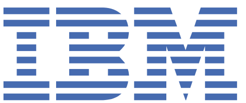
Db2 13 for z/OS

JSON Application Development
Last updated: 2024-06-06



Notes

Before using this information and the product it supports, be sure to read the general information under "Notices" at the end of this information.

Subsequent editions of this PDF will not be delivered in IBM Publications Center. Always download the latest edition from IBM Documentation.

2024-06-06 edition

This edition applies to Db2® 13 for z/OS® (product number 5698-DB2®), Db2 13 for z/OS Value Unit Edition (product number 5698-DBV), and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Specific changes are indicated by a vertical bar to the left of a change. A vertical bar to the left of a figure caption indicates that the figure has changed. Editorial changes that have no technical significance are not noted.

© Copyright International Business Machines Corporation 2014, 2024.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this information....................v
Who should read this information....................v
Db2 Utilities Suite for z/OS....................vi
Terminology and citations....................vi
Accessibility features for Db2 for z/OS....................vi
How to send comments....................vii
How to read syntax diagrams....................vii

Chapter 1. Key JSON concepts....................1
JSON namespaces....................2
JSON collections....................2
JSON documents....................2
JSON nested objects....................3

Chapter 2. JSON installation requirements....................5
NoSQL properties....................5
nosql.asyncMaxThreadCount property....................5
nosql.connectionPoolSize property....................5
nosql.traceFile property....................6
nosql.traceLevel property....................6

Chapter 3. Solution planning....................9
Connection management....................9
Query operators....................9
JSON logical operators....................9
JSON comparison operators....................10
Data evaluation operators....................11
Security model....................11
Wire listener authentication....................11
Performance features....................12
Lazy fetches....................12
JSON batching....................12
Fire and forget mode....................13

Chapter 4. Working with JSON documents with the Java API....................15
Java APIs....................15
Connect to a Db2 database....................15
Store JSON documents in a Db2 database....................16
Select JSON documents from a Db2 database....................17
Create indexes on JSON fields....................17
Import or export JSON data....................17
JSON command-line interface (CLI)....................18
Starting the command-line interface....................18
Command-line options....................19
Wire listener....................38
Configuring the wire listener....................38
wpListener -help - 39
wpListener -shutdown - 39
wpListener -start - 40

Chapter 5. Working with JSON documents by using SQL.................................... 43
JSON_VAL...43
JSON_LEN...46
JSON_TYPE...47
JSON_TABLE...48
JSON2BSON..49
BSON2JSON..50
Creating a column to store JSON data...50
Inserting JSON data into a table..51
Updating JSON data in a table...51
Retrieving data that is inside a JSON document..52
Example SQL statements with JSON2BSON, BSON2JSON, and JSON_VAL functions......................53

Information resources for Db2 for z/OS and related products...............................57

Notices...59
Programming interface information...60
Trademarks...60
Terms and conditions for product documentation..61
Privacy policy considerations...61

Glossary...63

Index...65

About this information

Throughout this information, "Db2" means "Db2 13 for z/OS". References to other Db2 products use complete names or specific abbreviations.

Important: To find the most up to date content for Db2 13 for z/OS, always use IBM® Documentation or download the latest PDF file from PDF format manuals for Db2 13 for z/OS (Db2 for z/OS in IBM Documentation).

Most documentation topics for Db2 13 for z/OS assume that the highest available function level is activated and that your applications are running with the highest available application compatibility level, with the following exceptions:

- The following documentation sections describe the Db2 13 migration process and how to activate new capabilities in function levels:
  - Migrating to Db2 13 (Db2 Installation and Migration)
  - What's new in Db2 13 (Db2 for z/OS What's New?)
  - Adopting new capabilities in Db2 13 continuous delivery (Db2 for z/OS What's New?)
- FL 500 A label like this one usually marks documentation changed for function level 500 or higher, with a link to the description of the function level that introduces the change in Db2 13. For more information, see How Db2 function levels are documented (Db2 for z/OS What's New?).

The availability of new function in Db2 13 depends on the type of enhancement, the activated function level, and the application compatibility levels of the applications. For a list of all available function levels in Db2 13, see Db2 13 function levels (Db2 for z/OS What's New?).

Function level 100

Db2 starts at function level 100 (V13R1M100) during migration to Db2 13, and fallback and coexistence with Db2 12 in data sharing remain possible. Many new capabilities in Db2 13 remain unavailable. For more information, see Function level 100 (for migrating to Db2 13 - May 2022) (Db2 for z/OS What's New?).

Function level 500

Activating function level 500 (V13R1M500) prevents coexistence with and fallback to Db2 12. Function level 500 is also the first opportunity for applications to use many of the new capabilities in Db2 13. However, new capabilities that depend on Db2 13 catalog changes remain unavailable. For more information, see Function level 500 (for migrating to Db2 13 - May 2022) (Db2 for z/OS What's New?).

Function level 501

Function level 501 (V13R1M501) is the first opportunity after migration to Db2 13 for applications to use new features and capabilities that depend on catalog changes in Db2 13. For more information, see Function level 501 (Db2 13 installation or migration - May 2022) (Db2 for z/OS What's New?).

Some virtual storage and optimization enhancements take effect in function level 100. Optimization enhancements become available after full prepare of the SQL statements, depending on the statement type:

- For static SQL statements, after bind or rebind of the package.
- For non-stabilized dynamic SQL statements, immediately, unless the statement is in the dynamic statement cache.
- For stabilized dynamic SQL statements, after invalidation, free, or changed application compatibility level.

Who should read this information

This information is for the following users:

- Db2 for z/OS application programmers with a knowledge of SQL and the JSON programming language.
- JSON application programmers with a knowledge of SQL and the JSON programming language.

Db2 Utilities Suite for z/OS

Important: Db2 Utilities Suite for z/OS is available as an optional product. You must separately order and purchase a license to such utilities, and discussion of those utility functions in this publication is not intended to otherwise imply that you have a license to them.

Db2 13 utilities can use the DFSORT program regardless of whether you purchased a license for DFSORT on your system. For more information about DFSORT, see https://www.ibm.com/support/pages/dfsort.

Db2 utilities can use IBM Db2 Sort for z/OS as an alternative to DFSORT for utility SORT and MERGE functions. Use of Db2 Sort for z/OS requires the purchase of a Db2 Sort for z/OS license. For more information about Db2 Sort for z/OS, see Db2 Sort for z/OS documentation.

Related concepts

Db2 utilities packaging (Db2 Utilities)

Terminology and citations

When referring to a Db2 product other than Db2 for z/OS, this information uses the product's full name to avoid ambiguity.

The following terms are used as indicated:

Db2
Represents either the Db2 licensed program or a particular Db2 subsystem.

IBM OMEGAMON® for Db2 Performance Expert on z/OS
Refers to any of the following products:

- IBM IBM OMEGAMON for Db2 Performance Expert on z/OS
- IBM Db2 Performance Monitor on z/OS
- IBM Db2 Performance Expert for Multiplatforms and Workgroups
- IBM Db2 Buffer Pool Analyzer for z/OS

C, C++, and C language
Represent the C or C++ programming language.

CICS®
Represents CICS Transaction Server for z/OS.

IMS
Represents the IMS Database Manager or IMS Transaction Manager.

MVS™
Represents the MVS element of the z/OS operating system, which is equivalent to the Base Control Program (BCP) component of the z/OS operating system.

RACF®
Represents the functions that are provided by the RACF component of the z/OS Security Server.

Accessibility features for Db2 for z/OS

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility features

The following list includes the major accessibility features in z/OS products, including Db2 for z/OS. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers and screen magnifiers.
- Customization of display attributes such as color, contrast, and font size

Tip: IBM Documentation (which includes information for Db2 for z/OS) and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard navigation

For information about navigating the Db2 for z/OS ISPF panels using TSO/E or ISPF, refer to the *z/OS TSO/E Primer*, the *z/OS TSO/E User's Guide*, and the *z/OS ISPF User's Guide*. These guides describe how to navigate each interface, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Related accessibility information

IBM and accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

How to send your comments about Db2 for z/OS documentation

Your feedback helps IBM to provide quality documentation.

Send any comments about Db2 for z/OS and related product documentation by email to db2zinfo@us.ibm.com.

To help us respond to your comment, include the following information in your email:

- The product name and version
- The address (URL) of the page, for comments about online documentation
- The book name and publication date, for comments about PDF manuals
- The topic or section title
- The specific text that you are commenting about and your comment

Related concepts
About Db2 13 for z/OS product documentation (Db2 for z/OS in IBM Documentation)
Related reference
PDF format manuals for Db2 13 for z/OS (Db2 for z/OS in IBM Documentation)

How to read syntax diagrams

Certain conventions apply to the syntax diagrams that are used in IBM documentation.

Apply the following rules when reading the syntax diagrams that are used in Db2 for z/OS documentation:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►─── symbol indicates the beginning of a statement.

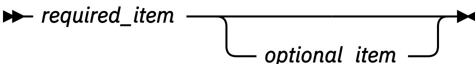
  The ───► symbol indicates that the statement syntax is continued on the next line.

  The ►─── symbol indicates that a statement is continued from the previous line.

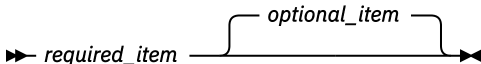
  The ───►◄ symbol indicates the end of a statement.
- Required items appear on the horizontal line (the main path).

  ►► *required_item* ►◄

- Optional items appear below the main path.

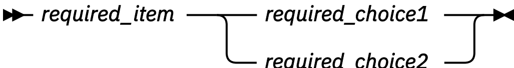


If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

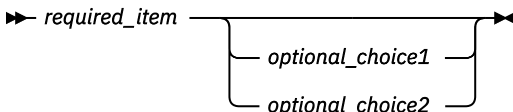


- If you can choose from two or more items, they appear vertically, in a stack.

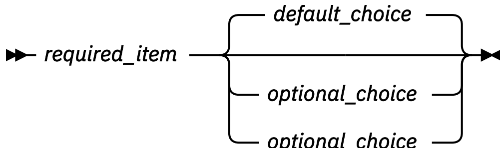
If you *must* choose one of the items, one item of the stack appears on the main path.



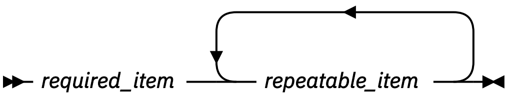
If choosing one of the items is optional, the entire stack appears below the main path.



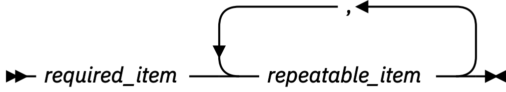
If one of the items is the default, it appears above the main path and the remaining choices are shown below.



- An arrow returning to the left, above the main line, indicates an item that can be repeated.

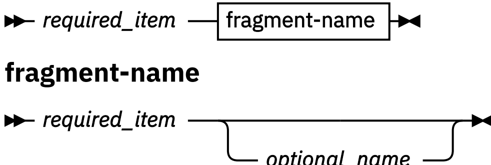


If the repeat arrow contains a comma, you must separate repeated items with a comma.



A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Sometimes a diagram must be split into fragments. The syntax fragment is shown separately from the main syntax diagram, but the contents of the fragment should be read as if they are on the main path of the diagram.



- For some references in syntax diagrams, you must follow any rules described in the description for that diagram, and also rules that are described in other syntax diagrams. For example:
  - For *expression*, you must also follow the rules described in Expressions (Db2 SQL).
  - For references to *fullselect*, you must also follow the rules described in fullselect (Db2 SQL).
  - For references to *search-condition*, you must also follow the rules described in Search conditions (Db2 SQL).
- With the exception of XPath keywords, keywords appear in uppercase (for example, `FROM`). Keywords must be spelled exactly as shown.
- XPath keywords are defined as lowercase names, and must be spelled exactly as shown.
- Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Related concepts

Commands in Db2 (Db2 Commands)
Db2 online utilities (Db2 Utilities)
Db2 stand-alone utilities (Db2 Utilities)

Chapter 1. Key JSON concepts

A JSON data store is a database that provides the capabilities to store, process, and manage data in JSON format. The JSON feature for Db2 enables a Db2 database to serve as a JSON data store.

JSON documents in the data store are stored in a binary format (extended BSON). A Java™ API supports a JSON-oriented query language that is derived from the popular MongoDB query language. For the JSON capability in Db2, this Java API is the backbone for extra application interfaces, such as the command-line interface and the wire listener.

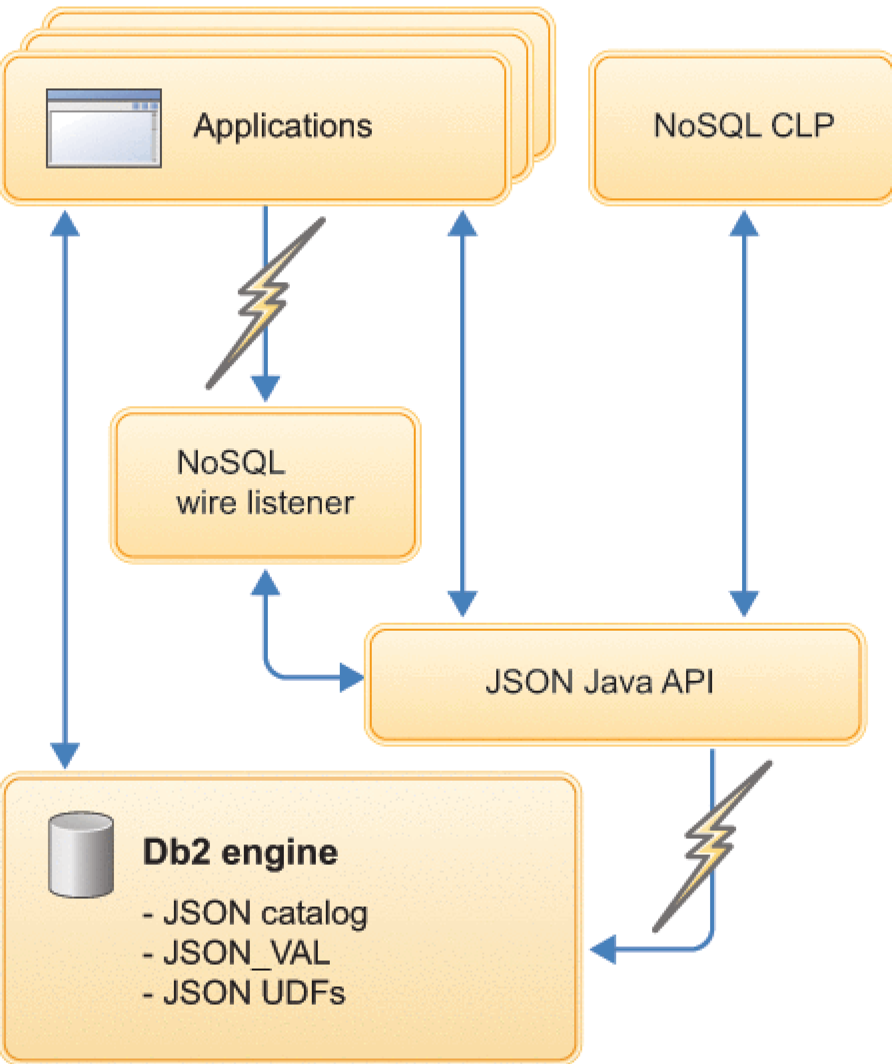


Figure 1. JSON components in Db2

In addition to supporting basic features of the MongoDB query language, the JSON capability in Db2 provides extensions that you can use to apply some Db2 database features to JSON documents, such as:

- Transaction control to group multiple operations into one commit scope
- Batch processing for multi-row insert operations

© Copyright IBM Corp. 2014, 2024

JSON namespaces

Each JSON data store can support multiple JSON namespaces. JSON namespaces are conceptually similar to a MongoDB database and are represented as an SQL schema. By default, the namespaces are not case-sensitive.

In the command-line interface, the namespace is set with the **use** command and referenced with the command prefix **db**.

The following code sample represents a JSON namespace:

```
nosql>use media
Switched to schema MEDIA
nosql>db.getCollectionNames()
[movies, books, audio]
```

JSON collections

A JSON collection is a named grouping of JSON documents. Document structures in a collection might differ significantly. However, usually documents in a collection are of a similar nature to enable finding and grouping data.

Collections are represented in a Db2 table with a custom-defined or automatically generated unique identifier and a binary large object to hold the semi-structured document content. A collection can be created explicitly with the **`createCollection()`** command. A collection is implicitly created when an insert is attempted for a collection that did not previously exist.

The document identifiers that serve as primary key must be of the same data type for all documents in the collection.

JSON documents

JSON documents consist of fields, which are name-value pair objects. The fields can be in any order, and be nested or arranged in arrays.

There is no enforcement of document structures. Therefore, other documents in the same collection might have a subset of fields, extra fields, or different representations of the same field.

The keys, that is, the field names are always String data and must be unique. The values can be any of the supported JSON data types.

Table 1. JSON data types

Data type	Example in JSON format
java.lang.String	"string"
java.lang.Integer	3
java.lang.Long	4294967296
java.lang.Double	6.2
java.lang.Byte []	true / false
java.util.Date (millisecond precision, in UTC)	{ "$binary": "(base64-encoded value)", "$type": "0" }
java.util.regex.Pattern	{ "$date" : "1998-03-11T17:14:12.456Z" }
java.util.regex.Pattern	{ "$regex" : "ab*" , "$options" : "" }
java.util.UUID	{ "$uuid" : "fb46f9a6-13df-41a2-a4e3-c77e85e747dd" }
com.ibm.nosql.bson.types.ObjectId	{ "$oid" : "51d2f200eefac17ea91d6831" }

Table 1. JSON data types (continued)

Data type	Example in JSON format
com.ibm.nosql.bson.types.Code	{ "$code" : "mycode" }
com.ibm.nosql.bson.types.CodeWScope	{ "$code" : "i=i+1", "$scope" : {} }
com.ibm.nosql.json.api.BasicDBObject	{ "a" : 1, "b": { "c" : 2 } }
com.ibm.nosql.json.api.BasicDBList	[1 , 2, "3", "abc", 5]

For `Date` string values, the client converts the value to UTC to be stored in Db2 databases. It is also retrieved as a `Date` in UTC format.

The following example represents a sample JSON document:

```
{
    name:"Joe",
    age:25,
    phone:["555-666-7777", "444-789-1234"],
    homeAddress:
        {street:"Sycamore Avenue",
        city: “Gilroy”,
        zipcode:"95046"
        }
    businessAddress:
        {
        street:"Bailey Avenue",
        City: “San Jose”,
        zipcode:"95141"
        }
}
```

JSON nested objects

JSON objects can be nested inside other JSON objects. Each nested object must have a unique access path.

The same field name can occur in nested objects in the same document. However, uniqueness must still apply for the full access name.

To access nested fields, concatenate the field names that are separated by a . (dot). For example, use `author.lastname` to access the surname for the author in this document:

```
{"isbn": "123-456-222",
 "author":
    {
      "lastname": "Doe",
      "firstname": "Jane"
    },
"editor":
    {
      "lastname": "Smith",
      "firstname": "Jane"
    },
  "title": "The Ultimate Database Study Guide",
  "category": ["Non-Fiction", "Technology"]
 }
```

In the example, the field name `lastname` occurs in the `author` and the `editor` object. The prefixes in `author.lastname` and `editor.lastname` provide unique access.

Chapter 2. JSON installation requirements

To enable support for programming with JSON in Db2, the server and client must be running specific software.

Table 2. JSON installation requirements

Installation requirements
• with all current maintenance applied • The feature of IBM Db2 Accessories Suite for z/OS, • IBM Data Server 10.5.0.3 from http://www.ibm.com/support/docview.wss?uid=swg24036705 (required only for the JSON Java API)

NoSQL properties

You can set properties that affect the behavior of the NoSQL client.

The properties are set in the `nosql.properties` file. The `nosql.properties` file is created manually and can be copied to any directory. The file directory must be included in the application classpath.

nosql.asyncMaxThreadCount property

Sets the maximum number of asynchronous threads. The threads are used to process `WriteConcern.NONE` and `WriteConcern.NORMAL` inserts, in which case inserts are queued, batched, and inserted with less frequent commits drastically increasing speed, but also increasing transaction log space requirements on the Db2 server.

If there are too many threads, each thread has less data to consolidate. If there are too few threads, they become a bottleneck in a system that can otherwise move more data.

The **`nosql.connPoolSize`** property limits the number of threads that you can use, because each thread needs a connection to work with. If there are not enough connections, some threads wait until one becomes available, and that decreases performance.

Default

`10`

Example

Set the asynchronous maximum thread count to 10:

```
nosql.asyncMaxThreadCount=10
```

nosql.connectionPoolSize property

Specifies the number of connections in the JDBC connection pool that is managed by the JSON API instance. If you use the **`NoSQLClient.getDB(URL, username , password)`** property, the API creates a JDBC connection pool and manages it for the database.

If the program needs another connection and no connections are available in the pool, the connection request must wait until a connection is returned to the pool. Keeping the value of the property too low might reduce concurrency. Setting the value too high might exceed the database connection limits and cause errors.

© Copyright IBM Corp. 2014, 2024

Default

20

Example

Set the connection pool size to 20:

```
nosql.connPoolSize=20
```

nosql.traceFile property

Specifies the name of a trace file that your program has permission to write to.

You can also configure NoSQL API tracing or logging by using the `java.util.logging` class. If you turn on logging in both an application and in the properties file, the log is written to both places. In general, logging has a performance cost.

Default

Null (no tracing)

Example

Set the name of the trace file to `/tmp/nosql.text`:

```
nosql.traceFile=/tmp/nosql.txt
```

nosql.traceLevel property

Sets the level of detail for the trace output. You must specify this property together with the **`nosql.traceFile`** property.

Values

Values are as follows. They are specified in order of lowest detail level to highest detail level.

OFF
: Specifies that logging is turned off.

SEVERE
: Specifies that only severe problems are logged. This level indicates a serious failure.

WARNING
: Specifies that potential problems are logged.

INFO
: Specifies that informational messages are logged. For the JSON API, if this level is chosen, most traceable messages are displayed. The following levels of granularity are supported:

  CONFIG
  : Specifies that static configuration are logged.

  FINE
  : Specifies that basic tracing messages are logged.

  FINER
  : Specifies that detailed tracing messages are logged.

  FINEST
  : Specifies that highly detailed tracing messages are logged.

  ALL
  : Indicates that all messages are logged.

Default

OFF

Example

Set the trace level to ALL:

```
nosql.traceLevel=ALL
```

Chapter 3. Solution planning

Information about connection management, query operators, JSON security, and performance features will help you plan your implementation of JSON.

Connection management

There are two options to connect to a JSON data store. The first option is through a single-mode connection, which is explicitly established with the provided connection information. The second option is through a shared connection pool.

By default, a connection pool is used. Operations such as `insert()` and `update()` attempt to obtain a connection from the pool, perform the operation with that connection, and return it to the pool when done. For details, see the information about the **`nosql.connectionPoolSize`** property.

For connections in single-mode, the JSON API allows control of transactional behavior when you are working with JSON documents. Therefore, it is possible to combine multiple operations in a transaction, control auto-commit behavior, and trigger a rollback if errors occur. The transaction APIs are not applicable for the fire and forget mode. For details, see the information about connecting to a Db2 database.

Query operators

Various query operators are supported, such as logical operators, comparison operators, and data evaluation operators.

JSON logical operators

The $and, $or, $not, and $nor logical operators are supported.

$and

The following example uses the $and operator:

```
{"$and":[{"age":5},{"name":"Joe"}]}
```

You can imply the $and operator by using a comma. The following version of the previous example uses this alternative syntax for the $and operator:

```
{"name":"Joe","age":5}
```

$not

The following example uses the $not operator:

```
{"$not":{"age":4}}
```

$or

The following example uses the $or operator:

```
{"$or":[{"age":4},{"name":"Joe"}]}
```

$nor

The following example uses the $nor operator:

```
{"$nor":[{"age":3},{"name":"Moe"}]}
```

© Copyright IBM Corp. 2014, 2024

JSON comparison operators

A number of different comparison operators are supported.

equality comparison operator

The following example uses the equality comparison operator:

```
{"name":"Joe"}
```

$ne (inequality comparator operator)

The following example uses the inequality comparator operator:

```
{"age":{"$ne":3}}
```

$in

The following example uses the $in operator to match at least one value in a set:

```
{"age":{"$in":[1,2,3,4,5]}}
```

You cannot use the $in operator, or the “in” operator in a index if the array contains different data types.

Do not use different types for the same field, as shown in the following incorrect example:

```
{"age":{"$in":[1,2,"A",3,4,5]}}
```

$nin

The following example uses the $nin operator to match to no values in a set, which is also referred to as the "not in" comparison operator:

```
{"age":{"$nin":[1,2,3,4]}}
```

$lt

The following example uses the $lt operator, which is also referred to as the "less than" comparison operator:

```
{"age":{"$lt":3}}
```

$lte

The following example uses the $lte operator, which is also referred to as the “less than or equals" operator:

```
{"age":{"$lte":3}}
```

$gt

The following example uses the $gt operator, which is also referred to as the “greater than" operator:

```
{"age":{"$gt":3}}
```

$gte

The following example uses the $gte operator, which is also referred to as the “greater than or equals" operator:

```
{"age":{"$gte":3}}
```

$regex (regular expression predicate)

The following example uses the regular expression predicate ($regex):

```
{"name":{"$regex":/^a.*\/}}
```

Data evaluation operators

JSON data evaluation operators provide ways to locate data within the database. They work in different manners for queries on individual documents and aggregation queries.

$size

The $size operator returns the number of documents if the array field is of the specified size. The following example uses the $size operator:

```
{"arr":{"$size":4}}
```

$mod

The $mod operator performs a modulo operation on the value of the field and returns the document that meets the specified result. The following examples use the $mod operator:

- The following query finds the value that has a remainder of 1 after it is divided by 4:

```
{"age":{"$mod":[4,1]}}
```

- The following query finds the value that has a remainder of 0 after it is divided by 5:

```
{"age":{"$mod":[5]}}
```

For details about supported data evaluation operators in queries that are constructed by using aggregation tasks, see the topic about aggregation tasks and operators.

Security model

The JSON security model is based on roles. To fulfill such a role, certain database privileges must be available to a user.

The three user roles are listed as shown:

- JSON administrator
  - This user role requires SYSCTRL or SYSADM authority to be set for the authorization ID.
- JSON collection manager
  - A JSON collection manager role requires authorizations for the CREATE TABLE, CREATE TRIGGER, and CREATE INDEX statements.
  - Might need authority to create new SQL schemas, if the JSON administrator has not already created the schemas.
  - Collection managers automatically have the document user role for their collections.
  - Collections are created with default access rights for database users.
- JSON document user
  - Can insert, update, and delete JSON documents.
  - Authorizations must be explicitly assigned by the JSON collection manager.
  - If implicit creation of documents is allowed, the document user must also have the collection manager role.

For applications that connect through the Java API, or for connections from the command-line interface, the authentication ID of the connected user is used to determine the access privileges. However, the wire listener uses proxy users.

Wire listener authentication

The wire listener uses an MD5-hash mechanism to verify that applications are a trusted source of messages.

The wire listener connects with a proxy user to the database. The proxy user must have DBADM privileges.

When the incoming message from a client application is authenticated by the wire listener, the listener submits the request to the database with a proxy user connection. The proxy user must have JSON collection manager and JSON document user roles for all JSON collections that should be accessible with this wire listener.

The application must authenticate users. It accepts user ID, password if applicable. Applications might accept unauthenticated users, in which case, the application must ensure that such users can only run approved queries.

The wire listener keeps a registration file on the host, which contains a list of registered applications and MD5-hash-tokens. These tokens might be per application, per application and schema, or just a single token per wire listener. This registration file is maintained by the JSON administrator by using a wire listener script. Access to the configuration list is controlled by operating system security, so that only those with access to the system and read or write access to the directory can work with the file.

The application sends messages with the user queries to the wire listener. The application must know the connection information (host, port) and have a valid token for the listener. The token is exchanged between the application manager and wire listener manager. The application id or token is then sent by the application by using the user ID and md5-hash mechanism in the message that is otherwise used for user ID or password.

The wire listener authenticates the application user ID only once, rather than for every message it receives from the application for a user. If the application switches the user ID then it is authenticated again by the wire listener.

The application and wire listener must be behind a firewall to prevent external snooping.

Remember: If a malicious user has access inside the firewall, they can snoop the message exchanges between the application and the wire listener, including the token, through some network sniffer, and can get access to the JSON data included in these messages.

Performance features

Various performance features are provided with the JSON capability for Db2, such as lazy fetching of queries, batching of JSON documents, and the fire and forget mode for insert operations.

Lazy fetches

Queries that use the `DBCollection.find()` method return a `DBCursor` object, which represents a forward-only cursor that iterates over results. To fetch blocks of results that load as you demand them, use the `DBCursor.lazyFetch()` method.By default, queries fetch results eagerly, or all at once.

The purpose of lazy fetching is memory optimization. You must specify the lazy fetch before opening the cursor. It is important to close the cursor after fetching the results. Otherwise, memory leaks might occur.

JSON batching

Batching refers to accumulating multiple JSON documents and then sending them together to the JSON store instead of sending each one separately. The JSON API provides a programmatic way to perform batching. The major advantage of batching is improved performance.

The following types of batching are supported:

Homogenous batching
: Occurs when the JSON documents in the batch are part of the same collection and the same operation is applied to all documents.

Heterogeneous batching
: Occurs when the JSON documents in the batch are part of different collections or different operations are applied to the documents.

To mark the start of a batch, use the `startBatch()` method. To trigger the insert, update, remove, or save operations for the documents, use the `endBatch()` method. If an operation in the batch fails, processing continues with the next operation in the batch.

Example 1:

In the following example, homogeneous batching is used to insert three documents in one collection:

```
DBCollection batch = db.getCollection("batch");

BasicDBObject dbObj1 = new BasicDBObject("name", "Joe1");
dbObj1.put("_id", 1);

BasicDBObject dbObj2 = new BasicDBObject("name", "Joe2");
dbObj2.put("_id", 2);

BasicDBObject dbObj3 = new BasicDBObject("name", "Joe3");
dbObj3.put("_id", 3);

db.startBatch();

batch.insert(dbObj1);
batch.insert(dbObj2);
batch.insert(dbObj3);

db.endBatch();
```

Example 2:

In the following example, heterogeneous batching is used to insert three documents in collection batch1 and two documents in collection batch2:

```
// Get collection batch1
DBCollection batch1 = db.getCollection("batch1");

// Create three documents to insert in collection batch1
BasicDBObject dbObj1 = new BasicDBObject("name", "Joe1");
dbObj1.put("_id", 1);

BasicDBObject dbObj2 = new BasicDBObject("name", "Joe2");
dbObj2.put("_id", 2);

BasicDBObject dbObj3 = new BasicDBObject("name", "Joe3");
dbObj3.put("_id", 3);

// Get collection batch2
DBCollection batch2 = _db.getCollection("batch2");

// Create two documents to insert in collection batch2
BasicDBObject dbObj4 = new BasicDBObject("name", "Joe4");
dbObj4.put("_id", 4);

BasicDBObject dbObj5 = new BasicDBObject("name", "Joe5");
dbObj5.put("_id", 5);

db.startBatch();

// Insert three documents into collection batch1

batch1.insert(dbObj1);
batch1.insert(dbObj2);
batch1.insert(dbObj3);

// Insert two documents into collection batch2

batch2.insert(dbObj4);
batch2.insert(dbObj5);

db.endBatch();
```

Fire and forget mode

Fire and forget mode enables multi-threaded, asynchronous inserts and can be set on the collection to enhance performance for inserts.

Fire and forget mode can only be activated when applications use a connection pool. If an application enables fire and forget for a single connection, the mode setting is ignored and the insert will be executed single-threaded.

The number of threads used for fire and forget is 10 by default. This value can be changed by setting the `asyncMaxThreadCount` in the `nosql.properties` file. For example, to set the number of threads to 100, use `nosql.asyncMaxThreadCount=100`.

To enable fire and forget mode, the collection must set the WriteConcern value to either `NONE` or `NORMAL`. Other WriteConcern values such as SAFE and `JOURNAL_SAFE` disable fire and forget mode because they guarantee writes to the database. See the Java documentation for more information on WriteConcern.

The downside to using this mode is that the data is not guaranteed to be written to the server. Moreover, the application will not see an exception raised if an error did occur during an insert. However, for application scenarios that can tolerate loss of data, the performance gain from using this mode can be significant.

Chapter 4. Working with JSON documents with the Java API

You can create, modify, or remove JSON documents that are stored in a Db2 database.

The NoSQL capability in Db2 enables JSON documents that are stored in a Db2 database server to be manipulated in the following three ways:

- The JSON capability for Java API provides a set of methods for storing, retrieving, and manipulating JSON documents. These methods can be called by Java applications directly through the API to work with the documents in the database. Because the Db2 database server is the data store, this component translates the operations that are requested in the method invocations into SQL statements.
- The JSON command-line interface (CLI) is a command shell for issuing administrative commands for JSON document collections, and for running queries and update operations against the JSON collections. The JSON CLI very similar to the Db2 command line processor.
- The JSON wire listener is a server application that intercepts the Mongo wire protocol. This wire listener acts as a mid-tier gateway server between MongoDB applications and Db2. It uses the NoSQL for JSON API to interface with the Db2 database server as the data store. You can run a MongoDB application that is written in application programming languages (such as Java, NodeJS, PHP, and Ruby), or you can use the MongoDB CLI to communicate with the Db2 server.

Java APIs

The Java API provides a set of methods for storing, retrieving, and manipulating JSON documents.

Connect to a Db2 database

The JSON API uses JDBC connections to perform various database operations, such as connecting to a Db2 database.

To use the database you must get an instance of a DB object. This object can be initialized with a `Connection` JDBC connection, or `DataSource`, or the database URL, user name, and password.

The following snippet from a Java program demonstrates how to connect to a Db2 database:

```
//Create a context and a dataSource
Context initialContext = new InitialContext();
DataSource dataSource = (DataSource)initialContext.lookup("jdbc/myDB2");
//The dataSource instance becomes a cache key for metadata for the target database
DB db = NoSQLClient.getDB(dataSource);
```

The following overloaded methods are available to get a DB instance:

- `NoSQLClient.getDB(java.sql.Connection)`
- `NoSQLClient.getDB(javax.sql.DataSource)`
- `NoSQLClient.getDB(String, String, String)`

  The string arguments for the third instance of the `getDB` method take the following arguments:

  jdbcUrl
  Specifies a URL that can be used to connect with a JDBC connection.

  user
  Specifies a user name that can be used to connect the database.

  password
  Specifies a password.

© Copyright IBM Corp. 2014, 2024

Note: This DB instance must not be shared between threads. Each thread must make its own `NoSQLClient.getDB()` method call.

Table 3. Methods on DB to control transactional behavior

Method	Description
`public void startTransaction()`	Gets a connection if needed, and sets auto-commit to false. If a connection pool or data source is being used, it puts the DB in single connection mode until a future `commitTransaction()` or `rollbackTransaction()`.
`public void commitTransaction()`	Commits a transaction started by `startTransaction()`.
`public void rollbackTransaction()`	Rollback a transaction that was started by `startTransaction()`.
`public void setAutoCommit(boolean autoCommit)`	Sets auto-commit if the Db2 server is using single connection mode. It is an error to call this method in `DataSource` mode.

Note: The transaction APIs described in this table are not applicable for the fire and forget mode.

Obtain a single-mode connection when you are using the transaction APIs to avoid situations where starting a transaction forcibly changes the connection pool mode into single-mode.

Documents are not inserted if an error occurs. With these APIs, either all or none of the documents are inserted.

Store JSON documents in a Db2 database

JSON documents are stored in collections (or tables) in the database. These documents are represented as a map of key value pairs where the keys are strings. The values can be primitive Java objects like String, Integer, Double, `java.util.Date`, as well as nested JSON documents represented by other `DBObject` instances.

JSON documents are represented by `DBObject` instances of the interface. A common implementation is by using `BasicDBObject` object. JSON arrays are represented by `BasicDBList` class. You can manually construct the objects, or you can use `BasicDBObjectBuilder` class for friendly syntax.

A collection is represented by `DBCollection` class. Before you can insert your documents, you must get a `DBCollection` from a DB instance.

The following snippet from a Java program demonstrates how to get a collection:

```
DBCollection empColl = db.getCollection("employees");
 BasicDBObject obj = new BasicDBObject();
 obj.put("name", "Joe");
 obj.put("age", 50);
 obj.put("salary", 60000);
 empColl.insert(obj);
```

By default, insert operations wait for success confirmation from the database. If you want to increase throughput at the expense of write safety, you can use `WriteConcern.NONE` or `WriteConcern.NORMAL`, in which case inserts are queued, batched, and inserted with less frequent commits drastically increasing speed, but also increasing transaction log space requirements on Db2 server side. The Db2 server is still providing atomicity, consistency, isolation, and durability (ACID). It is the API that is relaxing the rules for speed.

```
 DBCollection empColl = db.getCollection("employees");
 for(int i=0; i<100; i++){
     //Queues the object to be inserted
   empColl.insert(dbObject[i], com.ibm.nosql.json.api.WriteConcern.NORMAL);
```

```
  }
//Waits until all work sent from this thread on this db instance is processed
db.waitQueue();
```

Select JSON documents from a Db2 database

You can use one of several overloaded `DBCollection.find()` methods to select documents.

The following snippet from a Java program demonstrates how to select JSON documents:

```
DBCollection empColl = db.getCollection("employees");
 //Looking for employees that are named 'Joe'
 DBCursor cursor = empColl.find(new BasicDBObject("name", "Joe"));
 try{
   while(cursor.hasNext()){
     DBObject obj = cursor.next();
     doSomething(obj);
   }
 }finally{
  //Close the cursor no matter what
   cursor.close();
 }
```

Here are some example of the overloaded `DBCollection.find()` method:

```
DBCollection.find(com.ibm.nosql.json.api.DBObject query)
DBCollection.find(com.ibm.nosql.json.api.DBObject query, DBObject fields)
DBCollection.find(com.ibm.nosql.json.api.DBObject, com.ibm.nosql.json.api.DBObject, int, int)
DBCollection.findOne(com.ibm.nosql.json.api.DBObject)
```

To obtain specific the query output, consider using the following methods:

```
DBCursor.sort(com.ibm.nosql.json.api.DBObject)
DBCursor.limit(int)
DBCursor.skip(int)
```

Create indexes on JSON fields

You can create indexes on JSON data by using the `DBCollection.ensureIndex(DBObject)` method.

The following snippet from a Java program demonstrates how to create indexes on JSON fields:

```
 //Create an ascending integer index on 'age'
 db.collection.ensureIndex({age:[1, "$int"]});
 //Create ascending varchar(50) (default type) index
 //on 'manager.name' nested object field
 db.collection.ensureIndex({"manage.name":1}});
```

The following snippet from a Java program demonstrates how to create a compound (composite) index that contains multiple fields:

```
 //Create compound index with two fields: name ascending
 //with type varchar(20), and age descending as integer
 db.collection.ensureIndex({name:[1, "$string", 20], age:[-1, "$int"]});
```

For details, see the `DBCollection.ensureIndex(com.ibm.nosql.json.api.DBObject)` method for more details on index creation.

Import or export JSON data

You can use the `importFile()` and `exportFile()` Java APIs to import or export JSON data from a Db2 database.

You can use the `com.ibm.nosql.json.api.DBCollection#importFile(String)` method to import files with the `*.js` extension. A second integer parameter of this method indicates the commit frequency.

The following snippet from a Java program demonstrates how to import JSON data:

```
//Import data and commit after every 100 rows
db.collection.importFile("/temp/myjsondata.js", 100)
```

You can use the `DBCollection.exportFile(String)` method to export files with the `*.js` extension.

The following snippet from a Java program demonstrates how to export JSON data:

```
//Export data
 db.collection.exportFile("/temp/myjsondata.js")
```

Each of the `*.js` import or export files contains one JSON object on each line of the file in plain text format.

You can also use the JSON Java API to import files, where the first row of the file identifies field names. To import a csv file, the name must end with `.csv` instead of `.js`. The file extension provides input information to the API about the file format.

JSON command-line interface (CLI)

Use the command-line interface (CLI) to interact dynamically with the JSON collections and harness the capability of NoSQL. The CLI requires a Java runtime environment (JRE) of Version 1.5 or later.

There are three ways to interact with the command line:

- Start an interactive shell by specifying the URL, user ID, and password.

  The following is a sample command to start the interactive shell:

  ```
  java -cp nosqljson.jar;db2jcc4.jar;db2jcc_license_cisuz.jar;
  js.jar com.ibm.nosql.json.cmd.NoSqlCmdLine
  --url jdbc:db2://localhost:50000/jsondb --user tonysun --password ****
  ```

- Run a `.js` file. The command line runs the `.js` file instead of through a shell. The application ends after all the commands are completed.

  The following is a sample command to run a JavaScript file:

  ```
  java -cp nosqljson.jar;db2jcc4.jar;db2jcc_license_cisuz.jar;
  js.jar com.ibm.nosql.json.cmd.NoSqlCmdLine
  --url jdbc:db2://localhost:50000/jsondb; --user tonysun
  --password **** --file "file.js"
  ```

- Evaluate a command, by using the **eval** command. The command line runs a one line command, and then the applications ends.

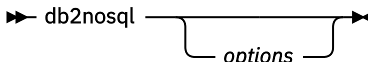
  The following is a sample command:

  ```
  java -cp nosqljson.jar;db2jcc4.jar;db2jcc_license_cisuz.jar;
  js.jar com.ibm.nosql.json.cmd.NoSqlCmdLine
  --url jdbc:db2://localhost:50000/jsondb; --user tonysun
  --password **** --eval "db.things.find()"
  ```

Starting the command-line interface

To start the JSON command-line interface (CLI), run the **db2nosql** script. The script uses the specified connection information to establish a connection to the Db2 database that is used as the JSON data store. If you do not specify connection parameters, the script prompts you for the required values.

Command syntax



Command parameters

--url *url*
Specifies a URL for connecting to the database.

--user *user_ID*
Specifies a user ID for connecting to the database.

--password *password*
Specifies a password for connecting to the database. JRE version 1.6 masks a typed password. However, JRE version 1.5 does not.

--file *file_name*
Specifies a JavaScript file that contains a list of one-line JavaScript commands that are used as input on the command-line.

--enable [`true` | `false`]
Specifies whether JSON artifacts are to be created. If the value is `true`, the **`--enable`** parameter creates all the necessary artifacts for JSON in the database. If the value is `false`, the SQL statements that create JSON artifacts are shown as output but are not issued.

--disable [`true` | `false`]
Specifies whether JSON artifacts are to be deleted. If the value is `true`, the **`--disable`** parameter deletes the artifacts that were created for JSON in the database. If the value is `false`, the SQL statements that delete JSON artifacts are shown as output but are not issued.

--schema *schema_name*
Specifies the optional schema name that represents the JSON namespace. If you specify this parameter, the schema name is used as the qualifier for the collection name. The JSON namespace is conceptually similar to a MongoDB database.

quiet
Limits the verbosity of the output.

Usage notes

Consider the following usage notes:

- The CLI requires a JRE of Version 1.5 or later.
- Using the **`eval`** and **`file`** commands together is not supported.

Command-line options

The following command-line options are supported.

aggregate() - Retrieve JSON documents command

Runs a sequence of tasks to retrieve documents from a collection, including attributes and calculated values. Tasks can occur multiple times as long as the syntax sequence is valid.

Syntax diagram

```
►► db — . — collection — . — aggregate — ( — task ►◄
```

Command parameters

task
This parameter specifies one or more of the following options:

`$distinct`
Retrieves distinct values for specific fields.

`$group`
Sets a grouping key and collects grouped values.

$limit
: Returns at most the specified number of documents from the result set.

$match
: Filters documents.

$project
: Selects fields for the document to retrieve. You can specify fields to include, not exclude.

$skip
: Skips the specified number of documents.

$sort
: Sorts the data on the specified fields.

Example

Calculate the average price of books per author in a category.

```
db.books.aggregate(
      {$match:   { category: "Fantasy" }},
      {$project: {author:1, price:1}},
      {$group:   {_id: {author:1}, avgPrice:{$avg:"$price"}}}
   )
```

Sample output it as follows:

```
Row 1:
 {
"_id":"Tolkien, J.R",
"avgPrice":15.5
 }
Row 2:
 {
"_id":"Verne, Jules",
"avgPrice":6.35
 }
2 rows returned
```

count() - Find the number of documents command

Finds the number of documents in a collection.

Syntax diagram

```
►► db ── . ── collection ── . ── count ── ( ── query ── ) ►◄
```

Command parameters

query
: This optional parameter specifies a filter for selecting a subset of documents and returns a count.

Example

Example 1: Get the number of documents in the books collection:

```
db.books.count()
```

Sample output is as follows:

```
15
```

Example 2: Get the number of documents for an author:

```
db.books.count({author: "Tolkien, J.R"})
```

Sample output is as follows:

```
2
```

createCollection() - Creates a collection

Creates a new collection with the specified characteristics.

Syntax diagram

```
>>- db — . — collection — . — createCollection — ( — name — , —┬───────────┬— ) -><
                                                                └─tableSpec─┘
```

Command arguments

name
Specifies a collection name. It must be alphanumeric with no special characters except $.

tableSpec
This optional parameter specifies the selection criteria for indexes. The options are as follows:

_id
Specifies ID definition. It must be one of the following data types:

- $int
- $long
- $number
- $date
- $timestamp
- $string: length
- $binary: length

compress (YES | NO) *YES* | *NO*)
Specifies whether to compress the collection.

tablespace
Specifies the name of the table space.

inline
Specifies the bufferpool space usage. It is of type integer.

systemTime (*sysStartField* | *sysEndField*)
Specifies the system time for *sysStartField* and *sysEndField* variables.

businessTime (*busStartField* | *busEndField*)
Specifies the business time for *busStartField* and *busEndField* variables.

Example

Example 1: Create a collection called "media" with _id of type SQL BIGINT.

```
db.createCollection(“media”, {_id: “$long”)})
```

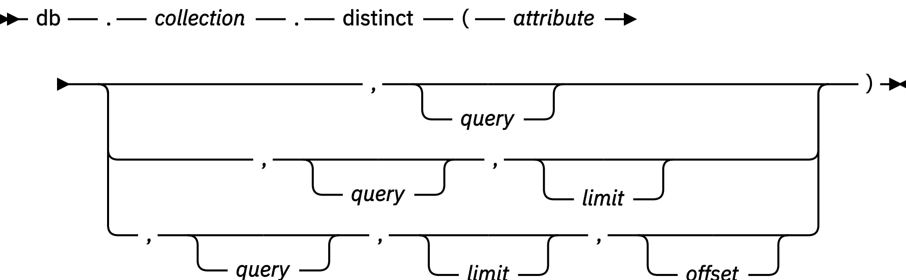
Sample output:

```
Collection: TEST."media" created. Use db.media.
```

distinct() - Find distinct values command

Finds distinct values for the submitted query.

Syntax diagram



Command parameters

attribute
: This string parameter specifies the field for which to find distinct values. It can be a string data type or an object.

query
: This optional string parameter specifies a filter for selecting a subset of documents.

limit
: This optional string parameter returns at most the specified number of documents from the result set.

offset
: This optional string parameter finds the document that the query requested after skipping a specified number of rows.

Example

Example 1: Issue an SQL query:

```
db.books.distinct("author")
```

Sample output is as follows:

```
[christie, granger, marsh]
```

Example 2: Get distinct values for the state and city, with no limit on the results, but skip the first three rows:

```
db.books.distinct({"state":1, "city":1}, {}, 0, 3)
```

Sample output is as follows:

```
Row 1:
{ state: AZ,
city: AB
}
Row 2:
(...)
```

drop() - Drop a collection command

Drops a collection and associated objects, such as indexes on the collection.

Syntax diagram

▶▶─ db ─ . ─ *collection* ─ . ─ drop ─ (─) ─▶◀

Command parameters

None

Example

Drop the books collection:

```
db.books.drop()
```

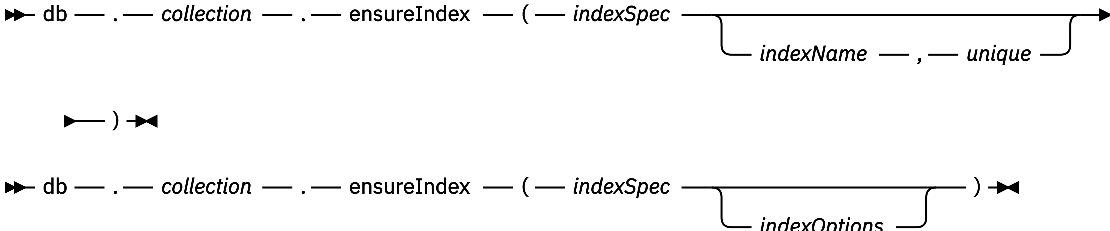
Sample output is as follows:

```
OK
```

ensureIndex() - Creates a new index

Creates a new index on a specific field of a collection.

Syntax diagram



Command arguments

indexSpec
: This argument specifies a field in the document on which the index is to be created. It might include a data type definition

indexName
: This argument specifies a name for an index. A name is generated if it is not specified.

unique (*true* | *false*)
: This optional argument specifies whether the index is unique.

indexOptions
: This optional argument can have one of the following options:

  name
  : This argument specifies a name of an index. A name is generated if not specified.

  array (*false*)
  : This argument specifies an array of elements. Db2 for z/OS does not support a JSON index on an array or an array element.

  unique (*true* | *false*)
  : This argument specifies whether the index is unique.

Example

Example 1: Create an index on field 'author' in ascending order, by using the default type string with default length 50.

```
db.books.ensureIndex({"author": 1})
```

Sample output:

```
Index <books_xauthor> was created successfully.
```

Example 2: Create an index on field 'category' with type string and field length 40.

```
db.books.ensureIndex({"category": [1, "$string", 40]})
```

Sample output:

```
Index <books_xcategory> was created successfully.
```

Example 3: Create an index on field price with type number in descending order, name it mypriceidx.

```
db.books.ensureIndex({"price": [-1, "$number"]}, “mypriceidx”)
```

Sample output:

```
Index <mypriceidx> was created successfully.
```

exportFile() - Export JSON documents to a file command

Exports JSON documents from a collection into a file.

Syntax diagram

▶▶ db — . — *collection* — . — exportFile — (— *file_name* —) ▶◀

Command parameters

file_name
This parameter specifies the fully qualified name of a file from which to export data.

Example

Export the books collection from the `books.js` file:

```
db.books.exportFile("C:\\books.js")
```

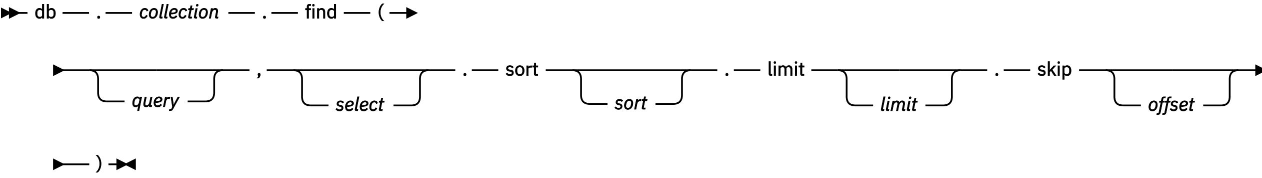
Sample output is as follows:

```
28 objects were exported in 200 milliseconds.
```

find() - Find JSON document command

Find JSON documents according to filter criteria. You can also specify fields to be included or excluded, determine the sort order, limit the number of documents, and page through results with an offset.

Syntax diagram



Command parameters

query
: This optional parameter specifies a filter for selecting a subset of documents.

select
: This optional parameter specifies fields to include or exclude in the query.

sort *sort*
: This optional parameter specifies a JSON object that indicates the sort criteria:

1
: Sort by ascending order.

-1
: Sort by descending order.

This parameter can include the data type, including $int, $long, $number, $date, $timestamp, $string: length, and $binary: length.

limit *limit*
: This optional parameter specifies a JSON object that restricts the number of documents.

offset *offset*
: This optional parameter specifies a JSON object that indicates the number of rows to skip.

Example

Find books in a collection. The collection name is case sensitive.

```
db.books.find()
```

Sample output is as follows:

```
Row 1:
{
"_id":{"$oid":"519b8727cd1552ed65b47a20"},
"isbn":"123-456-789",
"author":"Verne, Jules",
"title":"Journey to the Center of the Earth",
"abstract":"Classic science fiction novel in an unusual setting",
"price":6,
"pages":276,
"category":"Fantasy",
"sales": 500.5
}
```

findAndModify() - Find and update documents command

Finds documents and updates them with new values.

Syntax diagram

►► db — . — *collection* — . — findAndModify — (— *specification* —) ►◄

Command parameters

specification

This parameter specifies a JSON object with the following optional content:

query *document*

This parameter specifies a query object for filtering documents.

fields *document*

This parameter specifies a query object for selecting fields to return.

sort *document*

This parameter specifies a query object that defines the sort order to apply before selecting the first document.

remove `true`

This parameter specifies a query object that removes the document. When it is true, the selected document is removed. The default is false.

update *document*

This parameter specifies a query object that contains the update.

new (`true` | `false`)

If the value is `true`, this parameter returns the updated document instead of the original. If the value is `false`, the old document is returned (discarded except for use by time travel functionality).

insert (`true` | `false`)

If the value is `true`, this parameter inserts a new document, if the query does not return a document. If the value is `false`, this parameter updates the document or returns an error if the document is not present.

fullResponse (`true` | `false`)

If the value is `true`, the parameter returns the document or documents that are the result of the query. Also, a status object is returned that contains the write result and the last error. If the value is `false`, the parameter returns only the document.

Example

Find the first document with the name Joe and update the age field with the value 6:

```
db.mycollection.findAndModify({query : {name : "Joe"}, update : {$set : {age : 6}}})
```

Sample output is as follows:

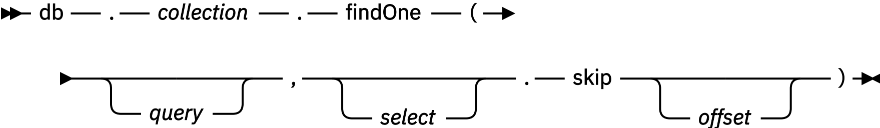
```
{
"lastErrorObject":
{
"updatedExisting":true,
"n":1,
"err": null ,
"ok":1
},
"value":
{
"_id":{"$oid":"51cdc007927a75e5a8c327e3"},
"name":"Joe",
"age":5
```

```
}
}
```

findOne() - Find the first offset document

Find the first document that matches the query after you skip a specified number of rows.

Syntax diagram



Command arguments

query
This optional argument specifies a filter to select a subset of documents.

select
This optional argument specifies a selection of fields to include or exclude in the query.

offset *offset*
This optional argument specifies a JSON object to skip some rows.

Example

Example 1: Use the `db.collection.findOne()` method to return a single record.

```
db.books.findOne({author: "Tolkien, J.R"}, {title: 1, price:1})
```

Sample output:

```
{
  "_id":{"$oid":"51f94b6a6bca2ee58280ef28"},
  "title":"The Hobbit",
  "price":5.0
  }
```

getCollectionNames() - View collection names for a schema command

Returns the names of all JSON collections in a Db2 database.

Syntax diagram

►► db — . — getCollectionNames — (—) ►◄

Command parameters

None.

Example

View all the collection names:

```
db.getCollectionNames()
```

Sample output is as follows:

```
[media, books, booksnest]
```

getIndexes() command - View JSON indexes for a collection command

Returns index information that identifies and describes the existing indexes for a collection.

Syntax diagram

db — . — *collection* — . — getIndexes — (— *filter* —)

Command parameters

filter
This optional parameter specifies the selection criteria for indexes. The options are as follows:

ID
Get details for the index with the specified identifier.

key
Get details for indexes with the specified attribute.

name
Get details for the index with the specified name.

Example

View a list of all indexes and type markers in the specified namespace:

```
db.books.getIndexes()
```

Sample output is as follows:

```
[{"v":0,"_id":57,"key":{"_id_":1},"ns":"TEST.books","name":"_id_","unique":true},
{"v":0,"_id":63,"key":{"author":1},"ns":"TEST.books","name":"books_xauthor","unique":false}]
```

Example 2: View a list of all indexes and type markers in the specified namespace:

```
db.books.getIndexes({"name": "_id_"})
```

Sample output is as follows:

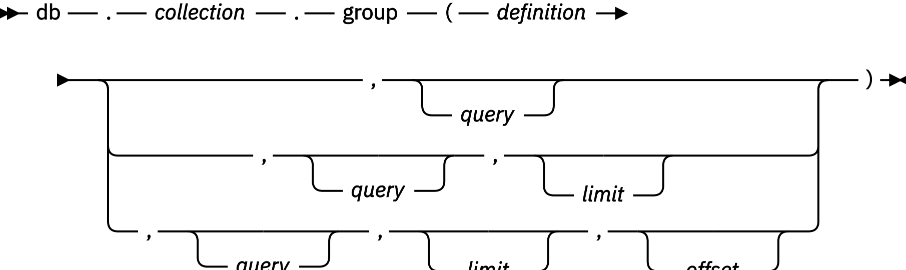
```
[{"v":0,"_id":57,"key":{"_id_":1},"ns":"TEST.books","name":"_id_","unique":true}]
```

group() - Group values JSON command

Groups values for the collection in a query. More command options are available by using the **aggregate()** command with the $group task.

Syntax diagram

db — . — *collection* — . — group — (— *definition* —



Command parameters

definition
This parameter specifies the fields that must be grouped.

query
This parameter specifies a filter for selecting a subset of documents.

limit
This parameter specifies an integer value that indicates the maximum size of the result set.

offset
This parameter finds the document that the query requested after skipping a specified number of rows.

Example

Calculate the average price of books per author:

```
db.books.group({"_id": {"author": 1}, "sumqty": {"$sum": "$qty"}})
```

Sample output is as follows:

```
Row 1:
{
"_id":"Doe",
"sumqty":17481
}
Row 2:
{
"_id":"Smith",
"sumqty":5926
}
```

help() - Lists help commands

Lists possible commands for the collection commands.

Command parameters

Lists commands that can be applied for a collection.

importFile() - Import JSON documents from file command

Imports JSON documents from a file into a collection.

Syntax diagram

```
>>- db -- . -- collection -- . -- importFile -- ( -- file_name -- , --+-------------------+-- ) -><
                                                                      '- commit_frequency -'
```

Command parameters

file_name
This parameter specifies the fully qualified name of a file from which to import data.

commit_frequency
This optional parameter specifies the batch size for commit operations. Larger batches generally improve performance.

Example

Import documents from a file that is named books.

```
db.books.importFile("C:\\books.js")
```

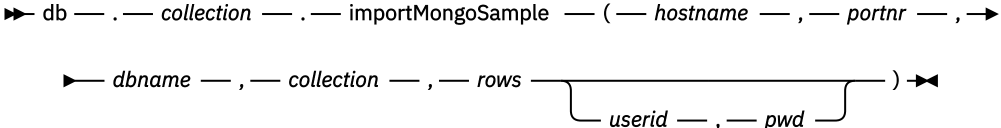
Sample output is as follows:

```
28 objects were imported in 4030 milliseconds.
```

importMongoSample() - Import some JSON documents from MongoDB

Import a specified number of JSON documents from a collection in a MongoDB database.

Syntax diagram



Command parameters

hostname
This argument specifies the name of the server that hosts the MongoDB database.

portnr
This argument specifies the port number of the MongoDB database.

dbname
This argument specifies the database name of the MongoDB database.

collection
This argument specifies the collection name in the MongoDB database.

rows
This argument specifies the - number of documents to import. If it is set to 0, then it imports all documents.

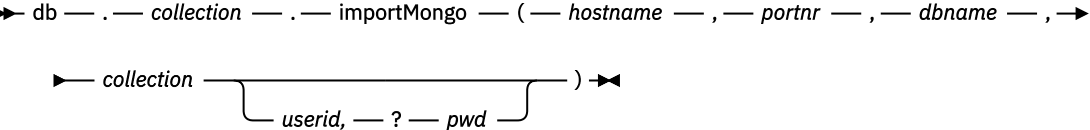
userid
This optional argument specifies the user name with which to connect to the MongoDB database.

pwd
This optional argument specifies the password for this user name.

importMongo() - Import JSON documents from MongoDB

Import JSON documents from a collection in a MongoDB database.

Syntax diagram



Command parameters

hostname
This argument specifies the name of the server that hosts the MongoDB database.

portnr
This argument specifies the port number of the MongoDB database.

dbname
This argument specifies the database name of the MongoDB database.

collection
This argument specifies the collection name in the MongoDB database.

userid
This optional argument specifies the user name with which to connect to the MongoDB database.

pwd
This optional argument specifies the password for this user name.

Example

Example 1: Update the specified fields in a document.

```
db.test.importMongo("remoteServer", 27017, "medialib", "DVD")
```

insert() - Insert a JSON document command

Inserts a JSON document into a collection. If the collection does not exist, it is created automatically.

Syntax diagram

▶▶─ db ── . ── *collection* ── . ── insert ── (── *document* ──) ─▶◀

Command parameters

document
This parameter specifies the JSON document to insert.

Example

Insert a JSON document into a collection:

```
db.books.insert({
          isbn: "123-456-789",
          author: "Verne, Jules",
          title: "Journey to the Center of the Earth",
          abstract: "Classic science fiction novel in an unusual setting",
          price: 6.00,
          pages: 276,
          category: "Fantasy",
          sales: 500.50
         })
```

Sample output is as follows:

```
OK
```

markType() - Specifies the data type

Specifies a data type for a JSON field in a collection.

Syntax diagram

▶▶─ db ── . ── *collection* ── . ── markType ── (── { ── *field* ── : ── *type* ── } ──) ─▶◀

Command parameters

field
This string argument specifies an attribute name.

type
: This argument specifies a data type, or null to remove the data type. The following data types are supported:

- $int
- $long
- $number
- $date
- $timestamp
- $string: length
- $binary: length

Example

Example 1: Mark the 'x' field as integer type, and sort by 'x' in ascending order.

```
db.c.insert({x:-100})
db.c.insert({x:-10})
db.c.markType({x:"$int"})
db.c.find().sort({x:1})
```

Sample output:

```
Row 1:
{
"_id":{"$oid":"51f08bbdeefadf1d37d08bd9"},
"x":-100
}
Row 2:
{
"_id":{"$oid":"51f08bc3eefadf1d37d08bda"},
"x":-10
}
```

Example 2: Mark the 'x' field as string type, and sort by 'x' in ascending order

```
db.c.markType({x:["$string",20]})
db.c.find().sort({x:1})
```

Sample output:

```
Row 1:
{
"_id":{"$oid":"51f08bc3eefadf1d37d08bda"},
"x":-10
}
Row 2:
{
"_id":{"$oid":"51f08bbdeefadf1d37d08bd9"},
"x":-100
}
```

printDDL() - Print the DDL

Prints the default DDL of a collection.

Example

Print the DDL for a collection:

```
db.collection.printDDL()
```

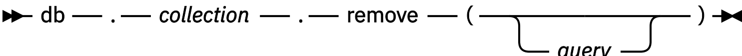
Sample output is as follows:

```
CREATE TABLE furniture (CLOB(16M) NOT NULL,DATA BLOB(16M)
INLINE LENGTH 3000,PRIMARY KEY(ID))
```

remove() - Remove data from a collection command

Removes all the data or specified data from a collection. Index definitions are retained.

Syntax diagram



Command parameters

query
This optional string parameter specifies a JSON query to filter a subset of data.

Example

Example 1: Remove all documents from the books collection.

```
db.books.remove()
```

Sample output is as follows:

```
Removed 15 row(s).
```

Example 2: Remove a book from the collection by specifying an ISBN:

```
db.books.remove({isbn: "123-456-789"})
```

Sample output is as follows:

```
Removed 1 row(s).
```

removeIndex() - Remove a JSON index command

Removes the specified index from a collection

Syntax diagram

▶▶ db — . — *collection* — . — removeIndex — (— *index_spec* —) ▶◀

Command parameters

index_spec
This parameter specifies an index name or the JSON definition of the index that is to be removed.

Example

Example 1: Remove the mypriceidx index by referring to the index name:

```
db.books.removeIndex("mypriceidx")
```

Sample output is as follows:

```
Index <mypriceidx> was removed successfully.
```

Example 2: Remove an index by specifying the index characteristics.

```
db.books.removeIndex({"author": 1})
```

Sample output is as follows:

```
Index <books_xauthor> was removed successfully.
```

rename() - Rename a collection command

Renames a collection.

Syntax diagram

```
>>- db — . — collection — . — rename — ( — new_name — , ─┬────────────┬─ ) ->◄
                                                          └─ forceDrop ─┘
```

Command parameters

new_name
This parameter specifies a new name for the collection. It must be alphanumeric, with no special characters except for $.

forceDrop(`true` | `false`)
If true, this optional parameter drops the existing collection *newName*.

Example

Rename the books collection to oldbooks:

```
db.books.rename("oldbooks")
```

Sample output is as follows:

```
Collection Renamed
```

sampleSchema() - Return document structure command

Returns the structure of JSON documents in a collection.

Syntax diagram

```
>>- db — . — collection — . — sampleSchema — ( ─┬──────┬─ ) ->◄
                                                  └ rows ┘
```

Command parameters

rows
This optional parameter specifies a number of rows to sample. If you set the parameter to `0` or do not provide a value, the structures of all documents in a collection are returned.

Example

View the structure of a JSON document in the books collection:

```
db.books.sampleSchema()
```

Sample output is as follows:

```
{
"._id":"15;type:ObjectId",
```

```
".abstract":"15;type:String",
".author":"15;type:String",
".category":"12;type:String",
".isbn":"15;type:String",
".pages":"14;type:Integer",
".price":"14;type:Double",
".sales":"13;type:Double",
".title":"15;type:String",
".year_published":"5;type:Integer"
}
```

save() - Save a document command

Saves an inserted or updated JSON document in a collection. If a document in a collection has the same ID as a new document, the existing document is replaced. Otherwise, the new document is inserted.

Syntax diagram

▶▶ db — . — *collection* — . — save — (— *document* —) ▶◀

Command parameters

document
This parameter specifies the JSON document to save.

Example

Save a JSON document in a collection:

```
db.books.save({isbn: "123-456-239",
              "author": "Verne, Jules",  "title": "Mysterious Island" })
```

Sample output is as follows:

```
OK
```

sqlQuery() - Run a JSON SQL query command

Runs an SQL query at the database level.

Syntax diagram

▶▶ db — . — sqlQuery — (— *sql* — , —┬——————————————┬—) ▶◀
　　　　　　　　　　　　　　　　　　　　　└— *params parameters* —┘

Command parameters

sql
This string parameter specifies an SQL query. It might contain parameter markers.

parameters
This optional string parameter specifies a list of parameter values to match any markers in the query.

Example

Issue an SQL query:

```
db.sqlQuery("select count(*) from test.products")
```

Sample output is as follows:

sqlUpdate() - Run a JSON SQL update query command

Runs an SQL update query at the database level.

Syntax diagram

db . sqlUpdate (*sql* , [*parameters*])

Command parameters

sql
: This string parameter specifies an SQL update query. It might contain parameter markers.

parameters
: This optional string parameter specifies a list of parameter values to match any markers in the query.

Example

Issue an SQL query:

```
db.sqlUpdate("update test.products set price = 9.80 where pid=45")
```

Sample output is as follows:

```

```

stats() - Prints collection statistics

Prints statistics for the collection. Unavailable values are set to -1.

Syntax diagram

Command arguments

scale
: This optional argument specifies the scale of the `size` field in the collection statistics. The following values are supported:

- 0 = bytes (default)
- 1 = KB
- 2 = MB
- 3 = GB

Any other numerical values are ignored.

Example

Example 1: Collect statistics for the books collection.

```
db.books.stats()
```

Sample output:

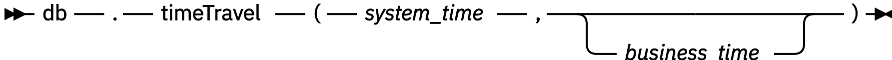
```
{
  "ns":"TEST.books",
  "count":15,
```

```
"size":234567,
"avgObjSize":309.0,
"numExtents":-1,
"nindexes":2,
"totalIndexSizes":-1,
"indexSizes":[{"books_xauthor": -1},{"_id_":-1}],
"ok":1
}
```

timeTravel() - Set JSON time travel queries command

Sets system or business time for JSON time travel queries against a Db2 database.

Syntax diagram



Command parameters

system_time
This parameter specifies a YYYY-MM-DD formatted string or time stamp. The default value is the current date.

business_time
This optional parameter specifies a YYYY-MM-DD formatted string or time stamp. The default value is the current date.

Example

Example 1: Create a collection with time travel enabled:

```
db.createCollection("timetravel", {systemTime:"sys",businessTime:"bus"})
```

Sample output is as follows:

```
Collection: TEST."timetravel" created. Use db.timetravel.
```

Example 2: Insert a document with business start and end times:

```
db.timetravel.insert({name:"Joe", age:5, bus_start:"1990-05-30", bus_end:"1996-08-13"})
```

Sample output is as follows:

```
OK
```

Example 3: Change the business time to 2012:

```
db.timeTravel(null, "2012-01-15")
```

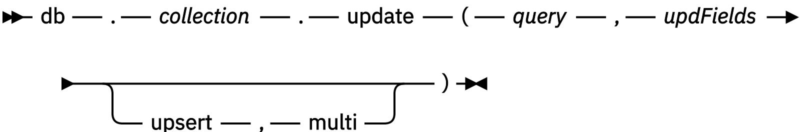
Sample output is as follows:

```
Setting system time: current date and business time: 2012-01-15
```

update() - Update JSON documents command

Updates a JSON document or documents in a collection.

Syntax diagram



Command parameters

query
This parameter specifies a filter to select a subset of documents. The command can return an empty result set.

updFields
This parameter specifies an object with attribute names and values that must be updated. This parameter uses the `$set` operator to set the new value for a field.

upsert (`true` | `false`)
If the value is `true`, this optional parameter updates a document or inserts missing documents. If the value is `false`, it only updates the documents. The default value is false.

multi (`true` | `false`)
If the value is `true`, this optional parameter updates all matching documents. If the value is `false`, only the first matching row is updated. The default value is false.

Example

Update the specified fields in a document:

```
db.books.update({isbn: "123-456-234"}, {$set: {pages: 299}})
```

Sample output is as follows:

```
Updated 1 rows.
```

Wire listener

The JSON wire listener acts as a mid-tier gateway server between MongoDB applications and the Db2 database server. This wire listener leverages JSON API to interface with the Db2 database.

You can use a Mongo application that is written in one of the supported application languages (such as Java, NodeJS, or pymongo), or use the Mongo command-line interface to communicate with the Db2 database.

Configuring the wire listener

You must configure the wire listener to communicate with the Db2 database server.

Before you begin

Install the following prerequisite software for the wire listener:

- Java JRE Version 1.6 or later
- The wire listener library (`db2NoSQLWireListener.jar`)
- The JSON library for NoSQL (`nosqljson.jar`)

- The JDBC driver (`db2jcc4.jar`)

Procedure

To configure the wire listener:

1. Configure the Db2 database server and client to access the JSON API for NoSQL.
2. Start the wire listener server in one of two ways:
   - On Windows operating systems, run the `wplistener.bat` script.
   - On Linux® and UNIX operating systems, run the `wplistener.sh` script.

   Applications that are based on community drivers must specify the wire listener host and port as well as a valid JSON namespace.

What to do next

Register the wire listener.

wpListener -help -

Displays usage help for the listener.

Syntax diagram

▶▶─ wpListener ── -help ─▶◀

Command parameters

—help
: Specifies usage help for the listener.

Example

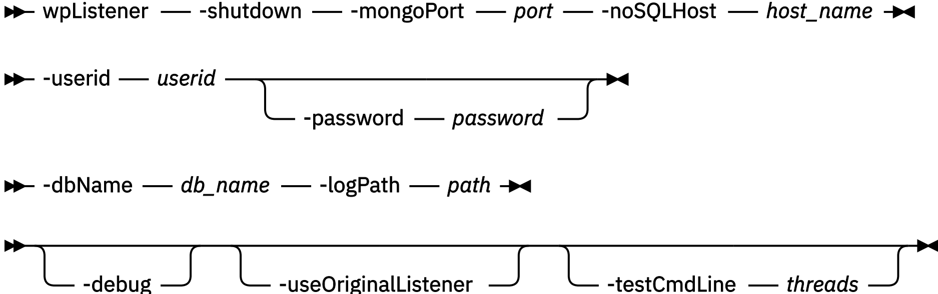
View usage information for the wire listener:

```
wpListener -help
```

wpListener -shutdown -

Stops the JSON wire listener.

Syntax diagram



Command parameters

-mongoPort *port*
: This parameter specifies a port that is listening for Mongo client requests.

-noSQLHost *host_name*
: This parameter specifies the name of the host that is to be shut down because it is running the wire listener.

-userid *userid*
: This parameter specifies the user ID for the backend server.

-password *password*
: This optional parameter specifies the password for the user ID for the backend server.

-dbName *db_name*
: This parameter specifies the name of the Db2 database that is used as the NoSQL data store.

-logPath *path*
: This parameter specifies the path for storing log files.

-debug
: This optional parameter enables debugging for the wire listener.

Example

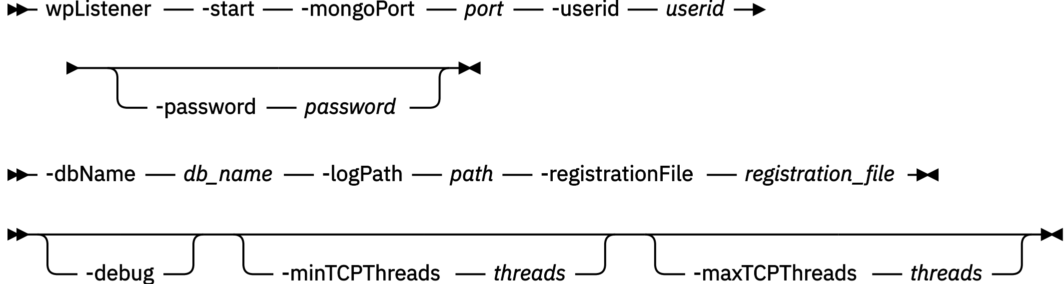
Stop the wire listener:

```
wpListener -shutdown -mongoPort 27017 -noSQLHost host -userid db2admin
```

wpListener -start -

Starts the JSON wire listener.

Syntax diagram



Command parameters

-mongoPort *port*
: This parameter specifies a port that is listening for messages that are based on the MongoDB wire protocol.

-userid *userid*
: This parameter specifies the user ID for the backend server.

-password *password*
: This optional parameter specifies the password for the user ID for the backend server.

-dbName *db_name*
: This parameter specifies the name of the Db2 database that is used as the NoSQL data store.

-logPath *path*
: This parameter specifies the path for storing log files.

-registrationFile *registration_file*
: This parameter specifies a registration file that contains credentials.

-debug
: This optional parameter enables debugging for the wire listener.

-minTCPThreads *threads*
This optional parameter specifies the maximum number of transaction control protocol threads.

-minTCPThreads *threads*
This optional parameter specifies the minimum number of transaction control protocol threads.

Example

Example 1: Start the wire listener without wire listener authentication:

```
wpListener -start -mongoPort 27017 -userid db2admin
-password pwd -debug dbName sample
-logPath c:/temp/logs
```

Example 2: Start the wire listener with wire listener authentication:

```
wpListener -start -mongoPort 27017 -userid db2admin
-password pwd -debug dbName sample
-logPath c:/temp/logs -registrationFile c:/temp/credentials.bin
```

Chapter 5. Working with JSON documents by using SQL

You can store and retrieve JSON data directly by using SQL.

The JSON functions rely on the JSON documents being stored in an internal binary format named BSON (Binary JSON) in Db2.

You can also store and retrieve JSON values in Db2 columns without parsing or indexing on specific fields inside the JSON document. Use a CLOB or VARCHAR column to hold the value and query and update based on another key column in the table.

Related tasks

Tuning routines with storage-consuming parameter lists (Db2 Performance)

JSON_VAL

The JSON_VAL function provides an SQL interface to extract and retrieve JSON data into SQL data types from BSON objects. The JSON_VAL function returns an element of a JSON document that is identified by the JSON field name that is specified in *search-string*. The value of the JSON element is returned in the data type and length that is specified in *result-type*.

```
►► JSON_VAL ── ( ── json-value ── , ── search-string ── , ── result-type ── ) ─►◄
```

The schema is SYSIBM.

json-value
An expression that returns a BLOB value. The *json-value* must contain a BSON representation of a JSON document.

search-string
A character or graphic string constant that is not longer than 2048 bytes. *search-string* contains the path-qualified JSON field name. If the value is not CCSID 1208, the value is converted to CCSID 1208 before the function is evaluated.

result-type
A character string constant that is not longer than 32 bytes that specifies the characteristics for the result of the function. The constant is not case-sensitive. The string ':na' can be appended to the data type to indicate that *json-value* must not contain a JSON array. Leading blanks can be specified before ':' and trailing blanks can be specified after ':'. For example, ' s:40', 's :40', 'b: 10', 'ts: na ', 's:40 :na', and 's:40: na ' can be specified. The *result-type* value must contain a data type specification from the following table.

Table 4. Supported constant values for the data type portion of result-type.

result-type	**SQL data type of the result**
n	DECFLOAT(34)
i	INTEGER
l	BIGINT
f	DOUBLE
d	DATE
ts	TIMESTAMP(6)

© Copyright IBM Corp. 2014, 2024

Table 4. Supported constant values for the data type portion of result-type. (continued)	
result-type	**SQL data type of the result**
t	TIME
s:*n*	VARCHAR(*n*), where *n* is an integer constant 1–32672. The CCSID of the result is the CCSID for Unicode mixed data (1208).
b:*n*	VARCHAR(*n*) FOR BIT DATA, where *n* is an integer constant 1–32672.
u	INTEGER, with values 0 or 1.

The JSON field name can identify a scalar value or a JSON array value. If ':na' is specified as part of *result-type*, the result must not be an array. If ':na' is not specified in *result-type* and the result is an array, the value of the first element of the identified array is returned.

The result of the JSON_VAL function is the data type and length that is specified by *result-type*. The result can be null.

The result is determined by the following criteria:

- If the JSON field name is not found in *json-value*, the result is the null value.
- If the BSON_TYPE of the JSON field in *json-value* is BSON_DBREF or BSON_UNDEFINED, the result is the null value.
- If *result-type* is 'u', the result value describes the value of the specified JSON field in *json-value*. The result depends on the following values of the JSON field:

  1
  A value exists and the value is not the null value or an empty string. For example: `{name:"Joe"}`.

  0
  A value exists and the value is the null value. For example: `{name:null}`.

  null
  A value does not exist. For example: `{ }`.

- Otherwise, if necessary, the value of the specified JSON field in *json-value* is converted to *result-type*. If the value of the JSON field cannot be converted to *result-type*, the null value is returned. For example, assume that the value of the JSON field is an arbitrary string that cannot be converted to an integer, and *result-type* is 'i'. The null value is returned. If the length of the data is longer than the length specified by *result-type*, the result is the null value. Also, if the result is VARCHAR or a datetime value (DATE, TIME, or TIMESTAMP), the encoding scheme is Unicode.

Notes

Assignment of arguments to parameters
When the JSON_VAL function is invoked and arguments are assigned to the parameter types of the function, or the result of the function is assigned to the target on return from the function, implicit cast is not used.

Statement concentrator
Constant values that are specified as arguments cannot be replaced with parameter markers for use with the statement concentrator.

CREATE INDEX statement considerations
The following considerations apply to CREATE INDEX statements that invoke the JSON_VAL function:

- If *key-expression* invokes the JSON_VAL function and the first argument is a LOB column, the column must be defined as an inline LOB.

- If *key-expression* invokes the JSON_VAL function, the function invocation must meet the following conditions:
  - The invocation of the JSON_VAL function must be the outermost expression for *key-expression*.
  - If the first argument is a column, that column must be contained in a table in a partition-by-growth table space.
  - The third argument must end with the string ':na', to indicate that the first argument does not contain a JSON array.
- If *key-expression* invokes the JSON_VAL built-in function, the CREATE INDEX statement must not reference any LOB columns other than the LOB column that is the argument to the JSON_VAL function. Such a CREATE INDEX statement can refer only to a single LOB column.

Date value restrictions

When $DATE is used in the JSON document, only date values from 01 January 1970 through 31 December 2037 are supported.

Examples

Example 1: Assume that the DATA column in table 'table1' contains a row with the following JSON document:

```
{
    name:"Joe",
    age:5,
    phone:["555-666-7777", "444-789-1234"],
    address:{
        street:"ABC st",
        zipcode:"95141"
               }
}
```

You can use the JSON_VAL function to find addresses with a zip code of '95141'.

```
SELECT * FROM "table1"
   WHERE JSON_VAL(DATA, 'address.zipcode', ‘s:5’)='95141';
```

Example 2: The following examples show how you can use the *search-string* argument. Assume that the table 'json_table' contains four rows with the following JSON objects. Each JSON object is stored as BSON in a BLOB column named DATA.

```
{
     name: "Joe",
      isMarried:true,
      phone: 82113456,
      partner:
      {
        name: "Kate",
         isMarried:true,
         phone: 82113111,
         son:
         {
            name: "Lock",
             isMarried:false,
             phone: 81231232, 81231233
}
}
}

{
     name:"Mary"
      isMarried:-1,
      phone: “82111432”
      partner: “NULL”
}

{
     name:"Henry",
      isMarried:0,
      phone: “NULL”
}
```

```
{
     name:"Bill",
      isMarried:false
}
```

Extract the name of the object:

You can use the JSON_VAL function to extract the name of the object whose partner's son's name (*search-string* of 'partner.son.name') is 'Lock' and return the value as VARCHAR(40).

```
SELECT JSON_VAL( data, 'name' , 's:40' ) FROM json_table
     WHERE JSON_VAL( data, 'partner.son.name' , 's:40' ) = 'Lock';
```

The result is 'Joe'.

Extract the name of the partner's partner:

You can use the JSON_VAL function to extract the name of the partner's partner (*search-string* of 'partner.partner') for each row and return the value as VARCHAR(40).

```
SELECT JSON_VAL( data, 'partner.partner ' , 's:40' ) FROM json_table ;
```

This example returns four rows, each containing the null value because the search string 'partner.partner' does not exist for the objects in any row.

JSON_LEN

The JSON_LEN user-defined function returns the number of elements in an element of type array inside a JSON document. NULL is returned if an element is not an array.

```
▶▶─ JSON_LEN( injson,inelem) ─▶◀
```

The schema is SYSTOOLS.

injson

An expression that returns a JSON document in binary format (extended BSON).

inelem

A character or graphic string constant that is not longer than 2048 bytes. *inelem* contains the path-qualified JSON field name to be searched for.

The result of the function is an integer.

Example

Assume that table T1 contains following JSON document in a column C1 as a BSON format.

```
{array: [10,20,30,40]}
```

Following query returns the number of elements in ‘array’.

```
SELECT SYSTOOLS.JSON_LEN(C1, 'array') as Length
FROM T1;
```

```
  +----------------+
  |     LENGTH     |
  +----------------+
1_|              4 |
  +----------------+
```

JSON_TYPE

The JSON_TYPE user-defined function returns the BSON type of the given element.

```
▸▸─ JSON_TYPE( injson,inelem,maxlength ) ─▸◂
```

The schema is SYSTOOLS.

injson
An expression that returns a JSON document in binary format (extended BSON).

inelem
A character or graphic string constant that is not longer than 2048 bytes. *inelem* contains the path-qualified JSON field name to be searched for.

maxlength
maxlength specifies the maximum length of the return value of the binary field of this user-defined function. The default is 2048. If the length of the field value is greater than 2048, then this function returns a null value.

The result of the function is an integer.

The following table shows the BSON type that is returned for the corresponding data types or null value.

Table 5. BSON type returned

BSON type	Data type
1	64-bit binary floating point
2	UTF-8 string
3	Embedded document
4	Array
5	Binary data
6	Undefined – deprecated
7	ObjectID
8	Boolean
9	UTC datetime
10	Null value
11	Regular expression
12	DBPointer – Deprecated
13	Javascript code
14	Deprecated
15	JavaScript code with Scope
16	32-bit integer
17	Timestamp
18	64-bit integer
0xFF	Min key
0x7F	Max key

Example

Assume that table T1 contains following simple JSON document in a column C1 as a BSON format.

```
{name: “Mike”, age: 20}
```

The following query returns BSON type for ‘name’ and ‘age’ in JSON document.

```
SELECT SYSTOOLS.JSON_TYPE(C1, 'name', 2048) AS Name,
       SYSTOOLS.JSON_TYPE(C1, 'age' , 2048) AS Age;
```

```
  +---------------------------------+
  |      NAME      |      AGE       |
  +---------------------------------+
1_|              2 |             16 |
  +---------------------------------+
```

The value 2 is the BSON type for a string and 16 is the BSON type for an integer.

JSON_TABLE

The JSON_TABLE user-defined function returns a table with two columns. First column is the BSON type and second column is the string value.

```
▶▶─ JSON_TABLE( injson,inelem,rettype ) ─▶◀
```

The schema is SYSTOOLS.

injson
An expression that returns a JSON document in binary format (extended BSON).

inelem
A character or graphic string constant that is not longer than 2048 bytes. *inelem* contains the path-qualified JSON field name to be searched for.

rettype
A character string constant that is no longer than 32 bytes that specifies the characteristics for the result of the function. *rettype* must contain a data type specification, see “JSON_VAL” on page 43. The value is not case-sensitive.

The result of the function is a table with the format that is shown in the following table. All the columns are nullable.

Table 6. Table returned

Column name	Data type	Contains
TYPE	INTEGER	The type of the element.
VALUE	VARCHAR(2048)	The value of the element.

Notes for JSON_TABLE

CCSID rules

The JSON_TABLE function follows the CCSID rules for user-defined functions, which means that it returns a string data type in the appropriate CCSID of the application encoding scheme. So, you might need to use the CAST function to avoid problems with the encoding of returned data.

Date value restrictions

When $DATE is used in the JSON document, only date values from 01 January 1970 through 31 December 2037 are supported.

Example

Assume that table T1 contains the following three rows of a JSON document, in BSON format, in a column C1.

```
{person: {name: "Mike",  age: 25}}
{person: {name: "John",  age: 42}}
{person: {name: "Kevin", age: 34}}
```

The following query returns a table with two columns: TYPE and VALUE.

```
SELECT X.*
FROM T1, TABLE(SYSTOOLS.JSON_TABLE(C1,'person', 's:50')) X
```

```
   +----------------------------------
   | TYPE    | VALUE
   +----------------------------------
 1_|       3 | {name:"Mike",age:25}
 2_|       3 | {name:"John",age:42}
 3_|       3 | {name:"Kevin",age:34}
   +----------------------------------
```

TYPE contains the BSON type of element 'person' and VALUE contains the value of 'person'. The TYPE 3 is the BSON type for an embedded document (nested JSON object).

JSON2BSON

The JSON2BSON user-defined function converts the specified JSON document in string format to an equivalent binary representation in BSON format. If you are using the JSON_VAL function to retrieve JSON field values from a JSON document, that document must be in BSON format.

```
►► JSON2BSON( injson ) ►◄
```

The schema is SYSTOOLS.

injson
An expression that returns a CLOB (JSON document in text format) that is no larger than 16 MB.

The result is a BLOB(16 MB) that contains the converted BSON value.

If the input string contains an invalid JSON format, or if the converted value cannot fit into a BLOB(16 MB), an error is returned.

Notes for JSON2BSON

Date value restrictions
When $DATE is used in the JSON document, only date values from 01 January 1970 through 31 December 2037 are supported.

Example

The following example inserts a row into a table, which contains a BLOB column that holds a JSON document:

```
INSERT INTO CUSTOMER_LIST
                  (CUSTID, ADDRESS_LINE1, CITY, STATE, ZIP, PROFILE)
       VALUES('12345',' 333 Oak Street', 'San Jose', 'CA', '95110',
                   SYSTOOLS.JSON2BSON('{"name":"Joe Stockton", "age":28, "interests":
["gardening","camping"]}'));
```

Related tasks

Tuning routines with storage-consuming parameter lists (Db2 Performance)

BSON2JSON

The BSON2JSON user-defined function converts a JSON document in binary format (extended BSON) into a readable JSON text format.

```
▶▶─ BSON2JSON( inbson ) ─▶◀
```

The schema is SYSTOOLS.

inbson
An expression that returns a document in binary format (extended BSON) that is no larger than 16 MB.

The result is a CLOB (16 MB) that contains a JSON document in a readable text format.

If the input value does not contain a valid BSON formatted expression, or if the converted value cannot fit into a 16 MB CLOB, an error is returned.

Notes for BSON2JSON

CCSID rules
The BSON2JSON function follows the CCSID rules for user-defined functions, which means that it returns a string data type in the appropriate CCSID of the application encoding scheme. So, you might need to use the CAST function to avoid problems with the encoding of returned data.

Date value restrictions
When $DATE is used in the JSON document, only date values from 01 January 1970 through 31 December 2037 are supported.

Example

The following SQL statement uses BSON2JSON to convert the content of the BLOB column that is named PROFILE to a readable format as part of a SELECT statement. The example assumes that PROFILE contains data that was inserted in a valid BSON format:

```
SELECT CUSTID, CITY, SYSTOOLS.BSON2JSON(PROFILE)
FROM CUSTOMER_LIST
WHERE CUSTID = '12345';
```

Related tasks
Tuning routines with storage-consuming parameter lists (Db2 Performance)

Creating a column to store JSON data

JSON data can be stored in BSON format in a column in Db2.

About this task

Procedure

Issue the CREATE TABLE SQL statement and specify BLOB as the data type for the column to contain JSON data.

Example

The following example creates a CALL_RECORDS column to store JSON data.

```
CREATE TABLE CLAIMS (CLAIM_ID       VARCHAR(10),
                     POLICY_NUM     VARCHAR(12),
                     CUSTOMER_ID    VARCHAR(8),
```

```
                    STATUS         VARCHAR(20),
                    CALL_RECORDS   BLOB);
```

Inserting JSON data into a table

You can use the JSON2BSON function to insert JSON data into a table.

About this task

The JSON2BSON function is used to convert a value into its binary representation before it is inserted into a table. JSON2BSON requires a CLOB data type as the single input parameter.

Procedure

Issue the INSERT SQL statement and specify the JSON2BSON function.

Example

The following example inserts a JSON document string into the table:

```
INSERT INTO CLAIMS (CLAIM_ID, POLICY_NUM, CUSTOMER_ID, STATUS, CALL_RECORDS)
            VALUES ( 'AB0033789',
                     'GL0000336512',
                     '09736814',
                     'CLOSED',
                     SYSTOOLS.JSON2BSON('{"calls": [ {"dateOfCall": "2014-02-26", "associate":
"Steven Barnes",
                                         "conversationNotes": "Customer provided details of
accident and witness
                                         contact information.  Stated that her local agent was
unavailable."}  ],
                                         "satisfactionSurveyResult":"needs improvement"  }'
                                       )
                     );
```

Updating JSON data in a table

You can use the JSON2BSON function to update JSON data in a table.

Procedure

To update JSON data in a table:

1. Query and retrieve the JSON data from a JSON column.
2. Modify the contents of the JSON column.
3. To push the updated column value back into the table, issue the UPDATE SQL statement and specify the JSON2BSON function.

Example

The following example updates the JSON document in the table:

```
UPDATE CLAIMS SET CALL_RECORDS = SYSTOOLS.JSON2BSON('{"calls": [
                                                     {"dateOfCall": "2014-02-26",
                                                      "associate": "Steven Barnes",
                                                      "conversationNotes": "Customer provided
details of accident
                                                      and witness contact information.  Stated
that her local
                                                      agent was unavailable."},
                                                     {"dateOfCall": "2014-02-29",
                                                      "associate": "Paula Garcia",
                                                      "conversationNotes": "Client asked about
status of the
                                                      damage estimate."}],
```

```
                                                       "satisfactionSurveyResult":"needs
improvement" }');
```

Retrieving data that is inside a JSON document

You can use the JSON_VAL function to retrieve data that is inside a JSON document.

Before you begin

The examples in this procedure assume that you have created and populated the following tables:

- The CLAIMS table:
  - Define CLAIMS as shown in "Creating a column to store JSON data" on page 50.
  - Populate CLAIMS as shown in "Inserting JSON data into a table" on page 51.
- The ASSOCIATES table:
  - Define ASSOCIATES like this:

    ```
    CREATE TABLE ASSOCIATES
     (ASSOC_ID CHAR(6), NAME VARCHAR(40));
    ```

  - Populate ASSOCIATES like this:

    ```
    INSERT INTO ASSOCIATES (ASSOC_ID, NAME) VALUES ('000100','Paula Garcia');
    INSERT INTO ASSOCIATES (ASSOC_ID, NAME) VALUES ('000200','Steven Barnes');
    INSERT INTO ASSOCIATES (ASSOC_ID, NAME) VALUES ('000300','Roger McCarthy');
    ```

Procedure

Issue the SELECT SQL statement and specify the JSON_VAL function.

Example

The following example uses the JSON_VAL function to extract data from a JSON document for retrieval through a SELECT clause.

```
SELECT CLAIM_ID,
 JSON_VAL(CALL_RECORDS, 'satisfactionSurveyResult', 's:25' )
 AS "SATISFACTION SURVEY RESULT"
 FROM CLAIMS
 WHERE CLAIM_ID = 'AB0033789';
```

The output should look like this.

CLAIM_ID	SATISFACTION SURVEY RESULT
AB0033789	needs improvement

The following example uses the JSON_VAL function to extract a value that is inside the JSON document for use in a WHERE clause predicate:

```
SELECT CUSTOMER_ID, CLAIM_ID, SYSTOOLS.BSON2JSON(CALL_RECORDS)
 AS "CALL RECORD"
 FROM SYSADM.CLAIMS
 WHERE JSON_VAL(CALL_RECORDS,
  'satisfactionSurveyResult', 's:25') = 'needs improvement' ;
```

The output should look like this.

CUSTOMER_ID	CLAIM_ID	CALL RECORD
09736814	AB0033789	{"calls":[{"dateOfCall":"2014-02-26","associate":"Steven Barnes","conversationNotes":"Customer provided details of accident and witness contact information. Stated that her local agent was unavailable."}],"satisfactionSurveyResult":"needs improvement"}

The following example uses the JSON_VAL function to extract a value that is inside the JSON document for use in a JOIN predicate:

```
SELECT C.CUSTOMER_ID, C.CLAIM_ID, SYSTOOLS.BSON2JSON(C.CALL_RECORDS)
 AS "CALL RECORD",
 A.NAME, A.ASSOC_ID
 FROM CLAIMS C, ASSOCIATES A
 WHERE JSON_VAL(C.CALL_RECORDS, 'satisfactionSurveyResult', 's:25') =
  'needs improvement'
  AND JSON_VAL(C.CALL_RECORDS, 'calls.associate', 's:40') = A.NAME;
```

You should receive results like these:

CUSTOMER_ID	CLAIM_ID	CALL RECORD	NAME	ASSOC_ID
09736814	AB0033789	{"calls": [{"dateOfCall":"2014-02-26","associate":"Steven Barnes","conversationNotes":"Customer provided details of accident and witness contact information. Stated that her local agent was unavailable."}],"satisfactionSurveyResult":"needs improvement"}	Steven Barnes	000200

Example SQL statements with JSON2BSON, BSON2JSON, and JSON_VAL functions

The following examples show how you can use SQL with the JSON2BSON, BSON2JSON, and JSON_VAL functions to work with JSON data.

Example: Creating tables to store JSON data

The following example creates tables to store JSON data.

```
CREATE TABLE  JSONPO( ID INTEGER NOT NULL,
                      DATA BLOB(16M) INLINE LENGTH 25000,
                      PRIMARY KEY(ID)) CCSID UNICODE;

CREATE TABLE  JSONCUSTOMER
                    ( ID INTEGER NOT NULL,
                      DATA BLOB(16M) INLINE LENGTH 25000,
                      PRIMARY KEY(ID)) CCSID UNICODE;

CREATE TABLE  JSONPOTXT( DATA VARCHAR(5000)) CCSID UNICODE;
```

Example: Creating an index on a table with JSON data

The following example creates an index on Customer.age column and Customer.name column in the JSONCUSTOMER table.

```
CREATE INDEX IX2 ON  JSONCUSTOMER (JSON_VAL(DATA, 'Customer.age', 'i:na'),
                                   JSON_VAL(DATA, 'Customer.name', 's:20:na'));
```

Example: Inserting JSON data into a table

The following example inserts purchase order JSON data into the JSONPO table, and customer information into the JSONCUSTOMER table.

```
INSERT INTO JSONPO VALUES (
 101,
 SYSTOOLS.JSON2BSON(
   '{"PO":{"@id": 101,
           "@orderDate": "2014-11-18",
           "customer": {"@cid": 999},
           "items": {
              "item": [{"@partNum": "872-AA",
                        "productName": "Lawnmower",
                        "quantity": 1,
                        "USPrice": 149.99,
                        "shipDate": "2014-11-20"
                       },
                       {"@partNum": "945-ZG",
                        "productName": "Sapphire Bracelet",
                        "quantity": 2,
                        "USPrice": 178.99,
                        "comment": "Not shipped"
                       }
                      ]
                    }
          }
    }')
);

INSERT INTO  JSONPO VALUES (
  102,
SYSTOOLS.JSON2BSON(
 '{"PO":{"@id": 102,
           "@orderDate": "2014-12-20",
           "customer": {"@cid": 888},
           "items": {
              "item": [{"@partNum": "872-AA",
                        "productName": "Lawnmower",
                        "quantity": 1,
                        "USPrice": 749.99,
                        "shipDate": "2014-12-21"
                       },
                       {"@partNum": "837-CM",
                        "productName": "Digital Camera",
                        "quantity": 2,
                        "USPrice": 199.99,
                        "comment": "2014-12-22"
                       }
                      ]
                   }
        }
  }'));

INSERT INTO  JSONCUSTOMER VALUES (
   101,
   SYSTOOLS.JSON2BSON(
     '{"Customer":{"@cid": 999,
                   "name": "Michael",
                   "age": 31,
                   "telephone": "234-343-2343",
                   "country": "USA"
                   }
      }'));

INSERT INTO  JSONCUSTOMER VALUES (
   102,
   SYSTOOLS.JSON2BSON(
      '{"Customer":{"@cid": 888,
                    "name": "George",
                    "age": 29,
                    "telephone": "133-144-9999",
                    "country": "USA"
                   }
      }'));
```

The following example inserts a JSON document into a VARCHAR column. JSON2BSON is not needed for this operation.

```
INSERT INTO  JSONPOTXT VALUES (
   '{"PO":{"@id": 103,
           "@orderDate": "2014-06-20",
           "customer": {"@cid": 888},
           "items": {
             "item": [ { "@partNum": "872-AA",
                         "productName": "Lawnmower",
                         "quantity": 1,
                         "USPrice": 749.99,
                         "shipDate": "2014-06-21"
                       },
                       { "@partNum": "837-CM",
                         "productName": "Digital Camera",
                         "quantity": 2,
                         "USPrice": 199.99,
                         "comment": "2014-06-22"
                       }
                     ]
                   }
          }
    }');
```

Examples: Displaying JSON data

The following example displays the entire JSON data as text from the JSONPO table.

```
SELECT ID, SYSTOOLS.BSON2JSON(DATA)
FROM  JSONPO;
```

The following example converts BSON format to JSON format and displays the entire JSON data as text.

```
SELECT SYSTOOLS.BSON2JSON(SYSTOOLS.JSON2BSON((SELECT DATA FROM  JSONPOTXT)))
FROM SYSIBM.SYSDUMMY1;
```

The following example displays customer ID JSON data in descending order.

```
SELECT SYSTOOLS.BSON2JSON(DATA)
FROM  JSONPO
ORDER BY JSON_VAL(DATA, 'PO.customer.@cid', 'i') DESC;
```

Examples: Retrieving JSON data

The following example retrieves the name of customers who are older than 30.

```
SELECT JSON_VAL(DATA,'Customer.name', 's:20') "Name"
FROM  JSONCUSTOMER
WHERE JSON_VAL(DATA,'Customer.age', 'i') > 30;
```

The following example retrieves 2014 purchase orders from table JSONPO.

```
SELECT SYSTOOLS.BSON2JSON(DATA)
FROM  JSONPO
WHERE YEAR(JSON_VAL(DATA,'PO.@orderDate', 'd')) = 2014;
```

The following example retrieves the first item that was purchased by a customer who is older than 30 years old.

```
SELECT JSON_VAL(T2.DATA,'PO.items.item.0.productName', 's:10') AS "Item"
FROM  JSONCUSTOMER T1,  JSONPO T2
WHERE JSON_VAL(T1.DATA, 'Customer.@cid', 'i') = JSON_VAL(T2.DATA, 'PO.customer.@cid', 'i')
AND
      JSON_VAL(T1.DATA,'Customer.age', 'i') > 30;
```

Example: Updating JSON data in a table

The following example updates the Customer information.

```
UPDATE  JSONCUSTOMER
SET DATA = SYSTOOLS.JSON2BSON('{"Customer":{"@cid": 777,
                                                  "name": "George",
                                                  "age": 29,
                                                  "telephone": "566-898-1111",
                                                  "country": "USA"
                                                   }
                                     }')
WHERE JSON_VAL(DATA, 'Customer.@cid', 'i:na') = 888;
```

Example: Deleting JSON data in a table

The following example deleted JSON data in a table:

```
DELETE FROM JSONCUSTOMER
WHERE JSON_VAL(DATA, 'Customer.@cid', 'i:na') = 777;
```

Information resources for Db2 for z/OS and related products

You can find the online product documentation for Db2 13 for z/OS and related products in IBM Documentation.

For all online product documentation for Db2 13 for z/OS, see IBM Documentation (https://www.ibm.com/docs/en/db2-for-zos/13).

For other PDF manuals, see PDF format manuals for Db2 13 for z/OS (https://www.ibm.com/docs/en/db2-for-zos/13?topic=zos-pdf-format-manuals-db2-13).

© Copyright IBM Corp. 2014, 2024

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785 US

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing Legal and Intellectual Property Law IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785 US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

© Copyright IBM Corp. 2014, 2024

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as shown below:

© (*your company name*) (*year*).
Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. (*enter the year or years*).

Programming interface information

This information is intended to help you to plan for and administer Db2 13 for z/OS. This information also documents General-use Programming Interface and Associated Guidance Information and Product-sensitive Programming Interface and Associated Guidance Information provided by Db2 13 for z/OS.

General-use Programming Interface and Associated Guidance Information

General-use Programming Interfaces allow the customer to write programs that obtain the services of Db2 13 for z/OS.

General-use Programming Interface and Associated Guidance Information is identified where it occurs by the following markings:

GUPI General-use Programming Interface and Associated Guidance Information... GUPI

Product-sensitive Programming Interface and Associated Guidance Information

Product-sensitive Programming Interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this IBM software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive Programming Interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs by the following markings:

PSPI Product-sensitive Programming Interface and Associated Guidance Information... PSPI

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered marks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at: http://www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions:

Applicability: These terms and conditions are in addition to any terms of use for the IBM website.

Personal use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights: Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM’s Privacy Policy at http://www.ibm.com/privacy and IBM’s Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled “Cookies, Web Beacons and Other Technologies” and the “IBM Software Products and Software-as-a-Service Privacy Statement” at http://www.ibm.com/software/info/product-privacy.

Glossary

The glossary is available in IBM Documentation

For definitions of Db2 for z/OS terms, see Db2 glossary (Db2 Glossary).

© Copyright IBM Corp. 2014, 2024

Index

A

accessibility
- keyboard vi
- shortcut keys vi

aggregate() JSON command 19

B

BSON2JSON
- example 53

BSON2JSON function 50

C

commands
- JSON
  - aggregate() 19
  - count() 20
  - distinct() 22
  - drop() 23
  - exportFile() 24
  - find() 25
  - findAndModify() 26
  - getCollectionNames() 27
  - getIndexes() 28
  - group() 28
  - help() 29
  - importFile() 29
  - insert() 31
  - remove() 33
  - removeIndex() 33
  - rename() 34
  - sampleSchema() 34
  - save() 35
  - sqlQuery() 35
  - sqlUpdate() 36
  - timeTravel() 37
  - update() 38
  - wpListener -help 39
  - wpListener -shutdown 39
  - wpListener -start 40

count() JSON command 20

D

disability vi
distinct() JSON command 22
drop() JSON command 23

E

example
- BSON2JSON 53
- JSON_VAL 53
- JSON2BSON 53

exportFile() JSON command 24

F

find() JSON command 25
findAndModify() JSON command 26
function
- BSON2JSON 50
- JSON_LEN 46–48
- JSON_VAL 43
- JSON2BSON 49

G

general-use programming information, described 60
getCollectionNames() JSON command 27
getIndexes() JSON command 28
group() JSON command 28
GUPI symbols 60

H

help() JSON command 29

I

importFile() JSON command 29
insert() JSON command 31

J

JSON
- batching 12
- collections 2
- command-line interface (CLI)
  - db2nosql script 18
- command-line options 19
- commands
  - aggregate() 19
  - count() 20
  - distinct() 22
  - drop() 23
  - exportFile() 24
  - find() 25
  - findAndModify() 26
  - getCollectionNames() 27
  - getIndexes() 28
  - group() 28
  - help() 29
  - importFile() 29
  - insert() 31
  - remove() 33
  - removeIndex() 33
  - rename() 34
  - sampleSchema() 34
  - save() 35

JSON *(continued)*
commands *(continued)*
sqlQuery() 35
sqlUpdate() 36
timeTravel() 37
update() 38
wpListener -help 39
wpListener -shutdown 39
wpListener -start 40
comparison operators 10
connecting to a Db2 database 15
connection management 9
create indexes on JSON fields 17
data evaluation operators 11
db.collection.ensureIndex() 23
db.collection.findOne() 27
db.collection.importMongo() 30
db.collection.importMongoSample() 30
db.collection.markType() 31
db.collection.stats() 36
db.createCollection() 21
documents 2
fire and forget mode 13
import or export JSON data 17
installation requirements 5
Java APIs 15
key concepts 1
lazy fetches 12
logical operators 9
namespaces 2
nested objects 3
NoSQL properties for the client 5
operators
comparison 10
data evaluation 11
logical 9
performance features 12
printDDL() 32
properties
nosql.asyncMaxThreadCount 5
nosql.connectionPoolSize 5
nosql.traceFile 6
nosql.traceLevel 6
query operators 9
security model 11
select JSON documents from a DB2 database 17
solution planning 9
store documents in a Db2 database 16
wire listener 38
wire listener authentication 11
wire protocol listener 38
JSON documents
working with 15
JSON_LEN function 46–48
JSON_VAL
example 53
JSON_VAL function 43
JSON2BSON
example 53
JSON2BSON function 49

L

links
links *(continued)*
non-IBM Web sites
61

P

product-sensitive programming information, described 60
programming interface information, described 60
PSPI symbols 60

R

remove() JSON command 33
removeIndex() JSON command 33
rename() JSON command 34

S

sampleSchema() JSON command 34
save() JSON command 35
shortcut keys
keyboard vi
sqlQuery() JSON command 35
sqlUpdate() JSON command 36
syntax diagram
how to read vii

T

timeTravel() JSON command 37

U

update() JSON command 38

W

wpListener -help JSON command 39
wpListener -shutdown JSON command 39
wpListener -start JSON command 40

Product Number: 5698-DB2
5698-DBV