elements or equal to one of them.</td>
</tr>
<tr>
<td></td>
<td><strong>Operand</strong></td>
</tr>
<tr>
<td></td>
<td>JSON array of two scalar elements. The first must be the smaller of the two. (For string values, smaller means first, lexicographically.) At most one of the elements can be <code>null</code>, which means no limit. An error is raised if both are <code>null</code> or if there are not exactly two array elements.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td></td>
<td>```json</td>
</tr>
<tr>
<td></td>
<td>{&quot;age&quot; : { &quot;$between&quot; : [49, 70] }}</td>
</tr>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>matches sample documents 2 and 3.</td>
</tr>
<tr>
<td></td>
<td>```json</td>
</tr>
<tr>
<td></td>
<td>{&quot;age&quot; : { &quot;$between&quot; : [45, null] }}</td>
</tr>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>matches sample documents 1, 2, and 3. It is equivalent to</td>
</tr>
<tr>
<td></td>
<td>```json</td>
</tr>
<tr>
<td></td>
<td>{&quot;age&quot; : { &quot;$gte&quot; : 45 }}</td>
</tr>
</tbody>
</table>
|  | ```
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$startsWith</code></td>
<td>Matches document only if field value starts with operand value.</td>
</tr>
<tr>
<td></td>
<td><strong>Operand</strong></td>
</tr>
<tr>
<td></td>
<td>JSON string.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td></td>
<td>```json</td>
</tr>
<tr>
<td></td>
<td>{&quot;name&quot; : {$startsWith&quot; : &quot;J&quot;}}</td>
</tr>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>matches sample document 1.</td>
</tr>
<tr>
<td><code>$hasSubstring</code></td>
<td>Matches document only if field value is a string with a substring equal to the operand.</td>
</tr>
<tr>
<td>or <code>$instr</code></td>
<td><strong>Operand</strong></td>
</tr>
<tr>
<td></td>
<td>Non-empty JSON string.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td></td>
<td>```json</td>
</tr>
<tr>
<td></td>
<td>{&quot;street&quot; : { &quot;$hasSubstring&quot; : &quot;street&quot; }}</td>
</tr>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>matches sample documents 1 and 2.</td>
</tr>
<tr>
<td><code>$regex</code></td>
<td>Matches document only if field value matches operand regular expression.</td>
</tr>
<tr>
<td></td>
<td><strong>Operand</strong></td>
</tr>
<tr>
<td></td>
<td>SQL regular expression, as a JSON string.</td>
</tr>
<tr>
<td></td>
<td>See Oracle Database SQL Language Reference.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td></td>
<td>```json</td>
</tr>
<tr>
<td></td>
<td>{&quot;name&quot; : {$regex&quot; : &quot;.*son&quot;}}</td>
</tr>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>matches sample document 1.</td>
</tr>
<tr>
<td><code>$like</code></td>
<td>Matches document only if field value matches operand pattern.</td>
</tr>
<tr>
<td></td>
<td><strong>Operand</strong></td>
</tr>
<tr>
<td></td>
<td>SQL LIKE condition pattern, as a JSON string.</td>
</tr>
<tr>
<td></td>
<td>See Oracle Database SQL Language Reference.</td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
</tr>
<tr>
<td></td>
<td>```json</td>
</tr>
<tr>
<td></td>
<td>{&quot;city&quot; : {$like&quot; : &quot;Mar_&quot; }}</td>
</tr>
<tr>
<td></td>
<td>```</td>
</tr>
<tr>
<td></td>
<td>matches sample documents 2 and 3.</td>
</tr>
<tr>
<td><code>$in</code></td>
<td>Matches document only if field exists and its value equals at least one value in the operand array.</td>
</tr>
<tr>
<td></td>
<td><strong>Operand</strong></td>
</tr>
</tbody>
</table>
|               | Non-empty JSON array of scalars.  
1                                                                                       |
|               | **Example**                                                                                     |
|               | ```json                                                                                          |
|               |   {"address.zip" : {$in" : [ 94088, 90001 ] }}                                                   |
|               | ```                                                                                             |
|               | matches sample documents 1 and 2.                                                                |
Table 5-1  (Cont.) Query-By-Example (QBE) Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$nin</td>
<td>Matches document only if one of these is true:</td>
</tr>
<tr>
<td></td>
<td>· Field exists, but its value is not equal to any value in the operand array.</td>
</tr>
<tr>
<td></td>
<td>· Field does not exist.</td>
</tr>
<tr>
<td>Operand</td>
<td>Non-empty JSON array of scalars.¹</td>
</tr>
<tr>
<td>Example</td>
<td><code>{&quot;address.zip&quot; : { &quot;$nin&quot; : [ 90001 ] }}</code></td>
</tr>
</tbody>
</table>

matches sample documents 1 and 2.

| $all     | Matches document only if one of these is true: |
|          | · Field value is an array that contains all values in the operand array. |
|          | · Field value is a scalar value and the operand array contains a single matching value. |
| Operand  | Non-empty JSON array of scalars.¹ |
| Example  | `{"drinks" : { "$all" : ["soda", "tea"]}}` |

matches sample document 2.

|         | `{"drinks" : { "$all" : ["tea"]}}` |

matches sample documents 1 and 2.

¹ A syntax error is raised if the array does not contain at least one element.

Related Topics

- **Overview of QBE Comparison Operators**
  A query-by-example (QBE) comparison operator tests whether a given JSON object field satisfies some conditions.

5.2.2.2 Not Clause (Reference)

A not clause logically negates the truth value of a set of comparison clauses. When any of the comparison clauses is true, the not clause evaluates to false; when all of them are false, the not clause evaluates to true.

A **not clause** is an object member whose field is operator $not and whose value is an object whose members are comparison clauses, which are implicitly ANDed before negating the truth value of that conjunction.

"$not" : { comparison-clause ... }

Example: "$not" : {"$eq" : 200, "$lt" : 40}. 
The following field-condition clause matches documents that have no field
address.zip, as well as documents that have such a field but whose value is a scalar
other than "90001" or an array that has no elements equal to "90001":

"address.zip" : {"$not" : {"$eq" : "90001"}}

In contrast, the following field-condition clause has the complementary effect: it
matches documents that have a field address.zip whose value is either the scalar
"90001" or an array that contains that scalar value.

"address.zip" : {"$eq" : "90001"}

Here is an example of a field-condition clause with field salary and with value a not
clause whose operand object has more than one comparison clause. It matches salary
values that are not both greater than 20,000 and less than 100,000. That is, it
matches salary values that are either less than or equal to 20,000 or greater than or
equal to 100,000.

"salary" : {"$not" : {"$gt" : 20000, "$lt" : 100000}}

Related Topics

- **Overview of QBE Operator $not**
  Query-by-example (QBE) operator $not negates the behavior of its operand,
  which is a JSON object containing one or more comparison clauses, which are
  implicitly ANDed.

5.2.2.3 Item-Method Clause (Reference)

An **item-method clause** is an item-method equality clause or an item-method modifier
clause. It applies an item method to the field of the field-condition clause in which it
appears, typically to modify the field value. It then matches the result against the
operand of the item-method.

For example, item-method operator $timestamp interprets as a time stamp a string-
valued field that is in one of the supported ISO 8601 date formats. After the operator is
applied to the value of the targeted field, other processing takes place, including the
evaluation of any not clause and comparison clauses that make up the item-method
modifier clause. The QBE uses the modified data in place of the raw field data that is
in your JSON documents.

In some cases, the application of an item-method operator acts only as a filter,
removing targeted data from the QBE result set. For example, if item-
method $timestamp is applied to a string value that is not in one of the supported ISO
8601 date formats then there is no match — the query treats that field occurrence as if
it were not present in the document.
Note:

An item-method field (operator) does not, itself, use or act on its associated value (its operand). Instead, it acts on the value of the JSON data that matches its parent field.

For example, in the QBE \{"birthday" : {"$date" : {"$gt" : "2000-01-01"}}}\), item-method operator $date acts on the JSON data that matches its parent field, birthday. It does not use or act on its operand, which is the JSON object (a comparison clause in this case) \{"$gt" : "2000-01-01"\}. The birthday data (a JSON string of format ISO 8601) in your JSON document is interpreted as a date, and that date is then matched against the condition that it be greater than the date represented by the (ISO date) string "2000-01-01" (later than January 1, 2000).

This can take some getting used to. The operand is used after the item-method operator does its job. It is matched against the result of the action of the operator on the value of its parent field. A item-method operator is a filter of sorts — it stands syntactically between the field (to its left) that matches the data it acts on and (to its right) some tests that are applied to the result of that action.

Item-Method Equality Clause

An item-method equality clause is an object member whose field is an item-method operator and whose value is a JSON scalar.\(^6\)

\[
\text{item-method-operator} : \text{scalar}
\]

The clause first applies the item method to the field of the field-condition clause. It then tests whether the result is equal to the scalar value (operand).

Example: "$upper" : "john"

(An item-method equality clause is equivalent to an item-method modifier clause (see next) whose field value (operand) is an object with a single comparison clause with comparison operator $eq. For example, "$upper" : "john" is equivalent to "$upper" : {"$eq" : "john"}).

Item-Method Modifier Clause

An item-method modifier clause is an object member whose field is an item-method operator and whose value (operand) is an object whose members are comparison clauses or at most one not clause. The operand of the item-method operator cannot be an empty object.

\[
\text{item-method-operator} : \{ \text{comparison-or-not-clause} \ldots \} \] \(^7\)

The clause first applies the item method to the field of the field-condition clause. It then tests whether the result of that operation satisfies all of the comparison clauses and not clause in its object value.

\(^6\) Reminder: a JSON scalar is a value other than an object or an array; that is, it is a JSON number, string, true, false, or null.

\(^7\) At most one not clause is allowed in the operand.
Example: "$upper" : { "$between" : [ "ALPHA", "LAMBDA" ], "$not" : 
{ "$startsWith" : "BE" } }

Item-Method Operators

Here is a brief description of each item-method operator. The target of the operator is the data matched by the field of the field-condition clause in which the item-method clause appears — the parent field of the operator. It is not the operand of the operator.

Table 5-2  Item-Method Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$abs</td>
<td>Absolute value of the targeted JSON number.</td>
</tr>
<tr>
<td><strong>Target of Operator</strong></td>
<td>JSON number</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>{&quot;ordinate&quot; : {&quot;$abs&quot; : {&quot;$gt&quot; : 1.0}}}} matches a negative or positive ordinate value whose magnitude is greater than 1.0. It matches, for example, -1.3 and 1.3.</td>
</tr>
<tr>
<td>$boolean</td>
<td>A Boolean interpretation of the targeted JSON value.</td>
</tr>
<tr>
<td><strong>Target of Operator</strong></td>
<td>JSON Boolean value (true or false) or a string that when converted to lowercase is either &quot;true&quot; or &quot;false&quot;</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>{&quot;&quot;retired&quot; : {&quot;$boolean&quot; : true}} matches (only) a retired value of true or a string that matches &quot;true&quot; case-insensitively.</td>
</tr>
<tr>
<td>$ceiling</td>
<td>The targeted JSON number, rounded up to the nearest integer.</td>
</tr>
<tr>
<td><strong>Target of Operator</strong></td>
<td>JSON number</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>{&quot;&quot;age&quot; : {&quot;$ceiling&quot; : {&quot;$lt&quot; : 65}}}} matches an age value of 63.9. It does not match a value of 64.1, because 64.1 rounds up to 65.</td>
</tr>
<tr>
<td>$date²</td>
<td>A date interpretation of the targeted JSON string.</td>
</tr>
<tr>
<td><strong>Target of Operator</strong></td>
<td>JSON string in supported ISO 8601 format</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>{&quot;&quot;birthday&quot; : {&quot;$date&quot; : &quot;2018-06-30&quot;}}}} matches a &quot;birthday&quot; value of &quot;2018-06-30&quot; or &quot;2018-06-30T17:29:08Z&quot;, because they are supported ISO 8601 formats for the same date.</td>
</tr>
<tr>
<td>$double</td>
<td>A SQL BINARY_DOUBLE interpretation of the targeted JSON number or numeric string value.</td>
</tr>
<tr>
<td><strong>Target of Operator</strong></td>
<td>JSON number or numeric string</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>{&quot;&quot;thickness&quot; : {&quot;$double&quot; : {&quot;$lt&quot; : 1.0}}}} matches a thickness value of &quot;0.999999999&quot;.</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>$floor</code></td>
<td>The targeted JSON number, rounded down to the nearest integer.</td>
</tr>
<tr>
<td>Target of Operator: JSON number</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>`{&quot;age&quot; : {&quot;$floor&quot; : { &quot;$le&quot; : 65}}} matches an age value of 65.2. It does not match a value of 66.3, because 66.3 rounds down to 66.</td>
</tr>
<tr>
<td><code>$length</code></td>
<td>The number of characters in the targeted JSON string.</td>
</tr>
<tr>
<td>Target of Operator: JSON string</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>`{&quot;name&quot; : {&quot;$length&quot; : { &quot;$gt&quot; : 4}}} matches &quot;Jason&quot;. It does not match &quot;Mary&quot; because that string has only 4 characters.</td>
</tr>
<tr>
<td><code>$lower</code></td>
<td>The lowercase string that corresponds to the characters in the targeted JSON string.</td>
</tr>
<tr>
<td>Target of Operator: JSON string</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>`{&quot;name&quot; : {&quot;$lower&quot; : &quot;mary&quot;}} matches &quot;Mary&quot;.</td>
</tr>
<tr>
<td><code>$number</code></td>
<td>A SQL NUMBER interpretation of the targeted JSON number or numeric string value.</td>
</tr>
<tr>
<td>Using <code>$number</code> is equivalent to specifying a numeric constant.</td>
<td></td>
</tr>
<tr>
<td>Target of Operator: JSON number or numeric string</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>`{&quot;thickness&quot; : {&quot;$number&quot; : { &quot;$lt&quot; : 1.0}}} matches a thickness value of &quot;0.9999&quot;.</td>
</tr>
<tr>
<td>`{&quot;thickness&quot; : {&quot;$number&quot; : { &quot;$lt&quot; : 1.0}}} is equivalent to {&quot;thickness&quot; : { &quot;$lt&quot; : 1.0}}.</td>
<td></td>
</tr>
<tr>
<td><code>$size</code></td>
<td>The number of elements in an array, or 1 for a scalar or an object.</td>
</tr>
<tr>
<td>Target of Operator: JSON value of any kind</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>`{&quot;drinks&quot; : {&quot;$size&quot; : { &quot;$gt&quot; : 1}}} matches a drinks value of [&quot;soda&quot;, &quot;coffee&quot;] because the value is an array with more than one element.</td>
</tr>
<tr>
<td>`{&quot;address&quot; : {&quot;$size&quot; : 1}} matches an address value that is a JSON object.</td>
<td></td>
</tr>
<tr>
<td><code>$string</code></td>
<td>A SQL VARCHAR2(4000) interpretation of the targeted JSON scalar.</td>
</tr>
<tr>
<td>Using <code>$string</code> is equivalent to specifying a string constant (literal).</td>
<td></td>
</tr>
<tr>
<td>Target of Operator: JSON scalar other than null</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>`{&quot;age&quot; : {&quot;$string&quot; : { &quot;$lt&quot; : &quot;45&quot;}}} matches a numeric age value of 100, because the string &quot;100&quot; is lexicographically less than the string &quot;45&quot;.</td>
</tr>
<tr>
<td>`{&quot;age&quot; : {&quot;$string&quot; : { &quot;$lt&quot; : &quot;45&quot;}}} is equivalent to {&quot;age&quot; : { &quot;$lt&quot; : &quot;45&quot;}}.</td>
<td></td>
</tr>
</tbody>
</table>
Table 5-2  (Cont.) Item-Method Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
</table>
| $timestamp$ | A date-with-time interpretation of the targeted JSON string.  
**Target of Operator**  
JSON string in supported ISO 8601 format  
**Example**  
{"meeting—time" : {"$timestamp" : VALUE}}, where VALUE is any of the following, matches any of the same values:  
- "2016-07-26T02:06:01Z"  
- "2016-07-26T02:06:01" (UTC by default)  
- "2016-07-26T01:06:01-01:00" (1:00 am in a time zone that is one hour behind UTC is equivalent to 2:00 am UTC.)  

If VALUE is a date-only ISO 8601 string then its equivalent date-with-time value is used. For example, a date value of "2016-07-26" is treated as the date-with-time zone value "2016-07-26T00:00:00Z". |
| $type$ | The name of the JSON-language data type of the targeted data, as a lowercase JSON string.  
- "null" for a value of null.  
- "boolean" for a value of true or false.  
- "number" for a number.  
- "string" for a string.  
- "array" for an array.  
- "object" for an object.  
**Target of Operator**  
JSON value of any kind  
**Example**  
{"address" : {"$type" : "object"}} matches an address value that is a JSON object. |
| $upper$ | The uppercase string that corresponds to the characters in the targeted JSON string.  
**Target of Operator**  
JSON string  
**Example**  
{"name" : {"$upper" : "MARY"}} matches "Mary". |

1 The scalar-equality abbreviation {field : {operator : value}} is used everywhere in examples here, in place of the equivalent {field : {operator : {$eq" : value}}}.  
2 The operand of operator $date must be a JSON string that has a supported ISO 8601 format. Otherwise, no match is found.  
3 The operand of operator $timestamp must be a JSON string that has a supported ISO 8601 format. Otherwise, no match is found.
Note:

- If an item-method conversion fails for any reason, such as the operand being of the wrong type, then the path cannot be matched (it refers to no data), and no error is raised.
- If an item-method operator is applied to an array then it is in effect applied to each of the array elements.

For example, QBE {"color" :{"$upper" : "RED"}} matches data
{"color" : ["Red", "Blue"]} because the array has an element that when converted to uppercase matches "RED". The QBE is equivalent to
{"color[*]" :{"$upper" : "RED"}} — operator $upper is applied to each array element of the target data.

Note:

- To use item method operator $abs, $date, $size, $timestamp, or $type you need Oracle Database Release 18c or later.
- To use any other item method you need Oracle Database Release 12c (12.2.0.1) or later.

Related Topics

- **Overview of QBE Item-Method Operators**
  A query-by-example (QBE) item-method operator acts on a JSON-object field value to modify or transform it in some way (or simply to filter it from the query result set). Other QBE operators that would otherwise act on the field value then act on the transformed field value instead.

- **SODA Index Specifications (Reference)**
  You can index the data in JSON documents using index specifications. A detailed definition of SODA index specifications is presented.

- **ISO 8601 Date, Time, and Duration Support**
  International Standards Organization (ISO) standard 8601 describes an internationally accepted way to represent dates, times, and durations. Oracle Database supports the most common ISO 8601 formats as proper Oracle SQL date, time, and interval (duration) values. The formats that are supported are essentially those that are numeric-only, language-neutral, and unambiguous.

See Also:

*Oracle Database JSON Developer’s Guide*
5.2.2.4 ISO 8601 Date, Time, and Duration Support

International Standards Organization (ISO) standard 8601 describes an internationally accepted way to represent dates, times, and durations. Oracle Database supports the most common ISO 8601 formats as proper Oracle SQL date, time, and interval (duration) values. The formats that are supported are essentially those that are numeric-only, language-neutral, and unambiguous.

(Simple Oracle Document Access (SODA) does not support durations.)

Oracle Database Syntax for ISO Dates and Times

This is the syntax that Oracle Database supports for ISO dates and times:

- **Date (only):** `YYYY-MM-DD`
- **Date with time:** `YYYY-MM-DDThh:mm:ss[.ssssss]`[Z]([+-]hh:mm]

where:

- `YYYY` specifies the **year**, as four decimal digits.
- `MM` specifies the **month**, as two decimal digits, 00 to 12.
- `DD` specifies the **day**, as two decimal digits, 00 to 31.
- `hh` specifies the **hour**, as two decimal digits, 00 to 23.
- `mm` specifies the **minutes**, as two decimal digits, 00 to 59.
- `ss[.ssssss]` specifies the **seconds**, as two decimal digits, 00 to 59, optionally followed by a decimal point and 1 to 6 decimal digits (representing the fractional part of a second).
- `Z` specifies **UTC** time (time zone 0). (It can also be specified by `+00:00`, but not by `-00:00`.)
- `([+-]hh:mm)` specifies the time-zone as difference from UTC. (One of `+` or `-` is required.)

For a time value, the time-zone part is optional. If it is absent then UTC time is assumed.

No other ISO 8601 date-time syntax is supported. In particular:

- **Negative dates** (dates prior to year 1 BCE), which begin with a hyphen (e.g. `–2018-10-26T21:32:52`), are not supported.
- **Hyphen and colon separators** are required: so-called “basic” format, `YYYYMMDDThhmmss`, is not supported.
- **Ordinal dates** (year plus day of year, calendar week plus day number) are not supported.
- **Using more than four digits for the year** is not supported.

Supported dates and times include the following:

- `2018-10-26T21:32:52`
- `2018-10-26T21:32:52+02:00`
- `2018-10-26T19:32:52Z`
Unsupported dates and times include the following:

- 2018-10-26T21:32:52+02:00 (the hours part, 25, is out of range)
- 18-10-26T21:32 (the year is not specified fully)
- 2018-10-26T21:32 (if a time is specified then all of its parts must be present)

**Oracle Database Syntax for ISO Durations**

There are two supported Oracle Database syntaxes for ISO durations, the `ds_iso_format` specified for SQL function `to_dsinterval` and the `ym_iso_format` specified for SQL function `to_yminterval`. (`to_dsinterval` returns an instance of SQL type `INTERVAL DAY TO SECOND`, and `to_yminterval` returns an instance of type `INTERVAL YEAR TO MONTH`.)

These formats are used for data types `daysecondInterval` and `yearmonthInterval`, respectively, which Oracle has added to the JSON language.

- **ds_iso_format:**
  
  \[PdDTHhmMsS\], where \(d\), \(h\), \(m\), and \(s\) are digit sequences for the number of days, hours, minutes, and seconds, respectively. For example: "P0DT06H23M34S".

  \(s\) can also be an integer-part digit sequence followed by a decimal point and a fractional-part digit sequence. For example: P1DT6H23M3.141593S.

  Any sequence whose value would be zero is omitted, along with its designator. For example: "PT3M3.141593S". However, if all sequences would have zero values then the syntax is "P0D".

- **ym_iso_format**
  
  \[PYYMm\], where \(y\) is a digit sequence for the number of years and \(m\) is a digit sequence for the number of months. For example: "P7Y8M".

  If the number of years or months is zero then it and its designator are omitted. Examples: "P7Y", "P8M". However, if there are zero years and zero months then the syntax is "P0Y".

**Related Topics**

- **Item-Method Clause (Reference)**
  
  An **item-method clause** is an item-method equality clause or an item-method modifier clause. It applies an item method to the field of the field-condition clause in which it appears, typically to modify the field value. It then matches the result against the operand of the item-method.
5.2.3 Logical Combining Clause (Reference)

A logical combining clause combines the effects of multiple non-empty filter conditions.

A logical combining clause is a logical combining operator — $and, $or, or $nor — followed by a non-empty array of one or more non-empty filter conditions.

This logical combining clause uses operator $or. It is satisfied if either of its conditions is true (or if both are true). That is, it is satisfied if the document contains a field name whose value is "Joe", or if it contains a field salary whose value is 10000.

"$or" : [ {"name" : "Joe"}, {"salary" : 10000} ]

The following logical combining clause uses operator $and. Its array operand has two filter conditions as its members. The second of these is a condition with a logical combining clause that uses operator $or. This logical combining clause is satisfied if both of its conditions are true. That is, it is satisfied if the document contains a field age whose value is at least 60, and either it contains a field name whose value is "Jason" or it contains a field drinks whose value is "tea".

"$and" : [ {"age" : {$gte : 60}},
          {"$or" : [{"name" : "Jason"}, {"drinks" : "tea"}]} ]

• Omitting $and

Sometimes you can omit the use of $and.

5.2.3.1 Omitting $and

Sometimes you can omit the use of $and.

A filter condition is true if and only if all of its clauses are true. And a field-condition clause can contain multiple condition clauses, all of which must be true for the field-condition clause as whole to be true. In each of these, logical conjunction (AND) is implied. Because of this you can often omit the use of $and, for brevity.

This is illustrated by Example 5-1 and Example 5-2, which are equivalent in their effect. Operator $and is explicit in Example 5-1 and implicit (omitted) in Example 5-2.

The filter specifies objects for which the name starts with "Fred" and the salary is greater than 10,000 and less than or equal to 20,000 and either address.city is "Bedrock" or address.zip is 12345 and married is true.

8 A syntax error is raised if the array does not contain at least one element.
A rule of thumb for $and omission is this: If you omit $and, make sure that no field or operator in the resulting filter appears multiple times at the same level in the same object.

This rule precludes using a QBE such as this, where field `salary` appears twice at the same level in the same object:

```json
{ "salary" : { "$gt" : 10000 },
  "age"    : { "$gt" : 40 },
  "salary" : { "$lt" : 20000 } }
```

And it precludes using a QBE such as this, where the same condition operator, $regex, is applied more than once to field `name` in the same condition clause:

```json
{ "name" : { "$regex" : "son", "$regex" : "Jas" } }
```

The behavior here is not that the field condition is true if and only if both of the $regex criteria are true. To be sure to get that effect, you would use a QBE such as this one:

```json
{ $and : [ { "name" : { "$regex" : "son" }, { "name" : { "$regex" : "Jas" } } ] }
```

If you do not follow the rule of thumb for $and omission then only one of the conflicting condition clauses that use the same field or operator is evaluated; the others are ignored, and no error is raised. For the `salary` example, only one of the `salary` field-condition clauses is evaluated; for the `name` example, only one of the $regex condition clauses is evaluated. Which one of the set of multiple condition clauses gets evaluated is undefined.

**Example 5-1  Filter Specification with Explicit $and Operator**

```json
{ "$and" : [ { "name" : { "$startsWith" : "Fred" } },
              { "salary" : { "$gt" : 10000, "$lte" : 20000 } },
              { "$or" : [ { "address.city" : "Bedrock" },
                           { "address.zip" : 12345 } ] },
              { "married" : true } ] }
```

**Example 5-2  Filter Specification with Implicit $and Operator**

```json
{ "name" : { "$startsWith" : "Fred" },
  "salary" : { "$gt" : 10000, "$lte" : 20000 },
  "$or" : [ { "address.city" : "Bedrock" },
             { "address.zip" : 12345 } ],
  "married" : true }
```
5.2.4 Nested-Condition Clause (Reference)

You use a QBE nested-condition clause to apply *multiple conditions* at the same time, to array elements that are objects.

A *nested-condition clause* consists of a *parent* path, followed by a colon (:), and a single, non-empty *nested filter condition*.

```
parent : filter-condition
```

The path targets a parent object whose value is a child object that satisfies the nested condition. If the *parent* path ends with [*], then it targets a parent object whose value is either such a child object or an array with such an object as *at least one* of its elements. The latter case is typical: you end the *parent* path with [*].

All fields contained in the nested condition are scoped to the parent object. They act as multiple conditions on each of the array objects (or the single child object, if the parent's value is not an array).

**Note:**

Because the condition of a nested-condition clause follows a field, it *cannot contain* an ID clause or a special-criterion clause. Those clauses can occur only at the root level.

For example, suppose that field `address` has an array value with object elements that have fields `city` and `state`. The following nested-condition clause tests whether array `address` has at least one object with both a field `address.city` that has the value "Boston" and a field `address.state` that has the value "MA":

```
"address[*]" : { "city" : "Boston", "state" : "MA" }
```

Similarly, this nested-condition clause tests whether the value of `address.city` starts with `Bos` and `address.state` has the value "MA":

```
"address[*]" : { "city" : { "$startsWith" : "Bos" }, "state" : "MA" }
```

Now suppose that you have this document:

```
{ "address" : [ { "city" : "Boston", "state" : "MA" },
    { "city" : "Los Angeles", "state" : "CA" } ] }
```

Both of the above nested-condition clauses match that document.
They also match the following document, whose address field value is an object, not an array of objects. The [*] in a nested-condition clause is needed to handle the array case, but it also handles the single-object case.

```
{ "address" : { "city" : "Boston", "state" : "MA" } }
```

If you mistakenly omit the [*], then each object element of the array is matched independently against each of the multiple conditions specified.

For example, the following two queries are equivalent. The first one has the form of a nested-condition clause but without the [*]. These queries match each address in a document independently. Each object element of an address array is matched to see if it has a city value of "Boston" or it has a state value of "CA" — it can, but it need not, have both. Each of these queries thus matches the document shown above, which has no single object with both city "Boston" and state "CA".

```
{ "address" : { "city": "Boston", "state" : "CA" } }
```

```
{ "address.city" : "Boston", "address.state" : "CA" } }
```

The following query, with a nested-condition clause for parent field address, does not match the preceding document with an address value that is an array, because that document has no single object in the array with both a field city of value "Boston" and a field state of value "CA".

```
{ "address[*]" : { "city" : "Boston", "state" : "CA" } }
```

Related Topics

- **Overview of Nested Conditions in QBEs**
  You can use a query-by-example (QBE) with a nested condition to match a document that has a field with an array value with object elements, where a given object in the array satisfies multiple conditions.

- **Special-Criterion Clause (Reference)**
  A special criterion clause is a spatial clause (with operator $near, $intersects, or $within), or a contains clause (with operator $contains).

### 5.2.5 ID Clause (Reference)

Other query-by-example (QBE) operators generally look for particular JSON fields within the content of documents and try to match their values. An ID clause, which uses operator $id, instead matches document keys. It thus matches document metadata, not document content.

Operator $id is not supported by SODA for Java or SODA for REST for use with a heterogeneous collection, that is, a collection that has the media type column.

A document key uniquely identifies a given document. It is metadata, like the creation time stamp, last-modified time stamp, and version. It pertains to the document as a whole and is not part of the document content.
The syntax of an ID clause is QBE operator \$id followed by either a scalar key (document identifier) or a non-empty array of scalar keys. The scalar key must be either an integer or a string. The array elements must be either all integers or all strings. For example:

```
"$id" : "USA"
"$id" : [1001,1002,1003]
```

Like a special-criterion clause or a text-contains clause, you can use operator \$id only in the outermost condition of a QBE, that is, in a condition used in a composite filter or in a filter-condition filter.

An ID clause can be combined with other clauses in a \$and clause. Only a single ID clause can be combined with other clauses in the same QBE.

### Example 5-3  Use of Operator \$id in the Outermost QBE Condition

Each of these equivalent QBEs finds documents that have at least one of the keys key1 and key2, and that have a color field with value "red". (Operator \$and is implicit in the first QBE.)

```json
{ "$id" : [ "key1", "key2" ], "color" : "red" }
{ "$and" : [ [ "$id" : [ "key1", "key2" ] ], { "color" : "red" } ] }
{ "$and" : [ [ "$id" : [ "key1", "key2" ], "color" : "red" ] ] }
```

### Related Topics

- **Overview of QBE Operator \$id**
  Other query-by-example (QBE) operators generally look for particular JSON fields within documents and try to match their values. Operator \$id is an exception in that it instead matches document keys. It thus matches document metadata, not document content. You use operator \$id in the outermost condition of a QBE.

- **Special-Criterion Clause (Reference)**
  A special criterion clause is a spatial clause (with operator \$near, \$intersects, or \$within), or a contains clause (with operator \$contains).

### 5.2.6 Text-Contains Clause (Reference)

A text-contains clause is operator \$textContains with a string value that is used as a full-text search pattern: \$textcontains:pattern. It matches a non-JSON document only if some text in the document matches that pattern. Matching is Oracle Text full-text matching.

(Operator \$textContains is not supported by SODA for Java or SODA for REST.)

Oracle Text technology underlies SODA operator \$textContains. This means, for instance, that you can query for text that is near some other text, or query use fuzzy pattern-matching. Operator \$textContains acts like SQL function contains, and the syntax of the search-pattern is the same.

To use full-text search with \$textContains you must first create an Oracle Text search index for the heterogeneous collection to be searched. (See **Overview of QBE Operator \$textContains** for an example of how to do this.)
You can use a text-contains clause only in the outermost condition of a QBE, either in the sole top-level filter condition or in the sole condition in a query clause ($query). In either case, you can only use one text-contains clause in the condition.

Here is a typical filter condition containing a text-contains clause:

```
{ "$textContains" : "beth" }
```

You can use $textContains in conjunction with $id to search only particular documents. For example, these (equivalent) examples search the documents that have keys 1001, 1002, and 1003.

```
{ "$textContains" : "beth", "$id" : [1001,1002,1003] }
{ "$and" : [ { "$textContains" : "beth" }, { "$id" : [1001,1002,1003] } ] }
```

You cannot use a text-contains clause in conjunction with a clause other than ID. For example, this is not allowed:

```
• { "$textcontains" : "beth", "age" : 42 }
• { "$and" : [ { "$textcontains" : "beth" }, { "age" : 42 } ] }
```

Related Topics

- **Contains Clause (Reference)**
  A contains clause is a field followed by an object with one $contains operator, whose value is a string. It matches a JSON document only if a string or number in the field value matches the string operand somewhere, including in array elements. Matching is Oracle Text full-text.

- **Overview of QBE Operator $textContains**
  Query-by-example (QBE) operator $textContains performs full-text search of documents in a heterogeneous SODA collection, that is, a collection that has the media type column.

See Also:

- *Oracle Text Reference* for complete information about Oracle Text operator contains, including its search-pattern syntax
- *Oracle Text Reference* for information about the use of special characters in SQL function `contains` search patterns
- *Oracle Text Reference* for information about the words and characters that are reserved with respect to Oracle Text search, and *Oracle Text Reference* for information about how to escape them.
5.2.7 Special-Criterion Clause (Reference)

A special criterion clause is a spatial clause (with operator \$near, \$intersects, or \$within), or a contains clause (with operator \$contains).

Like an ID clause or a text-contains clause, you can use a special-criterion clause only in the outermost condition of a QBE, that is, in a condition used in a composite filter or in a filter-condition filter.

More precisely, if a QBE also uses other operators, in addition to the operators for a special-criterion clause, then its outermost condition must have operator \$and, and the special-criterion clauses must be members of elements in the array argument to that \$and occurrence.

- Spatial Clause (Reference)
  GeoJSON objects are JSON objects that represent geographic data. You can use a SODA QBE spatial clause to match GeoJSON geometry objects in your documents.

- Contains Clause (Reference)
  A contains clause is a field followed by an object with one \$contains operator, whose value is a string. It matches a JSON document only if a string or number in the field value matches the string operand somewhere, including in array elements. Matching is Oracle Text full-text.

Related Topics

- ID Clause (Reference)
  Other query-by-example (QBE) operators generally look for particular JSON fields within the content of documents and try to match their values. An ID clause, which uses operator \$id, instead matches document keys. It thus matches document metadata, not document content.

- Contains Clause (Reference)
  A contains clause is a field followed by an object with one \$contains operator, whose value is a string. It matches a JSON document only if a string or number in the field value matches the string operand somewhere, including in array elements. Matching is Oracle Text full-text.

5.2.7.1 Spatial Clause (Reference)

GeoJSON objects are JSON objects that represent geographic data. You can use a SODA QBE spatial clause to match GeoJSON geometry objects in your documents.

Note:

To use QBE spatial operators you need Oracle Database Release 12c (12.2.0.1) or later.

A spatial QBE clause is a field followed by an object with a spatial operator: \$near, \$intersects, or \$within. It matches the field only if it contains GeoJSON geographic data that is near a specified position, intersects a specified geometric object, or is within a specified geometric object, respectively.
Each of the spatial QBE operators is followed by a JSON object whose fields must include $geometry. Operator $near must also include field $distance, and it can include $unit. A compile-time error is raised if $geometry is missing or if $distance or $unit is present with operator $intersects or $within.

The value of field $geometry is interpreted as a GeoJSON geometry object (other than a geometry collection), such as a point or a polygon. Each such object has a type field, with the geometry type, such as "Point" or "Polygon" as value, and a coordinates field, which defines the shape and location of the object, respectively.

(For a single position, such as an object of type "Point", field coordinates is an array of numbers, the first three of which generally represent longitude, latitude, and altitude, in that order.)

The value of field $distance must be a positive number, the distance from the field preceding spatial operator $near to the geometry object specified by $geometry. For non-point geometry objects, such as lines and polygons, the distance is the minimum distance between them. The distance between two adjacent polygons is zero.

The value of field $unit is a string such as "mile" that specifies the GeoJSON unit to use for the $distance value. The available units are defined in database table SDO_UNITS_OF_MEASURE. The default unit is "mile".

Example 5-4  QBE With a Spatial Clause

This example matches a location field whose value is GeoJSON geometry object of type Point, and which is within 60 miles of the coordinates [-122.417, 37.783] (San Francisco, California). It would match data with a "location" value of [-122.236, 37.483] (Redwood City, California). (Note that the first element of array "coordinates" is the longitude, and the second is the latitude.)

```json
{"location" : { "$near" : { "$geometry" : { "type" : "Point", "coordinates" : [-122.417, 37.783] }, "$distance" : 60, "$unit" : "mile" } } }
```

The default error-handling behavior for a QBE spatial clause is that the targeted field need not be present, but if it is present then its value must be a single GeoJSON geometry object. An error is raised at query time if, for any matching document, that is not the case.\(^9\)

A spatial clause can specify an alternative error-handling behavior from the default by including one of the following Boolean fields with a true value in the object that a spatial operator ($near, $within, $intersects) applies to. (Only one of these error-handling fields can be specified as true; otherwise, a syntax error is raised at query time.)

- $scalarRequired — Boolean. Optional. The targeted field must be present and have a GeoJSON geometry object as its value. Raise an error at query time if, for any matched document, that is not the case.

---

\(^9\) The default error-handling behavior corresponds to the use of SQL clauses ERROR ON ERROR and NULL ON EMPTY for a json_value expression.

\(^1\) A true value of $scalarRequired corresponds to the use of SQL clause ERROR ON ERROR for a json_value expression.
• **$lax** — Boolean. *Optional.* The targeted field need *not* be present or have a GeoJSON geometry object as its value. Do not raise an error at query time if, for any matched document, that is the case.\(^1\)

![Note:]

If you have created a SODA spatial index for a field whose value is a GeoJSON geometry object, and if you use a QBE that targets that field, then the index can be picked up for the QBE *only* if both index and QBE specify the same error-handling behavior for that field. Both must specify the same one of these:

- scalarRequired : true
- lax : true
- Neither scalarRequired : true nor lax : true

**Related Topics**

- **Overview of QBE Spatial Operators**
  You can use query-by-example (QBE) operator $near, $intersects, or $within to select documents that have a field whose value is a GeoJSON geometry object that is near a specified position, intersects a specified geometric object, or is within another specified geometric object, respectively.

- **SODA Index Specifications (Reference)**
  You can index the data in JSON documents using index specifications. A detailed definition of SODA index specifications is presented.

**See Also:**

- *Oracle Spatial Developer's Guide* for information about using GeoJSON data with Oracle Spatial and Graph
- *Oracle Spatial Developer's Guide* for information about Oracle Spatial and Graph and SDO_GEOMETRY object type
- GeoJSON.org for information about GeoJSON
- *The GeoJSON Format Specification* for details about GeoJSON data
- *Oracle Database JSON Developer’s Guide* for information about using GeoJSON geographic data with SQL/JSON functions

### 5.2.7.2 Contains Clause (Reference)

A contains clause is a field followed by an object with one $contains operator, whose value is a string. It matches a JSON document only if a string or number in the field

\(^1\) A true value of $lax corresponds to the use of SQL clause NULL ON ERROR for a functional index created on a json_value expression.
value matches the string operand somewhere, including in array elements. Matching is Oracle Text full-text.

The string operand is matched as a full word or number against strings and numbers in the field value, including in array elements.

For example, $contains operand "beth" matches the string "Beth Smith", but not the string "Elizabeth Smith". Operand "10" matches the number 10 or the string "10 Main Street", but not the number 110 or the string "102 Main Street".

**Note:**

To use operator $contains you need Oracle Database Release 12c (12.2.0.1) or later.

Oracle Text technology underlies SODA QBE operator $contains. This means, for instance, that you can query for text that is near some other text, or query use fuzzy pattern-matching.

For details about the behavior of a SODA QBE contains clause see the Oracle Database documentation for SQL condition json_textcontains.

To be able to use operator $contains you first must create a JSON search index; otherwise, a QBE with $contains raises a SQL error.

You can use a contains clause only in the outermost condition of a QBE. You can have multiple contains clauses at the top level, provided their fields are different (objects in QBEs must not have duplicate fields).

For example, this QBE checks for a "name" field that contains the word "beth" (case-insensitively) and an "address" field that contains the number 10 or the string "10" as a word:

```json
{ "name" : { "$contains" : "beth" },
  "address" : { "$contains" : "10" } }
```

To have the effect of multiple contains clauses for the same field (search the same field for multiple word or number patterns), the outermost condition must have operator $and, and the contains clauses must occur in object elements of the array argument to that $and occurrence.

For example, this QBE checks for an "address" field that contains both the word "street" and either the number 10 or the word "10":

```json
{"$and" : [ { "address" : { "$contains" : "street" } },
              { "address" : { "$contains" : "10" } } ] }
```

**Related Topics**

- Logical Combining Clause (Reference)

  A logical combining clause combines the effects of multiple non-empty filter conditions.
• **Overview of SODA Indexing**
  The performance of SODA QBEs can sometimes be improved by using indexes. You define a SODA index with an index specification, which is a JSON object that specifies how particular QBE patterns are to be indexed for quicker matching.

• **SODA Index Specifications (Reference)**
  You can index the data in JSON documents using index specifications. A detailed definition of SODA index specifications is presented.

• **Overview of QBE Operator $contains**
  Query-by-example (QBE) operator `$contains` performs *full-text search* of *JSON* documents in a SODA collection.

---

**See Also:**

- *Oracle Database SQL Language Reference* for reference information about SQL condition `json_textcontains`
- *Oracle Database JSON Developer’s Guide* for information about full-text search of JSON documents using SQL condition `json_textcontains`
You can index the data in JSON documents using index specifications. A detailed definition of SODA index specifications is presented.

### Note:

- To create a B-tree index you need Oracle Database Release 12c (12.2.0.1) or later. To create a B-tree index that indexes a `DATE` or a `TIMESTAMP` value you need Oracle Database Release 18c (18.1) or later.
- To create a spatial index or a search index you need Oracle Database Release 12c (12.2.0.1) or later.

An **index specification** is a JSON object that specifies a particular kind of database index, which is used for operations on JSON documents. You can specify these kinds of index:

- **B-tree**: Used to index scalar JSON values. It is identified by the presence of field `fields`. (Only a B-tree index has this field.)
- **Spatial**: Used to index GeoJSON geographic data. It is identified by the presence of field `spatial`. (Only a spatial index has this field.)
- **Search**: Used for one or both of the following:
  - Ad hoc structural queries or full-text searches
  - JSON data guide

  A search index specification is identified by the lack of fields `fields` and `spatial`.

Each kind of index specification requires a **name** field, which is a string that names the index.

### B-Tree Index Specifications

A SODA B-tree index specification specifies a B-tree function-based index on SQL/JSON function `json_value`, which is used by SODA to query JSON documents for scalar values. A B-tree index specification can have the following fields. Field `fields` is required for a B-tree index specification. The other fields are optional.

- **fields** — Array of objects, each of which targets a field in the indexed documents that has a scalar JSON value. When the array has more than one element the index specification results in the creation of a **composite** B-tree index.

  The order of the elements in array `fields` specifies the primary order of indexing, that is, the order among the targeted fields. The field of the first array element has the highest priority; the field of the last element has the lowest priority.

  Each object in the array can have the following fields:

  - **path** — String specifying the path to the targeted field, whose value is expected to be a scalar. **Required.**
If there are any array steps in the path then only the first element of each such array is used for indexing. In your documents, only scalar values for the targeted field are handled by the index — any non-scalar values for the field are ignored by the index.

- **datatype** — String naming the data type of the targeted-field value, for indexing purposes. *Optional.* Possible values (all are interpreted case-insensitively): "varchar2" (default), "number", "date", "timestamp", and the "varchar2" synonyms "string" and "varchar".

An index can be used to improve performance when evaluating a QBE filter criterion if the effective type of the input data matched by QBE filter criteria matches the index *datatype* value.

For an index to be picked up, to evaluate a given QBE, it is sufficient that the scalar JSON value targeted by the QBE be interpreted as being of the same SQL data type as the value of index-specification field *datatype*. This is the case for a JSON number value or string value and an index *datatype* of "number" or "varchar2" (or a "varchar2" synonym or no *datatype*), respectively.

For other *datatype* values there is no directly corresponding JSON scalar data type, so for a QBE to pick up the index it needs to use an item-method operator, to transform the JSON value to a SQL value of the appropriate data type.

For example, in a QBE such as \{"dateField" : {"$date" : "2017-07-25"}} the input string value "2017-07-25" (which has one of the supported ISO 8601 date formats) is converted by QBE item-method operator $date to data type "date". An index specified with a *datatype* value of "date" can be picked up to evaluate the QBE.

A QBE that does not explicitly use item-method operator $number or $string can pick up an index whose *datatype* is "number" or "varchar2" (or one of its synonyms), respectively, because of the direct correspondence between JSON and SQL data types for such values. For example:

* Using QBE \{"numField" : 20\}, like using \{"numField" : {$number : 20}\}, can pick up an index created with *datatype* value "number".

* Using QBE \{"stringField" : "my string"\}, like using \{"stringField" : {$string" : "my string"}\}, can pick up an index created with *datatype* value "varchar2" (or one of its synonyms).

- **maxlength** — Number specifying the maximum length of the value to index. *Optional.* Ignored if the *datatype* is one (such as number) that has no length. If *maxlength* is not specified then the length of the value indexed is 4000 divided by the number of string fields that are indexed.

- **order** — Index sorting order. *Optional.* The value of field *order* can be the string "asc" or the number 1, meaning ascending order, or the string "desc" or the number -1, meaning descending order. Default: ascending order.

- **unique** — Boolean. *Optional.* Whether the index is unique. Default: nonunique (false).

- **indexNulls** — Boolean. *Optional.* Whether to index NULL values for the selected columns (by appending the numeric value 1 to the list of columns to index). Default: do not index NULL values (false).
Note:
You must specify a true value for indexNulls for the index to be picked up for the orderby clause of a QBE.

The default error-handling behavior is that the targeted field need not be present, but if it is present then its value must be a JSON scalar that is convertible to data type datatype. An error is raised at query time if, for any document, that is not the case. In addition, if such an index exists, and you try to write a document where that is not the case, then an error is raised for the write operation.1

A B-tree index specification can specify an alternative error-handling behavior from the default by including field scalarRequired with a true value. That requires that the targeted field be present and have a value convertible to data type datatype. If, for any document to be indexed, that is not the case then an error is raised at indexing time. In addition, if such an index exists, and you try to write a document where that is not the case, then raise an error for the write operation.2

Note:
A JSON null value in your data is always convertible to the data type specified for the index. That data is simply not indexed. (This is true regardless of the value of scalarRequired.)

Spatial Index Specifications

A SODA spatial index specification specifies an Oracle Spatial and Graph index, which indexes GeoJSON data. A spatial index specification has a spatial field, whose value is a string specifying the path to the JSON field to be indexed. The value of that targeted JSON field is expected to be a single GeoJSON geometry object, that is, a JSON scalar that is also a GeoJSON geometry object.

The default error-handling behavior is that the targeted field need not be present, but if it is present then its value must be a single GeoJSON geometry object. An error is raised at indexing time if, for any document to be indexed, that is not the case. In addition, if such an index exists, and you try to write a document where that is not the case, then an error is raised for the write operation.3

A spatial index specification can specify an alternative error-handling behavior from the default by including one of the following Boolean fields with a true value. (Only one of these error-handling fields can be specified as true; otherwise, a syntax error is raised at index-creation time.)

- scalarRequired — Boolean. Optional. The targeted field must be present and have a GeoJSON geometry object as its value. Raise an error at indexing time if, for any

---

1 The default error-handling behavior corresponds to the use of SQL clauses ERROR ON ERROR and NULL ON EMPTY for a functional index created on a json_value expression.

2 A true value of scalarRequired corresponds to the use of SQL clause ERROR ON ERROR for a functional index created on a json_value expression.

3 The default error-handling behavior corresponds to the use of SQL clauses ERROR ON ERROR and NULL ON EMPTY for a functional index created on a json_value expression.
document to be indexed, that is not the case. In addition, if such an index exists, and you try to write a document where that is not the case, then raise an error for the write operation.

- **lax** — Boolean. *Optional.* The targeted field need not be present or have a GeoJSON geometry object as its value. Do not raise an error at indexing time for any document to be indexed that lacks the field or for which the field value is not geometry. In addition, if such an index exists, and you try to write a document where that is the case, do not raise an error for the write operation.\(^5\)

### Note:

If you have created a SODA spatial index for a field whose value is a GeoJSON geometry object, and if you use a QBE that targets that field, then the index can be picked up for the QBE only if both index and QBE specify the same error-handling behavior for that field. Both must specify the same one of these:

- scalarRequired : true
- lax : true
- Neither scalarRequired : true nor lax : true

### Search Index Specifications

A SODA search index specification specifies a **JSON search index**, which indexes the textual context of your JSON documents in a general way. A search index can improve the performance of both (1) *ad hoc* structural queries, that is, queries that you might not anticipate or use regularly, and (2) queries that make use of *full-text* search. It is an Oracle Text index that is designed specifically for use with JSON data.

A JSON search index can also accumulate and update aggregate information about your documents. In this it provides a JSON data guide, which is a summary of the structural and type information contained in a set of JSON documents. It records metadata about the fields used in those documents.

You can use data-guide information to:

- Generate a JSON Schema document that describes the set of JSON documents.
- Create database views that you can use to perform SQL operations on the data in the documents.
- Automatically add or update virtual database columns that correspond to added or changed fields in the documents.

The data-guide information contained in a JSON search index is updated automatically as new JSON content is added.

By default, a search index specification creates an index that provides both of these features: a general index and a data guide. These features are specified by fields **search_on** (string) and **dataguide** (string), respectively.

**If field search_on is present with value "none" then the index provides only the data-guide functionality (no general search index). If field dataguide is present with value "off" then only the general search index functionality is provided (no data guide).**

---

4 A true value of scalarRequired corresponds to the use of SQL clause **ERROR ON ERROR** for a functional index created on a json_value expression.

5 A true value of lax corresponds to the use of SQL clause **NULL ON ERROR** for a functional index created on a json_value expression.
(A dataguide value of "on", or no field dataguide, specifies data-guide support).

Besides none, field search_on can also have value "text" or "text_value". Both of these support full-text queries, which use QBE operator $contains, and they both support ad hoc queries that make of other QBE operators, such as $eq, $ne, and $gt.

In addition, search_on value "text_value" indexes numeric ranges. This is a separate value because it has an added performance cost. If you do not need range indexing then you can save some index maintenance time and some disk space by specifying value text instead of text_value. The default value of search_on is text_value.

Related Topics

- **Overview of SODA Indexing**
  The performance of SODA QBEs can sometimes be improved by using indexes. You define a SODA index with an index specification, which is a JSON object that specifies how particular QBE patterns are to be indexed for quicker matching.

- **Item-Method Clause (Reference)**
  An item-method clause is an item-method equality clause or an item-method modifier clause. It applies an item method to the field of the field-condition clause in which it appears, typically to modify the field value. It then matches the result against the operand of the item-method.

- **Orderby Clause Sorts Selected Objects**
  A filter specification (query-by-example, or QBE) with an orderby clause returns the selected JSON documents in sorted order.

See Also:

- *Oracle Database JSON Developer’s Guide* for information about using SQL to create json_value B-tree indexes
- *Oracle Database JSON Developer’s Guide* for information about using SQL to index multiple JSON fields with a composite json_value B-tree index
- Ordering Columns in an Index in *Oracle Database Performance Tuning Guide*, and *Oracle Database SQL Language Reference*, section ASC | DESC, for information about indexing order
- *Oracle Database JSON Developer’s Guide* for information about the use of a NULL ON EMPTY clause for a B-tree index created on a json_value expression
- *Oracle Database JSON Developer’s Guide* for information about JSON search indexes
- ISO 8601 for information about the ISO date formats
- *Oracle Spatial Developer’s Guide* for information about spatial indexes
SODA Collection Metadata Components (Reference)

Collection metadata is composed of multiple components. A detailed definition of the components is presented.

Note:
The identifiers used for collection metadata components (schema name, table name, view name, database sequence name, and column names) must be valid Oracle quoted identifiers. Some characters and words that are allowed in Oracle quoted identifiers are strongly discouraged. For details, see Oracle Database SQL Language Reference.

• Default Collection Metadata
  The kind of database you use determines the collection metadata that is used by default, and which of its field values you can modify for custom metadata.

• Schema
  The collection metadata component that specifies the name of the Oracle Database schema that owns the table or view to which the collection is mapped.

• Table or View
  The collection metadata component that specifies the name of the table or view to which the collection is mapped.

• Key Column Name
  The collection metadata component that specifies the name of the column that stores the document key.

• Key Column Type
  The collection metadata component that specifies the SQL data type of the column that stores the document key.

• Key Column Max Length
  The collection metadata component that specifies the maximum length of the key column in bytes. This component applies only to keys of type VARCHAR2.

• Key Column Assignment Method
  The collection metadata component that specifies the method used to assign keys to objects that are inserted into the collection.

• Key Column Path
  The collection metadata component that specifies the top-level document path to the field used as an embedded document key.

1 Reminder: letter case is significant for a quoted SQL identifier; it is interpreted case-sensitively.
• **Key Column Sequence Name**
  The collection metadata component that specifies the name of the database sequence that generates keys for documents that are inserted into a collection if the key assignment method is `SEQUENCE`.

• **Content Column Name**
  The collection metadata component that specifies the name of the column that stores the database content.

• **Content Column Type**
  The collection metadata component that specifies the SQL data type of the column that stores the document content.

• **Content Column Format**
  The collection metadata component that specifies the format of the column that stores the document content.

• **Content Column Max Length**
  The collection metadata component that specifies the maximum length of the content column in bytes. This component applies only to content of type `VARCHAR2`.

• **Content Column JSON Validation**
  The collection metadata component that specifies the syntax to which JavaScript Object Notation (JSON) content must conform—strict or lax.

• **Content Column SecureFiles LOB Compression**
  The collection metadata component that specifies the SecureFiles LOB compression setting.

• **Content Column SecureFiles LOB Cache**
  The collection metadata component that specifies the SecureFiles LOB cache setting.

• **Content Column SecureFiles LOB Encryption**
  The collection metadata component that specifies the SecureFiles LOB encryption setting.

• **Version Column Name**
  The collection metadata component that specifies the name of the column that stores the document version.

• **Version Column Generation Method**
  The collection metadata component that specifies the method used to compute version values for objects when they are inserted into a collection or replaced.

• **Last-Modified Time Stamp Column Name**
  The collection metadata component that specifies the name of the column that stores the last-modified time stamp of the document.

• **Last-Modified Column Index Name**
  The collection metadata component that specifies the name of the index on the last-modified column.

• **Creation Time Stamp Column Name**
  The collection metadata component that specifies the name of the column that stores the creation time stamp of the document. This time stamp is generated during the `insert`, `insertAndGet`, `save`, or `saveAndGet` operation.

• **Media Type Column Name**
  The collection metadata component that specifies the name of the column that stores the media type of the document. A media type column is needed if the
collection is to be heterogeneous, that is, it can store documents other than JavaScript Object Notation (JSON).

- **Read Only**
The collection metadata component that specifies whether the collection is read-only.

**Related Topics**

- **Overview of SODA Document Collections**
  A SODA collection is a set of documents that is backed by an Oracle Database table or view.

### 7.1 Default Collection Metadata

The kind of database you use determines the collection metadata that is used by default, and which of its field values you can modify for custom metadata.

In particular, the default SQL data type of the column used to store JSON content (the **content column**), and the default method for computing object version values (the **version column generation method**), depend on your database.

If the Oracle Database you use is an Oracle Autonomous Database — Autonomous JSON Database (AJD), Autonomous Transaction Processing (ATP), or Autonomous Data Warehouse (ADW) — then SODA always uses Oracle's native JSON format, OSON, to store the JSON content. Fields `contentColumn.sqlType` and `contentColumn.jsonFormat` in the metadata reflect this. These fields are not customizable for an autonomous database.

- If database initialization parameter `compatible` is at least 20 then:
  - The value of field `contentColumn.sqlType` in the default metadata is "JSON". JSON data type uses OSON format.
  - The value of field `versionColumn.method` in the default metadata is "UUID".
- Otherwise (parameter `compatible` is less than 20).

  For an autonomous database:
  - The value of field `contentColumn.sqlType` is "BLOB". This Binary Large Object (BLOB) data uses Oracle's native JSON format, OSON. (For an autonomous database this field is not customizable.)
  
  The additional field `contentColumn.jsonFormat` is present, with value "OSON". (For an autonomous database this field is not customizable.)
  
  The value of field `versionColumn.method` is "UUID".

  For a nonautonomous database, that is, for an on-premise database or a nonautonomous cloud database:
  - The value of field `contentColumn.sqlType` in the default metadata is "BLOB". BLOB textual data is used for JSON content, by default. The data is character data encoded using a Unicode encoding, either UTF-8 or UTF-16.
  - The following additional `contentColumn` fields are present: `compress`, `cache`, `encrypt`, and `validation`.
  - The value of field `versionColumn.method` in the default metadata is "SHA256".

You can define custom metadata, whose values for some fields differ from the default values, as follows:
• If the database you use is an autonomous database, then:
  – You can change metadata field `keyColumn.assignmentMethod` to `CLIENT` (instead of the default value, `UUID`), to specify client-assignment of document keys.
  – If the autonomous database is Autonomous Transaction Processing (ATP) or Autonomous Data Warehouse (ADW), but not Autonomous JSON Database (AJD), then you can add metadata field `mediaTypeColumn.name`, to specify the name of the column that stores the media type of a document. (By default, this field is absent.) A media-type column is needed for a heterogeneous collection, that is, a collection that can store documents other than JSON documents.

• If the database you use is not an autonomous database, then you can customize any metadata fields.

Note:
You need certain versions of SODA drivers and related software to support collection content type that uses Oracle’s native JSON binary format, OSON, that is, for JSON type or for BLOB type with format OSON. See SODA Drivers

Related Topics
• Content Column Type
  The collection metadata component that specifies the SQL data type of the column that stores the document content.
• Content Column Format
  The collection metadata component that specifies the format of the column that stores the document content.
• Content Column JSON Validation
  The collection metadata component that specifies the syntax to which JavaScript Object Notation (JSON) content must conform—strict or lax.
• Content Column SecureFiles LOB Compression
  The collection metadata component that specifies the SecureFiles LOB compression setting.
• Content Column SecureFiles LOB Cache
  The collection metadata component that specifies the SecureFiles LOB cache setting.
• Content Column SecureFiles LOB Encryption
  The collection metadata component that specifies the SecureFiles LOB encryption setting.
7.2 Schema

The collection metadata component that specifies the name of the Oracle Database schema that owns the table or view to which the collection is mapped.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>None</td>
</tr>
<tr>
<td>Allowed values</td>
<td>Valid Oracle quoted identifier(^1). If this value contains double quotation marks (&quot;&quot;) or control characters, SODA replaces them with underscore characters (_).</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>schemaName</td>
</tr>
</tbody>
</table>

See Also:
Oracle Database SQL Language Reference for information about valid Oracle quoted identifiers

7.3 Table or View

The collection metadata component that specifies the name of the table or view to which the collection is mapped.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>None</td>
</tr>
<tr>
<td>Allowed values</td>
<td>Valid Oracle quoted identifier(^1). If this value contains double quotation marks (&quot;&quot;) or control characters, SODA replaces them with underscore characters (_).</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>tableName or viewName</td>
</tr>
</tbody>
</table>

See Also:
Oracle Database SQL Language Reference for information about valid Oracle quoted identifiers
### 7.4 Key Column Name

The collection metadata component that specifies the name of the column that stores the document key.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>ID</td>
</tr>
<tr>
<td>Allowed values</td>
<td>Valid Oracle quoted identifier (^1) (as defined in Oracle Database SQL Language Reference). If this value contains double quotation marks (&quot;*) or control characters, SODA replaces them with underscore characters (_).</td>
</tr>
</tbody>
</table>

**JSON collection metadata document path**

keyColumn.name

### 7.5 Key Column Type

The collection metadata component that specifies the SQL data type of the column that stores the document key.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td>Allowed values</td>
<td>VARCHAR2</td>
</tr>
<tr>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td></td>
<td>RAW(16)</td>
</tr>
</tbody>
</table>

**JSON collection metadata document path**

keyColumn.sqlType

---

**Caution:**

If client-assigned keys are used and the key column type is VARCHAR2 then Oracle recommends that the database character set be AL32UTF8. This ensures that conversion of the keys to the database character set is lossless.

Otherwise, if client-assigned keys contain characters that are not supported in your database character set then conversion of the key into the database character set during a read or write operation is lossy. This can lead to duplicate-key errors during insert operations. More generally, it can lead to unpredictable results. For example, a read operation could return a value that is associated with a different key from the one you expect.
7.6 Key Column Max Length

The collection metadata component that specifies the maximum length of the key column in bytes. This component applies only to keys of type \texttt{VARCHAR2}.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>255</td>
</tr>
<tr>
<td>Allowed values</td>
<td>At least 32 bytes if key assignment method is \texttt{UUID} or \texttt{GUID}. See Key Column Assignment Method.</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>keyColumn.maxLength</td>
</tr>
</tbody>
</table>

Related Topics

- Key Column Type
  The collection metadata component that specifies the SQL data type of the column that stores the document key.

7.7 Key Column Assignment Method

The collection metadata component that specifies the method used to assign keys to objects that are inserted into the collection.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>UUID</td>
</tr>
<tr>
<td>Allowed values</td>
<td>UUID, GUID, SEQUENCE, CLIENT, EMBEDDED_OID</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>keyColumn.assignmentMethod</td>
</tr>
</tbody>
</table>

Table 7-1    Key Assignment Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{UUID} (default — but not used by Oracle Database API for MongoDB)</td>
<td>Keys are generated by SODA, based on the \texttt{UUID} (a Universal Unique Identifier).</td>
</tr>
<tr>
<td>\texttt{GUID}</td>
<td>Keys are generated in Oracle Database by SQL function \texttt{SYS_GUID}, described in \texttt{Oracle Database SQL Language Reference}.</td>
</tr>
<tr>
<td>\texttt{SEQUENCE}</td>
<td>Keys are generated in Oracle Database by a database sequence. If you specify the key assignment method as \texttt{SEQUENCE} then you must also specify the name of that sequence — see Key Column Sequence Name.</td>
</tr>
</tbody>
</table>
Table 7-1  (Cont.) Key Assignment Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIENT</td>
<td>Keys are assigned by the client application.</td>
</tr>
</tbody>
</table>

**Caution:**

If client-assigned keys are used and the key column type is `VARCHAR2` then Oracle recommends that the database character set be `AL32UTF8`. This ensures that conversion of the keys to the database character set is lossless. Otherwise, if client-assigned keys contain characters that are not supported in your database character set then conversion of the key into the database character set during a read or write operation is lossy. This can lead to duplicate-key errors during insert operations. More generally, it can lead to unpredictable results. For example, a read operation could return a value that is associated with a different key from the one you expect.

**EMBEDDED_OID**

(used always by Oracle Database API for MongoDB)

This method is used only, and automatically, with Oracle Database API for MongoDB, which is available only for an Oracle Autonomous Database. (You cannot create a collection that uses this method with any of the SODA implementations/languages.)

The column value is a string (data type `VARCHAR2`). The value is also embedded in the documents of the collection, as the value of top-level field `_id`.

- If a document already has a top-level field `_id`, then a string representation of that field value is used for the key column.
- If a document has no such field, then a field `_id` with a `generated` value is embedded in the document at top level, and that value is used for the key column. The value is a string hexadecimal representation of a 12-byte binary OID (ObjectID).

### 7.8 Key Column Path

The collection metadata component that specifies the top-level document path to the field used as an embedded document key.

The component value is top-level path `_id`, the embedded key for documents in a collection used by Oracle Database API for MongoDB.

This metadata component is required for key-column assignment method `EMBEDDED_OID`, and it is used only with that assignment method. It is provided automatically when `EMBEDDED_OID` is used; you need not specify it explicitly.

(If you do specify this component then an error is raised if `keyColumn.path` is specified for an assignment method other than `EMBEDDED_OID`, or if the value specified is not `_id`.)
7.9 Key Column Sequence Name

The collection metadata component that specifies the name of the database sequence that generates keys for documents that are inserted into a collection if the key assignment method is `SEQUENCE`.

If you specify the key assignment method as `SEQUENCE` then you must also specify the name of that sequence. If the specified sequence does not exist then SODA creates it.

**Note:**

If you drop a collection using SODA, the sequence used for key generation is not dropped. This is because it might not have been created using SODA. To drop the sequence, use SQL command `DROP SEQUENCE`, after first dropping the collection.

**Related Topics**

- **Key Column Assignment Method**
  The collection metadata component that specifies the method used to assign keys to objects that are inserted into the collection.

**See Also:**

- *Oracle Database SQL Language Reference* for information about `DROP SEQUENCE`
- *Oracle Database Concepts* for information about database sequences
### 7.10 Content Column Name

The collection metadata component that specifies the name of the column that stores the database content.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>JSON_DOCUMENT</td>
</tr>
<tr>
<td>Allowed values</td>
<td>Valid Oracle quoted identifier(^1) (as defined in Oracle Database SQL Language Reference). If this value contains double quotation marks (&quot;) or control characters, SODA replaces them with underscore characters (_).</td>
</tr>
</tbody>
</table>

**JSON collection metadata document path**

`contentColumn.name`

### 7.11 Content Column Type

The collection metadata component that specifies the SQL data type of the column that stores the document content.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
</table>
| Default value                   | • JSON, if database initialization parameter `compatible` is at least 20 and component `mediaTypeColumn.name` is not specified (the content is homogeneous — JSON data only).  
  • BLOB, otherwise.  
  If the content type is BLOB then the format is native JSON binary (OSON) if the database you use is an Oracle Autonomous Database (Autonomous JSON Database, Autonomous Transaction Processing or Autonomous Data Warehouse); otherwise, it is textual (unparsed Unicode character data.).  
  If the database you use is an Autonomous JSON Database (AJD) then component `mediaTypeColumn.name` cannot be specified — the content must be JSON data (homogeneous). |
| Allowed values                  | JSON (only if database initialization parameter `compatible` is at least 20)  
  VARCHAR2  
  BLOB  
  CLOB |

**JSON collection metadata document path**

`contentColumn.sqlType`
Note:
You need certain versions of SODA drivers and related software to support collection content type that uses Oracle's native JSON binary format, OSON, that is, for JSON type or for BLOB type with format OSON. See SODA Drivers

Related Topics
• Default Collection Metadata
  The kind of database you use determines the collection metadata that is used by default, and which of its field values you can modify for custom metadata.

7.12 Content Column Format

The collection metadata component that specifies the format of the column that stores the document content.

The value of this metadata component is automatically OSON — you cannot change it. This component is available only when the database is an Oracle Autonomous Database: Autonomous JSON Database (AJD), Autonomous Transaction Processing (ATP), or Autonomous Data Warehouse (ADW), and only when the value of metadata component content type (field contentColumn.sqlType) is BLOB.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allowed value</td>
<td>&quot;OSON&quot;</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>contentColumn.jsonFormat</td>
</tr>
</tbody>
</table>

Note:
You need certain versions of SODA drivers and related software to support collection content type that uses Oracle's native JSON binary format, OSON, that is, for JSON type or for BLOB type with format OSON. See SODA Drivers

Related Topics
• Default Collection Metadata
  The kind of database you use determines the collection metadata that is used by default, and which of its field values you can modify for custom metadata.
7.13 Content Column Max Length

The collection metadata component that specifies the maximum length of the content column in bytes. This component applies only to content of type VARCHAR2.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>4000</td>
</tr>
<tr>
<td>Allowed values</td>
<td>32767 if extended data types are enabled. Otherwise, 4000 if content column type is VARCHAR2.</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>contentColumn.maxLength</td>
</tr>
</tbody>
</table>

Related Topics

- Content Column Type
  The collection metadata component that specifies the SQL data type of the column that stores the document content.

See Also:

Oracle Database SQL Language Reference for information about extended data types

7.14 Content Column JSON Validation

The collection metadata component that specifies the syntax to which JavaScript Object Notation (JSON) content must conform—strict or lax.

Note:

If the content column stores JSON data using Oracle’s native binary format OSON — either JSON type or BLOB with format OSON, then this metadata component is absent.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>STANDARD</td>
</tr>
<tr>
<td>Allowed values</td>
<td>STANDARD, STRICT, LAX (default for SQL condition is json)</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>contentColumn.validation</td>
</tr>
</tbody>
</table>


• **STANDARD** validates according to the JSON RFC 8259 standard (or the RFC 4627 standard, if database initialization parameter compatible is less than 20). It corresponds to the strict syntax defined for Oracle SQL condition is json.²

• **STRICT** is the same as **STANDARD**, except that it also verifies that the document does not contain duplicate JSON field names. (It corresponds to the strict syntax defined for Oracle SQL condition is json when the SQL keywords WITH UNIQUE KEYS are also used.)

• **LAX** validates more loosely. (It corresponds to the lax syntax defined for Oracle SQL condition is json.) Some of the relaxations that LAX allows include the following:
  – It does not require JSON field names to be enclosed in double quotation marks (").
  – It allows uppercase, lowercase, and mixed case versions of true, false, and null.
  – Numerals can be represented in additional ways.

Related Topics

• **Default Collection Metadata**
  The kind of database you use determines the collection metadata that is used by default, and which of its field values you can modify for custom metadata.

---

See Also:

• *Oracle Database JSON Developer’s Guide* for information about strict and lax JSON syntax

• IETF RFC 8259 for the JSON RFC 8259 standard

• IETF RFC 4627 for the JSON RFC 4627 standard

---

### 7.15 Content Column SecureFiles LOB Compression

The collection metadata component that specifies the SecureFiles LOB compression setting.

---

**Note:**

If the content column stores JSON data using Oracle’s native binary format OSON — either JSON type or BLOB with format OSON, then this metadata component is absent.

---

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>NONE</td>
</tr>
<tr>
<td>Allowed values</td>
<td>NONE, HIGH, MEDIUM, LOW</td>
</tr>
</tbody>
</table>

² In database releases prior to 20c only IETF RFC 4627 was supported. It allows only a JSON object or array, not a scalar, at the top level of a JSON document. RFC 8259 support includes RFC 4627 support (and RFC 7159 support).
7.16 Content Column SecureFiles LOB Cache

The collection metadata component that specifies the SecureFiles LOB cache setting.

**Note:**

If the content column stores JSON data using Oracle's native binary format OSON — either JSON type or BLOB with format OSON, then this metadata component is *absent*.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>TRUE</td>
</tr>
<tr>
<td>Allowed values</td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>contentColumn.cache</td>
</tr>
</tbody>
</table>

Related Topics

- **Default Collection Metadata**
  The kind of database you use determines the collection metadata that is used by default, and which of its field values you can modify for custom metadata.

**See Also:**

*Oracle Database SecureFiles and Large Objects Developer's Guide* for information about SecureFiles LOB storage
7.17 Content Column SecureFiles LOB Encryption

The collection metadata component that specifies the SecureFiles LOB encryption setting.

**Note:**

If the content column stores JSON data using Oracle's native binary format OSON — either JSON type or BLOB with format OSON, then this metadata component is absent.

Before you create a collection that uses SecureFiles LOB encryption you must set up an encryption wallet.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>NONE</td>
</tr>
<tr>
<td>Allowed values</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NONE</td>
</tr>
<tr>
<td></td>
<td>3DES168</td>
</tr>
<tr>
<td></td>
<td>AES128</td>
</tr>
<tr>
<td></td>
<td>AES192</td>
</tr>
<tr>
<td></td>
<td>AES256</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>contentColumn.encrypt</td>
</tr>
</tbody>
</table>

**Related Topics**

- Default Collection Metadata
  The kind of database you use determines the collection metadata that is used by default, and which of its field values you can modify for custom metadata.

**See Also:**

- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about SecureFiles LOB storage
- *Oracle Database SQL Language Reference* for information about how to set up an encryption wallet using the `SET ENCRYPTION WALLET` clause of the `ALTER SYSTEM` statement

7.18 Version Column Name

The collection metadata component that specifies the name of the column that stores the document version.
7.19 Version Column Generation Method

The collection metadata component that specifies the method used to compute version values for objects when they are inserted into a collection or replaced.

Table 7-2 describes the version generation methods.

Table 7-2  Version Generation Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UUID</td>
<td>Ignoring object content, SODA generates a universally unique identifier (UUID) when the document is inserted and for every replace operation. Efficient, but the version changes even if the original and replacement documents have identical content. Version column type value is VARCHAR2(255).</td>
</tr>
</tbody>
</table>
Table 7-2  (Cont.) Version Generation Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP</td>
<td>Ignoring object content, SODA generates a value from the time stamp and converts it to LONG. This method might require a round trip to the database instance to get the time stamp. As with UUID, the version changes even if the original and replacement documents have identical content. Version column type value is NUMBER.</td>
</tr>
<tr>
<td>MD5</td>
<td>SODA uses the MD5 algorithm to compute a hash value of the document content. This method is less efficient than UUID, but the version changes only if the document content changes. Version column type value is VARCHAR2(255).</td>
</tr>
<tr>
<td>SHA256</td>
<td>SODA uses the SHA256 algorithm to compute a hash value of the document content. This method is less efficient than UUID, but the version changes only if the document content changes. Version column type value is VARCHAR2(255).</td>
</tr>
<tr>
<td>SEQUENTIAL</td>
<td>Ignoring object content, SODA assigns version 1 when the object is inserted and increments the version value every time the object is replaced. Version values are easily understood by human users, but the version changes even if the original and replacement documents have identical content. Version column type value is NUMBER.</td>
</tr>
<tr>
<td>NONE</td>
<td>If the version column is present, NONE means that the version is generated outside SODA (for example, by a database trigger).</td>
</tr>
</tbody>
</table>

7.20 Last-Modified Time Stamp Column Name

The collection metadata component that specifies the name of the column that stores the last-modified time stamp of the document.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>LAST_MODIFIED</td>
</tr>
<tr>
<td>Allowed values</td>
<td>Valid Oracle quoted identifier⁴ (as defined in Oracle Database SQL Language Reference). If this value contains double quotation marks (&quot;) or control characters, SODA replaces them with underscore characters (_).</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>lastModifiedColumn.name</td>
</tr>
</tbody>
</table>

7.21 Last-Modified Column Index Name

The collection metadata component that specifies the name of the index on the last-modified column.

The value of this component is the name of a nonunique index on the last-modified time-stamp column. The index is created if a name is specified. This index can improve the performance of read and write operations that are driven by last-modified time stamps.
Only SODA for REST provides such an operation (operation GET collection with time-stamp parameters since and until). Other implementations do not use this component, since they do not provide any read or write operations that are driven by last-modified time stamps. Even for SODA for REST, it is typically better not to set this component if you are sure that your application does not use any read or write operations that are driven by time stamps, because creating and maintaining an index carries a cost.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>None</td>
</tr>
<tr>
<td>Allowed values</td>
<td>Valid Oracle quoted identifier (as defined in Oracle Database SQL Language Reference). If this value contains double quotation marks (&quot;), or control characters, SODA replaces them with underscore characters (_).</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>lastModifiedColumn.index</td>
</tr>
</tbody>
</table>

See Also:
Oracle REST Data Services SODA for REST Developer's Guide

### 7.22 Creation Time Stamp Column Name

The collection metadata component that specifies the name of the column that stores the creation time stamp of the document. This time stamp is generated during the insert, insertAndGet, save, or saveAndGet operation.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>CREATED_ON</td>
</tr>
<tr>
<td>Allowed values</td>
<td>Valid Oracle quoted identifier (as defined in Oracle Database SQL Language Reference). If this value contains double quotation marks (&quot;), or control characters, SODA replaces them with underscore characters (_).</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>creationTimeColumn.name</td>
</tr>
</tbody>
</table>

### 7.23 Media Type Column Name

The collection metadata component that specifies the name of the column that stores the media type of the document. A media type column is needed if the collection is to
be heterogeneous, that is, it can store documents other than JavaScript Object Notation (JSON).

**Note:**

You cannot use query-by-example (QBE) with a heterogeneous collection. An error is raised if you try to do so.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>None</td>
</tr>
<tr>
<td>Allowed values</td>
<td>Valid Oracle quoted identifier(^1) (as defined in <em>Oracle Database SQL Language Reference</em>). If this value contains double quotation marks (&quot;) or control characters then SODA replaces them with underscore characters (_).</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>mediaTypeColumn.name</td>
</tr>
</tbody>
</table>

### 7.24 Read Only

The collection metadata component that specifies whether the collection is read-only.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default value</td>
<td>FALSE</td>
</tr>
<tr>
<td>Allowed values</td>
<td>TRUE</td>
</tr>
<tr>
<td>JSON collection metadata document path</td>
<td>readOnly</td>
</tr>
</tbody>
</table>
SODA Drivers

The drivers needed for different SODA implementations (languages) are described. The latest version of each driver is recommended, in all cases. Minimum required versions are described, for different SODA implementations with different Oracle databases.

Starting with Oracle Database 21c, the JSON content of SODA collections is, by default, stored as JSON data type, that is, the content column of a collection is JSON type.

To avoid compatibility problems of SODA drivers with database release 21c and above, Oracle recommends the following:

- Use the driver versions that are needed for working with JSON type, even if your database release is lower than 21c.
- For projects that were started using a database release prior to 21c, explicitly specify the metadata for the default collection, as specified in Example 8-1. For projects started using release 21c or later, just use the default metadata.

Table 8-1  SODA Driver Minimum Required Versions

<table>
<thead>
<tr>
<th>SODA Implementation (Language)</th>
<th>JSON Data Type Content</th>
<th>Content Other Than JSON Data Type</th>
</tr>
</thead>
</table>
| Java                           | • SODA for Java, version 1.1.7, available from Maven Central at these coordinates:  
  – Group ID: com.oracle.database.soda  
  – Artifact ID: orajsoda  
  – Version: 1.1.7.1  
  • Oracle JDBC driver: ojdbc8.jar for Oracle Database 21c, available from Maven Central or Oracle Database JDBC and UCP Downloads.  
  • javax.json-1.1.4.jar, available from Maven Central. | • SODA for Java, version 1.1.4, available from Maven Central at these coordinates:  
  – Group ID: com.oracle.database.soda  
  – Artifact ID: orajsoda  
  – Version: 1.1.4  
  Obtaining SODA for Java from Maven automatically picks up its dependencies (Oracle JDBC driver and javax.json-1.1.4.jar), as well. |
| REST                           | Oracle REST Data Services (ORDS), release 20.4.1 or later. | If your database is not an Oracle Autonomous Database then Oracle REST Data Services (ORDS), release 19.4.6 or later.  
  (If your database is an Oracle Autonomous Database then the required release of ORDS is preinstalled.) |
| C                              | Oracle Instant Client libraries must be 21c or later. Obtain them from Oracle Instant Client Downloads. | Oracle Instant Client libraries must be 19.6 or later. Obtain them from Oracle Instant Client Downloads. |
### Table 8-1 (Cont.) SODA Driver Minimum Required Versions

<table>
<thead>
<tr>
<th>SODA Implementation (Language)</th>
<th>JSON Data Type Content</th>
<th>Content Other Than JSON Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node.js</strong></td>
<td>Minimum recommended Node.js driver version is 4.0. Oracle Instant Client libraries must be 21c or later. Obtain them from Oracle Instant Client Downloads.</td>
<td>Minimum recommended Node.js driver version is 4.0. Oracle Instant Client libraries must be 19.6 or later. Obtain them from Oracle Instant Client Downloads.</td>
</tr>
<tr>
<td><strong>Python</strong></td>
<td>Minimum recommended Python driver version is 7.1. Oracle Instant Client libraries must be 21c or later. Obtain them from Oracle Instant Client Downloads.</td>
<td>Minimum recommended Python driver version is 7.1. Oracle Instant Client libraries must be 19.6 or later. Obtain them from Oracle Instant Client Downloads.</td>
</tr>
<tr>
<td><strong>Oracle SQLcl</strong> (SQL Developer Command Line)</td>
<td>SODA for SQLcl is <em>not yet available</em> for use with JSON data type content. Use Oracle SQLcl version 20.2 (<em>not</em> 20.3).</td>
<td></td>
</tr>
</tbody>
</table>

1 Content of JSON type requires database initialization parameter `compatible to be at least 20.`

### Example 8-1 Workaround To Use BLOB Content With Oracle Database 21c Or Later

If you have an on-premises Oracle Database that is release 21c or later, but you do not have the minimum required client versions for using JSON data type, you can use SODA with collections whose document content is BLOB data type instead. You can create such a collection by supplying this custom metadata when creating the collection:

```json
{ "keyColumn" : { "name":"ID" },
  "contentColumn" : { "name" : "JSON_DOCUMENT", "sqlType" : "BLOB" },
  "versionColumn" : { "name" : "VERSION", "method" : "UUID" },
  "lastModifiedColumn" : { "name" : "LAST_MODIFIED" },
  "creationTimeColumn" : { "name" : "CREATED_ON" } }
```

### Related Topics
- **SODA Feature Support**
  Features that are supported in different SODA implementations (languages) are listed, along with their required client versions and database releases.
SODA Feature Support

Features that are supported in different SODA implementations (languages) are listed, along with their required client versions and database releases.

Table 9-1 provides a general idea of the minimal database release and minimal driver version needed for a given feature. The feature names or descriptions are only roughly indicative. Refer to the documentation for a particular implementation for definitive and detailed information about any given feature.

Legend: the following abbreviations are used in the table.

- X — The feature is not available.
- NA — The feature is not applicable to the given implementation.

Table 9-1 Minimal Client and Server Versions for SODA Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Oracle Database Release</th>
<th>PL/SQL Driver</th>
<th>OCI C Driver¹</th>
<th>Node.js Driver²</th>
<th>Python Driver²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Collection Operations (Create, Open, List, Drop)</td>
<td>Java or REST: 12.2</td>
<td>18.3</td>
<td>v18: 18.5</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td>v19: 19.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metadata Cache</td>
<td>Java or REST: 12.2</td>
<td>NA</td>
<td>v19: 19.11</td>
<td>5.2</td>
<td>8.2</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td>v21: 21.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert Document, Replace Document, Remove Document</td>
<td>Java or REST: 12.2</td>
<td>18.5</td>
<td>v18: 18.5</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>PL/SQL: 18.5</td>
<td></td>
<td>v19: 19.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bulk Insert</td>
<td>Java or REST: 12.2</td>
<td>NA</td>
<td>19.11</td>
<td>4.0</td>
<td>7.2</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Save Document</td>
<td>Java or REST: 12.2</td>
<td>19.9</td>
<td>19.9</td>
<td>5.0</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>PL/SQL: 19.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Truncate Collection</td>
<td>Java or REST: 12.2</td>
<td>19.11</td>
<td>19.11</td>
<td>5.0</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>PL/SQL: 19.11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 9-1 (Cont.) Minimal Client and Server Versions for SODA Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Oracle Database Release</th>
<th>PL/SQL Driver</th>
<th>OCI C Driver&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Node.js Driver&lt;sup&gt;2&lt;/sup&gt;</th>
<th>Python Driver&lt;sup&gt;2&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>QBE Filter</td>
<td>Java or REST: 12.2</td>
<td>v18: 18.5</td>
<td>v18: 18.5</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>PL/SQL:</td>
<td>v19: 19.3</td>
<td>v19: 19.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v18: 18.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v19: 19.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pagination</td>
<td>Java or REST: 12.2</td>
<td>v18: 18.5</td>
<td>v18: 18.5</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>PL/SQL:</td>
<td>v19: 19.3</td>
<td>v19: 19.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v18: 18.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v19: 19.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimistic Locking</td>
<td>Java or REST: 12.2</td>
<td>v18: 18.5</td>
<td>v18: 18.5</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>PL/SQL:</td>
<td>v19: 19.3</td>
<td>v19: 19.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v18: 18.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v19: 19.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration Document Cursor</td>
<td>Java or REST: 12.2</td>
<td>v18: 18.5</td>
<td>v18: 18.5</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>PL/SQL:</td>
<td>v19: 19.3</td>
<td>v19: 19.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v18: 18.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v19: 19.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create Index (B-Tree, Search, Spatial) Drop Index</td>
<td>Java or REST: 12.2</td>
<td>v18: 18.5</td>
<td>v18: 18.5</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>PL/SQL:</td>
<td>v19: 19.3</td>
<td>v19: 19.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v18: 18.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v19: 19.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Guide Based on a Search Index</td>
<td>Java or REST: 12.2</td>
<td>v18: 18.5</td>
<td>v18: 18.5</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>PL/SQL:</td>
<td>v19: 19.3</td>
<td>v19: 19.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v18: 18.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· v19: 19.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OSON Native JSON Storage Format</td>
<td>Autonomous Database 19c</td>
<td>19.1</td>
<td>19.1</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td>JSON Data Type Storage</td>
<td>21.1</td>
<td>21.1</td>
<td>21.1</td>
<td>5.1</td>
<td>8.1</td>
</tr>
</tbody>
</table>
## Table 9-1  (Cont.) Minimal Client and Server Versions for SODA Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Oracle Database Release</th>
<th>PL/SQL Driver</th>
<th>OCI C Driver&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Node.js Driver&lt;sup&gt;2&lt;/sup&gt;</th>
<th>Python Driver&lt;sup&gt;2&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create View From Data Guide</td>
<td>Java or REST: 12.2</td>
<td>19.1</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>PL/SQL: 19.1</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Array Fetch Documents</td>
<td>Java or REST: 12.2</td>
<td>NA</td>
<td>19.9</td>
<td>5.0</td>
<td>8.0</td>
</tr>
<tr>
<td></td>
<td>PL/SQL: 19.1</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Online Redefinition of a Collection</td>
<td>21.1</td>
<td>21.1</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Pessimistic Locking</td>
<td>Java or REST: 12.2</td>
<td>19.11</td>
<td>19.11</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>PL/SQL: 19.11</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Flashback</td>
<td>Java or REST: 12.2</td>
<td>19.11</td>
<td>19.11</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>PL/SQL: 19.11</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Dynamic Data Guide</td>
<td>Java or REST: 12.2</td>
<td>19.11</td>
<td>v19: 19.11</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>PL/SQL: 19.11</td>
<td></td>
<td>v21: 21.3</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Hint (Monitoring)</td>
<td>Java or REST: 12.2</td>
<td>19.11</td>
<td>v19: 19.11</td>
<td>5.2</td>
<td>8.2</td>
</tr>
<tr>
<td></td>
<td>PL/SQL: 19.11</td>
<td></td>
<td>v21: 21.3</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Text Search of Non-JSON Documents</td>
<td>Java or REST: 12.2</td>
<td>v19: 19.13</td>
<td>v19: 19.13</td>
<td>3.0</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Sampling Documents</td>
<td>Java or REST: 12.2</td>
<td>v19: 19.13</td>
<td>v19: 19.13</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Others: 18.3</td>
<td></td>
<td></td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

<sup>1</sup> Oracle Call Interface

<sup>2</sup> Support of Node.js and Python for a feature also requires the relevant minimal OCI driver version for that feature.
Related Topics

- **SODA Drivers**
  The drivers needed for different SODA implementations (languages) are described. The latest version of each driver is recommended, in all cases. Minimum required versions are described, for different SODA implementations with different Oracle databases.

Related Topics

- **Overview of SODA Document Collections**
  A SODA collection is a set of documents that is backed by an Oracle Database table or view.
10
SODA Guidelines and Restrictions

General guidelines and restrictions that apply across SODA implementations are presented.

- **SODA Guidelines**
  Guidelines that apply across SODA implementations are described.

- **SODA Restrictions (Reference)**
  Restrictions that apply across SODA implementations are described.

10.1 SODA Guidelines

Guidelines that apply across SODA implementations are described.

- **AL32UTF8 database character set** — Oracle recommends¹ that you use AL32UTF8 (Unicode) for your database character set. Otherwise:
  - Data can be altered by SODA when documents are written to a collection, because of lossy conversion to the database character set. (This affects only collections stored as VARCHAR2 and CLOB data; collections stored as BLOB data do not depend on the database character set.
  - Query-by-example (QBE) can return unpredictable results.

- **Re-creating a collection²** — Do not drop a collection and then re-create it with different metadata if there is any application running that uses the collection in any way. Shut down any such applications before re-creating the collection, so that all live SODA objects are released.

  There is no problem just dropping a collection. Any read or write operation on a dropped collection raises an error. And there is no problem dropping a collection and then re-creating it with the same metadata. But if you re-create a collection with different metadata, and if there are any live applications using SODA objects, then there is a risk that a stale collection is accessed, and no error is raised in this case.

  **Note:**

  In SODA implementations that allow collection metadata caching, such as SODA for Java, this risk is increased if such caching is enabled. In that case, a cache can return an entry for a stale collection object even if the collection has been dropped.

¹ SODA for C requires that you use AL32UTF8 as the database character set.
² Day-to-day use of a typical application that makes use of SODA does not require that you drop and re-create collections. But if you need to do that for any reason then this guideline applies.
10.2 SODA Restrictions (Reference)

Restrictions that apply across SODA implementations are described.

- **Document size:**
  - For SODA for REST and SODA for Java the limit is approximately 2 gigabytes.
  - For SODA for PL/SQL the size is limited by the maximum possible LOB size.

- **JSON document content:**
  In SODA, JSON content must conform to the JSON RFC 8259 standard, if database initialization parameter `compatible` is at least 20, or to the RFC 4627 standard, if `compatible` is less than 20. RFC 8259 support includes RFC 4627 support (and RFC 7159 support).

  In Oracle Database release 19c and prior, only RFC 4627 was supported. RFC 4627 allows only a JSON object or an array, not a scalar, at the top level of a JSON document. For example, according to RFC 8259, the string value "hello" is, by itself, valid JSON content; but according to RFC 4627, it is not.

  In addition, SODA JSON content can be UTF-8 or UTF-16 (big endian (BE) or little endian (LE)). Although RFC 4627 also allows UTF-32 (BE and LE) encodings, SODA does not support them. Some implementations may support additional, non-Unicode, encodings.

- **See Also:**
  - IETF RFC 8259 for the JSON RFC 8259 standard
- IETF RFC 4627 for the JSON RFC 4627 standard
Index

Symbols

, character (comma), path syntax, 4-1
. character (period), path syntax, 4-1
[ and ] characters (brackets), path syntax, 4-1
* character (asterisk), path syntax, 4-1
` character (backquote), path syntax, 4-1
$ character (dollar sign)
    escaping in QBE path, 4-1
    in operator names, 5-1
$, prefix for QBE operator names, 2-1
$abs operator, 5-14
$all operator, 5-9
$and operator, 5-22
    omitting, 2-11, 5-22
    overview, 2-11
$between operator, 5-9
$boolean operator, 5-14
$ceiling operator, 5-14
$contains operator
    overview, 2-18
    reference, 5-30
$date operator, 5-14
$double operator, 5-14
$eq operator, 5-9
    omitting, 5-9
$exists operator, 5-9
$string operator, 5-14
$id operator, 5-25
    overview, 2-14
    used with $textContains, 5-26
$in operator, 5-9
$intersects operator, 5-28
$length operator, 5-14
$lower operator, 5-14
$lt operator, 5-9
$lte operator, 5-9
$ne operator, 5-9
$near operator, 5-28
$nin operator, 5-9
$nor operator, 5-22
    overview, 2-11
$not operator, 5-13

$not QBE operator
    overview, 2-9
$number operator, 5-14
$or operator, 5-22
    overview, 2-11
$orderby operator, 5-2
    overview, 2-15
$query operator, 5-2
$regex operator, 5-9
$size operator, 5-14
$startsWith operator, 5-9
$string operator, 5-14
$textContains operator
    overview, 2-19
    reference, 5-26
$timestamp operator, 5-14
$type operator, 5-14
$upper operator, 5-14
$within operator, 5-28

A
array index (position), 4-1
array step (QBE path), definition, 4-1
array, object satisfying multiple conditions, 2-13
asterisk character, path syntax, 4-1

B
B-tree index
    details, 6-1
    overview, 3-1
backquote character, path syntax, 4-1
bracket characters, path syntax, 4-1

C
collection, 1-5
    heterogeneous, 2-1
    definition, 1-5
    media type column name, 7-18
collection configuration, 7-1
collection metadata
collection configuration, 7-1
collection metadata (continued)
content column format, 7-11
content column JSON validation, 7-12
content column max length, 7-12
content column name, 7-10
content column SecureFiles LOB cache, 7-14
content column SecureFiles LOB compression, 7-13
content column SecureFiles LOB encryption, 7-15
content column type, 7-10
creation time stamp column name, 7-18
default, 7-3
default and custom, 1-1
key column assignment method, 7-7
key column max length, 7-7
key column name, 7-6
key column path, 7-8
key column sequence name, 7-9
key column type, 7-6
last-modified column index name, 7-17
last-modified time stamp column name, 7-17
media type column name, 7-18
read only, 7-19
schema, 7-5
table or view, 7-5
version column name, 7-15
version generation method, 7-16
collection table name, 1-10
collections, database view of, 1-11
comma character, path syntax, 4-1
collection table name, 1-10
collections, database view of, 1-11
comma character, path syntax, 4-1
collection table name, 1-10
collections, database view of, 1-11
comma character, path syntax, 4-1
collection table name, 1-10
collections, database view of, 1-11
comma character, path syntax, 4-1
G

guidelines, 10-1

H

heterogeneous collection, 2-1
  definition, 1-5
  media type column name, 7-18
hint
  SQL monitoring, 1-11

I

ID clause, 5-25
  implicit $and operator, 5-22
index (position), array, 4-1
index specification
details, 6-1
index specifications
  overview, 3-1
ISO 8601 formats, 5-20
item-method clause
definition, 5-14
item-method equality clause
definition, 5-14
item-method modifier clause
definition, 5-14
item-method QBE operators, 5-14
  $abs, 5-14
  $boolean, 5-14
  $ceiling, 5-14
  $date, 5-14
  $double, 5-14
  $floor, 5-14
  $length, 5-14
  $lower, 5-14
  $number, 5-14
  $size, 5-14
  $string, 5-14
  $timestamp, 5-14
  $type, 5-14
  $upper, 5-14
  overview, 2-9

J

JSON data format used for document content, 7-11
JSON data type used for document content, 7-10

K

key column assignment method collection
  metadata component, 7-7
key column max length collection metadata
  component, 7-7
key column name collection metadata
  component, 7-7
key column path collection metadata component, 7-6
key column sequence name collection metadata
  component, 7-9
key column type collection metadata component, 7-6
key, document, 1-1
  matching in QBE, 5-25

L

last-modified column index name collection
  metadata component, 7-17
last-modified time stamp column name collection
  metadata component, 7-17
lax field, spatial index specification
details, 6-1
  overview, 3-1
limitations, 10-2
logical combining clause
definition, 5-22
logical combining operator
definition, 5-22
logical combining operators
  overview, 2-11

M

media type column name collection metadata
  component, 7-18
metadata, collection, 1-1
modifiers
  See item-method QBE operators
MONITOR SQL hint, 1-11
monitoring performance, 1-11

N

nested conditions, 2-13
nested-condition clause, 5-24
not clause
definition, 5-13

O

object in array, satisfying multiple conditions, 2-13
omitting $and operator, 5-22
omitting $eq operator, 5-9
operand, for QBE operator
definition, 5-1
operand, QBE, 2-1, 5-1
operator
-abs, 5-14
all, 5-9
and, 5-22
omitting, 2-11
overview, 2-11
-between, 5-9
boolean, 5-14
ceiling, 5-14
contains
overview, 2-18
reference, 5-30
date, 5-14
double, 5-14
eq, 5-9
omitting, 5-9
exists, 5-9
floor, 5-14
gt, 5-9
gte, 5-9
id
overview, 2-14
reference, 5-25
in, 5-9
intersects, 5-28
length, 5-14
lower, 5-14
lte, 5-9
lt, 5-9
ne, 5-9
near, 5-28
nin, 5-9
nor, 5-22
overview, 2-11
not, 5-13
overview, 2-9
number, 5-14
or, 5-22
overview, 2-11
orderby, 5-2
overview, 2-15
regex, 5-2
startsWith, 5-9
size, 5-14
string, 5-14
textContains
overview, 2-19
reference, 5-26
timestamp, 5-14
operator (continued)
type, 5-14
upper, 5-14
within, 5-28
operator, QBE, 2-1, 5-1
orderby clause, 5-3
ordinary characters, definition, 4-1

P

path, QBE, 2-5, 4-1
period character, path syntax, 4-1

Q

QBE
definition, 5-1
operand, 2-1, 5-1
operator, 2-1, 5-1
QBE (query by example), 1-1
QBE (query-by-example), 2-1
QBE operators
all, 5-9
and, 5-22
between, 5-9
contains
overview, 2-18
reference, 5-30
eq, 5-9
exists, 5-9
gt, 5-9
gte, 5-9
id
overview, 2-14
reference, 5-25
in, 5-9
intersects, 5-28
lax
for orderby, 5-3
for QBE spatial clause, 5-28
lt, 5-9
lte, 5-9
ne, 5-9
near, 5-28
nin, 5-9
nor, 5-22
not, 5-13
overview, 2-9
or, 5-22
overview, 2-11
sortBy, 5-2
overview, 2-15
regex, 5-9
size, 5-14
startsWith, 5-9
string, 5-14
textContains
overview, 2-19
reference, 5-26
timestamp, 5-14
scalarRequired
for orderby, 5-3
QBE operators *(continued)*

$scalarRequired *(continued)*

- for QBE spatial clause, 5-28
- $startsWith, 5-9
- $textContains
  - overview, 2-19
  - reference, 5-26
- $within, 5-28
- comparison
  - overview, 2-7
  - item-method, 5-14
  - overview, 2-9
- spatial
  - overview, 2-16

QBE path, 2-5, 4-1
query by example (QBE), 1-1
query-by-example, 2-1
query-by-example (QBE)
  - definition, 5-1

R

- read and write operations
  - monitoring performance, 1-11
- read only collection metadata component, 7-19
- restrictions, 10-2

S

- sample JSON documents used in examples, 2-4
- scalar-equality clause
  - definition, 5-8
- scalarRequired field, index specification
  - details, 6-1
- scalarRequired field, index specifications
  - overview, 3-1
- schema collection metadata component, 7-5
- scoping fields to a parent field in QBE, 5-24
- search index
  - details, 6-1
  - overview, 3-1
- search, full-text
  - JSON, overview, 2-18
  - non-JSON, overview, 2-19
- SODA database, 1-1
- SODA guidelines, 10-1
- SODA implementations, 1-1
- SODA limitations, 10-2
- SODA operator
  - definition, 5-1
- SODA restrictions, 10-2
- sorting JSON data returned by a QBE, 5-3
- spatial clause *(continued)*
- spatial index
  - details, 6-1
  - overview, 3-1
- spatial operator
  - $intersects, 5-28
  - $near, 5-28
  - $within, 5-28
- spatial QBE operators
  - overview, 2-16
- specification
  - filter
    - details, 5-1
  - index
    - details, 6-1
- specifications
  - index
    - overview, 3-1
- SQL/JSON operators, 1-1
- square bracket characters, path syntax, 4-1
- syntactic characters, definition, 4-1
- syntax, QBE path, 4-1

T

- table name, collection, 1-10
- table or view collection metadata component, 7-5
- text-contains clause
  - definition, 5-26
- time formats, ISO 8601, 5-20

U

- USER_SODA_COLLECTIONS database view, 1-11

V

- version column name collection metadata component, 7-15
- version generation method collection metadata component, 7-16
- view of your SODA collections, 1-11

W

- wildcard character, path syntax, 4-1
- write and read operations
  - monitoring performance, 1-11

Y

- ym_iso_format ISO 8601 duration format, 5-20