Informix Product Family
Informix
Version 12.10

IBM Informix JSON Compatibility Guide

SC27-5556-05

Informix Product Family
Informix
Version 12.10

IBM Informix JSON Compatibility Guide

SC27-5556-05

Note
Before using this information and the product it supports, read the information in "Notices" on page B-1.

This edition replaces SC27-5556-04.

Contents

Introduction

This introduction provides an overview of the information in this publication and describes the conventions that this publication uses.

About This Publication

This publication contains information about using the IBM® Informix® JSON capability.

This section discusses the intended audience for this publication and the associated software products that you must have to use the administrative utilities.

Types of Users

This publication is written for the following users:

- Database administrators
- System administrators
- Performance engineers

This publication is written with the assumption that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with database server administration, operating-system administration, or network administration

You can access the Informix information centers, as well as other technical information such as technotes, white papers, and IBM Redbooks publications online at http://www.ibm.com/software/data/sw-library/.

Assumptions about your locale

IBM Informix products can support many languages, cultures, and code sets. All the information related to character set, collation and representation of numeric data, currency, date, and time that is used by a language within a given territory and encoding is brought together in a single environment, called a Global Language Support (GLS) locale.

The IBM Informix OLE DB Provider follows the ISO string formats for date, time, and money, as defined by the Microsoft OLE DB standards. You can override that default by setting an Informix environment variable or registry entry, such as GL_DATE.

If you use Simple Network Management Protocol (SNMP) in your Informix environment, note that the protocols (SNMPv1 and SNMPv2) recognize only English code sets. For more information, see the topic about GLS and SNMP in the *IBM Informix SNMP Subagent Guide*.

The examples in this publication are written with the assumption that you are using one of these locales: en_us.8859-1 (ISO 8859-1) on UNIX platforms or

en_us.1252 (Microsoft 1252) in Windows environments. These locales support U.S. English format conventions for displaying and entering date, time, number, and currency values. They also support the ISO 8859-1 code set (on UNIX and Linux) or the Microsoft 1252 code set (on Windows), which includes the ASCII code set plus many 8-bit characters such as é and ñ.

You can specify another locale if you plan to use characters from other locales in your data or your SQL identifiers, or if you want to conform to other collation rules for character data.

For instructions about how to specify locales, additional syntax, and other considerations related to GLS locales, see the *IBM Informix GLS User's Guide*.

Demonstration databases

The DB-Access utility, which is provided with your IBM Informix database server products, includes one or more of the following demonstration databases:

- The **stores_demo** database illustrates a relational schema with information about a fictitious wholesale sporting-goods distributor. Many examples in IBM Informix publications are based on the **stores_demo** database.
- The **superstores_demo** database illustrates an object-relational schema. The **superstores_demo** database contains examples of extended data types, type and table inheritance, and user-defined routines.

For information about how to create and populate the demonstration databases, see the *IBM Informix DB-Access User's Guide*. For descriptions of the databases and their contents, see the *IBM Informix Guide to SQL: Reference*.

The scripts that you use to install the demonstration databases are in the `$INFORMIXDIR/bin` directory on UNIX platforms and in the `%INFORMIXDIR%\bin` directory in Windows environments.

What's new in JSON, Version 12.10

This publication includes information about new features and changes in existing functionality.

For a complete list of what's new in this release, go to http://www.ibm.com/support/knowledgecenter/SSGU8G_12.1.0/com.ibm.po.doc/new_features_ce.htm.

Table 1. What's new in JSON for IBM Informix Version 12.10.xC6

Overview	Reference
Parallel sharded queries You can now run SELECT statements in sharded queries in parallel instead of serially on each shard. Parallel sharded queries return results faster, but also have the following benefits: • Reduced memory consumption: Table consistency is enforced on the shard servers, which eliminates the processing of data dictionary information among the shard servers. • Reduced network traffic: Client connections are multiplexed over a common pipe instead of being created individual connections between each client and every shard server. Client connections are authenticated on only one shard server instead of on every shard server. Network traffic to check table consistency is eliminated. To enable parallel sharded queries, set the new SHARD_ID configuration parameter in the `onconfig` file to a unique value on each shard server in the shard cluster. Also set the new **sharding.parallel.query.enable=true** and **sharding.enable=true** parameters in the wire listener configuration file for each shard server. You can customize how shared memory is allocated for parallel sharded queries on each shard server by setting the new SHARD_MEM configuration parameter. You can reduce latency between shard servers by increasing the number of pipes for SMX connections with the new SMX_NUMPIPES configuration parameter. If you plan to upgrade your existing shard cluster from a previous version of Informix 12.10, upgrade and set the SHARD_ID configuration parameter on all the shard servers to enable parallel sharded queries.	Chapter 3, "JSON data sharding," on page 3-1

Table 1. What's new in JSON for IBM Informix Version 12.10.xC6 (continued)

Overview	Reference
MongoDB 2.6 and 3.0 compatibility Informix now supports the following MongoDB commands: • The following database management commands: – The query and write operation commands **insert**, **update**, and **delete**. – The instance administration commands **createIndexes**, **listCollections**, and **listIndexes**. – The user management commands, for MongoDB 2.6 and later, **createUser**, **updateUser**, **dropUser**, **dropAllUserFromDatabase**, **grantRolesToUser**, **revokeRolesFromUser**, and **usersInfo**. – The role management commands: **createRole**, **updateRole**, **dropRole**, **dropAllRolesFromDatabase**, **grantPrivilegesToRole**, **revokePrivilegesFromRole**, **grantRolesToRole**, **revokeRolesFromRole**, and **rolesInfo**. • The query and projection command **$eq**. • The field update operators **$mul**, **$min**, **$max**, and **$currentDate**. • The pipeline aggregation operator **$out**. You can authenticate MongoDB clients with the MongoDB 3.0 SCRAM-SHA-1 authentication method. You must upgrade the user schema for existing users. You upgrade to MongoDB 3.0 by setting the new **mongo.api.version** parameter to 3.0 in the wire listener configuration file.	"Query and projection operators" on page 4-18 "Database commands" on page 4-4 "Aggregation framework operators" on page 4-22 "Configuring MongoDB authentication" on page 2-38

Table 1. What's new in JSON for IBM Informix Version 12.10.xC6 (continued)

Overview	Reference
Wire listener enhancements The wire listener has the following new parameters that you can set to customize the wire listener. **MongoDB compatibility** Specify the version of MongoDB API compatibility with the **mongo.api.version** parameter. **Security** Disable commands with the **command.blacklist** parameter. Specify the authentication type with the **db.authentication** parameter. Specify an IP address as the administrative host with the **listener.admin.ipAddress** parameter. Set authentication timeout with the **listener.authentication.timeout** parameter. Add information to HTTP headers with the **listener.http.headers** parameter. **Resource management** Configure a memory monitor to reduce resource usage with the **listener.memoryMonitor** parameters. Create a separate thread pool for administrative connections with the **listener.pool.admin.enable** parameter. Specify the timeout periods for socket connections with the **listener.socket.accept.timeout** and **listener.socket.read.timeout** parameters. Suppress pooled connection checking with the **pool.lenient.return.enable** and the **pool.lenient.dispose.enable** parameters. Specify the number of maintenance threads for connection pools with the **pool.service.threads** parameter.	"The wire listener configuration file" on page 2-3
Authenticate wire listener connections with Informix You can configure the database server to authenticate MongoDB client users, who connect through the wire listener, with a pluggable authentication module (PAM). Because you administer user accounts through the database server, you can audit user activities and configure fine-grained access control. In contrast, if you use MongoDB authentication, MongoDB clients connect to the database server as the wire listener user that is specified by the **url** parameter.	"Configuring database server authentication with PAM (UNIX, Linux)" on page 2-39
Starting the wire listener for the REST API You no longer need to provide the path to tomcat when you start the wire listener for the REST API.	"Starting the wire listener" on page 2-33

Table 2. What's new in JSON for IBM Informix Version 12.10.xC5

Overview	Reference
High availability for MongoDB and REST clients You can provide high availability to MongoDB and REST clients by running a wire listener on each server in your Informix high-availability cluster. You can also provide high availability between the wire listener and the Informix database server. Connect the wire listener to the database server through the Connection Manager or specify an `sqlhosts` file in the **`url`** parameter in the wire listener properties file.	"High availability support in the wire listener" on page 2-45
Wire listener configuration enhancements You can set these new or updated parameters in the wire listener properties file: • **`url`** parameter can include JDBC environment variables. • New: **`listener.hostName`** parameter can specify the listener host name to control the network adapter or interface to which the wire listener connects. • New: **`collection.informix.options`** parameter can specify table options to automatically add shadow columns or enable auditing when you create a JSON collection. • New: **`command.listDatabases.sizeStrategy`** parameter can specify a strategy for computing the size of a database when you run the MongoDB **listDatabases** command. • New: **`fragment.count`** parameter can specify the number of fragments to create for a collection. • New: **`jdbc.afterNewConnectionCreation`** parameter can specify SQL statements, for example, SET ENVIRONMENT statements, to run after connecting to the database server.	"The wire listener configuration file" on page 2-3
Wire listener query support The wire listener now supports these types of queries: • Join queries on JSON data, relational data, or both JSON and relational data. • Array queries on JSON data with the MongoDB **$elemMatch** query operator.	"Running join queries by using the wire listener" on page 2-43 "Query and projection operators" on page 4-18
Enhanced account management through the wire listener You can control user authorization to Informix databases through the wire listener by locking and unlocking user accounts or individual databases with the new Informix JSON **lockAccount** and **unlockAccounts** commands.	"Informix JSON commands" on page 4-11

Table 2. What's new in JSON for IBM Informix Version 12.10.xC5 (continued)

Overview	Reference
Manipulate JSON and BSON data with SQL statements You can use SQL statements to manipulate BSON data. You can create BSON columns with the SQL CREATE TABLE statement. You can manipulate BSON data in a collection that was created by a MongoDB API command. You can use the CREATE INDEX statement to create an index on a field in a BSON column. You can insert data with SQL statements or Informix utilities. You can view BSON data by casting the data to JSON format or running the new BSON value functions to convert BSON field values into standard SQL data types, such as INTEGER and LVARCHAR. You can use the new BSON_GET and BSON_UPDATE functions to operate on field-value pairs.	"Manipulate BSON data with SQL statements" on page 1-5

Table 3. What's new in JSON for IBM Informix Version 12.10.xC4W1

Overview	Reference
Support for CORS requests in the REST API (12.10.xC4W1) You can now set up cross-origin resource sharing (CORS) with the REST API. To do so, set the following optional parameters that were added to the `jsonListener.properties` file: • listener.http.accessControlAllowCredentials • listener.http.accessControlAllowHeaders • listener.http.accessControlAllowMethods • listener.http.accessControlAllowOrigin • listener.http.accessControlExposeHeaders • listener.http.accessControlMaxAge Use these parameters to configure the HTTP headers of all responses. The HTTP headers provide access to JSON fields that are required by synchronous JavaScript + XML (AJAX) applications in a web browser when these applications access the REST listener.	"The wire listener configuration file" on page 2-3

Table 4. What's new in JSON for IBM Informix Version 12.10.xC4

Overview	Reference
Basic text searching support for JSON and BSON data You can now create a basic text search index on columns that have JSON or BSON data types. You can create the basic text search index on JSON or BSON data types through SQL with the CREATE INDEX statement or on BSON data types through the Informix extension to MongoDB with the **createTextIndex** command. You can control how JSON and BSON columns are indexed by including JSON index parameters when you create the basic text search index. You can run a basic text query on JSON or BSON data with the **bts_contains()** search predicate in SQL queries or the **$ifxtext** query operator in JSON queries.	"Informix JSON commands" on page 4-11 "Informix query operators" on page 4-22

Table 4. What's new in JSON for IBM Informix Version 12.10.xC4 (continued)

Overview	Reference
Enhanced JSON compatibility Informix now supports the following MongoDB 2.4 features: • Cursor support so that you can query large volumes of data. • Text search of string content in collections and tables. • Geospatial indexes and queries. • Pipeline aggregation operators. • The array update modifiers: **$each**, **$slice**, **$sort**. You can perform the following new tasks that extend MongoDB functionality in your JSON application: • Import and export data directly with the wire listener by using the Informix JSON commands **exportCollection** and **importCollection**. • Configure a strategy for calculating the size of your database by using the Informix extension to the MongoDB **listDatabases** command: **sizeStrategy** option or **command.listDatabases.sizeStrategy** property. You can customize the behavior of the wire listener by setting new properties. For example, you can control logging, caching, timeout, memory pools, and the maximum size of documents.	"Database commands" on page 4-4 "Query and projection operators" on page 4-18 "Update operators" on page 4-20 "Aggregation framework operators" on page 4-22 "Informix JSON commands" on page 4-11 "The wire listener configuration file" on page 2-3
Access Informix from REST API clients You can now directly connect applications or devices that communicate through the REST API to Informix. You create connections by configuring the wire listener for the REST API. With the REST API, you can use MongoDB and SQL queries against JSON and BSON document collections, traditional relational tables, and time series data. The REST API uses MongoDB syntax and returns JSON documents.	Chapter 5, "REST API," on page 5-1
Create a time series with the REST API or the MongoDB API If you have applications that handle time series data, you can now create and manage a time series with the REST API or the MongoDB API. Previously, you created a time series by running SQL statements. For example, you can program sensor devices that do not have client drivers to load time series data directly into the database with HTTP commands from the REST API. You create time series objects by adding definitions to time series collections. You interact with time series data through a virtual table.	Chapter 6, "Create time series through the wire listener," on page 6-1

Table 5. What's new in JSON for IBM Informix Version 12.10.xC3

Overview	Reference
Use the Mongo API to access relational data You can write a hybrid MongoDB application that can access both relational data and JSON collections that are stored in Informix. You can work with records in SQL tables as though they were documents in JSON collections by either referencing the tables as you would collections, or by using the **$sql** operator on an abstract collection.	Chapter 1, "About the Informix JSON compatibility," on page 1-1 "Running SQL commands by using a MongoDB API" on page 2-40 "Running MongoDB operations on relational tables" on page 2-42

Table 5. What's new in JSON for IBM Informix Version 12.10.xC3 (continued)

Overview	Reference
Improved JSON compatibility Informix now supports the following MongoDB features: • The **findAndModify** command, which performs multiple operations at the same time. • The MongoDB authentication methods for adding users and authenticating basic roles, such as read and write permissions for database and system level users.	"Collection methods" on page 4-1 "Database commands" on page 4-4 "The wire listener configuration file" on page 2-3

Java technology dependencies

IBM Informix software supports Java™ Platform Standard Edition (Java SE) to create and run Java applications, including user-defined routines (UDRs). Java SE 7 is supported as of Informix 12.10.xC5, while Java SE 6 is supported in earlier fix packs.

Important:

- Check the machine notes to learn about Java technology exceptions and other requirements for specific operating system platforms. The machine notes are available on the product media and in the online release information.
- In general, any application that ran correctly with earlier versions of Java technology will run correctly with this version. If you encounter problems, recompile the application with the next available fix pack or version. However, because there are frequent Java fixes and updates, not all of them are tested.
- To develop Java UDRs for the database server, use the supported Java software development kit or an earlier version according to Java compatibility guidelines. The supported version provides a known and reliable Java environment for UDRs in this database server release.

For details about Java requirements, check the following sections:

"Java runtime environment"

"Software development kit for Java" on page xiv

"Java Database Connectivity (JDBC) specification" on page xiv

Java runtime environment

On most supported operating system platforms, the Informix installation application bundles a Java runtime environment that it requires. However, check the machine notes for your operating system platform to determine whether the installation application requires a particular Java runtime environment to be preinstalled.

Also, IBM Runtime Environment, Java Technology Edition is supported for general use of the database server. It is installed on most operating system platforms by default in the following directory: `$INFORMIXDIR/extend/krakatoa/jre/`.

MongoDB API and REST API access supports IBM Runtime Environment, Java Technology Edition, Version 7.

Software development kit for Java

The following products and components require a software development kit for Java, but one is not installed:

- Informix DataBlade® Developers Kit (DBDK)
- IBM Informix JDBC Driver
- J/Foundation component
- Spatial Java API
- TimeSeries Java API

The software development kit that you use must be compatible with the supported Java runtime environment. Informix does not support OpenJDK. You can download a development kit from the following web sites:

- **Recommended for AIX and Linux:** IBM SDK, Java Technology Edition (http://www.ibm.com/developerworks/java/jdk/)
- **Recommended for HP-UX:** HP-UX 11i Java Development Kit for the Java 2 Platform Standard Edition (https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=HPUXJAVAHOME)
- Oracle Java Platform, Standard Edition Development Kit (JDK) (http://www.oracle.com/technetwork/java/javase/downloads/index.html)

Java Database Connectivity (JDBC) specification

Informix products and components support the Java Database Connectivity (JDBC) 3.0 specification.

Example code conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept that is being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional documentation

Documentation about this release of IBM Informix products is available in various formats.

You can access Informix technical information such as information centers, technotes, white papers, and IBM Redbooks® publications online at http://www.ibm.com/software/data/sw-library/.

Compliance with industry standards

IBM Informix products are compliant with various standards.

IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

How to read the syntax diagrams

Syntax diagrams use special components to describe the syntax for SQL statements and commands.

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

The double right arrowhead and line symbol ►►— indicates the beginning of a syntax diagram.

The line and single right arrowhead symbol —► indicates that the syntax is continued on the next line.

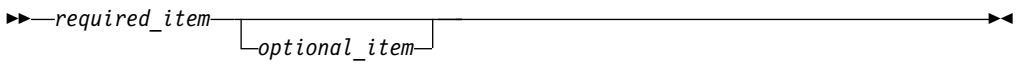
The right arrowhead and line symbol ►— indicates that the syntax is continued from the previous line.

The line, right arrowhead, and left arrowhead symbol —►◄ symbol indicates the end of a syntax diagram.

Syntax fragments start with the pipe and line symbol |— and end with the —| line and pipe symbol.

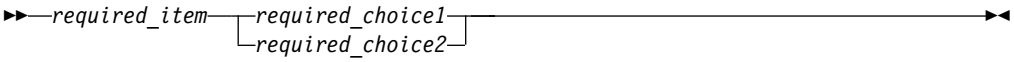
Required items appear on the horizontal line (the main path).

Optional items appear below the main path.

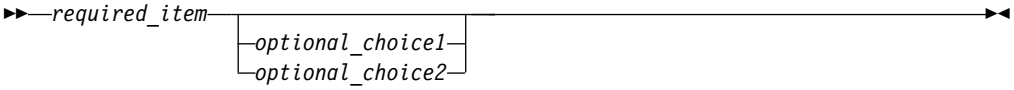


If you can choose from two or more items, they appear in a stack.

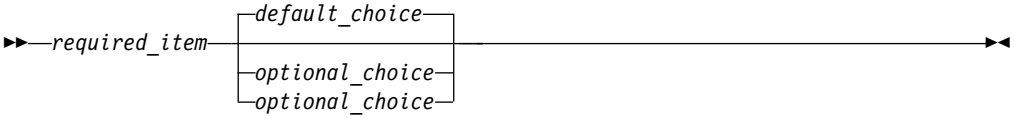
If you *must* choose one of the items, one item of the stack appears on the main path.



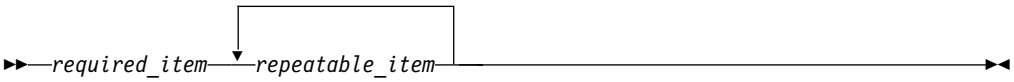
If choosing one of the items is optional, the entire stack appears below the main path.



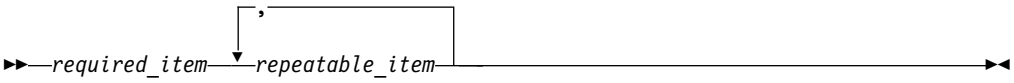
If one of the items is the default, it will appear above the main path, and the remaining choices will be shown below.



An arrow returning to the left, above the main line, indicates an item that can be repeated. In this case, repeated items must be separated by one or more blanks.



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 make more than one choice from the stacked items or repeat a single choice.

SQL keywords appear in uppercase (for example, `FROM`). They must be spelled exactly as shown. Variables appear in lowercase (for example, `column-name`). They represent user-supplied names or values in the syntax.

If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Sometimes a single variable represents a syntax segment. For example, in the following diagram, the variable `parameter-block` represents the syntax segment that is labeled **parameter-block**:

parameter-block:

How to provide documentation feedback

You are encouraged to send your comments about IBM Informix product documentation.

Add comments about documentation to topics directly in IBM Knowledge Center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more!

Feedback is monitored by the team that maintains the user documentation. The comments are reserved for reporting errors and omissions in the documentation. For immediate help with a technical problem, contact IBM Software Support at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Chapter 1. About the Informix JSON compatibility

You can combine relational and JSON data into a single query by using the Informix JSON compatibility features.

Applications that use the JSON-oriented query language can interact with relational and non-relational data that is stored in Informix databases by using the wire listener. The Informix database server also provides built-in JSON and BSON (binary JSON) data types. You can use MongoDB community drivers and the REST API to insert, update, and query JSON documents in Informix.

With Informix, you can use both SQL and MongoDB drivers to access SQL tables, JSON collections, time series data, and WebSphere® MQ data. You can join two JSON collections with each other or with relational tables.

Table 1-1. Relational data and JSON collection access by API type

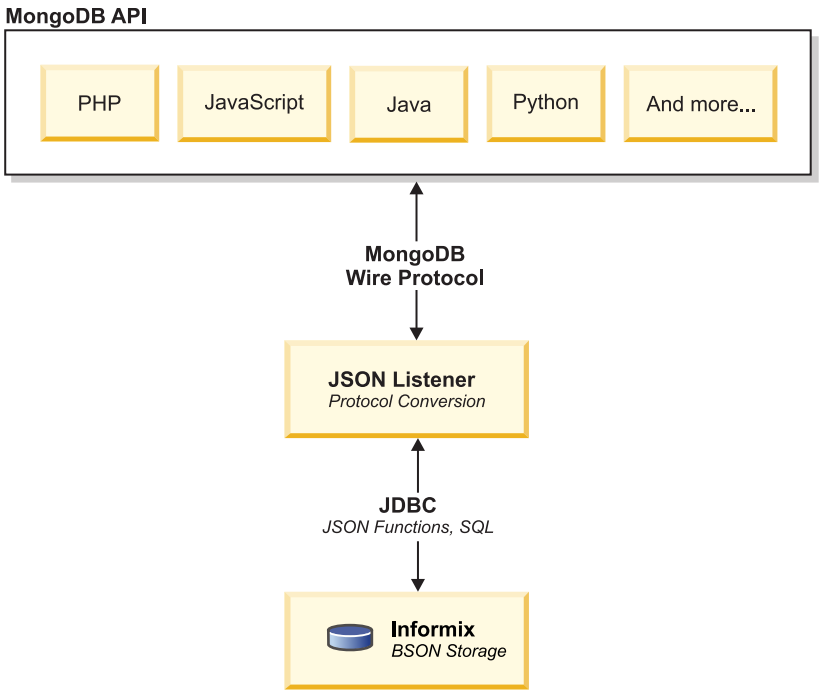
API type	Relational table access	JSON collection access
SQL API	Uses SQL language and standard ODBC, JDBC.NET, OData, and so on.	Uses direct SQL access, dynamic views, and row types.
MongoDB API	Uses MongoDB APIs for Java, JavaScript, C++, C#, and so on.	Uses MongoDB APIs for Java, JavaScript, C++, C#, and so on.

The JSON document format provides a way to transfer object information in a way that is language neutral, similar to XML. Language-neutral data transmission is a requirement for working in a web application environment, where data comes from various sources and software is written in various languages. With Informix, you can choose which parts of your application data are better suited unstructured, non-relational storage, and which parts are better suited in a traditional relational framework.

You can enable dynamic scaling and high-availability for data-intensive applications by taking the following steps:

- Define a sharded cluster to easily add or remove servers as your requirements change.
- Use shard keys to distribute subsets of data across multiple servers in a sharded cluster.
- Query the correct servers in a sharded cluster and return the consolidated results to the client application.
- Use secondary servers (similar to subordinates in MongoDB) in the sharded cluster to maximize availability and throughput. Secondary servers also have update capability.

You can choose to authenticate MongoDB clients in the wire listener with a MongoDB authentication method, or in the database server, with a pluggable authentication module.



Getting started with Informix JSON

You can begin using the Informix JSON features after installing Informix.

If you create the Informix server instance as a part of your installation, the wire listener is automatically started and connected to the MongoDB API and the database server with the default operational instance. You can use the MongoDB shell and any of the standard MongoDB command utilities and tools. To use the REST API, you must modify the default configuration.

If you create the Informix server instance outside of the installation process, you must configure and start the wire listener manually.

Related concepts:

Chapter 4, "MongoDB API and commands," on page 4-1

Chapter 5, "REST API," on page 5-1

Related tasks:

"Modifying the wire listener configuration file" on page 2-36

"Configuring the wire listener for the first time" on page 2-1

Software dependencies for JSON compatibility

Informix JSON compatibility is based on MongoDB version 2.4, 2.6, and 3.0, and has specific software dependencies.

Informix JSON compatibility requires IBM Informix version 12.10.xC2 or later, with the J/Foundation component, which enables services that use Java.

You must use a supported Java runtime environment .

You set the version of the MongoDB API that the wire listener uses by setting the **mongo.api.version** parameter in the wire listener configuration file. The MongoDB API version affects the type of authentication that you can use. For example, MongoDB version 3.0 supports the MongoDB SCRAM-SHA-1 authentication method, but does not support database server authentication or connections with the REST API.

MongoDB to Informix term mapping

The commonly used MongoDB terminology and concepts are mapped to the equivalent Informix terminology and concepts.

The following table provides a summary of commonly used MongoDB terms and their Informix conceptual equivalents.

Table 1-2. MongoDB concepts mapped to one or more Informix concepts.

MongoDB concept	Informix concept	Description
collection	table	This is the same concept. In Informix this type of collection is sometimes referred to as a JSON collection. A JSON collection is similar to a relational database table, except it does not enforce a schema.
document	record	This is the same concept. In Informix, this type of document is sometimes referred to as a JSON document.
field	column	This is the same concept.
master / slave	primary server / secondary server	This is the same concept. However, Informix secondary servers have additional capabilities. For example, data on a secondary server can be updated and propagated to primary servers.
replica set	high-availability cluster	This is the same concept. However, when the replica set is updated, it is then sent to all servers, not only to the primary server.
sharded cluster	shard cluster	This is the same concept. In Informix, a shard cluster is a group of servers (sometimes called shard servers) that contain sharded data.
shard key	shard key	This is the same concept.

Support for dots in field names

Unlike MongoDB, which does not allow dots, (.), in JSON or BSON field names, IBM Informix conforms to the JSON standard and allows dots. For example: {"user.fn" : "Jake"}. However, you cannot run a query or an operation directly on a field that has a dot in its name. In queries, a dot in between field names indicates a hierarchy.

Here the rules of using field names with dots in them with Informix:

- You can insert a document that has a field name with a dot in it. You do not get an error.
- You cannot use a field name with a dot in it in a query or operation. Informix ignores the field. The query does not return the matching document. The operation does not affect the value of the field.
- You can return a document that includes a field name with a dot in it by querying on a field name in the same document that does not have a dot in it.

Allowing dots in field names is useful when you do not have control over the field names because your data comes from external sources, for example, the Google API. You still want to store those documents in your database, even though some fields might have dots in their names.

The following examples to illustrate how dots in field names work in Informix. The table name is **tab1** and the column that contains JSON data is named **data**.

Suppose that you have the following document:

```
{user : {fn : "Bob", ln : "Smith"}, "user.fn" : "Jake"}
```

You run the following statement to update a field:

```
SELECT data::json FROM tab1 WHERE BSON_UPDATE(data, '$set : {"user.fn" :
      "John:}}');
```

The following document is returned:

```
{user : {fn : "John", ln : "Smith"}, "user.fn" : "Jake"}
```

The value of the **fn** field that is in a subdocument to the **user** field is updated. The value of the **user.fn** field is not updated, but the value is returned. You cannot update the value of a field with a dot in its name, but you can retrieve the value.

Suppose that you have the following document:

```
{"user.firstname" : "Jake"}
```

You run this query to return the value of the **user.firstname** field:

```
SELECT data::json FROM tab1 WHERE BSON_KEYS_EXIST(data,
      "user.firstname");
```

No documents are returned.

If you have documents where all the fields have dots in their names, you must run a query to return all documents in the database to see them: for example:

```
SELECT data::json FROM tab1;
```

Manipulate BSON data with SQL statements

As an alternative to using the MongoDB API, you can use Informix SQL to manipulate BSON data. However, if you plan to query JSON and BSON data through the wire listener, you must create your database objects, such as collections and indexes, through the wire listener. You can use SQL statements to query JSON and BSON data whether you created your database objects through the wire listener or with SQL statements.

You might have an existing application on relational tables that uses SQL to access the data, but you want to add BSON data to your database. You can create a table with a BSON column, insert the data, and manipulate the data with SQL statements. BSON documents that you insert through SQL statements or Informix utilities do not contain generated ObjectId field-value pairs or other MongoDB metadata.

Alternatively, you might use a MongoDB client for daily data processing, but need the querying capabilities of SQL for data analysis. For example, you can use SQL statements to join tables that have BSON columns with other tables based on BSON field values. You can create views that have columns of BSON field values. You can run warehouse queries on BSON data with Informix Warehouse Accelerator. If you have spatial, time series, or spatiotemporal data, you can use the corresponding specialized SQL routines to analyze the data.

You can use BSON processing functions to manipulate BSON data in SQL statements. The BSON value functions convert BSON field values to standard SQL data types, such as INTEGER and LVARCHAR. The BSON_GET and BSON_UPDATE functions manipulate field-value pairs. You can convert all or part of a relational table to a BSON document with the genBSON function.

Example: Using SQL to query a collection

In the following example, a table that is named **people** is created with **names** and **ages** fields that are inserted by using the interactive JavaScript shell interface to MongoDB:

```
db.createCollection("people");
db.people.insert({"name":"Anne","age":31});
db.people.insert({"name":"Bob","age":39});
db.people.insert({"name":"Charlie","age":29});
```

For SQL statements, the table name is **people** and the BSON column name is **data**. When you create a collection through a MongoDB API command, the name of the BSON column is set to **data**.

The following statement selects the **name** and **age** fields with dot notation and displays the results in a readable format by casting the results to JSON:

```
> SELECT data.name::JSON, data.age::JSON FROM people;

(expression)  {"name":"Anne"}
(expression)  {"age":31}

(expression)  {"name":"Bob"}
(expression)  {"age":39}

(expression)  {"name":"Charlie"}
(expression)  {"age":29}

3 row(s) retrieved.
```

Related information:
BSON and JSON built-in opaque data types
BSON processing functions

Chapter 2. Wire listener

The wire listener is a mid-tier gateway server that enables communication between MongoDB applications and the Informix database server.

The wire listener is a Java application and is provided as an executable JAR file, `$INFORMIXDIR/bin/jsonListener.jar`, that is included with the database server. The JAR file provides access to the MongoDB API and REST API.

MongoDB API access
You can connect to a JSON collection with the MongoDB API by using the MongoDB Wire Protocol.

When a MongoDB client is connected to the wire listener and requests a connection to a database, the wire listener creates a connection.

REST API access
You can connect to a JSON collection by using the REST API.

When a client is connected to the wire listener by using the REST API, each database is registered. The wire listener registers to receive session events such as create or drop a database. If a REST request refers to a database that exists but is not registered, the database is registered and a redirect to the root of the database is returned.

The wire listener connection properties file, named `jsonListener.properties` by default, defines every operational characteristic.

When you create a database or a table through the wire listener, automatic location and fragmentation are enabled. Databases are stored in the dbspace that is chosen by the server. Tables are fragmented among dbspaces that are chosen by the server. More fragments are added when tables grow.

The default logging mechanism for the wire listener is Logback. Logback is pre-configured and installed along with the JSON components.

Related information:

SQL administration API portal: Arguments by privilege groups

Managing automatic location and fragmentation

Configuring the wire listener for the first time

You must configure the wire listener by specifying an authorized user and customizing the wire listener configuration file.

Before you begin

The wire listener JAR file is included in the database server installation.

About this task

If you create a server instance as a part of the Informix installation process, the wire listener is configured with default properties and started:

- A wire listener configuration file, `$INFORMIXDIR/etc/jsonListener.properties`, is created.

- The **ifxjson** user, which has REPLICATION privilege group access, is created and added to the **url** parameter in the wire listener configuration file. This user ID is used by the wire listener to connect to Informix.
- The wire listener is started and connected to the MongoDB API and the database server.

If you want to use the REST API, or make other changes, edit the wire listener configuration file and restart the wire listener.

Procedure

To configure the wire listener for the first time:

1. Choose an authorized user. An authorized user is required in wire listener connections to the database server. The authorized user must have access to the databases and tables that are accessed through the MongoDB API and REST API.
   - **Windows:** Specify an operating system user.
   - **UNIX or Linux**: Specify an operating system user or a database user. For example, here is the argument to create a database user in UNIX or Linux:

     ```
     CREATE USER userID WITH PASSWORD 'password' ACCOUNT unlock PROPERTIES
      USER daemon;
     ```
2. Optional: If you want to shard data, grant the user REPLICATION privilege by running the **admin** or **task** SQL administration API command with the **grant admin** argument. The **ifxjson** user has REPLICATION privilege. For example:

   ```
   EXECUTE FUNCTION task('grant admin','userID','replication');
   ```
3. Create a wire listener configuration file in `$INFORMIXDIR/etc` with the `.properties` file extension. You can use the `$INFORMIXDIR/etc/jsonListener-example.properties` file as a template. For more information, see "The wire listener configuration file" on page 2-3.
4. Customize the wire listener configuration file to your needs. To include parameters in the wire listener, uncomment the row and customize the parameter. The **url** parameter is required. All other parameters are optional.

   Tip: Review the defaults for the following parameters and verify that they are appropriate for your environment: **mongo.api.verion**,**authentication.enable**, **listener.type**, **listener.port**, and **listener.hostName**.
5. If you are using a Dynamic Host Configuration Protocol (DHCP) on your IPv6 host, you must verify that the connection information between JDBC and Informix is compatible.

   For example, you can connect from the IPv6 host through an IPv4 connection by using the following steps:

   a. Add a server alias to the DBSERVERALIASES configuration parameter for the wire listener on the local host. For example: `lo_informix1210`.

   b. Add an entry to the `sqlhosts` file for the database server alias to the loopback address `127.0.0.1`. For example:

      ```
      lo_informix1210 onsoctcp 127.0.0.1 9090
      ```

   c. In the wire listener configuration file, update the **url** entry with the wire listener alias. For example:

      ```
      url=jdbc:informix-sqli://localhost:9090/sysmaster:
      INFORMIXSERVER=lo_informix1210;
      ```

What to do next

Start the wire listener.

Related concepts:

Chapter 3, "JSON data sharding," on page 3-1

Related tasks:

"Running SQL commands by using a MongoDB API" on page 2-40

Related information:

CREATE USER statement (UNIX, Linux)

grant admin argument: Grant privileges to run SQL administration API commands

What is JDBC?

The wire listener configuration file

The settings that control the wire listener and the connection between the client and database server are set in the wire listener configuration file.

The default name for the configuration file is `$INFORMIXDIR/etc/jsonListener.properties`. You can rename this file, but the suffix must be `.properties`.

If you create a server instance during the installation process, a configuration file that is named `jsonListener.properties` is automatically created with default properties, otherwise you must manually create the configuration file. You can use the `$INFORMIXDIR/etc/jsonListener-example.properties` file as a template.

In the configuration file that is created during installation, and in the template file, all of the parameters are commented out by default. To enable a parameter, you must uncomment the row and customize the parameter.

Important: The **`url`** parameter is required. All other parameters are optional.

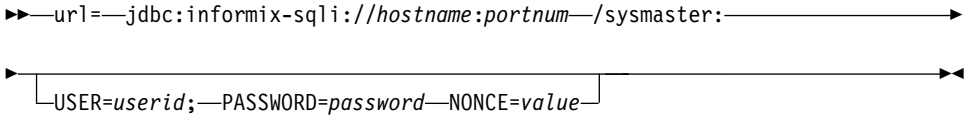
- Required
  - "url" on page 2-6
- Setup and configuration
  - "documentIdAlgorithm" on page 2-6
  - "include" on page 2-7
  - "listener.onException" on page 2-7
  - "listener.hostName" on page 2-8
  - "listener.port" on page 2-8
  - "listener.type" on page 2-8
  - "response.documents.count.default" on page 2-8
  - "response.documents.count.maximum" on page 2-8
  - "response.documents.size.maximum" on page 2-9
  - "sharding.enable" on page 2-9
  - "sharding.parallel.query.enable" on page 2-9
- Command and operation configuration
  - "collection.informix.options" on page 2-9
  - "command.listDatabases.sizeStrategy" on page 2-10

Required parameter

You must configure the **url** parameter before using the wire listener.

url
This required parameter specifies the host name, database server, user ID, and password that are used in connections to the database server.

You must specify the **sysmaster** database in the **url** parameter. That database is used for administrative purposes by the wire listener.



You can include additional JDBC properties in the **url** parameter such as INFORMIXCONTIME, INFORMIXCONRETRY, LOGINTIMEOUT, and IFX_SOC_TIMEOUT. For a list of Informix environment variables that are supported by the JDBC driver, see Informix environment variables with the IBM Informix JDBC Driver.

hostname*:*portnum
The host name and port number of your computer. For example, `localhost:9090`.

USER=*userid*
This optional attribute specifies the user ID that is used in connections to the Informix database server. If you plan to use this connection to establish or modify collection shards by using the Informix sharding capability, the specified user must be granted the REPLICATION privilege group access.

If you do not specify the user ID and password, the JDBC driver uses operating system authentication and all wire listener actions are run by using the user ID and password of the operating system user who runs the wire listener **start** command.

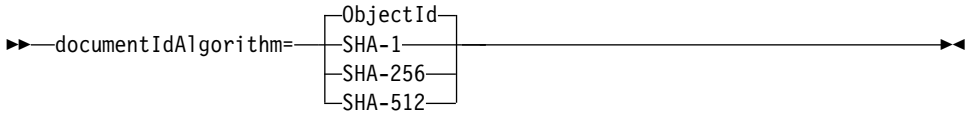
PASSWORD=*password*
This optional attribute specifies the password for the specified user ID.

NONCE=*value*
This optional attribute specifies a 16-character value that consists of numbers and the letters a, b, c, d, e, and f. This property triggers password encoding when a pluggable authentication module is configured for the wire listener. Applicable only if the **db.authentication** parameter is set to **informix-mongodb-cr**.

Setup and configuration

These parameters provide setup and configuration options.

documentIdAlgorithm
This optional parameter determines the algorithm that is used to generate the unique Informix identifier for the ID column that is the primary key on the collection table. The _id field of the document is used as the input to the algorithm. The default value is `documentIdAlgorithm=ObjectId`.



ObjectId
Indicates that the string representation of the ObjectId is used if the _id field is of type ObjectId; otherwise, the MD5 algorithm is used to compute the hash of the contents of the _id field.

- The string representation of an ObjectId is the hexadecimal representation of the 12 bytes that comprise an ObjectId.
- The MD5 algorithm provides better performance than the secure hashing algorithms (SHA).

ObjectId is the default value and it is suitable for most situations.

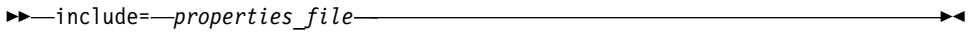
Important: Use the default unless a unique constraint violation is reported even though all documents have a unique _id field. In that case, you might need to use a non-default algorithm, such as SHA-256 or SHA-512.

SHA-1
Indicates that the SHA-1 hashing algorithm is used to derive an identifier from the _id field.

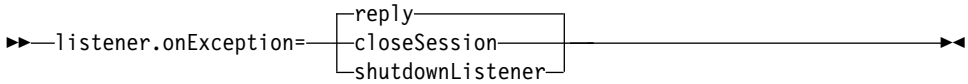
SHA-256
Indicates that the SHA-256 hashing algorithm is used to derive an identifier from the _id field.

SHA-512
Indicates that the SHA-512 hashing algorithm is used to derive an identifier from the _id field. This option generates the most unique values, but uses the most processor resources.

include
This optional parameter specifies the properties file to reference. The path can be absolute or relative. For more information, see "Running multiple wire listeners" on page 2-35.



listener.onException
This optional parameter specifies an ordered list of actions to take if an exception occurs that is not handled by the processing layer.



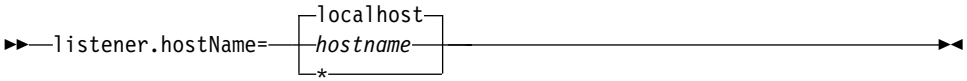
reply
When an unhandled exception occurs, reply with the exception message. This is the default value.

closeSession
When an unhandled exception occurs, close the session.

shutdownListener
When an unhandled exception occurs, shut down the wire listener.

listener.hostName
This optional parameter specifies the host name of the wire listener. The host name determines the network adapter or interface that the wire listener binds the server socket to.

Tip: If you enable the wire listener to be accessed by clients on remote hosts, turn on authentication by using the **authentication.enable** parameter.



localhost
Bind the wire listener to the localhost address. The wire listener is not accessible from clients on remote machines. This is the default value.

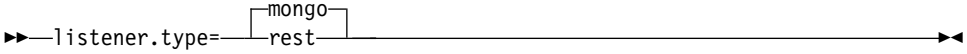
hostname
The host name or IP address of host machine where the wire listener binds to.

***** The wire listener can bind to all interfaces or addresses.

listener.port
This optional parameter specifies the port number to listen on for incoming connections from MongoDB clients. This value can be overridden from the command line by using the **-port** argument. The default value is 27017.

Important: If you specify a port number that is less than 1024, the user that starts the wire listener might require additional operating system privileges.

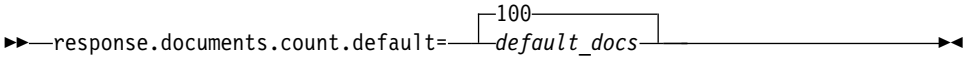
listener.type
This optional parameter specifies the type of wire listener to start.



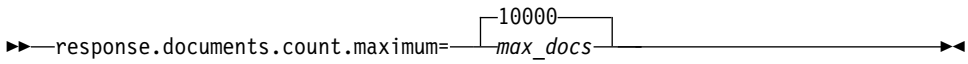
mongo
Connect the wire listener to the MongoDB API. This is the default value.

rest
Connect the wire listener to the REST API.

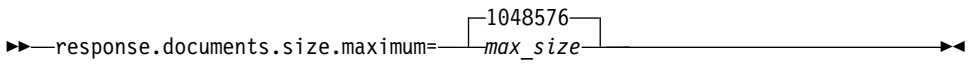
response.documents.count.default
This optional parameter specifies the default number of documents in a single response to a query. The default value is 100.



response.documents.count.maximum
This optional parameter specifies the maximum number of documents in a single response to a query. The default value is 10000.



response.documents.size.maximum
This optional parameter specifies the maximum size, in bytes, of all documents in a single response to a query. The default value is 1048576.

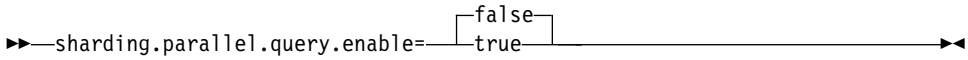


sharding.enable
This optional parameter indicates whether to enable the use of commands and queries on sharded data.

false
Do not enable the use of commands and queries on sharded data. This is the default value.

true
Enable the use of commands and queries on sharded data.

sharding.parallel.query.enable
This optional parameter indicates whether to enable the use of parallel sharded queries. Parallel sharded queries require that the SHARD_ID configuration parameter be set to unique IDs on all shard servers. The **sharding.enable** parameter must also be set to **true**.



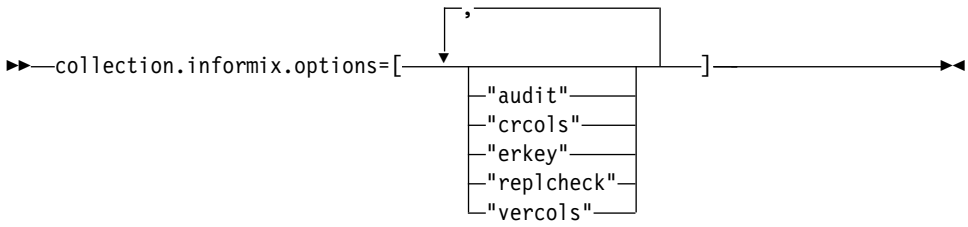
false
Do not enable parallel sharded queries. This is the default value.

true
Enable parallel sharded queries.

Command and operation configuration

These parameters provide configuration options for JSON commands and operations.

collection.informix.options
This optional parameter specifies which table options for shadow columns or auditing to use when creating a JSON collection.



audit
Use the AUDIT option of the CREATE TABLE statement to create a table to be included in the set of tables that are audited at the row level if selective row-level is enabled.

crcols
Use the CRCOLS option of the CREATE TABLE statement to create two shadow columns that Enterprise Replication uses for conflict resolution.

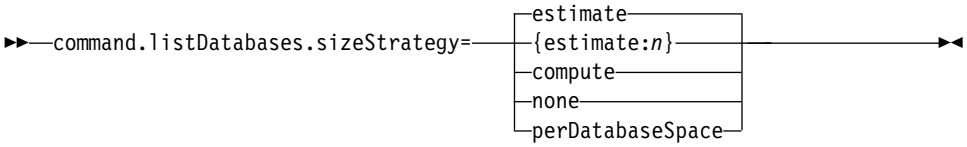
erkey
Use the ERKEY option of the CREATE TABLE statement to create the ERKEY shadow columns that Enterprise Replication uses for a replication key.

replcheck
Use the REPLCHECK option of the CREATE TABLE statement to create the **ifx_replcheck** shadow column that Enterprise Replication uses for consistency checking.

vercols
Use the VERCOLS option of the CREATE TABLE statement to create two shadow columns that Informix uses to support update operations on secondary servers.

command.listDatabases.sizeStrategy
This optional parameter specifies a strategy for calculating the size of your database when the MongoDB listDatabases command is run. The listDatabases command estimates the size of all collections and collection indexes for each database. However, relational tables and indexes are excluded from this size calculation.

Important: The MongoDB listDatabases command performs expensive and CPU-intensive computations on the size of each database in the database server instance. You can decrease the expense by using the **command.listDatabases.sizeStrategy** parameter.



estimate
Estimate the size of the database by sampling documents in every collection. This is the default value. This strategy is the equivalent of {estimate: 1000}, which takes a sample size of 0.1% of the documents in every collection. This is the default value.

```
command.listDatabases.sizeStrategy=estimate
```

estimate: *n*
Estimate the size of the database by sampling one document for every *n* documents in every collection. The following example estimates the collection size by using sample size of 0.5% or 1/200th of the documents:

```
command.listDatabases.sizeStrategy={estimate:200}
```

compute
Compute the exact size of the database.

```
command.listDatabases.sizeStrategy=compute
```

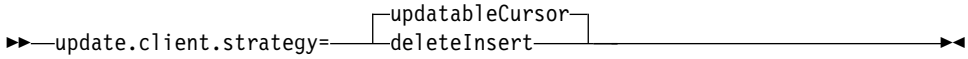
none
List the databases but do not compute the size. The database size is listed as 0.

```
command.listDatabases.sizeStrategy=none
```

perDatabaseSpace
Calculate the size of a database by adding the sizes for all dbspaces, sbspaces, and blobspaces that are assigned to the tenant database.

Important: The **perDatabaseSpace** option applies only to tenant databases that are created by the multi-tenancy feature.

update.client.strategy
This optional parameter specifies the method that is used by the wire listener to send updates to the database server. When the wire listener does the update processing, it queries the server for the existing document and then updates the document.

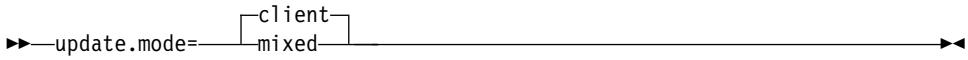


updatableCursor
Updates are sent to the database server by using an updatable cursor. This is the default value.

deleteInsert
The original document is deleted when the updated document is inserted.

Important: If the collection is sharded, you must use this method.

update.mode
This optional parameter determines where document updates are processed. The default value is `update.mode=client`.



client
Use the wire listener to process updates. For example, if your document updates are complicated or use document replacement, you can use **client** to process these updates by using the wire listener. This is the default value.

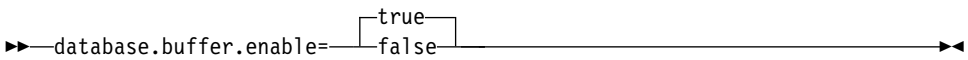
mixed
Attempt to process updates on the database server first, then fallback to the wire listener. For example, if your document updates consist mainly of single operation updates on a single field (for example, $set, $inc), you can use **mixed** to process these updates directly on the server.

Database resource management

These parameters provide database resource management options.

database.buffer.enable

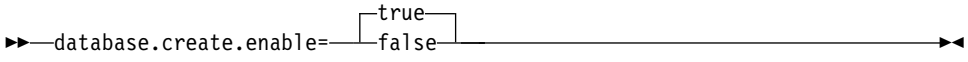
Prerequisite: `database.log.enable=true`
This optional parameter indicates whether to enable buffered logging when you create a database by using the wire listener.



true
Enable buffered logging. This is the default value.

false
Do not enable buffered logging.

database.create.enable
This optional parameter indicates whether to enable the automatic creation of a database, if a database does not exist.

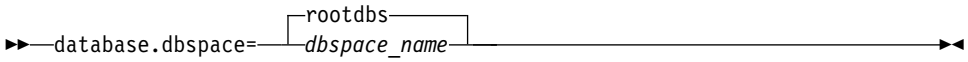


true
If a database does not exist, create a database. This is the default value.

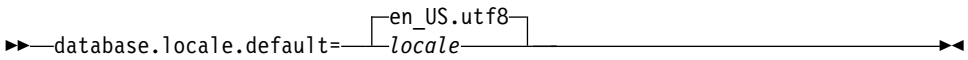
false
If a database does not exist, do not create a database. With this option, you can access only existing databases.

database.dbspace

Prerequisite: `dbspace.strategy=fixed`
This optional parameter specifies the name of the dbspace databases that are created. The default value is `database.dbspace=rootdbs`.



database.locale.default
This optional parameter specifies the default locale to use when a database is created by using the wire listener. The default value is en_US.utf8.



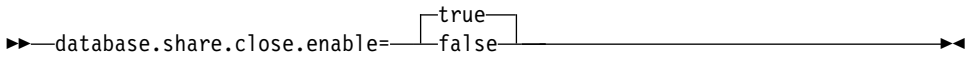
database.log.enable
This optional parameter indicates whether to create databases that are enabled for logging.

true
Create databases that are enabled for logging. This is the default value. Use the **database.buffer.enable** parameter to enable buffered logging.

false
Do not create databases that are enabled for logging.

database.share.close.enable

Prerequisite: `database.share.enable=true`
This optional parameter indicates whether to close a shared database and its associated resources, including connection pools, when the number of active sessions drops to zero.

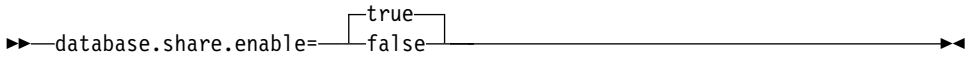


true
Close a shared database when the number of active sessions drops to zero. This is the default value.

false
Keep the shared database open when the number of active sessions drops to zero.

Important: If shared databases are enabled and this property is set to false, the connection pool associated with a database is never closed.

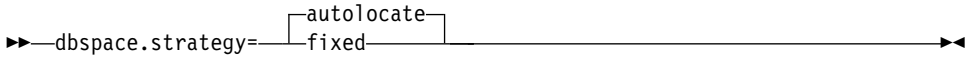
database.share.enable
This optional parameter indicates whether to share database objects and associated resources. For example, you can share connection pools between sessions.



true
Share database objects and associated resources. Use the **database.share.enable** parameter to control when to close the shared database. This is the default value.

false
Do not share database objects and associated resources.

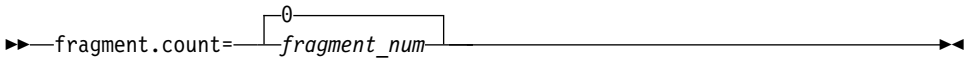
dbspace.strategy
This optional parameter specifies the strategy to use when determining the location of new databases, tables, and indexes.



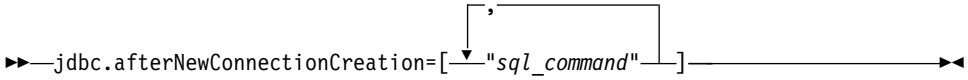
autolocate
The database server automatically determines the dbspace for the new databases, tables, and indexes. This is the default value.

fixed
Use a specific dbspace, as specified by the **database.dbspace** property.

fragment.count
This optional parameter specifies the number of fragments to use when creating a collection. If you specify 0, the database server determines the number of fragments to create. If you specify a *fragment_num* greater than 0, that number of fragments are created when the collection is created. The default value is 0.



jdbc.afterNewConnectionCreation
This optional parameter specifies one or more SQL commands to run after a new connection to the database is created.



For example, to accelerate queries run through the wire listener by using Informix Warehouse Accelerator:

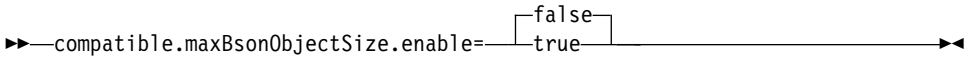
```
jdbc.afterNewConnectionCreation=["SET ENVIRONMENT USE_DWA 'ACCELERATE ON'"]
```

MongoDB compatibility

These parameters provide options for MongoDB compatibility.

compatible.maxBsonObjectSize.enable
This optional parameter indicates whether the maximum BSON object size is compatible with MongoDB.

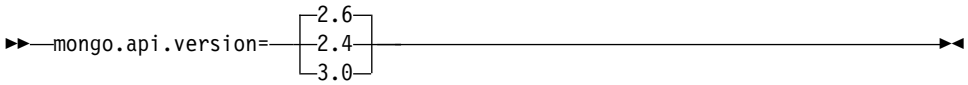
Tip: If you insert a BSON document by using an SQL operation, Informix supports a maximum document size of 2 GB.



false
Use a maximum document size of 256 MB with the wire listener. This is the default value.

true
Use a maximum document size of 16 MB. The maximum document size for MongoDB is 16 MB.

mongo.api.version
This optional parameter specifies the MongoDB API version with which the wire listener is compatible. The version affects authentication methods as well as MongoDB commands.



Important: Do not set **mongo.api.version=3.0** if you want to use the REST API or database server authentication. See "User authentication with the wire listener" on page 2-37.

update.one.enable
This optional parameter indicates whether to enable support for updating a single JSON document.

Important: The **update.one.enable** parameter applies to JSON collections only. For relational tables, the MongoDB multi-parameter is ignored and all documents that meet the query criteria are updated.

false
All collection updates are treated as multiple JSON document updates. This is the default value.

With the `update.one.enable=false` setting, the MongoDB **db.collection.update** multi-parameter is ignored and all documents that meet the query criteria are updated.

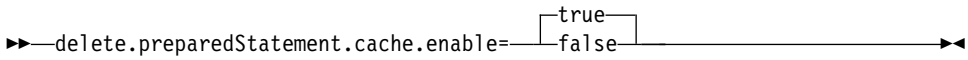
true
Allow updates on collections to a single document or multiple documents.

With the `update.one.enable=true` setting, the MongoDB **db.collection.update** multi-parameter is accepted. The **db.collection.update** multi-parameter controls whether you can update a single document or multiple documents.

Performance

These parameters provide performance options for databases and collections.

delete.preparedStatement.cache.enable
This optional parameter indicates whether to cache prepared statements that delete documents for reuse.



true
Use a prepared statement cache for statements that delete documents. This is the default value.

false
Do not use a prepared statement cache for statements that delete documents. A new statement is prepared for each query.

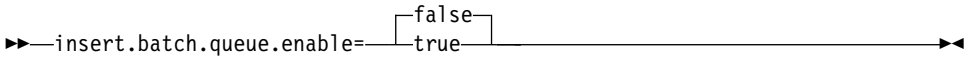
insert.batch.enable
If multiple documents are sent as a part of a single INSERT statement, this optional parameter indicates whether to batch document inserts operations into collections.

true
Batch document inserts into collections by using JDBC batch calls to perform the insert operations. This is the default value.

false
Do not batch document insert operations into collections.

insert.batch.queue.enable
This optional parameter indicates whether to queue INSERT statements into larger batches. You can improve insert performance by queuing INSERT statements, however, there is decreased durability.

This parameter batches all INSERT statements, even a single INSERT statement. These batched INSERT statements are flushed at the interval that is specified by the **insert.batch.queue.flush.interval** parameter, unless another operation arrives on the same collection. If another operation arrives on the same collection, the batch inserts are immediately flushed to the database server before proceeding with the next operation.

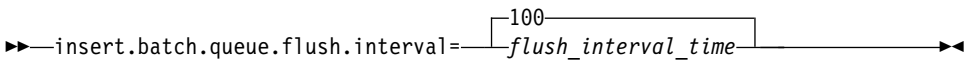


false
Do not queue INSERT statements. This is the default.

true
Queue INSERT statements into larger batches. Use the **insert.batch.queue.flush.interval** parameter to specify the amount of time between insert queue flushes.

insert.batch.queue.flush.interval

Prerequisite: `insert.batch.queue.enable=true`
This optional parameter specifies the number of milliseconds between flushes of the insert queue to the database server. The default value is `insert.batch.queue.flush.interval=100`.

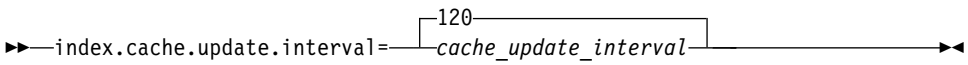


index.cache.enable
This optional parameter indicates whether to enable index caching on collections. To write the most efficient queries, the wire listener must be aware of the existing BSON indexes on your collections.

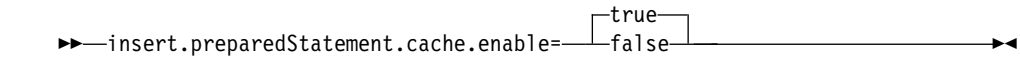
true
Cache indexes on collections. This is the default value.

false
Do not cache indexes on collections. The wire listener queries the database for indexes each time a collection query is translated to SQL.

index.cache.update.interval
This optional parameter specifies the amount of time, in seconds, between updates to the index cache on a collection table. The default value is `index.cache.update.interval=120`.



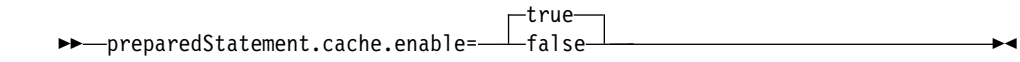
insert.preparedStatement.cache.enable
This optional parameter indicates whether to cache the prepared statements that are used to insert documents.



true
Cache the prepared statements that are used to insert documents. This is the default value.

false
Do not cache the prepared statements that are used to insert documents.

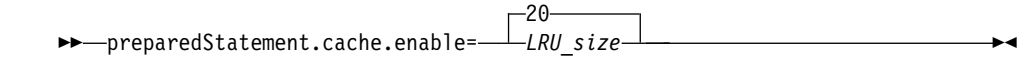
preparedStatement.cache.enable
This optional parameter indicates whether to cache prepared statements for reuse.



true
Use a prepared statement cache. This is the default value.

false
Do not use a prepared statement cache. A new statement is prepared for each query.

preparedStatement.cache.size
This optional parameter specifies the size of the least-recently used (LRU) map that is used to cache prepared statements. The default value is `preparedStatement.cache.size=20`.

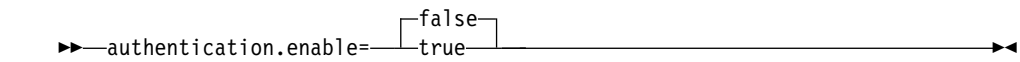


Security

The parameters provide security enablement options.

authentication.enable
This optional parameter indicates whether to enable user authentication.

You can choose to authenticate MongoDB clients in the wire listener with a MongoDB authentication method, or in the database server, with a pluggable authentication module.

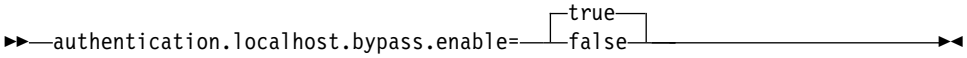


false
Do not authenticate users. This is the default value.

true
Authenticate users. Use the **authentication.localhost.bypass.enable** parameter to control the type of authentication.

authentication.localhost.bypass.enable

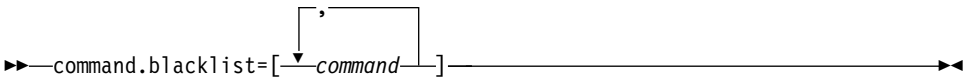
Prerequisite: `authentication.enable=true`
If you connect from the localhost to the Informix **admin** database, and the **admin** database contains no users, this optional parameter indicates whether to grant full administrative access. The Informix **admin** database is similar to the MongoDB admin database. The Informix **authentication.localhost.bypass.enable** parameter is similar to the MongoDB **enableLocalhostAuthBypass** parameter.



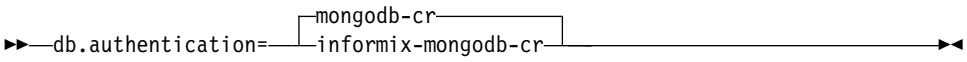
true
Grant full administrative access to the user. This is the default value.

false
Do not grant full administrative access to the user.

command.blacklist
This optional parameter lists commands that are removed from the command registry and cannot be called. By default, the black list is empty.



db.authentication
This optional parameter specifies the user authentication method. See "User authentication with the wire listener" on page 2-37.

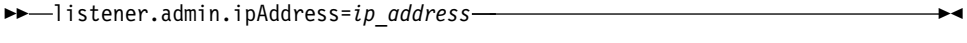


mongodb-cr
Authenticate through the wire listener with a MongoDB authentication method. The MongoDB authentication method depends on the setting of the **mongo.api.version** parameter.

informix-mongodb-cr
Authenticate through the database server with a pluggable authentication module.

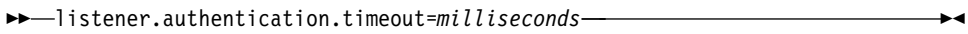
listener.admin.ipAddress
This optional parameter specifies the IP address for the administrative host. Must be a loopback IP address. The default value is 127.0.0.1.

Important: If you specify an address that is not a loopback IP address, an attacker might perform a remote privilege escalation and obtain administrative privileges without knowing a user password.

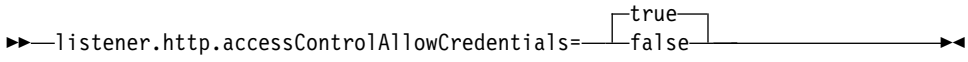


listener.authentication.timeout
This optional parameter specifies the number of milliseconds that the wire

listener waits for a client connection to authenticate. The default value is 0, which indicates that the wire listener waits indefinitely for client connections to authenticate.



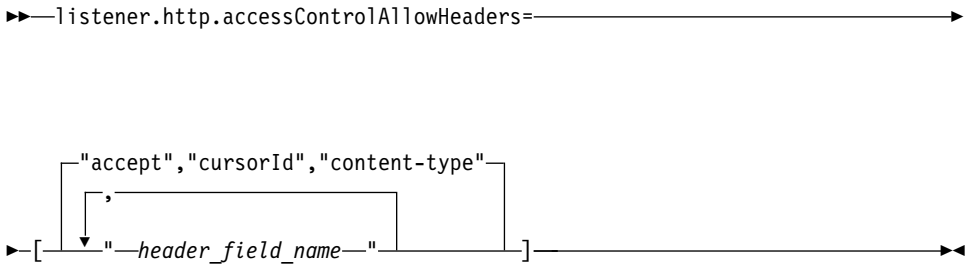
listener.http.accessControlAllowCredentials
This optional parameter indicates whether to display the response to the request when the omit credentials flag is not set. When this parameter is part of the response to a preflight request, it indicates that the actual request can include user credentials.



true
Display the response to the request. This is the default value.

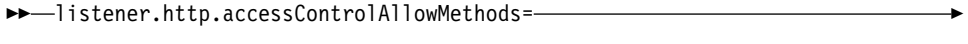
false
Do not display the response to the request.

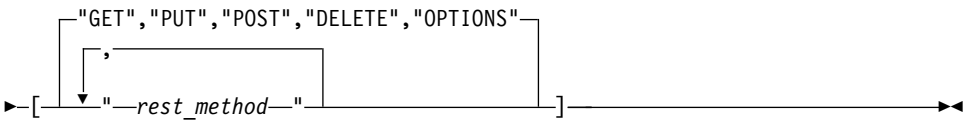
listener.http.accessControlAllowHeaders
This optional parameter, which is part of the response to a preflight request, specifies the header field names that are used during the actual request. You must specify the value by using a JSON array of strings. Each string in the array is the case-insensitive header field name. The default value is `listener.http.accessControlAllowHeaders=["accept","cursorId","content-type"]`.



For example, to allow the headers `foo` and `bar` in a request:
`listener.http.accessControlAllowHeaders=["foo","bar"]`

listener.http.accessControlAllowMethods
This optional parameter, which is part of the response to a preflight request, specifies the REST methods that are used during the actual request. You must specify the value by using a JSON array of strings. Each string in the array is the name of an HTTP method that is allowed. The default value is `listener.http.accessControlAllowMethods=["GET","PUT","POST","DELETE","OPTIONS"]`.



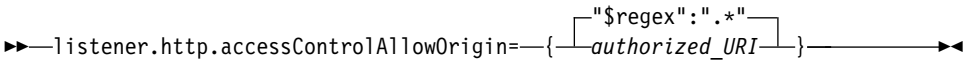


listener.http.accessControlAllowOrigin

This optional parameter specifies which uniform resource identifiers (URI) are authorized to receive responses from the REST listener when processing cross-origin resource sharing (CORS) requests. You must specify the value by using a JSON array of strings, with a separate string in the array for each value for the HTTP Origin header in a request. The values that are specified in this parameter are validated to ensure that they are identical to the Origin header.

HTTP requests include an Origin header that specifies the URI that served the resource that processes the request. When a resource from a different origin is accessed, the resource is validated to determine whether sharing is allowed.

The default value, `listener.http.accessControlAllowOrigin={"$regex":".*"}`, means that any origin is allowed to perform a CORS request.



Here are some usage examples:

- Grant access to the localhost:

  ```
  listener.http.accessControlAllowOrigin="http://localhost"
  ```

- Grant access to all hosts in the subnet 10.168.8.0/24. The first 3 segments are validated as 10, 168, and 8, and the fourth segment is validated as a value 1 - 255:

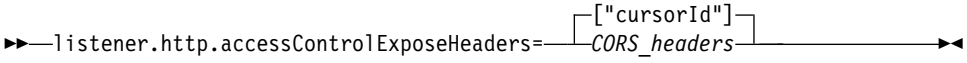
  ```
  listener.http.accessControlAllowOrigin={"$regex":"^
  http://10\\\\.168\\\\.8\\\\.([01]?\\\\
  d\\\\d?|2[0-4]\\\\d|25[0-5])$" }
  ```

- Grant access to all hosts in the subnet 10.168.8.0/24. The first 3 segments are validated as 10, 168, and 8, and the fourth segment must contain one or more digits:

  ```
  listener.http.accessControlAllowOrigin={"$regex":
  "^http://10\\\\.168\\\\.8\\\\.\\\\d+$" }
  ```

listener.http.accessControlExposeHeaders

This optional parameter specifies which headers of a CORS request to expose to the API. You must specify the value by using a JSON array of strings. Each string in the array is the case-insensitive name of a header to be exposed. The default value is `listener.http.accessControlExposeHeaders=["cursorId"]`.



For example, to expose the headers `foo` and `bar` to a client:

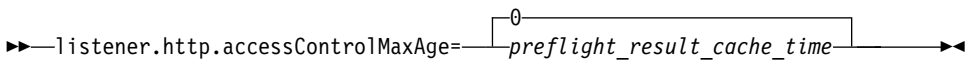
```
listener.http.accessControlExposeHeaders=["foo","bar"]
```

listener.http.accessControlMaxAge

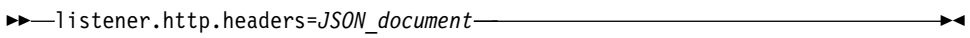
This optional parameter specifies the amount of time, in seconds, that the result of a preflight request is cached in a preflight result cache. A value of 0 indicates that the Access-Control-Max-Age header is not included in the

response to a preflight request. A value greater than 0 indicates that the Access-Control-Max-Age header is included in the response to a preflight request.

The default value is `listener.http.accessControlMaxAge=0`.



listener.http.headers
This optional parameter specifies the information to include in the HTTP headers of responses, as a JSON document. The default value is no additional information in the HTTP headers.



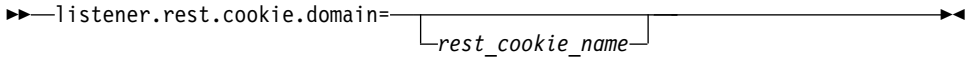
For example, you set this parameter to the following value:

```
listener.http.headers={ "Access-Control-Allow-Origin" : "http://192.168.0.1",
"Access-Control-Allow-Credentials" : "true" }
```

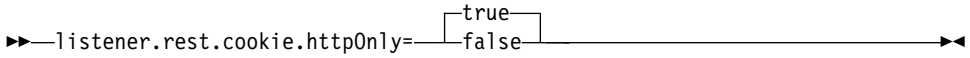
Then the HTTP headers for all responses look like this:

```
Access-Control-Allow-Origin : http://192.168.0.1
Access-Control-Allow-Credentials : true
```

listener.rest.cookie.domain
This optional parameter specifies the name of the cookie that is created by the REST wire listener. If not specified, the domain is the default value as determined by the Apache Tomcat web server.



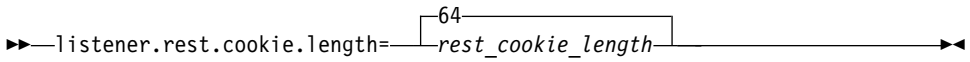
listener.rest.cookie.httpOnly
This optional parameter indicates whether to set the HTTP-only flag.



true
Set the HTTP-only flag. This flag helps to prevent cross-site scripting attacks. This is the default value.

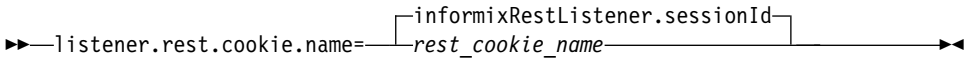
false
Do not set the HTTP-only flag.

listener.rest.cookie.length
This optional parameter specifies the length, in bytes, of the cookie value that is created by the REST wire listener, before Base64 encoding. The default value is `listener.rest.cookie.length=64`.

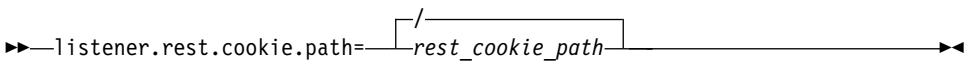


listener.rest.cookie.name
This optional parameter specifies the name of the cookie that is created by the

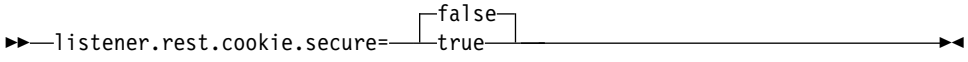
REST wire listener to identify a session. The default value is `listener.rest.cookie.name=informixRestListener.sessionId`.



listener.rest.cookie.path
This optional parameter specifies the path of the cookie that is created by the REST wire listener. The default value is `listener.rest.cookie.path=/`.



listener.rest.cookie.secure
This optional parameter indicates whether the cookies that are created by the REST wire listener have the secure flag on. The secure flag prevents the cookies from being used over an unsecure connection.

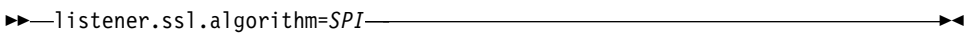


false
Turn off the secure flag. This is the default value.

true
Turn on the secure flag.

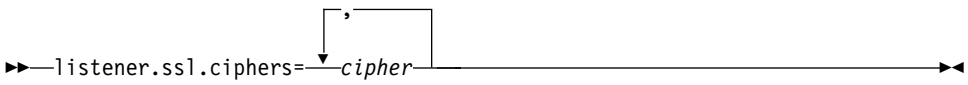
listener.ssl.algorithm
This optional parameter specifies the Service Provider Interface (SPI) for the KeyManagerFactory that is used to access the network encryption keystore. On an Oracle Java Virtual Machine (JVM), this value is typically SunX509. On an IBM JVM, this value is typically IbmX509. The default value is no SPI.

Important: Do not set this property if you are not familiar with Java Cryptography Extension (JCE).



listener.ssl.ciphers
This optional parameter specifies a list of Secure Sockets Layer (SSL) or Transport Layer Security (TLS) ciphers to use with network encryption. The default value is no ciphers, which means that the default list of enabled ciphers for the JVM are used.

Important: Do not set this property if you are not familiar with Java Cryptography Extension (JCE) and the implications of using multiple ciphers. Consult a security expert for advice.



You can include spaces between ciphers.

For example, you can set the following ciphers:

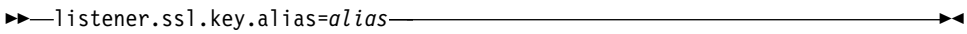
```
listener.ssl.ciphers=TLS_RSA_WITH_AES_128_CBC_SHA, TLS_RSA_WITH_AES_256_CBC_SHA,
TLS_EMPTY_RENEGOTIATION_INFO_SCSV
```

listener.ssl.enable
This optional parameter enables SSL or TLS network encryption on the socket for client connections.

false
Disable network encryption. This is the default.

true
Allow network encryption.

listener.ssl.key.alias
This optional parameter specifies the alias, or identifier, of the entry into the keystore. The default value is no alias, which indicates that the keystore contains one entry. If the keystore contain more than one entry and a key password is needed to unlock the keystore, set this parameter to the alias of the entry that unlocks the keystore.

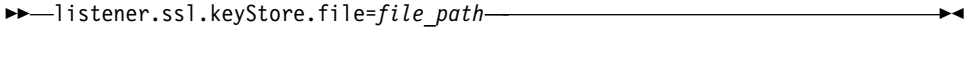


This parameter is effective when the **listener.ssl.enable** parameter is set to true.

listener.ssl.key.password
This optional parameter specifies the password to unlock the entry into the keystore, which is identified by the **listener.ssl.key.alias** parameter. The default value is no password, which means to use the keystore password. If the entry into the keystore requires a password that is different from the keystore password, set this parameter to the entry password.

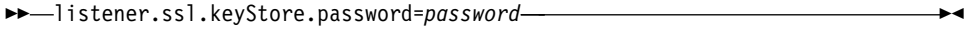
This parameter is effective when the **listener.ssl.enable** parameter is set to true.

listener.ssl.keyStore.file
This optional parameter specifies the fully-qualified path and file name of the Java keystore file to use for network encryption. The default value is no file.



This parameter is effective when the **listener.ssl.enable** parameter is set to true.

listener.ssl.keyStore.password
This optional parameter specifies the password to unlock the Java keystore file for network encryption. The default value is no password.



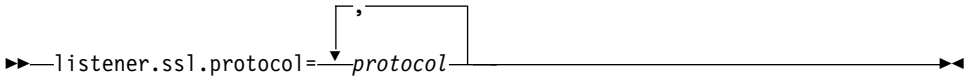
This parameter is effective when the **listener.ssl.enable** parameter is set to true.

listener.ssl.keyStore.type
This optional property specifies the provider identifier for the network encryption keystore SPI. The default value is JKS.

Important: Do not set this property if you are not familiar with Java Cryptography Extension (JCE).

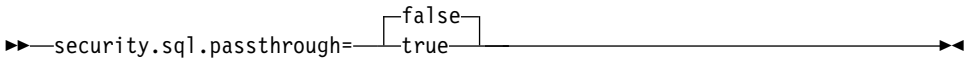
This parameter is effective when the **listener.ssl.enable** parameter is set to true.

listener.ssl.protocol
This optional parameter specifies the SSL or TLS protocols. The default value is TLS.



This parameter is effective when the **listener.ssl.enable** parameter is set to true.

security.sql.passthrough
This optional parameter indicates whether to enable support for issuing SQL statements by using JSON documents.



false
Disable the ability to issue SQL statements by using the MongoDB API. This is the default.

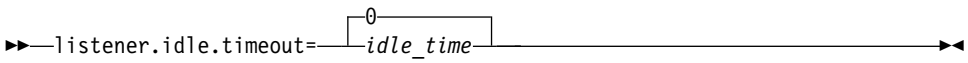
true
Allow SQL statements to be issued by using the MongoDB API.

Wire listener resource management

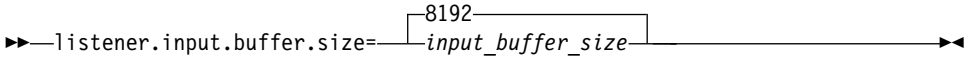
These parameters provide wire listener resource management options.

listener.idle.timeout
This optional parameter specifies the amount of time, in milliseconds, that a client connection to the wire listener can idle before it is forcibly closed. You can use this parameter to close connections and free associated resources when clients are idle. The default value is 0 milliseconds.

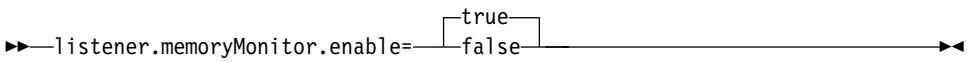
Important: When set to a nonzero value, the wire listener socket that is used to communicate with a MongoDB client is forcibly closed after the specified time. To the client, the forcible closure appears as an unexpected disconnection from the server the next time there is an attempt to write to the socket.



listener.input.buffer.size
This optional parameter specifies the size, in MB, of the input buffer for each wire listener socket. The default value is 8192 MB.



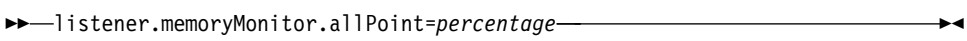
listener.memoryMonitor.enable
This optional parameter enables the wire listener memory monitor. When memory usage for the wire listener is high, the memory monitor attempts to reduce resources, such as removing cached JDBC prepared statements, removing idle JDBC connections from the connection pools, and reducing the maximum size of responses.



true
Enable the memory monitor. This is the default.

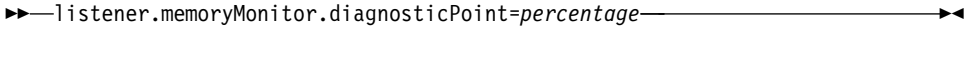
false
Disable the memory monitor.

listener.memoryMonitor.allPoint
This optional parameter specifies the maximum percentage of heap usage before the memory monitor reduces resources. The default value is 80.



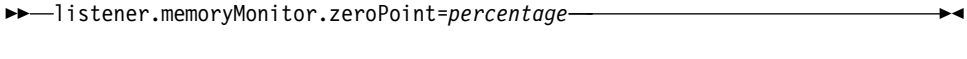
This parameter is effective when the **listener.memoryMonitor.enable** parameter is set to true.

listener.memoryMonitor.diagnosticPoint
This optional parameter specifies the percentage of heap usage before diagnostic information about memory usage is logged. The default value is 99.



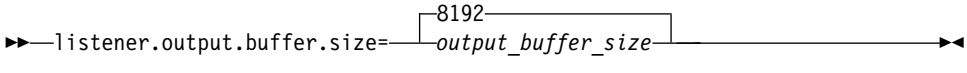
This parameter is effective when the **listener.memoryMonitor.enable** parameter is set to true.

listener.memoryMonitor.zeroPoint
This optional parameter specifies the percentage of heap usage before the memory manager reduces resource usage to the lowest possible levels. The default value is 95.



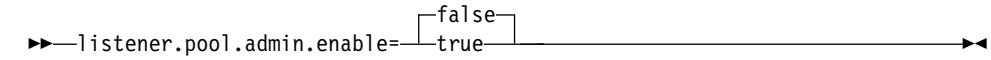
This parameter is effective when the **listener.memoryMonitor.enable** parameter is set to true.

listener.output.buffer.size
This optional parameter specifies the size, in MB, of the output buffer for each listener socket. The default value is 8192 MB.



listener.pool.admin.enable
This optional parameter enables a separate thread pool for connections from the administrative IP address, which is set by the **listener.admin.ipAddress**

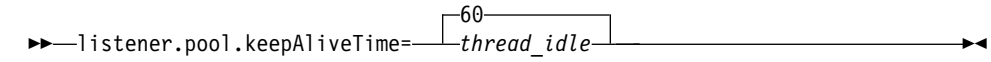
parameter. The default value is false. A separate thread pool ensures that administrative connections succeed even if the listener thread pool lacks available resources.



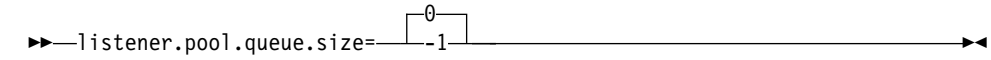
false
Prevents a separate thread pool. This is the default.

false
Creates a separate thread pool for administrative connections.

listener.pool.keepAliveTime
This optional parameter specifies the amount of time, in seconds, that threads above the core pool size are allowed to idle before they are removed from the wire listener JDBC connection pool. The default value is 60 seconds.



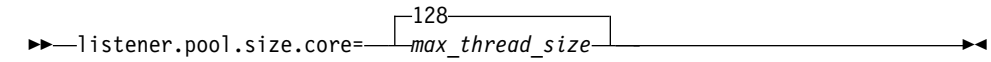
listener.pool.queue.size
This optional parameter specifies the number of requests to queue above the core wire listener pool size before expanding the pool size up to the maximum. A positive integer specifies the queue size to use before expanding the pool size up to the maximum.



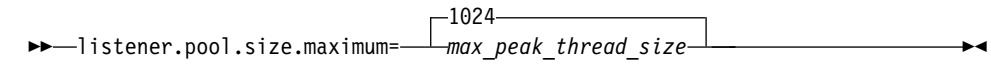
0 Do not allocate a queue size for tasks. All new sessions are either run on an available or new thread up to the maximum pool size, or are rejected if the maximum pool size is reached. This is the default value.

-1 Allocate an unlimited queue size for tasks.

listener.pool.size.core
This optional parameter specifies the maximum sustained size of the thread pool that listens for incoming connections from MongoDB clients. The default value is 128.



listener.pool.size.maximum
This optional parameter specifies the maximum peak size of the thread pool that listens for incoming connections from MongoDB clients. The default value is 1024.

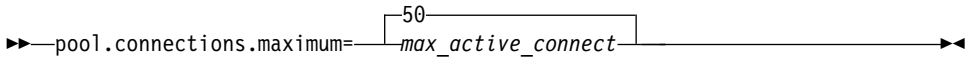


listener.socket.accept.timeout
This optional parameter specifies the number of milliseconds that a server

socket waits for an **accept()** function. The default value is 1024. The value of 0 indicates to wait indefinitely. The value of this parameter can affect how quickly the wire listener shuts down.

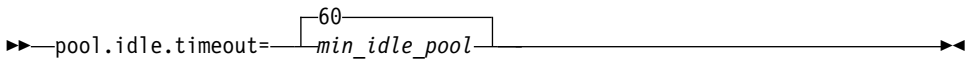
listener.socket.read.timeout
This optional parameter specifies the number of milliseconds to block when calling a **read()** function on the socket input stream. The default value is 1024. A value of 0 might prevent the wire listener from shutting down because the threads that poll the socket might never unblock.

pool.connections.maximum
This optional parameter specifies the maximum number of active connections to a database. The default value is 50.



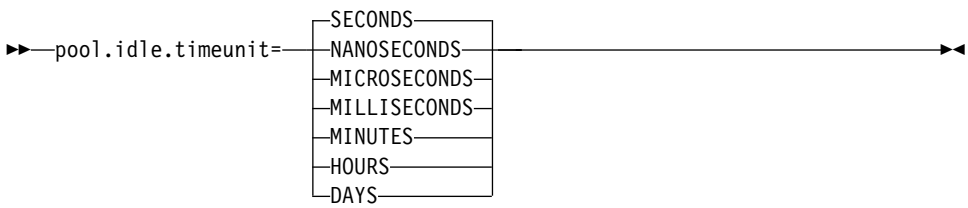
pool.idle.timeout
This optional parameter specifies the minimum amount of time that an idle connection is in the idle pool before it is closed. The default value is 60 and the default time unit is seconds.

Important: Set the unit of time in the **pool.idle.timeunit** parameter. The default value is seconds.



pool.idle.timeunit

Prerequisite: pool.idle.timeout=*time*
This optional parameter specifies the unit of time that is used to scale the **pool.idle.timeout** parameter.



SECONDS
Use seconds as the unit of time. This is the default value.

NANOSECONDS
Use nanoseconds as the unit of time.

MICROSECONDS
Use microseconds as the unit of time.

MILLISECONDS
Use milliseconds as the unit of time.

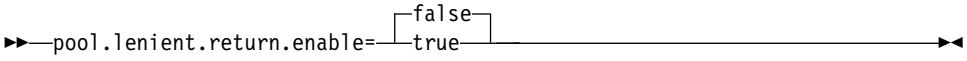
MINUTES
Use minutes as the unit of time.

HOURS
Use hours as the unit of time.

DAYS
Use days as the unit of time.

pool.lenient.return.enable
This optional parameter suppresses the following checks on a connection that is being returned that might throw exceptions:

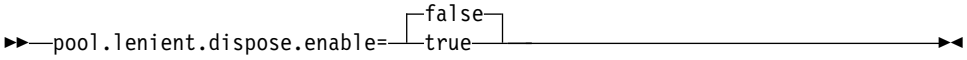
- An attempt to return a pooled connection that is already returned.
- An attempt to return a pooled connection that is owned by another pool.
- An attempt to return a pooled connection that is an incorrect type.



false
Connection checks are enabled. This is the default.

false
Connection checks are disabled.

pool.lenient.dispose.enable
This optional parameter suppresses the checks on a connection that is being disposed of that might throw exceptions.

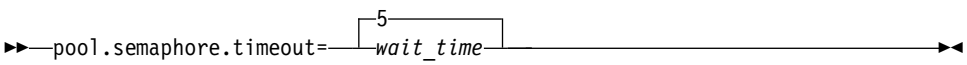


false
Connection checks are enabled. This is the default.

false
Connection checks are disabled.

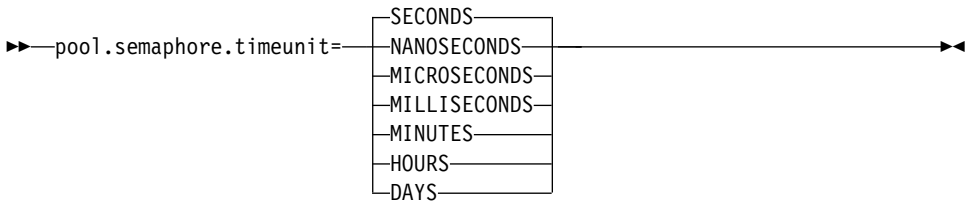
pool.semaphore.timeout
This optional parameter specifies the amount of time to wait to acquire a permit for a database connection. The default value is 5 and the default time unit is seconds.

Important: Set the unit of time in the **pool.semaphore.timeunit** parameter.



pool.semaphore.timeunit

Prerequisite: `pool.semaphore.timeout=wait_time`
This optional parameter specifies the unit of time that is used to scale the **pool.semaphore.timeout** parameter.



SECONDS
Use seconds as the unit of time. This is the default value.

NANOSECONDS
Use nanoseconds as the unit of time.

MICROSECONDS
Use microseconds as the unit of time.

MILLISECONDS
Use milliseconds as the unit of time.

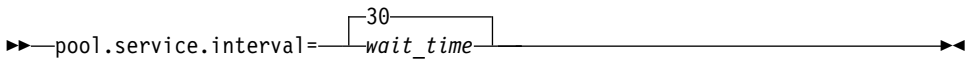
MINUTES
Use minutes as the unit of time.

HOURS
Use hours as the unit of time.

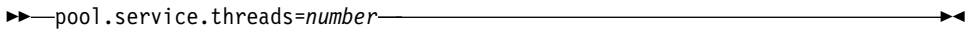
DAYS
Use days as the unit of time.

pool.service.interval
This optional parameter specifies the amount of time to wait between scans of the idle connection pool. The idle connection pool is scanned for connections that can be closed because they have exceeded their maximum idle time. The default value is 30.

Important: Set the unit of time in the **pool.service.timeunit** parameter.

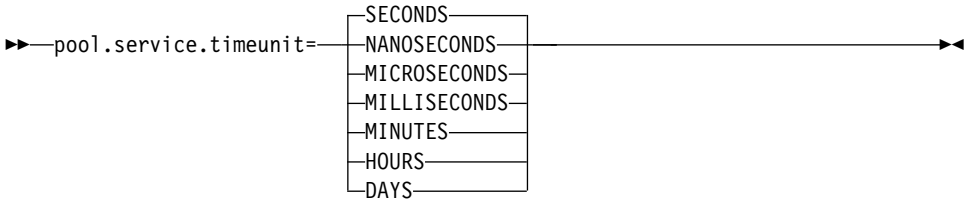


pool.service.threads
This optional parameter specifies the number of threads to use for the maintenance of connection pools that share a common service thread pool. The default value is 1.



pool.service.timeunit

Prerequisite: pool.service.interval=*wait_time*
This optional parameter specifies the unit of time that is used to scale the **pool.service.interval** parameter.



SECONDS
Use seconds as the unit of time. This is the default value.

NANOSECONDS
Use nanoseconds as the unit of time.

MICROSECONDS
Use microseconds as the unit of time.

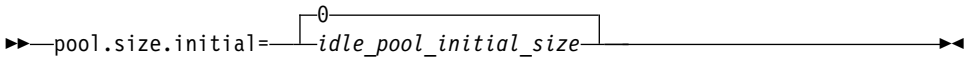
MILLISECONDS
Use milliseconds as the unit of time.

MINUTES
Use minutes as the unit of time.

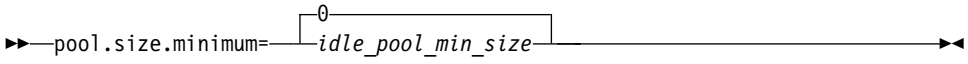
HOURS
Use hours as the unit of time.

DAYS
Use days as the unit of time.

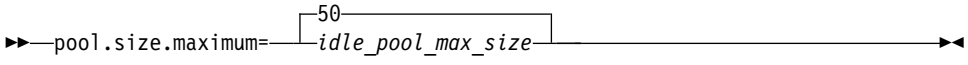
pool.size.initial
This optional parameter specifies the initial size of the idle connection pool. The default value is 0.



pool.size.minimum
This optional parameter specifies the minimum size of the idle connection pool. The default value is 0.



pool.size.maximum
This optional parameter specifies the maximum size of the idle connection pool. The default value is 50.



pool.type
This optional parameter specifies the type of pool to use for JDBC connections. The available pool types are:

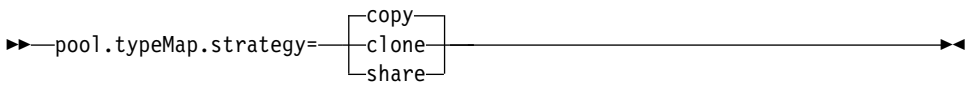
basic
Thread pool maintenance of idle threads is run each time that a connection is returned. This is the default value.

none
No thread pooling occurs. Use this type for debugging purposes.

advanced
Thread pool maintenance is run by a separate thread.

perThread
Each thread is allocated a connection for its exclusive use.

pool.typeMap.strategy
This optional parameter specifies the strategy to use for distribution and synchronization of the JDBC type map for each connection in the pool.

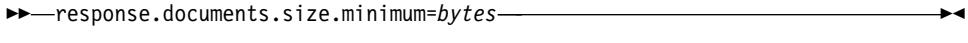


copy
Copy the connection pool type map for each connection. This is the default value.

clone
Clone the connection pool type map for each connection.

share
Share a single type map between all connections. You must use this strategy with a thread-safe type map.

response.documents.size.minimum
This optional parameter specifies the number of bytes for the lower threshold for the maximum response size, which is set by the **response.documents.size.maximum** parameter. The memory manager can reduce the response size to this size when resources are low. The default value is 65536 bytes.



This parameter is effective when the **listener.memoryMonitor.enable** parameter is set to true.

Related tasks:

"Configuring database server authentication with PAM (UNIX, Linux)" on page 2-39

Related reference:

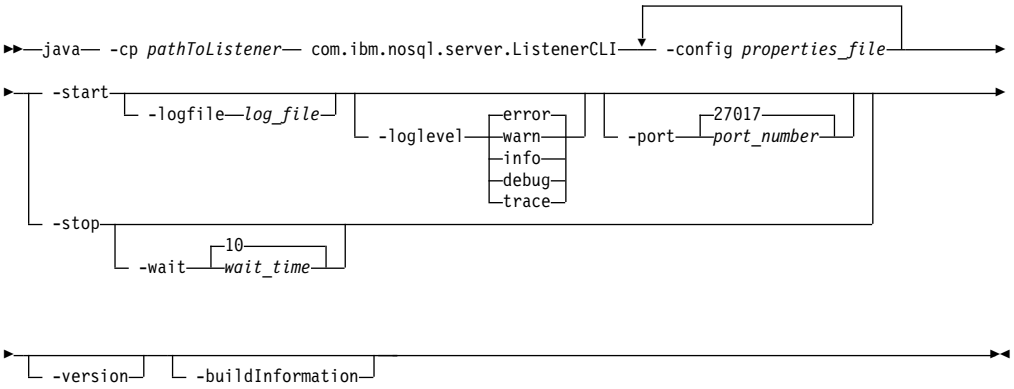
"Collection methods" on page 4-1

"REST API syntax" on page 5-1

Wire listener command line options

You can use command line options to control the wire listener.

Syntax



Argument	Purpose
-cp *pathToListener*	Specifies the fully qualified or relative path to the `jsonListener.jar` file.
com.ibm.nosql.server .ListenerCLI	Specifies the Java main method for the JSON wire listener.
-config *properties_file*	Specifies the name of the wire listener configuration file to run. This argument is required to start or stop the wire listener.
-start	Starts the wire listener. You must also specify the configuration file.
-stop	Stops the wire listener. You must also specify the configuration file. The **stop** command is similar to the MongoDB **shutdown** command.
-logfile *log_file*	Specifies the name of the log file that is used. If this option is not specified, the log messages are sent to `std.out`. **Important:** If you have customized the Logback configuration or specified another logging framework, the settings for **-loglevel** and **-logfile** are ignored.
-loglevel	Specifies the logging level. **error** Errors are sent to the log file. This is the default value. **warn** Errors and warnings are sent to the log file. **info** Informational messages, warnings, and errors are sent to the log file. **debug** Debug, informational messages, warnings, and errors are sent to the log file. **trace** Trace, debug, informational messages, warnings, and errors are sent to the log file. **Important:** If you have customized the Logback configuration or specified another logging framework, the settings for **-loglevel** and **-logfile** are ignored.
-port *port_number*	Specifies the port number. If a port is specified on the command line, it overrides the port properties set in the wire listener configuration file. The default port is 27017.

Argument	Purpose
-wait *wait_time*	Specifies the amount of time, in seconds, to wait for any active sessions to complete before the wire listener is stopped. The default is 10 seconds. To force an immediate shutdown, set the *wait_time* to 0 seconds.
-version	Prints the wire listener version.
-buildInformation	Prints the wire listener build information.

Examples

In this example, the wire listener is started and the log is specified as `$INFORMIXDIR/jsonListener.log`:

```
java -cp $INFORMIXDIR/bin/jsonListener.jar
 com.ibm.nosql.informix.server.ListenerCLI
-config $INFORMIXDIR/etc/jsonListener.properties
-logfile $INFORMIXDIR/jsonListener.log -start
```

In this example, the wire listener is started with the log level set to debug:

```
java -cp $INFORMIXDIR/bin/jsonListener.jar
 com.ibm.nosql.informix.server.ListenerCLI
-config $INFORMIXDIR/etc/jsonListener.properties
–loglevel debug -start
```

In this example, port 6388 is specified:

```
java –cp $INFORMIXDIR/bin/jsonListener.jar
 com.ibm.nosql.informix.server.ListenerCLI
-config $INFORMIXDIR/etc/jsonListener.properties
–port 6388 -start
```

In this example, the wire listener is paused 10 seconds before the wire listener is stopped:

```
java –cp $INFORMIXDIR/bin/jsonListener.jar
 com.ibm.nosql.informix.server.ListenerCLI
-config $INFORMIXDIR/etc/jsonListener.properties
–wait 10 -stop
```

In this example, the wire listener version is printed:

```
java –cp $INFORMIXDIR/bin/jsonListener.jar
 com.ibm.nosql.informix.server.ListenerCLI
–version
```

In this example, the wire listener build information is printed:

```
java –cp $INFORMIXDIR/bin/jsonListener.jar
 com.ibm.nosql.informix.server.ListenerCLI
–buildInformation
```

Related reference:

"Wire listener logging" on page 2-37

Starting the wire listener

You can start the wire listener for the REST API or the MongoDB API by using the **start** command.

Before you begin

- Stop all wire listeners that are currently running. If you create a server instance during the installation process, the MongoDB API wire listener is started automatically and connected to the MongoDB API.
- If you plan to customize the Logback logger or another custom Simple Logging Facade for Java (SLF4J) logger, you must configure the logger before starting the wire listener.
- "Configuring the wire listener for the first time" on page 2-1
- "Software dependencies for JSON compatibility" on page 1-2

Procedure

To start the wire listener, run the wire listener command with the **-start** option. For example:

```
java -cp $INFORMIXDIR/bin/jsonListener.jar
 com.ibm.nosql.server.ListenerCLI
-config $INFORMIXDIR/etc/jsonListener.properties -start
```

The `listener.type` property in the configuration file that you specify defines whether to start the wire listener for the MongoDB API or the REST API.

Results

The wire listener starts.

Examples

In the following example, the wire listener is started with the configuration file specified as `jsonListener_mongo.properties`, the log file specified as `jsonListener_mongo.log`, and the log level specified as **info**:

```
java -cp $INFORMIXDIR/bin/jsonListener.jar
 com.ibm.nosql.server.ListenerCLI
-config $INFORMIXDIR/etc/jsonListener_mongo.properties
-logfile $INFORMIXDIR/jsonListener_mongo.log
-loglevel info -start
```

Here is the output from starting the wire listener:

```
starting mongo listener on port 27017
```

In the following example, the wire listener is started with the configuration file specified as `jsonListener_rest.properties`:

```
java -cp $INFORMIXDIR/bin/jsonListener.jar
 com.ibm.nosql.server.ListenerCLI
-config $INFORMIXDIR/etc/jsonListener_rest.properties -start
```

Here is the output from starting the REST API wire listener:

```
starting rest listener on port 27017
```

Related concepts:

Chapter 5, "REST API," on page 5-1

Related tasks:

"Running multiple wire listeners" on page 2-35

Related reference:

"Wire listener logging" on page 2-37

Related information:

start json listener argument: Start the API wire listener

Running multiple wire listeners

You can run multiple wire listeners.

About this task

By running multiple wire listeners, you can use both the REST API and the MongoDB API. For example, you can create a configuration file to start the MongoDB API and a configuration file to start the REST API.

Procedure

1. Create the individual configuration files in the `$INFORMIXDIR/etc` directory. You can use the `$INFORMIXDIR/etc/jsonListener-example.properties` file as a template.
2. Customize each configuration file and assign a unique name.

   Important: The **`url`** parameter must be specified, either in each individual configuration file or in the file that is referenced by the **`include`** parameter.

   a. Specify the **`include`** parameter to reference an additional configuration file. The path can be relative or absolute. If you have multiple configuration files, you can avoid duplicating parameter settings in the multiple configuration files by specifying a subset of shared parameters in a single configuration file, and the unique parameters in the individual configuration files.
3. Start the wire listeners.

Example: Running multiple wire listeners that share parameter settings

In this example, the same **`url`**, **`authentication.enable`**, and **`security.sql.passthrough`** parameters are used to run two wire listeners:

1. Create a configuration file named `shared.properties` that includes the following parameters:

   ```
   url=jdbc:informix-sqli://localhost:9090/sysmaster:
   INFORMIXSERVER=lo_informix1210;
   authentication.enable=true
   security.sql.passthrough=true
   ```
2. Create a configuration file for use with the MongoDB API that is named `mongo.properties`, with the parameter `include=shared.properties` set:

   ```
   include=shared.properties
   listener.type=mongo
   listener.port=27017
   ```
3. Create a configuration file for use with the REST API that is named `rest.properties`, with the parameter `include=shared.properties` set:

   ```
   include=shared.properties
   listener.type=rest
   listener.port=8080
   ```
4. From the command line, run the **`start`** command. Include separate `-config` arguments for each wire listener API type.

   ```
   java -cp $INFORMIXDIR/bin/jsonListener.jar:pathname/
   tomcat-embed-core.jar com.ibm.nosql.server.ListenerCLI
   -config json.properties
   -config rest.properties -start
   ```

Related tasks:

"Starting the wire listener" on page 2-33

Related reference:

"REST API syntax" on page 5-1

"Wire listener command line options" on page 2-32

Modifying the wire listener configuration file

You can modify the wire listener connection properties that are set in the configuration file.

About this task

The wire listener configuration file, named `%INFORMIXDIR%\etc\jsonListener.properties` by default, controls the wire listener and the connection between the client and database server.

Procedure

To modify the wire listener configuration file:

1. Stop the wire listener.
2. Update the wire listener configuration file.
3. Start the wire listener.

Related tasks:

"Stopping the wire listener"

Related reference:

"The wire listener configuration file" on page 2-3

Stopping the wire listener

You can stop the wire listener by using the **stop** command.

About this task

You must stop the wire listener before you modify any configuration settings.

Procedure

From the command line, run the **stop** command with the configuration file specified. For example:

```
java -cp $INFORMIXDIR/bin/jsonListener.jar -config
 $INFORMIXDIR/etc/jsonListener.properties -stop
```

Important: You must specify the **-config** argument to stop the wire listener from the command line.

Results

The wire listener is stopped.

Related information:

stop json listener: Stop the wire listener

Wire listener logging

The wire listener can output trace, debug, informational messages, warnings, and error information to a log.

The default logging mechanism for the wire listener is Logback. Logback is pre-configured and installed along with the JSON components. For more information on how to customize Logback, see http://logback.qos.ch/.

If you start the MongoDB API wire listener from the command line, you can specify the amount of detail, name, and location of your log file by using the **-loglevel** and **-logfile** command-line arguments.

Important: If you have customized the Logback configuration or specified another logging framework, the settings for **-loglevel** and **-logfile** are ignored.

If the MongoDB API wire listener is started automatically after you create a server instance or if you run the SQL administration API **task()** or **admin()** function with the **start json listener** argument, errors are sent to a log file:

- **UNIX:** The log file is in `$INFORMIXDIR/jsonListener.log`.
- **Windows:** The log file is named *servername*`_jsonListener.log` and is in your home directory. For example, `C:\Users\ifxjson\ol_informix1210_1_jsonListener.log`.

Related tasks:

"Starting the wire listener" on page 2-33

Related reference:

"Wire listener command line options" on page 2-32

User authentication with the wire listener

You can configure authentication for MongoDB clients. You can choose to authenticate users with MongoDB authentication or with the database server, through a pluggable authentication module (PAM).

MongoDB authentication

The wire listener authenticates users with the MongoDB authentication method outside of the database server environment. MongoDB clients connect to the database server as the wire listener user that is specified by the **url** parameter. The database server cannot access MongoDB user account information. The type of authentication depends on the MongoDB version:

MongoDB 2.4
The MONGODB-CR challenge-response method. User information and privileges are stored in the **system_users** collection in each database.

MongoDB 2.6
The MONGODB-CR challenge-response method. User information and privileges are stored in the **system.users** collection in the **admin** database.

MongoDB 3.0
The SCRAM-SHA-1 two-step conversation method. User information and privileges are stored in the **system.users** collection in the **admin** database.

Important: Do not use the MongoDB 3.0 authentication method with the REST API. HTTP protocols do not support SCRAM authentication.

If you are upgrading your MongoDB version and you have existing users, you must upgrade your user schema.

Database server authentication with a PAM

The database server performs authentication through a PAM that implements the MONGODB-CR challenge-response method. The database server controls all user accounts and privileges. You can audit user activities and configure fine-grained access control.

Important: Database server authentication is not compatible with MongoDB version 3.0.

Configuring MongoDB authentication

You can configure the wire listener to use MongoDB authentication.

Before you begin

If you are upgrading your MongoDB version and you have existingMongoDB users, you must upgrade your user schema.

Procedure

To configure MongoDB authentication:

1. Set the following parameters in the wire listener configuration file:
   - Enable authentication: Set **authentication.enable=true**.
   - Specify MongoDB authentication: Set **db.authentication=mongodb-cr**.
   - Specify the MongoDB connection pool: Set **database.connection.strategy=mongodb-cr**.
   - Set the MongoDB version: Set **mongo.api.version** to the version that you want.

     Important: Do not set **mongo.api.version=3.0** if you want to use the REST API. HTTP protocols do not support SCRAM authentication.
   - Optional. Specify the authentication timeout period: Set the **listener.authentication.timeout** parameter to the number of milliseconds for authentication timeout.
2. Restart the wire listener.
3. If necessary, upgrade your user schema by running the **authSchemaUpgrade** command in the **admin** database. For example:

   ```
   use admin
   db.runCommand({authSchemUpgrade : 1})
   ```

   The **authSchemaUpgrade** command upgrades the user schema to the MongoDB version that is specified by the **mongo.api.version** parameter.

Related tasks:

"Starting the wire listener" on page 2-33

"Stopping the wire listener" on page 2-36

Related reference:

"The wire listener configuration file" on page 2-3

Adding users
Procedure

To add authorized users:

1. Start the wire listener with authentication turned off: Set **authentication.enable=false** in the wire listener configuration file.
2. Add users:
   - For MongoDB version 2.4, run the **addUser** command for each user in each database.
   - For MongoDB version 2.6 and 3.0, run the **createUser** command for each user.
3. Turn on authentication: Set **authentication.enable=true** in the wire listener configuration file.
4. Restart the wire listener.

Configuring database server authentication with PAM (UNIX, Linux)

You can configure the database server to authenticate MongoDB client users with a pluggable authentication module (PAM).

About this task

You create a user for the wire listener for PAM connections. The wire listener uses the PAM user to look up system catalog-related information before sending client connection requests to the database server for authentication. The database server authenticates the client users through PAM.

Procedure

To configure PAM authentication for MongoDB clients:

1. Set the **IFMXMONGOAUTH** environment variable. For example:

   ```
   setenv IFMXMONGOAUTH 1
   ```

2. Create a PAM service file that is named /etc/pam.d/pam_mongo and has the following contents:

   ```
   auth    required $INFORMIXDIR/lib/pam_mongo.so file=mongohash
   account required $INFORMIXDIR/lib/pam_mongo.so
   ```

   Replace *$INFORMIXDIR* with the value of the **$INFORMIXDIR** environment variable.

3. On IBM AIX® 64-bit computers, create a symbolic link that is named 64 that points to the lib directory by running the following commands:

   ```
   cd $INFORMIXDIR/lib
   ln -s . 64
   ```

4. Edit the sqlhosts file to add a connection that uses PAM. Include the **s=4** option. Specify the PAM service pam_mongo with the **pam_serv** option. Specify the password authentication mode with the **pamauth** option. For example:

   ```
   ol_informix1210   onsoctcp   myhost  8777  s=4,pam_serv=pam_mongo,pamauth=password
   ```

5. Enable connections from mapped users by setting the USERMAPPING configuration parameter to BASIC or ADMIN in the onconfig file.
6. Set up mapping to an operating system user that has no privileges. For example, on a typical Linux system, the user **nobody** is appropriate. Add the following line to the /etc/informix/allowed.surrogates file:

```
users:nobody
```

7. Restart the database server.
8. Create a PAM user for the wire listener. The user must be internally authenticated and map to the user **nobody**. For example, create a user that is named **mongo** by running the following SQL in the **sysmaster** database:

```
CREATE USER 'mongo' WITH PASSWORD 'aPassword'
     PROPERTIES USER 'nobody';
GRANT CONNECT TO 'mongo';
```

9. Verify the creation of the user by running the following statement:

```
SELECT * FROM sysuser:sysmongousers
        WHERE username='mongo';
```

   The result of the query shows the user and hashed password:

```
username   mongo
hashed_password  bbb8f9630d5c6e094b9aedd945893faf
```

10. Set the following parameters in the wire listener configuration file:
    - Enable authentication: Set **authentication.enable=true**.
    - Specify PAM authentication: Set **db.authentication=informix-mongodb-cr**.
    - Specify the PAM connection pool: Set **database.connection.strategy=informix-mongodb-cr**.
    - Set the MongoDB version: Set **mongo.api.version=2.6** or **mongo.api.version=2.4**. The PAM authentication method is not compatible with MongoDB version 3.0.
    - Optional. Specify the authentication timeout period: Set the **listener.authentication.timeout** parameter to the number of milliseconds for authentication timeout.
    - Specify the mapped user and password for connections and specify to encode and hash the password: Set the **url** parameter. Include the **NONCE** property set to a 16 character string. For example:

```
url=jdbc:informix-sqli://10.168.8.135:40000/sysmaster:USER=mongo;
     PASSWORD=aPassword;NONCE=0123456789abcdef
```

11. Restart the wire listener.
12. Create users that the database server authenticates with PAM by running the SQL statement CREATE USER. If you have existing MongoDB users, you must re-create those users in the database server.

Related reference:

"The wire listener configuration file" on page 2-3

Related information:

sqlhosts file and SQLHOSTS registry key options

IFMXMONGOAUTH environment variable

Pluggable authentication modules (UNIX or Linux)

CREATE USER statement (UNIX, Linux)

USERMAPPING configuration parameter (UNIX, Linux)

Internal users (UNIX, Linux)

Running SQL commands by using a MongoDB API

You can run SQL statements by using the MongoDB API and retrieve results back. The results of the SQL statements are treated like they are documents in a JSON collection.

Before you begin

You must enable SQL operations by setting **security.sql.passthrough=true** in the wire listener properties file.

Procedure

From the MongoDB API, use the abstract system collection `system.sql` as the collection name and `$sql` as the query operator in the MongoDB shell command, followed by the SQL statement. For example:

```
> db.getCollection("system.sql").find({ "$sql": "sql_statement" })
```

Examples

Create an SQL table by using the MongoDB API

In this example, an SQL table is created by running the Informix CREATE TABLE command by using the MongoDB API:

```
> db.getCollection("system.sql").find({ "$sql": "create table foo
(c1 int)" })
```

Drop an SQL table by using the MongoDB API

In this example, an SQL table is dropped by running the Informix DROP TABLE command by using the MongoDB API:

```
> db.getCollection("system.sql").find({"$sql": "drop table foo" })
```

Delete SQL customer call records that are more than 5 years old by using the MongoDB API

In this example, customer call records stored in SQL tables are deleted by running the Informix DELETE command by using the MongoDB API:

```
> db.getCollection("system.sql").findOne({ "$sql": "delete from
cust_calls where (call_dtime + interval(5) year to year) < current" })
```

Result: 7 rows were deleted.

```
{ "n" : 7 }
```

Delete SQL customer call records that are more than 5 years old by using the MongoDB API

In this example, customer call records stored in SQL tables are deleted by running the Informix DELETE command by using the MongoDB API:

```
> db.getCollection("system.sql").findOne({ "$sql": "delete
from cust_calls where (call_dtime + interval(5) year to year) < current" })
```

Result: 7 rows were deleted.

```
{ "n" : 7 }
```

Join JSON collections

In this example, a query counts the number of orders a customer has placed by using an outer join to include the customers who have not placed orders.

```
> db.getCollection("system.sql").find({ "$sql": "select
 c.customer_num,o.customer_num as order_cust,count(order_num) as
 order_count from customer c left outer join orders o on
 c.customer_num = o.customer_num group by 1, 2 order by 2" })
```

Result:

```
{ "customer_num" : 113, "order_cust" : null, "order_count" : 0 }
{ "customer_num" : 114, "order_cust" : null, "order_count" : 0 }
{ "customer_num" : 101, "order_cust" : 101, "order_count" : 1 }
{ "customer_num" : 104, "order_cust" : 104, "order_count" : 4 }
{ "customer_num" : 106, "order_cust" : 106, "order_count" : 2 }
```

Related tasks:

"Configuring the wire listener for the first time" on page 2-1

Related reference:

"The wire listener configuration file" on page 2-3

Running MongoDB operations on relational tables

You can run MongoDB operations on relational tables by using the MongoDB API.

About this task

Use the MongoDB database methods to run read and write operations on a relational table as if the table were a collection. The wire listener examines the database and if the accessed entity is a relational table, it converts the basic operations on that table to SQL and converts the returned values into a JSON document. At the first access to an entity, the wire listener caches the name and type of that entity. The first access results in an extra call to the Informix server, but subsequent operations do not.

Procedure

From the MongoDB API, enter the relational table name as the collection name in the MongoDB collection method. For example:

```
>db.getCollection("tablename");
```

Examples

The following examples use the **customer** table in the **stores_demo** sample database. All of the tables in the **stores_demo** database are relational tables, but you can use the same MongoDB collection methods to access and modify the tables, as if they were collections.

Get the customer count

In this example, the number of customers is returned.

```
> db.customer.count()
28
```

Query for a particular customer

In this example, a specific customer record is retrieved.

```
> db.customer.find({customer_num:101})
{ "customer_num" : 101, "fname" : "Ludwig", "lname" : "Pauli", "company" :
 "All Sports Supplies", "address1" : "213 Erstwild Court", "address2" :
 null, "city" : "Sunnyvale", "state" : "CA", "zipcode" : "94086",
 "phone" : "408-555-8075" }
```

Update a customer phone number

In this example, the customer phone number is updated.

```
> db.customer.update({"customer_id":101}, {"$set":{"phone":"408-555-1234"}})
```

Related reference:

"Collection methods" on page 4-1

Running join queries by using the wire listener

You can use the wire listener to run join queries on JSON and relational data. The syntax supports collection-to-collection joins, relational-to-relational joins, and collection-to-relational joins. Join queries are not supported in sharded environments.

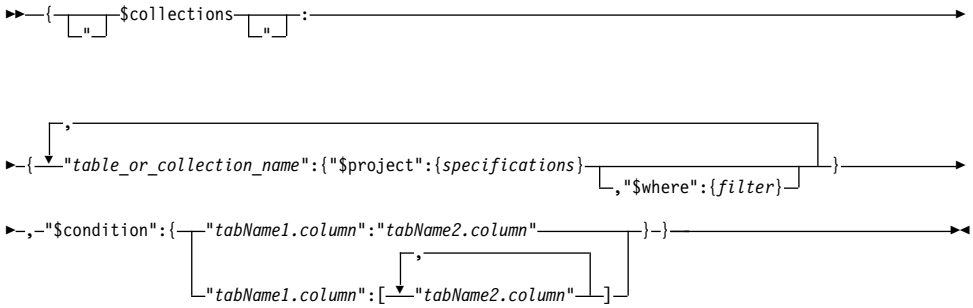
About this task

Join queries in the wire listener are done by submitting a join query document to the **system.join** pseudo table.

- Wire listener join queries support the sort, limit, skip, and explain options that you can set on a MongoDB cursor.
- Fields that are specified in the sort clause must also be included in the projection clause.
- The **$hint** operator is not supported.

Procedure

1. Create a join query document. The join query document has the following syntax:



   $collections
   This required Informix JSON operator defines the two or more collections or relational tables, which are separated by commas, that are included in the join.

   $project
   This required MongoDB JSON operator applies a projection clause to the *table_or_collection_name* that is specified.

   $where
   This optional MongoDB JSON operator applies a query filter to the table or relational table. You can use any of the supported query operators that are listed here: "Query and projection operators" on page 4-18.

   $condition
   This required Informix JSON operator defines how the specified collections or tables are joined. You can specify a condition by mapping a single table column to another single table column, or a single table column to multiple other table columns.

2. Run a **find** query against a pseudo table that is named **system.join** with the join query document specified. For example, in the MongoDB shell:

```
> db.system.join.find({join_query_document})
```

Results

The query results are returned.

Examples of join query document syntax

This example retrieves customer orders that total more than $100. The join query document joins the **customer** and **orders** tables, on the **customer_num** field where the order total is greater than 100. The same query document works if the customers and orders tables are collections, relational tables, or a combination of the two.

```
{"$collections":
        {
             "customers":
                 {"$project":{customer_num:1,name:1,phone:1}},
             "orders":
                 {"$project":{order_num:1,nitems:1,total:1,_id:0},
                  "$where":{total:{"$gt":100}}}
        },
  "$condition":
        {"customers.customer_num":"orders.customer_num"}
}
```

This example retrieves the order, shipping, and payment information for order number 1093. The array syntax is used in the **$condition** syntax of the join query document.

```
{"$collections":
         {
             "orders":
                  {"$project":{order_num: 1,nitems: 1,total: 1,_id:0},
                   "$where":{order_num:1093}},
             "shipments":
                  {"$project":{shipment_date:1,arrival_date:1}},
             "payments":
                  {"$project":{payment_method:1,payment_date:1}}
         },
  "$condition":
    {"orders.order_num":["shipments.order_num","payments.order_num"]}
}
```

This example retrieves the order and customer information for orders that total more than $1000 and that are shipped to the postal code 10112.

```
{"$collections":
        {
            "orders":
                 {"$project":{order_num:1,nitems:1,total:1,_id:0},
                  "$where":{total:{$gt:1000}}},
            "shipments":
                 {"$project":{shipment_date:1,arrival_date:1,_id:0},
                  "$where":{address.zipcode:10112},
            "customer":
                 {"$project":{customer_num:1,name:1,company:1,_id:0}}
       },
  "$condition":
       {
            "orders.order_num":"shipments.order_num",
          "orders.customer_num":"customer.customer_num",
       }
}
```

High availability support in the wire listener

The wire listener provides high availability support.

To provide high availability to client applications, use the appropriate method:

- For REST clients, you can use a reverse proxy for multiple wire listeners.
- For MongoDB clients, use a high-availability cluster configuration for your Informix database servers. For each database server in the cluster, run a wire listener that is directly connected to that database server. Each wire listener must be on the same computer as the database server that it is connected to and all wire listeners must run on the port 27017. For more information, see http://docs.mongodb.org/meta-driver/latest/legacy/connect-driver-to-replica-set/.

To provide high availability between the wire listener and the Informix database server, use one of the following methods:

- Route the connection between the wire listener and the database server through the Connection Manager.
- Configure the **url** parameter in the wire listener configuration file to use one of the Informix JDBC Driver methods of connecting to a high-availability cluster. For more information, see Dynamically reading the Informix sqlhosts file or Properties for connecting directly to an HDR pair of servers.

Related information:

Dynamically reading the sqlhosts file

Chapter 3. JSON data sharding

You can shard data with IBM Informix. Documents from a collection or rows from a table can be sharded across a cluster of database servers, reducing the number of documents or rows and the size of the index for the database of each server. When you shard data across database servers, you also distribute performance across hardware. As your database grows in size, you can scale up by adding more shard servers to your shard cluster.

Documents or rows that are inserted on a shard server are distributed to the appropriate shard servers in a shard cluster based on the sharding schema. Queries on a sharded table automatically retrieve data from all relevant shard servers in a shard cluster. When data is sharded based on a field or column that specifies certain segmentation characteristics, queries can skip shard servers that do not contain relevant data.

A shard cluster of Informix database servers is a special form of Enterprise Replication. You can create a shard cluster with Enterprise Replication commands or with MongoDB commands.

Informix shard cluster architecture is very flexible:

- Shard servers can run on different hardware and operating systems.
- Shard servers can run different version of Informix. For example, you can upgrade Informix on shard servers individually.
- Shard servers can have high-availability secondary servers from which users can query the sharded table.

To start sharding data:

1. Prepare shard servers for sharding.
2. Create a shard cluster.
3. Define a schema for sharding data against an existing table.

Related tasks:

"Configuring the wire listener for the first time" on page 2-1

Related information:

Shard cluster setup

Sharded queries

Preparing shard servers

You must prepare shard servers before you can shard data.

Procedure

To set up shard servers:

1. On each shard server, set the SHARD_ID configuration parameter to a positive integer value that is unique in the shard cluster by running the following command:

   ```
   onmode -wf SHARD_ID=unique_positive_integer
   ```

If the SHARD_ID configuration parameter is already set to a positive integer, you can change the value by editing the `onconfig` file and then restarting the database server. You can also set the SHARD_MEM configuration parameter to customize the number of memory pools that are used during shard queries.

2. Specify trusted hosts information for all shard servers. On each shard server, use one of the following methods to add trusted host information about all the other shard servers:
   - Use the OpenAdmin Tool (OAT) for Informix. Go to the **Server Administration** > **Configuration** page, and click the **Trusted Hosts** tab.
   - Run the SQL administration API **task()** or **admin()** function with the **cdr add trustedhost** argument and include the appropriate host values for all the other shard servers. You must be a Database Server Administrator (DBSA) to run these functions.
3. On each shard server, edit the wire listener configuration file:
   a. Set the **`sharding.enable`** parameter to `true`.
   b. Set the **`sharding.query.parallel.enable`** parameter to `true`.
   c. Set the **`update.client.strategy`** parameter to `deleteInsert`.
   d. Set the **`USER`** attribute in the **`url`** parameter to a user who has the REPLICATION privilege. If you created a database server instance during installation, the **ifxjson** user, who has the REPLICATION privilege, is automatically set as the value of the **`USER`** attribute. Otherwise, see "Configuring the wire listener for the first time" on page 2-1 for instructions.
4. On each shard server, restart the wire listener.

Related information:

cdr add trustedhost argument: Add trusted hosts (SQL administration API)

cdr list trustedhost argument: List trusted hosts (SQL administration API)

Starting the wire listener

onmode -wf, -wm: Dynamically change certain configuration parameters

SHARD_ID configuration parameter

SHARD_MEM configuration parameter

Creating a shard cluster with MongoDB commands

You create a shard cluster by adding shard servers with the The MongoDB **sh.addShard** shell command or the **db.runCommand** command with the **addShard** syntax.

Before you begin

The shard servers must be prepared for sharding. See "Preparing shard servers" on page 3-1.

Procedure

To create a shard cluster from the MongoDB shell:

1. Run the **mongo** command to start the MongoDB shell.
2. Run one of the following commands with the host name and port that is specified for the Informix server that you want to add. The specified port must run the Informix network-based listener, for example the **onsoctcp** protocol.
   a. Run the **sh.addShard** command.

b. Run the **db.runCommand** with the **addShard** command syntax. You can include the fully qualified domain name of the server instead of the host name. You can specify multiple servers.

Results

A shard cluster is created with the specified shard servers. Each shard server is set up with Enterprise Replication and assigned an Enterprise Replication group name in its `sqlhosts` file. The default Enterprise Replication group name for a database server is the database server name with a suffix of `g_`. For example, the default Enterprise Replication group name for a database server that is named **myserver** is **g_myserver**.

Examples

Add a server to a shard cluster with addShard

The following command adds the database server that is at port **9202** of **myhost2.ibm.com** to a shard cluster:

```
> sh.addShard("myhost2.ibm.com:9202")
```

Add a server to a shard cluster with db.runCommand and addShard

The following command adds the database server that is at port **9204** of **myhost4.ibm.com** to a shard cluster.

```
> db.runCommand({"addShard":"myhost4.ibm.com:9204"})
```

Add multiple servers to a shard cluster

This example adds the database servers that are at port **9205** of **myhost5.ibm.com**, port **9206** of **myhost6.ibm.com**, and port **9207** of **myhost7.ibm.com** to a shard cluster.

```
> db.runCommand({"addShard":["myhost5.ibm.com:9205",
    "myhost6.ibm.com:9206","myhost7.ibm.com:9207"]})
```

Related reference:

"Database commands" on page 4-4

Related information:

cdr define shardCollection

cdr add trustedhost argument: Add trusted hosts (SQL administration API)

cdr remove trustedhost argument: Remove trusted hosts (SQL administration API)

cdr list trustedhost argument: List trusted hosts (SQL administration API)

Shard-cluster definitions for distributing data

A cluster of shard servers uses a definition to distribute data across shard servers.

You must create a shard-cluster definition to distribute data across the shard servers. The definition contains the following information:

- The Informix Enterprise Replication group name of each participating shard server.
- The name of the database and collection or table that is distributed across the shard servers of a shard cluster.
- The field or column that is used as a shard key for distributing data. Shard key values determine which shard server a document or row is stored on.
- The sharding method by which documents or rows are distributed to specific shard servers. The sharding method is either a hash-based or expression-based.

Related information:

cdr change shardCollection

cdr delete shardCollection

Defining a sharding schema with a hash algorithm

The **shardCollection** command in the MongoDB shell creates a definition for distributing data across the database servers of a shard cluster.

Procedure

To create a shard-cluster definition that uses a hash algorithm for distributing data across database servers:

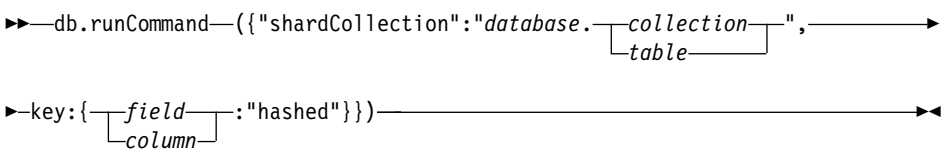
1. Run the **mongo** command. The command starts the MongoDB shell.
2. Run the **shardCollection** command. There are two ways to run the command:
   - Run the **sh.shardCollection** MongoDB command. For example:

     ```
     > sh.shardCollection("database1.collection1",
       {customer_name:"hashed"})
     ```

   - Run the **db.runCommand** from the MongoDB shell, with **shardCollection** command syntax. For example:

     ```
     > db.runCommand({"shardCollection":"database2.collection_2",
       key:{customer_name:"hashed"}})
     ```

     The **shardCollection** command syntax for using a hash algorithm is shown in the following diagram:



Element	Description	Restrictions
database	The name of the database that contains the collection that is distributed across database servers.	The database must exist.
collection	The name of the collection that is distributed across database servers.	The collection must exist.
column	The shard key that is used to distribute data across the database servers of a shard cluster.	The column must exist. Composite shard keys are not supported.
field	The shard key that is used to distribute data across the database servers of a shard cluster.	The field must exist. Composite shard keys are not supported.
table	The name of the table that is distributed across database servers.	The table must exist.

3. For optimal query performance, connect to the wire listener and run the MongoDB **ensureIndex** command on the shard key of each of a cluster's shard servers. The **ensureIndex** command ensures that an index for the collection or table is created on the shard server.

Results

The name of a shard-cluster definition that is created by a **shardCollection** command that is run through the wire listener is:

Example

The following command defines a shard cluster that uses a hash algorithm on the shard key value **year** to distribute data across multiple database servers.

```
> sh.shardCollection("mydatabase.mytable",{year:"hashed"})
```

The name of the created shard-cluster definition is **sh_mydatabase_mytable**.

Related reference:

"Database commands" on page 4-4

Related information:

cdr change shardCollection

cdr delete shardCollection

Defining a sharding schema with an expression

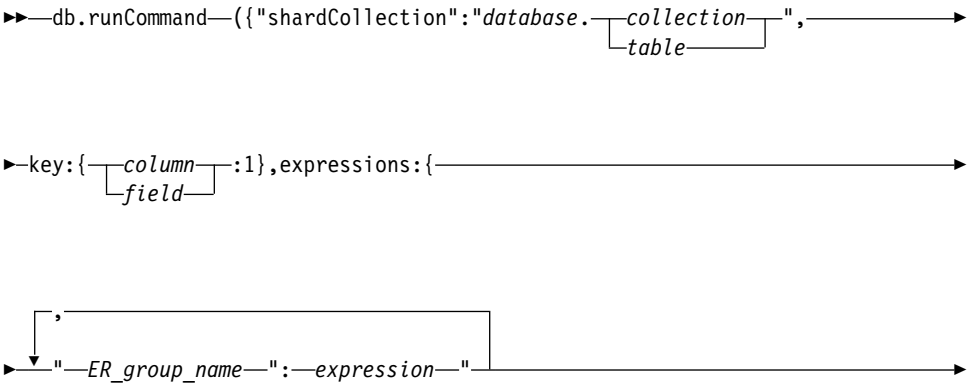
The MongoDB shell **db.runCommand** command with **shardCollection** command syntax creates a definition for distributing data across the database servers of a shard cluster.

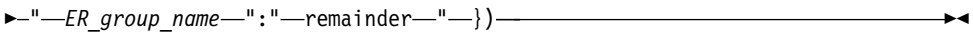
Procedure

To create a shard-cluster definition that uses an expression for distributing data across database servers:

1. Run the **mongo** command. The command starts the MongoDB shell.
2. Run the **db.runCommand** from the MongoDB shell, with **shardCollection** command syntax.

   The **shardCollection** command syntax for using an expression is shown in the following diagram:





Element	Description	Restrictions
collection	The name of the collection that is distributed across database servers.	The collection must exist.
column	The shard key that is used to distribute data across the database servers of a shard cluster.	The column must exist. Composite shard keys are not supported.
database	The name of the database that contains the collection that is distributed across database servers.	The database must exist.
ER_group_name	The Enterprise Replication group name of a database server that receives copied data. The default Enterprise Replication group name for a database server is the database server's name prepended with `g_`. For example, the default Enterprise Replication group name for a database server that is named **myserver** is **g_myserver**.	None.
expression	The expression that is used to select documents by shard key value.	None.
field	The shard key that is used to distribute data across the database servers of a shard cluster.	The field must exist. Composite shard keys are not supported.
remainder	Specifies a database server that receives documents with shard key values that are not selected by expressions. The remainder expression is required.	
table	The name of the table that is distributed across database servers.	The table must exist.

3. For optimal query performance, connect to the wire listener and run the MongoDB **ensureIndex** command on the shard key of each of a cluster's shard servers. The **ensureIndex** command ensures that an index is created for the collection or table on the shard server.

Results

The name of a shard-cluster definition that is created by a **shardCollection** command that is run through the wire listener is:

Examples

Define a shard cluster that uses an expression to distribute data across multiple database servers

The following command defines a shard cluster that uses an expression on the field value **state** for distributing **collection1** across multiple database servers.

```
> db.runCommand({"shardCollection":"database1.collection1",
   key:{state:1},expressions:{"g_shard_server_1":"in ('KS','MO')",
   "g_shard_server_2":"in ('CA','WA')","g_shard_server_3":"remainder"}})
```

The name of the created shard-cluster definition is **sh_database1_collection1**.

- Inserted documents with **KS** and **MO** values in the **state** field are sent to **g_shard_server_1**.
- Inserted documents with **CA** and **WA** values in the **state** field are sent to **g_shard_server_2**.
- All inserted documents that do not have **KS**, **MO**, **CA**, or **WA** values in the **state** field are sent to **g_shard_server_3**.

Define a shard cluster that uses an expression to distribute data across multiple database servers

The following command defines a shard cluster that uses an expression on the column value **animal** for distributing **table2** across multiple database servers.

```
> db.runCommand({"shardCollection":"database1.table2",
   key:{animal:1},expressions:{"g_shard_server_1":"in ('dog','coyote')",
   "g_shard_server_2":"in ('cat')","g_shard_server_3":"in ('rat')",
   "g_shard_server_4":"remainder"}})
```

The name of the created shard-cluster definition is **sh_database2_table2**.

- Inserted rows with **dog** or **coyote** values in the **animal** column are sent to **g_shard_server_1**.
- Inserted rows with **cat** values in the **animal** column are sent to **g_shard_server_2**.
- Inserted rows with **rat data** values in the **animal** column are sent to **g_shard_server_3**.
- All inserted rows that do not have **dog**, **coyote**, **cat**, or **rat** values in the **animal** column are sent to **g_shard_server_4**.

Define a shard cluster that uses an expression to distribute collections across multiple database servers

The following command defines a shard cluster that uses an expression on the field value **year** for distributing **collection3** across multiple database servers.

```
> db.runCommand({"shardCollection":"database1.collection3",
   key:{year:1},expressions:{"g_shard_server_1":"between 1980 and 1989",
   "g_shard_server_2":"between 1990 and 1999",
   "g_shard_server_3":"between 2000 and 2009",
   "g_shard_server_4":"remainder"}})
```

The name of the created shard-cluster definition is **sh_database3_collection3**.

- Inserted documents with values of **1980** to **1989** in the **year** field are sent to **g_shard_server_1**.
- Inserted documents with values of **1990** to **1999** in the **year** field are sent to **g_shard_server_2**.
- Inserted documents with values of **1980** to **1989** in the **year** field are sent to **g_shard_server_3**.
- Inserted documents with values below **1980** or above **2009** in the **year** field are sent to **g_shard_server_4**.

Related reference:

"Database commands" on page 4-4

Shard cluster management

You can display information about shard cluster participants and about the shard cache on each shard server. You can add or remove shard servers from a shard cluster.

To display information about shard cluster participants, run the **db.runCommand** from the MongoDB shell, with **listShard** command syntax.

To display information about shard caches, run the **onstat -g shard** command.

Add a shard server

To add a shard server to the shard cluster, prepare the new shard server and add it to the shard cluster with the **addShard** command. Make sure to add the trusted host information for the new shard server to the existing shard servers.

Remove a shard server

To remove a shard server, run the **db.runCommand** from the MongoDB shell, with **removeShard** command syntax.

Change the sharding definition

After you add or remove a shard server, you might need to update the sharding definition:

- A definition that uses a hash algorithm to shard data is modified automatically.
- You must modify a sharding definition that uses an expression by running the **changeShardCollection** command.

When you change the sharding definition, existing data on shard servers is redistributed to match the new definition.

Related tasks:

"Preparing shard servers" on page 3-1

"Creating a shard cluster with MongoDB commands" on page 3-2

Related information:

cdr list trustedhost argument: List trusted hosts (SQL administration API)

onstat -g shard command

Changing the definition for a shard cluster

The **db.runCommand** command with **changeShardCollection** command syntax changes the definition for a shard cluster.

Before you begin

If the shard cluster uses an expression for distributing data across multiple database servers, you must add database servers to a shard cluster and remove database servers from a shard cluster by running the **changeShardCollection** command. If the shard-cluster definition uses a hash algorithm, database servers are automatically added to the shard cluster when you run the **sh.addShard** MongoDB shell command.

If you change a shard-cluster definition to include a new shard server, that server must first be added to a shard cluster by running the **db.runCommand** command with **addShard** command syntax.

When a shard-cluster definition changes, existing data on shard servers is redistributed to match the new definition.

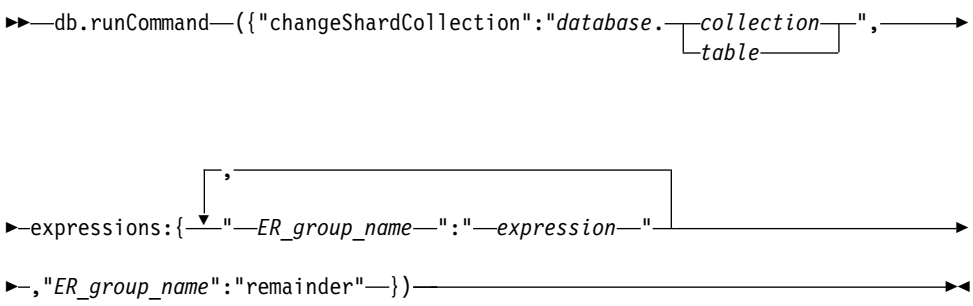
About this task

The following steps apply to changing the definition for shard cluster that uses an expression for distributing documents in a collection across multiple database servers.

Procedure

To change the definition for a shard cluster:

1. Run the **mongo** command. The command starts the MongoDB shell.
2. Change the shard-cluster definition by running the **changeShardCollection** command. You must redefine all expressions for all shard servers, not just newly added or changed shard servers.



Element	Description	Restrictions
collection	The name of the collection that is distributed across database servers.	The collection must exist.
database	The name of the database that contains the collection that is distributed across database servers.	The database must exist.
ER_group_name	The Enterprise Replication group name of a database server that receives copied data. The default Enterprise Replication group name for a database server is the database server's name prepended with `g_`. For example, the default Enterprise Replication group name for a database server that is named **myserver** is **g_myserver**.	None.
expression	The expression that is used to select documents by shard key value.	None.
remainder	The database server that receives documents with shard key values that are not selected by expressions.	
table	The name of the table that is distributed across database servers.	The table must exist.

3. For optimal query performance, connect to the wire listener and run the MongoDB **ensureIndex** command on the shard key each of a cluster's shard servers. The **ensureIndex** command ensures that an index for the collection or table is created on the shard server.

Example

You have a shard cluster that is composed of three database servers, and the shard cluster is defined by the following command:

```
> db.runCommand({"shardCollection":"database1.collection1",
   expressions:{"g_shard_server_1":"in ('KS','MO')",
   "g_shard_server_2":"in ('CA','WA')","g_shard_server_3":"remainder"})
```

To add **g_shard_server_4** and **g_shard_server_5** to the shard cluster and change where data is sent to, run the following command:

```
> db.runCommand({"changeShardCollection":"database1.collection1",
   expressions:{"g_shard_server_1":"in ('KS','MO')",
   "g_shard_server_2":"in ('TX','OK')","g_shard_server_3":"in ('CA','WA')",
   "g_shard_server_4":"in ('OR','ID')","g_shard_server_5":"remainder"})
```

The new shard cluster contains five database servers:

- Inserted documents with a **state** field value of `KS` or `MO` are sent to **g_shard_server_1**.
- Inserted documents with a **state** field value of `TX` or `OK` are sent to **g_shard_server_2**.
- Inserted documents with a **state** field value of `CA` or `WA` are sent to **g_shard_server_3**.
- Inserted documents with a **state** field value of `OR` or `ID` are sent to **g_shard_server_4**.
- Inserted documents with a **state** field value that is not in the expression are sent to **g_shard_server_5**.

To then remove **g_shard_server_2** and change where the data that was on **g_shard_server_2** is sent to, run the following command:

```
> db.runCommand({"changeShardCollection":"database1.collection1",
   expressions:{"g_shard_server_1":"in ('KS','MO')",
   "g_shard_server_3":"in ('TX','CA','WA')",
   "g_shard_server_4":"in ('OK','OR','ID')",
   "g_shard_server_5":"remainder"})
```

The new shard cluster contains four database servers.

- Inserted documents with a **state** field value of `TX` are now sent to **g_shard_server_3**.
- Inserted documents with a **state** field value of `OK` are now sent to **g_shard_server_4**.

Existing data on shard servers is redistributed to match the new definition.

Viewing shard-cluster participants

Run the **db.runCommand** MongoDB shell command with **listShards** syntax to list the Enterprise Replication group names, hosts, and port numbers of all shard servers in a shard cluster.

Procedure

1. Run the **mongo** command. The command starts the MongoDB shell.
2. Run the **listShards** command:

```
db.runCommand({listShards:1})
```

Results

The **listShards** command produces output in the following structure:

```
{
        "serverUsed" : "server_host/IP_address",
        "shards" : [
                {
                        "_id" : "ER_group_name_1",
                        "host" : "host_1:port_1"
                },
                {
                        "_id" : "ER_group_name_2",
                        "host" : "host_2:port_2"
                },
                {
                        "_id" : "ER_group_name_x",
                        "host" : "host_x:port_x"
                }
        ],
        "ok" : 1
}
```

ER_group_name
The Enterprise Replication group name of a shard server.

host The host for a shard-cluster participant. The host can be a localhost name or a full domain name.

IP_address
The IP address of the database server that the listener is connected to.

port The port number that a shard-cluster participant uses to communicate with other shard-cluster participants.

server_host
The host for the database server that the listener is connected to. The host can be a localhost name or a full domain name.

Example

For this example, you have a shard cluster defined by the following command:

```
prompt> db.runCommand({"addShard":["myhost1.ibm.com:9201",
   "myhost2.ibm.com:9202","myhost3.ibm.com:9203",
   "myhost4.ibm.com:9204","myhost5.ibm.com:9205"]})
```

The following example output is shown when the **listShards** command is run in the MongoDB shell, and the listener is connected to the database server at `myhost1.ibm.com`.

```
{
        "serverUsed" : "myhost1.ibm.com/192.0.2.0:9200",
        "shards" : [
                {
                        "_id" : "g_myserver1",
                        "host" : "myhost1.ibm.com:9200"
                },
                {
                        "_id" : "g_myserver2",
                        "host" : "myhost2.ibm.com:9202"
                },
                {
                        "_id" : "g_myserver3",
                        "host" : "myhost3.ibm.com:9203"
                }
                {
                        "_id" : "g_myserver4",
                        "host" : "myhost4.ibm.com:9204"
                }
                {
                        "_id" : "g_myserver5",
                        "host" : "myhost5.ibm.com:9205"
                }
        ],
        "ok" : 1
}
```

Figure 3-1. **listShards** *command output for a shard cluster*

Related reference:

"Database commands" on page 4-4

Related information:

cdr list trustedhost argument: List trusted hosts (SQL administration API)

Installing the OpenAdmin Tool for Informix with the Client SDK

cdr list shardCollection

onstat -g shard command: Print information about the shard cache

Chapter 4. MongoDB API and commands

The Informix support for MongoDB application programming interfaces and commands are described here.

Language drivers

The wire listener parses messages that are based on the MongoDB Wire Protocol.

You can use the MongoDB community drivers to store, update, and query JSON documents with Informix as a JSON data store. These drivers can include Java, C/C++, Ruby, PHP, PyMongo, and so on.

Download the MongoDB drivers for the programming languages at http://docs.mongodb.org/ecosystem/drivers/.

Command utilities and tools

You can use the MongoDB shell and any of the standard MongoDB command utilities and tools.

The supported MongoDB shell is version 2.4, 2.6, and 3.0.

You can run the MongoDB mongodump and mongoexport utilities against MongoDB to export data from MongoDB to Informix.

You can run the MongoDB mongorestore and mongoimport utilities against Informix to import data from MongoDB to Informix.

Collection methods

Informix supports a subset of the MongoDB collection methods.

The collection methods are run on a JSON collection or a relational table. The syntax for collection methods in the **mongo** shell is `db.collection_name.collection_method()`, where `db` refers to the current database, *collection_name* is the name of the JSON collection or relational table, *collection_method* is the MongoDB collection method. For example, `db.cartype.count()` determines the number of documents that are contained in the **cartype** collection.

Table 4-1. Supported collection methods

Collection method	JSON collections	Relational tables	Details
aggregate	No	No	
count	Yes	Yes	
createIndex	Yes	Yes	For more information, see “Index creation” on page 4-3.
dataSize	Yes	No	
distinct	Yes	Yes	
drop	Yes	Yes	

Table 4-1. Supported collection methods (continued)

Collection method	JSON collections	Relational tables	Details
dropIndex	Yes	Yes	
dropIndexes	Yes	No	
ensureIndex	Yes	Yes	For more information, see "Index creation" on page 4-3.
find	Yes	Yes	
findAndModify	Yes	Yes	For relational tables, findAndModify is supported only for tables that have a primary key. This method is not support sharded data.
findOne	Yes	Yes	
getIndexes	Yes	No	
getShardDistribution	No	No	
getShardVersion	No	No	
getIndexStats	No	No	
group	No	No	
indexStats	No	No	
insert	Yes	Yes	
isCapped	Yes	Yes	This command returns false because capped collections are not supported in Informix.
mapReduce	No	No	
reIndex	No	No	
remove	Yes	Yes	The justOne option is not supported. This command deletes all documents that match the query criteria.
renameCollection	No	No	
save	Yes	No	
stats	Yes	No	
storageSize	Yes	No	
totalSize	Yes	No	
update	Yes	Yes	The multi option is supported for JSON collections if `update.one.enable=true` in the wire listener properties file. For relational tables, the multi-parameter is ignored and all documents that meet the query criteria are updated. If `update.one.enable=false`, all documents that match the query criteria are updated.
validate	No	No	

For more information about the MongoDB features, see http://docs.mongodb.org/manual/reference/.

Related tasks:

"Running MongoDB operations on relational tables" on page 2-42

Related reference:

"The wire listener configuration file" on page 2-3

Index creation

Informix supports the creation of indexes on collections and relational tables by using the MongoDB API and the wire listener.

Index creation by using the MongoDB syntax

For JSON collections and relational tables, you can use the MongoDB createIndex and ensureIndex syntax to create an index that works for all data types. For example:

```
db.collection.createIndex( { zipcode: 1 } )
db.collection.createIndex( { state: 1, zipcode: -1} )
```

Tip: If you are creating an index for a JSON collection on a field that has a fixed data type, you can get the best query performance by using the Informix extended syntax.

The following options are supported:

- name
- unique

The following options are not supported:

- background
- default_language
- dropDups
- expireAfterSeconds
- language_override
- sparse
- v
- weights

Index creation for a specific data type by using the Informix extended syntax

You can use the Informix createIndex or ensureIndex syntax on collections to create an index for a specific data type. For example:

```
db.collection.createIndex( { zipcode : [1, "$int"] } )
db.collection.createIndex( { state: [1, "$string"],  zipcode: [-1, "$int"] } )
```

This syntax is supported for collections only. It not supported for relational tables.

Tip: If you are creating an index on a field that has a fixed data type, you can get better query performance by using the Informix createIndex or ensureIndex syntax.

The following data types are supported:

- $binary
- $boolean
- $date
- $double[2]

- $int[3]
- $integer[3]
- $lvarchar[1]
- $number[2]
- $string[1]
- $timestamp
- $varchar

Notes:

1. $string and $lvarchar are aliases and create lvarchar indexes.
2. $number and $double are aliases and create double indexes.
3. $int and $integer are aliases.

Index creation for text, geospatial, and hashed

Text indexes

Text indexes are supported. You can search string content by using text search in documents of a collection.

You can create text indexes by using the MongoDB or Informix syntax. For example, here is the MongoDB syntax:

```
db.articles.ensureIndex( { abstract: "text" } )
```

The Informix syntax provides additional support for the Informix basic text search functionality. For more information, see "createTextIndex" on page 4-11.

Geospatial indexes

2dsphere indexes are supported by using the GeoJSON objects, but not the MongoDB legacy coordinate pairs.

2d indexes are not supported.

Hashed indexes

Hashed indexes are not supported. If a hashed index is specified, a regular untyped index is created.

For more information about the MongoDB features, see http://docs.mongodb.org/manual/reference/.

Database commands

Informix supports a subset of the MongoDB database commands.

The basic syntax for database commands in the **mongo** shell is `db.`*`command`*`()`, where *db* refers to the current database, and *command* is the database command. You can use the **mongo** shell helper method **db.runCommand()** to run database commands on the current database.

- "User commands" on page 4-5
- "Database operations" on page 4-6

User commands

Aggregation commands

Table 4-2. Aggregation commands

MongoDB command	JSON collections	Relational tables	Details
aggregate	Yes	Yes	The wire listener supports version 2.4 of the MongoDB aggregate command, which returns a command result. For more information, see “Aggregation framework operators” on page 4-22.
count	Yes	Yes	
distinct	Yes	Yes	
group	No	No	
mapReduce	No	No	

Geospatial commands

Table 4-3. Geospatial commands

MongoDB command	JSON collections	Relational tables	Details
geoNear	Yes	No	Supported by using the GeoJSON format. The MongoDB legacy coordinate pairs are not supported.
geoSearch	No	No	
geoWalk	No	No	

Query and write operation commands

Table 4-4. Query and write operation commands

MongoDB command	JSON collections	Relational tables	Details
delete	Yes	Yes	
eval	No	No	
findAndModify	Yes	Yes	For relational tables, the findAndModify command is supported only for tables that have a primary key. This command does not support sharded data.
getLastError	Yes	Yes	
getPrevError	No	No	
insert	Yes	Yes	
resetError	No	No	
text	No	No	Text queries are supported by using the $text or $ifxtext query operators, not through the text command.
update	Yes	Yes	

Database operations

Authentication commands

Table 4-5. Authentication commands

Name	Supported	Details
authenticate	Yes	
authSchemaUpgrade	Yes	This command upgrades user data to MongoDB API version 2.6 or higher.
logout	Yes	
getnonce	Yes	

User management commands

Table 4-6. User management commands

Name	Supported	Details
createUser	Yes	Supported for MongoDB API version 2.6 or higher.
dropAllUsersFromDatabase	Yes	Supported for MongoDB API version 2.6 or higher.
dropUser	Yes	Supported for MongoDB API version 2.6 or higher.
grantRolesToUser	Yes	Supported for MongoDB API version 2.6 or higher.
revokeRolesFromUser	Yes	Supported for MongoDB API version 2.6 or higher.
updateUser	Yes	Supported for MongoDB API version 2.6 or higher.
usersInfo	Yes	Supported for MongoDB API version 2.6 or higher.

Role management commands

Table 4-7. Role management commands

Name	Supported	Details
createRole	Yes	Supported for MongoDB API version 2.6 or higher.
dropAllRolesFromDatabase	Yes	Supported for MongoDB API version 2.6 or higher.
dropRole	Yes	Supported for MongoDB API version 2.6 or higher.
grantPrivilegesToRole	Yes	Supported for MongoDB API version 2.6 or higher.
grantRolesToRole	Yes	Supported for MongoDB API version 2.6 or higher.
invalidateUserCache	No	
rolesInfo	Yes	Supported for MongoDB API version 2.6 or higher.
revokePrivilegesFromRole	Yes	Supported for MongoDB API version 2.6 or higher.
revokeRolesFromRole	Yes	Supported for MongoDB API version 2.6 or higher.
updateRole	Yes	Supported for MongoDB API version 2.6 or higher.

Diagnostic commands

Table 4-8. Diagnostic commands

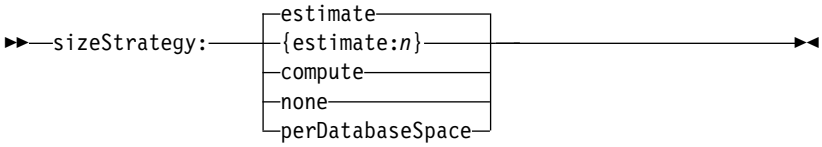
Name	Supported	Details
buildInfo	Yes	Whenever possible, the Informix output fields are identical to MongoDB. There are additional fields that are unique to Informix.
collStats	Yes	The value of any field that is based on the collection size is an estimate, not an exact value. For example, the value of the field 'size' is an estimate.
connPoolStats	No	

Table 4-8. Diagnostic commands (continued)

Name	Supported	Details
cursorInfo	No	
dbStats	Yes	The value of any field that is based on the collection size is an estimate, not an exact value. For example, the value of the field 'dataSize' is an estimate.
features	Yes	
getCmdLineOpts	Yes	
getLog	No	
hostInfo	Yes	The `memSizeMB`, `totalMemory`, and `freeMemory` fields indicate the amount of memory that is available to the Java virtual machine (JVM) that is running, not the operating system values.
indexStats	No	
listCommands	Yes	

Table 4-8. Diagnostic commands (continued)

Name	Supported	Details

listDatabases	Yes	The value of any field that is based on the collection size is an estimate, not an exact value. For example, the value of the field 'sizeOnDisk' is an estimate.
		
		The listDatabases command estimates the size of all collections and collection indexes for each database. However, relational tables and indexes are excluded from this size calculation.
		Important: The listDatabases command performs expensive and CPU-intensive computations on the size of each database in the Informix instance. You can decrease the expense by using the sizeStrategy option.
		
		sizeStrategy
		You can use this option to configure the strategy for calculating database size when the listDatabases command is run.
		
		 
		
		estimate
		Estimate the size of the documents in the collection by using 1000 (or 0.1%) of the documents. This is the default value.
		
		The following example estimates the collection size by using the default of 1000 (or 0.1%) of the documents:
		`db.runCommand({listDatabases:1,`
		`sizeStrategy:"estimate"})`
		
		estimate: *n*
		Estimate the size of the documents in a collection by sampling one document for every *n* documents in the collection.
		
		The following example estimates the collection size by using sample size of 0.5% or 1/200th of the documents:
		`db.runCommand({listDatabases:1,`
		`sizeStrategy:{estimate:200}})`
		
		compute
		Compute the exact size of each database.
		`db.runCommand({listDatabases:1,`
		`sizeStrategy:"compute"})`
		
		none
		List the databases but do not compute the size. The database size is listed as 0.
		`db.runCommand({listDatabases:1,`
		`sizeStrategy:"none"})`
		
		perDatabaseSpace
		Calculate the size of a database by adding the sizes for all dbspaces, sbspaces, and blobspaces that are assigned to the tenant database.
		Important: The **perDatabaseSpace** option applies only to tenant databases that are created by the multi-tenancy feature.
		`db.runCommand({listDatabases:1 ,`
		`sizeStrategy:"perDatabaseSpace"})`

ping	Yes	

Table 4-8. Diagnostic commands (continued)

Name	Supported	Details
serverStatus	Yes	
top	No	
whatsmyuri	Yes	

Instance administration commands

Table 4-9. Instance administration commands

Name	JSON collections	Relational tables	Details
clone	No	No	
cloneCollection	No	No	
cloneCollectionAsCapped	No	No	
collMod	No	No	
compact	No	No	
convertToCapped	No	No	
copydb	No	No	
create	Yes	No	Informix does not support the following flags: • capped • autoIndexID • size • max
createIndexes	Yes	Yes	
drop	Yes	Yes	Informix does not lock the database to block concurrent activity.
dropDatabase	Yes	Yes	
dropIndexes	Yes	No	The MongoDB deleteIndexes command is equivalent.
filemd5	No	No	
fsync	No	No	
getParameter	No	No	
listCollections	Yes	Yes	The includeRelational and includeSystem flags are supported to include or exclude relational or system tables in the results. Default is includeRelational=true and includeSystem=false.
listIndexes	Yes	Yes	
logRotate	No	No	
reIndex	No	No	
renameCollection	No	No	
repairDatabase	No	No	
setParameter	No	No	

Table 4-9. Instance administration commands (continued)

Name	JSON collections	Relational tables	Details
shutdown	Yes	Yes	The timeoutSecs flag is supported. In the Informix, the timeoutSecs flag determines the number of seconds that the wire listener waits for a busy client to stop working before forcibly terminating the session. The force flag is not supported.
touch	No	No	

Replication commands

Table 4-10. Replication commands

Name	Supported
isMaster	Yes
replSetFreeze	No
replSetGetStatus	No
replSetInitiate	No
replSetMaintenance	No
replSetReconfig	No
replSetStepDown	No
replSetSyncFrom	No
Resync	No

Sharding commands

Table 4-11. Replication commands

Name	JSON collections	Relational tables	Details
addShard	Yes	Yes	The MongoDB maxSize and name options are not supported. In addition to the MongoDB command syntax for adding a single shard server, you can use the Informix specific syntax to add multiple shard servers in one command by sending the list of shard servers as an array. For more information, see "Creating a shard cluster with MongoDB commands" on page 3-2.
enableSharding	Yes	Yes	This action is not required for Informix and therefore this command has no affect for Informix.
flushRouterConfig	No	No	
isdbgrid	Yes	Yes	
listShards	Yes	Yes	The equivalent Informix command is **cdr list server**.
movePrimary	No	No	
removeShard	No	No	

Table 4-11. Replication commands (continued)

Name	JSON collections	Relational tables	Details
shardCollection	Yes	Yes	The equivalent Informix command is **cdr define shardCollection**. The MongoDB unique and numInitialChunks options are not supported.
shardingState	No	No	
split	No	No	

For more information about the MongoDB features, see http://docs.mongodb.org/manual/reference/.

Related tasks:

"Defining a sharding schema with an expression" on page 3-5

"Viewing shard-cluster participants" on page 3-10

"Creating a shard cluster with MongoDB commands" on page 3-2

"Defining a sharding schema with a hash algorithm" on page 3-4

Informix JSON commands

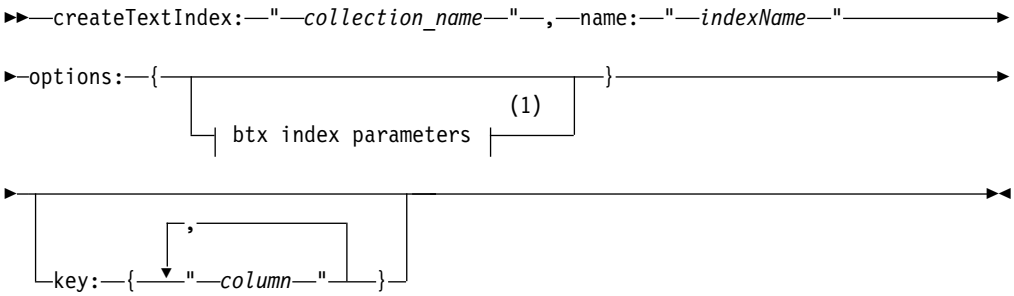
The Informix JSON commands are available in addition to the supported MongoDB commands. These commands enable functionality that is supported by Informix and they are run by using the MongoDB API.

- "createTextIndex"
- "exportCollection" on page 4-12
- "importCollection" on page 4-14
- "lockAccounts" on page 4-15
- "transaction" on page 4-16
- "unlockAccounts" on page 4-17

createTextIndex

Create Informix **bts** indexes.

Important: If you create text indexes by using the Informix createTextIndex command, you must query them by using the Informix $ifxtext query operator. If you create text indexes by using the MongoDB syntax for text indexes, you must query them by using the MongoDB $text query operator.



Notes:

1 See bts access method syntax.

createTextIndex
This required parameter specifies the name of the collection or relational table where the **bts** index is created.

name
This required parameter specifies the name of the **bts** index.

options
This required parameter specifies the name-value pairs for the **bts** parameters that are used when creating the index. If no parameter values are required, you can specify an empty document.

Use **bts** index parameters to customize the behavior of the index and how text is indexed. Include JSON index parameters to control how JSON and BSON documents are indexed. For example, you can index the documents as field name-value pairs instead of as unstructured text so that you can search for text by field. The name and values of the **bts** index parameters in the options parameter are the same as the syntax for creating a **bts** access method with the SQL CREATE INDEX statement.

key
This parameter is required if you are indexing relational tables, but optional if you are indexing collections. This parameter specifies which columns to index for relational tables.

The following example creates an index named myidx in the mytab relational table on the title and abstract columns:

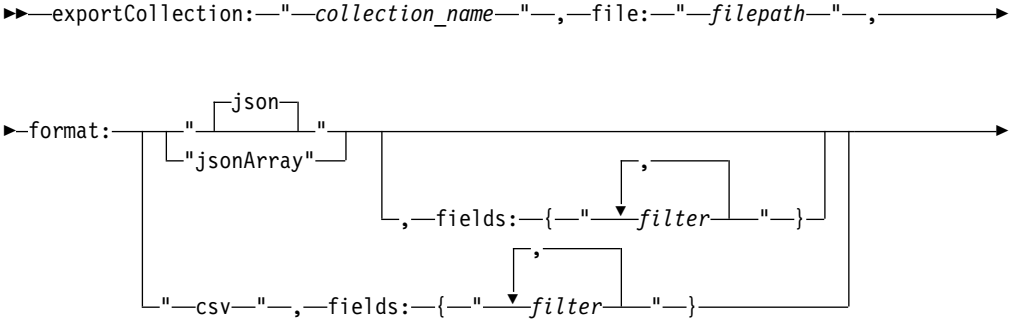
```
db.runCommand({
 createTextIndex:"mytab",
 name:"myidx",
 key:{"title":"text","abstract":"text"},
 options:{}})
```

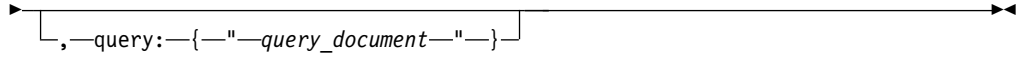
The following example creates an index named articlesIdx on the articles collection by using the **bts** paramter `all_json_names="yes"`.

```
db.runCommand({
 createTextIndex:"articles",
 name:"articlesIdx",
 options:{all_json_names:"yes"}})
```

exportCollection

Export JSON collections from the wire listener to a file.





exportCollection
This required parameter specifies the collection name to export.

file
This required parameter specifies the output file path on the host machine where the wire listener is running. For example:

- UNIX is `file:"/tmp/export.out"`
- Windows is `file:"C:/temp/export.out"`

format
This required parameter specifies the exported file format.

json
The `.json` file format. One JSON-serialized document per line is exported. This is the default value.

The following command exports all documents from the collection that is named c by using the json format:

```
> db.runCommand({exportCollection:"c",file:"/tmp/export.out"
 ,format:"json"})
{
 "ok":1,
 "n":1000,
 "millis":NumberLong(119),
 "rate":8403.361344537816
}
```

Where `"n"` is the number of documents that are exported, `"millis"` is the number of milliseconds it took to export, and `"rate"` is the number of documents per second that are exported.

jsonArray
The `.jsonArray` file format. This format exports an array of JSON-serialized documents with no line breaks. The array format is JSON-standard.

The following command exports all documents from the collection c by using the jsonArray format:

```
> db.runCommand({exportCollection:"c",file:"/tmp/export.out"
 , format:"jsonArray"})
{
 "ok":1,
 "n":1000,
 "millis":NumberLong(81),
 "rate":12345.67901234568
}
```

Where `"n"` is the number of documents that are exported, `"millis"` is the number of milliseconds it took to export, and `"rate"` is the number of documents per second that are exported.

csv
The `.csv` file format. Comma-separated values are exported. You must specify which fields to export from each document. The first line of the `.csv` file contains the fields and all subsequent lines contain the comma-separated document values.

fields
This parameter specifies which fields are included in the output file. This parameter is required for the csv format, but optional for the json and jsonArray formats.

The following command exports all documents from the collection that is named c by using the csv format, only output the "_id" and "name" fields:

```
> db.runCommand({exportCollection:"c",file:"/tmp/export.out"
 ,format:"csv",fields:{"_id":1 ,"name":"1"}})
{
 "ok":1,
 "n":1000,
 "millis":NumberLong(57),
 "rate":17543.859649122805
}
```

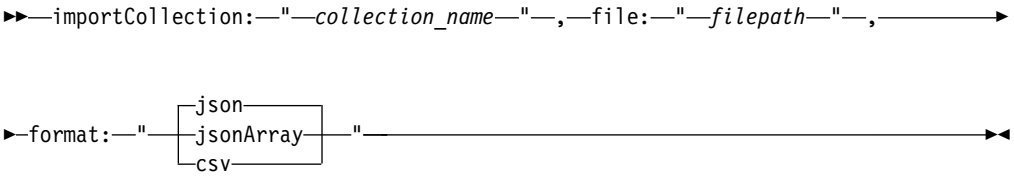
Where "n" is the number of documents that are exported, "millis" is the number of milliseconds it took to export, and "rate" is the number of documents per second that are exported.

query
This optional parameter specifies a query document that identifies which documents are exported. The following example exports all documents from the collection that is named c that have a "qty" field that is less than 100:

```
> db.runCommand({exportCollection:"c",file:"/tmp/export.out"
 ,format:"json",query:{"qty":{"$lt":100}}})
{"ok":1,"n":100,"millis":NumberLong(5),"rate":20000}
```

importCollection

Import JSON collections from the wire listener to a file.



importCollection
The required parameter specifies the collection name to import.

file
This required parameter specifies the input file path. For example, file: "/tmp/import.json".

Important: The input file must be located on the same host machine where the wire listener is running.

format
This required parameter specifies the imported file format.

json
The .json file format. This is the default value.

The following example imports documents from the collection that is named c by using the json format:

```
> db.runCommand({importCollection:"c",file:"/tmp/import.out"
 ,format:"json"})
```

jsonArray
The `.jsonArray` file format.

The following example imports documents from the collection c by using the jsonArray format:

```
> db.runCommand({exportCollection:"c",file:"/tmp/import.out"
 ,format:"jsonArray"})
```

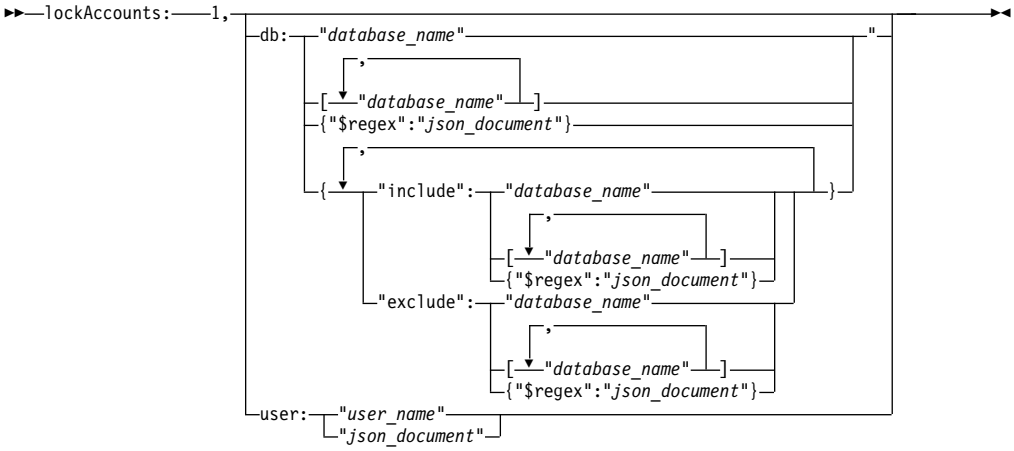
csv
The `.csv` file format.

lockAccounts

Lock a database or user account.

Important:

- To run this command, you must be the instance administrator.
- If you specify the `lockAccounts:1` command without specifying a **db** or **user** argument, all accounts in all databases are locked.



lockAccounts:1
This required parameter locks a database or user account.

db This optional parameter specifies the database name of an account to lock. For example, to lock all accounts in database that is named `foo`:

```
> db.runCommand({lockAccounts:1,db:"foo"})
```

exclude
This optional parameter specifies the databases to exclude. For example, to lock all accounts on the system except those in the databases named `alpha` and `beta`:

```
> db.runCommand({lockAccounts:1,db:{"exclude":["alpha","beta"]})
```

include
This optional parameter specifies the databases to include. For example, to lock all accounts in the databases named `delta` and `gamma`:

```
> db.runCommand({lockAccounts:1,db:{"include":["delta","gamma"]})
```

$regex
This optional MongoDB evaluation query operator selects values from a specified JSON document. For example, to lock accounts for databases that begin with the character `a.` and end in `e`:

```
> db.runCommand({lockAccounts:1,db:{"$regex":"a.*e"})
```

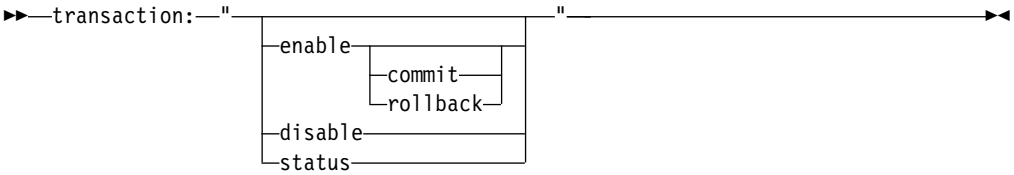
user
This optional parameter specifies the user accounts to lock. For example, to lock the account of all users that are not named `alice`:

```
> db.runCommand({lockAccounts:1,user:{$ne:"alice"}});
```

transaction

Enable or disable transaction support for a session. This command binds or unbinds a connection to the current MongoDB session in a database. The relationship between a MongoDB session and the Informix JDBC connection is not static.

Important: This command is not supported for queries that are run on shard servers.



enable
This optional parameter enables transaction mode for the current session in the current database. The following example shows how to enable transaction mode:

```
> db.runCommand({transaction:"enable"})
{"ok":1}
```

commit
If transactions are enabled, this optional parameter commits the current transaction. If transactions are disabled, an error is shown. The following example shows how to commit the current transaction:

```
> db.c.insert({a:1})
> db.runCommand({transaction:"commit"})
{"ok":1}
```

rollback
If transactions are enabled, this optional parameter rolls back the current transaction. If transactions are disabled, an error is shown. The following example shows how to roll back the current transaction:

```
> db.c.insert({a:2})
> db.c.find()
{"_id":ObjectId("52a8f9c477a0364542887ed4"),"a":1}
{"_id":ObjectId("52a8f9e877a0364542887ed5"),"a":2}
> db.runCommand({transaction:"rollback"})
{"ok":1}
```

disable
This optional parameter disables transaction mode for the current session in the current database. The following example shows how to disable for transaction mode:

```
> db.c.find()
{"_id":ObjectId("52a8f9c477a0364542887ed4"),"a":1}
> db.runCommand({transaction:"disable"})
{"ok":1}
```

status
This optional parameter prints status information to indicate whether transaction mode is enabled, and if transactions are supported by the current database. The following example shows how to print status information:

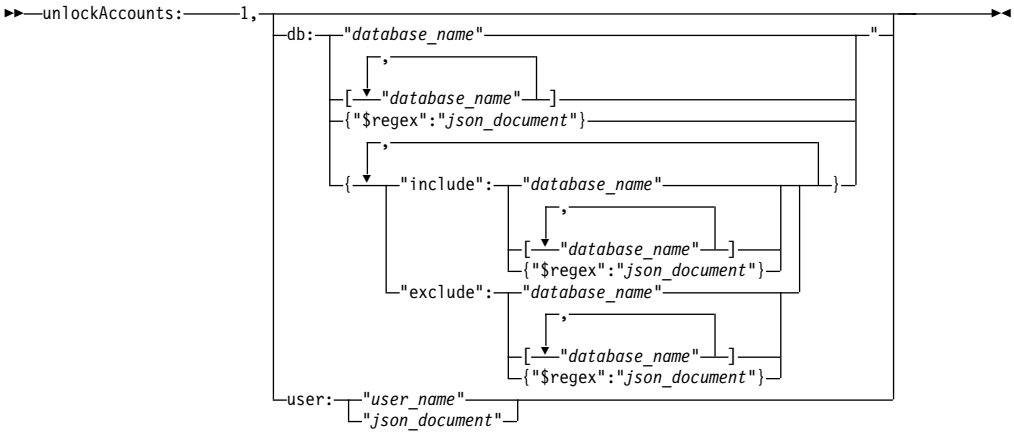
```
> db.runCommand({transaction:"status"})
{"enabled":true,"supports":true,"ok":1}
```

unlockAccounts

Unlock a database or user account.

Important:

- To run this command, you must be the instance administrator.
- If you specify the `unlockAccounts:1` command without specifying a **db** or **user** argument, all accounts in all databases are unlocked.



unlockAccounts:1
This required parameter unlocks a database or user account.

db This optional parameter specifies the database name of an account to unlock. For example, to unlock all accounts in database that is named `foo`:

```
> db.runCommand({unlockAccounts:1,db:"foo"})
```

exclude
This optional parameter specifies the databases to exclude. For example, to unlock all accounts on the system except those in the databases named `alpha` and `beta`:

```
> db.runCommand({unlockAccounts:1,db:{"exclude":["alpha","beta"]})
```

include
This optional parameter specifies the databases to include. For example, to unlock all accounts in the databases named `delta` and `gamma`:

```
> db.runCommand({unlockAccounts:1,db:{"include":["delta","gamma"]})
```

$regex
This optional MongoDB evaluation query operator selects values from a specified JSON document. For example, to unlock accounts for databases that begin with the character `a.` and end in `e`:

```
> db.runCommand({unlockAccounts:1,db:{"$regex":"a.*e"})
```

user

This optional parameter specifies the user accounts to unlock. For example, to unlock the account of all users that are not named `alice`:

```
> db.runCommand({unlockAccounts:1,user:{$ne:"alice"}});
```

Operators

The MongoDB operators that are supported by Informix are sorted into logical areas.

MongoDB read and write operations on existing relational tables are run as if the table were a collection. The wire listener determines whether the accessed entity is a relational table and converts the basic MongoDB operations on that table to SQL, and then converts the returned values back into a JSON document. The initial access to an entity results in an extra call to the Informix server. However, the wire listener caches the name and type of an entity so that subsequent operations do not require an extra call.

MongoDB operators are supported on both JSON collections and relational tables, unless explicitly stated otherwise.

Query and projection operators

Informix supports a subset of the MongoDB query and projection operators.

You can refine your queries with the MongoDB query and projection operators. For example, in the **mongo** shell, to find members of the **cartype** collection with an age greater than 10, you can use the $gt operator:
`db.cartype.find({"age":{"$gt":10.0}})`.

The JSON wire listener supports the skip, limit, and sort query options. You can set these options by using the **mongo** shell or MongoDB drivers.

- "Query selectors"
- "Projection operators" on page 4-20

Query selectors

Use query selectors to select specific data from queries.

Array query operators

Table 4-12. Array query operators

MongoDB command	JSON collections	Relational tables	Details
$elemMatch	Yes	No	
$size	Yes	No	

Comparison query operators

Table 4-13. Comparison query operators

MongoDB command	JSON collections	Relational tables	Details
$all	Yes	Yes	Supported for primitive values and simple queries only. The operator is only supported when it is the only condition in the query document.

Table 4-13. Comparison query operators (continued)

MongoDB command	JSON collections	Relational tables	Details
$gt	Yes	Yes	
$gte	Yes	Yes	
$in	Yes	Yes	
$lt	Yes	Yes	
$lte	Yes	Yes	
$ne	Yes	Yes	
$nin	Yes	Yes	
$query	Yes	Yes	

Element query operators

Table 4-14. Element query operators

MongoDB command	JSON collections	Relational tables	Details
$exists	Yes	No	
$type	Yes	No	

Evaluation

Table 4-15. Evaluation query operators

MongoDB command	JSON collections	Relational tables	Details
$mod	Yes	Yes	
$regex	Yes	Yes	Supported for string matching, similar to queries that use the SQL LIKE condition. Pattern matching that uses regular expression special characters is not supported.
$text	Yes	Yes	The $text query operator support is based on MongoDB version 2.6. You can customize your text index and take advantage of additional text query options by creating a basic text search index with the createTextIndex command. For more information, see "Informix JSON commands" on page 4-11.
$where	No	No	

Geospatial query operators

Geospatial queries are supported by using the GeoJSON format. The legacy coordinate pairs are not supported.

Table 4-16. Geospatial query operators

MongoDB command	JSON collections	Relational tables	Details
$geoWithin	Yes	No	
$geoIntersects	Yes	No	
$near	Yes	No	
$nearSphere	Yes	No	

JavaScript query operators

The JavaScript query operators are not supported.

Logical query operators

Table 4-17. Logical query operators

MongoDB command	JSON collections	Relational tables	Details
$and	Yes	Yes	
$or	Yes	Yes	
$not	Yes	Yes	
$nor	Yes	Yes	

Projection operators

Use projection operators to select specific data from a document.

Comparison query operators

Table 4-18. Comparison query operators

MongoDB command	JSON collections	Relational tables	Details
$	No	No	
$elemMatch	Yes	No	
$eq	Yes	Yes	
$comment	No	No	
$explain	Yes	Yes	
$hint	Yes	No	
$maxScan	No	No	
$max	No	No	
$meta	Yes	Yes	
$min	No	No	
$orderby	Yes	Yes	
$returnkey	No	No	
$showdiskLoc	No	No	
$slice	No	No	
$snapshot	No	No	

For more information about the MongoDB features, see http://docs.mongodb.org/manual/reference/.

Update operators

Informix supports a subset the MongoDB update operators.

You can use update operators to modify or add data in your database. For example, in the **mongo** shell, to change the username to atlas in the document with the _id of 101 in the users collection, you can use the $set operator: `db.users.update({"_id":101},{"$set":{"username":"atlas"}})`.

Array update operators

Table 4-19. Array update operators

MongoDB command	JSON collections	Relational tables	Details
$	No	No	
$addToSet	Yes	No	Supported for primitive values only. The operator is not supported on arrays and objects.
$pop	Yes	No	
$pullAll	Yes	No	Supported for primitive values only. The operator is not supported on arrays and objects.
$pull	Yes	No	Supported for primitive values only. The operator is not supported on arrays and objects.
$pushAll	Yes	No	
$push	Yes	No	

Array update operators modifiers

Table 4-20. Array update modifiers

MongoDB command	JSON collections	Relational tables	Details
$each	Yes	No	
$slice	Yes	No	
$sort	Yes	No	
$position	No	No	

Bitwise update operators

Table 4-21. Bitwise update operators

MongoDB command	JSON collections	Relational tables	Details
$bit	Yes	No	

Field update operators

Table 4-22. Field update operators

MongoDB command	JSON collections	Relational tables	Details
$currentDate	Yes	Yes	
$inc	Yes	Yes	
$max	Yes	Yes	
$min	Yes	Yes	
$mul	Yes	Yes	
$rename	Yes	No	
$setOnInsert	Yes	No	
$set	Yes	Yes	
$unset	Yes	Yes	

Isolation update operators
The isolation update operators are not supported.

For more information about the MongoDB features, see http://docs.mongodb.org/manual/reference/.

Informix query operators

The Informix query operators are extensions to the MongoDB API.

You can use the Informix query operators in all MongoDB functions that accept query operators, for example find() or findOne().

`$ifxtext`
The $ifxtext query operator is similar to the MongoDB $text operator, except that it passes the search string as-is to the **bts_contains()** function.

When using relational tables, the MongoDB $text and Informix $ifxtext query operators both require a column name, specified by $key, in addition to the $search string.

The search string can be a word or a phrase as well as optional query term modifiers, operators, and stopwords. You can include field names to search in specific fields. The syntax of the search string in the $ifxtext query operator is the same as the syntax of the search criteria in the **bts_contains()** function that you include in an SQL query.

In the following example, a single-character wildcard search is run for the strings `text` or `test`:

```
db.collection.find( { "$ifxtext" : { "$search" : "te?t" } } )
```

`$like`
The $like query operator tests for matching character strings and maps to the SQL LIKE query operator. For more information about the SQL LIKE query operator, see LIKE Operator.

In the following example, a wildcard search is run for strings that contain `Informix`:

```
db.collection.find( { "$like" : "%Informix%" )
```

Related information:

Basic Text Search query syntax

Aggregation framework operators

The MongoDB aggregation framework operators that are supported by Informix are sorted into logical areas.

You can use aggregation framework operators to aggregate and manipulate documents as they move through the aggregation pipeline stages.

- "Pipeline operators" on page 4-23
- "Expression operators" on page 4-23

Pipeline operators

Table 4-23. Pipeline operators

MongoDB command	JSON collections	Relational tables	Details
$project	Partial	Partial	• You can use $project to include fields from the original document, for example `{ $project : { title : 1 , author : 1 }}`. • You cannot use $project to insert computed fields, rename fields, or create and populate fields that hold subdocuments. • Projection operators are not supported.
$match	Yes	Yes	
$redact	No	No	
$limit	Yes	Yes	
$skip	Yes	Yes	
$unwind	Yes	No	
$group	Yes	Yes	
$sort	Yes	Yes	
$geoNear	Yes	No	• Supported by using the GeoJSON format. The MongoDB legacy coordinate pairs are not supported. • You cannot use dot notation for the distanceField and includeLocs parameters.
$out	Yes	Yes	

Expression operators

$group operators

Table 4-24. $group operators

MongoDB command	JSON collections	Relational tables
$addToSet	Yes	No
$first	Yes	Yes
$last	Yes	Yes
$max	Yes	Yes
$min	Yes	Yes
$avg	Yes	Yes
$push	Yes	No
$sum	Yes	Yes

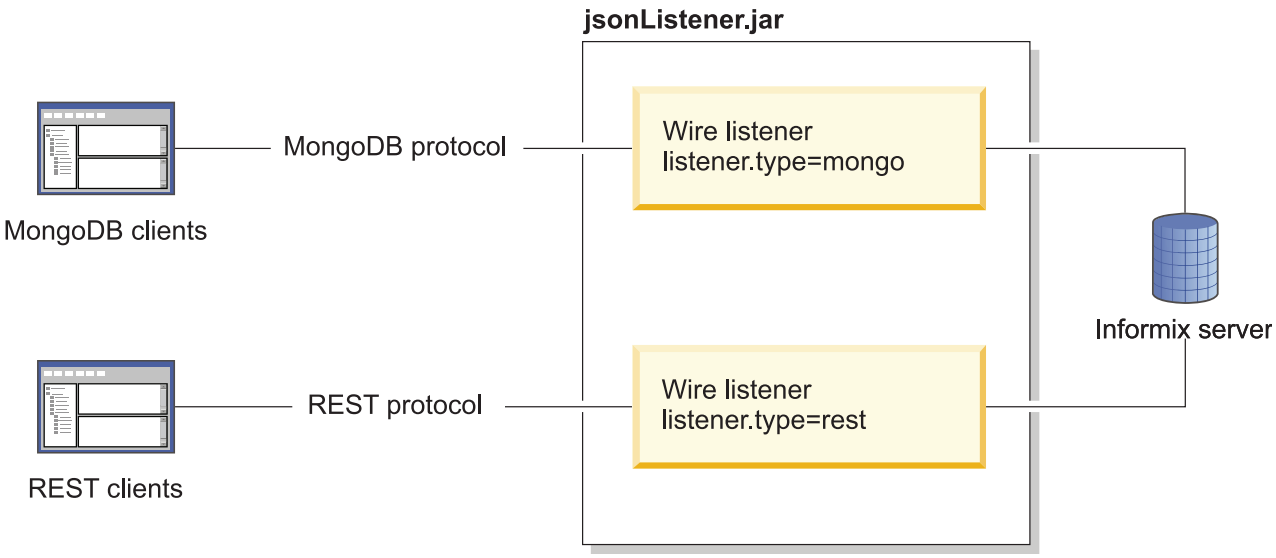
For more information about the MongoDB features, see http://docs.mongodb.org/manual/reference/.

Chapter 5. REST API

The REST API provides an alternative method for accessing JSON collections in Informix and provides driverless access to your data.

With the REST API, you can use MongoDB and SQL queries against JSON and BSON document collections, traditional relational tables, and time series data. The REST API uses MongoDB syntax and returns JSON documents.

The `jsonListener.jar` file is the executable file that includes the wire listener configuration file, named `jsonListener.properties` by default, which defines the operational characteristics for theMongoDB API and REST API.



Related tasks:

"Starting the wire listener" on page 2-33

REST API syntax

A subset of the HTTP methods are supported by the REST API. These methods are DELETE, GET, POST, and PUT.

The examples shown in this topic contain line breaks for page formatting; however, the REST API does not allow line breaks.

POST

The POST method maps to the MongoDB insert or create command.

Table 5-1. Supported POST method syntax

Method	Path	Description
POST	/	Create a database.
POST	/*databaseName*	Create a collection. ***databaseName*** The database name.
POST	/*databaseName*/*collectionName*	Create a document. ***databaseName*** The database name. ***collectionName*** The collection name.

Create a database

This example creates a database with the locale specified.

Request:

Specify the POST method:

```
POST /
```

Data: Specify database name mydb and an English UTF-8 locale:

```
{name:"mydb",locale:"en_us.utf8"}
```

Response:

The following response indicates that the operation was successful:

```
{"msg":"created db 'mydb'","ok":true}
```

Create a collection

This example creates a collection in the mydb database.

Request:

Specify the POST method and the database name as mydb:

```
POST /mydb
```

Data: Specify the collection name as bar:

```
{name:"bar"}
```

Response:

The following response indicates that the operation was successful:

```
{"msg":"created collection mydb.bar","ok":true}
```

Create a relational table

This example creates a relational table in an existing database.

Request:

Specify the POST method and stores_mydb as the database:

```
POST /stores_mydb
```

Data: Specify the table attributes:

```
{ name: "rel",
  options: {
  columns:[{name:"id",type:"int",primaryKey:true,},
```

```
        {name:"name",type:"varchar(255)"},
         {name:"age",type:"int",notNull:false}]
    }
}
```

Response:

The following response indicates that the operation was successful:

```
{msg: "created collection stores_mydb.rel" ok: true}
```

Insert a single document

This example inserts a document into an existing collection.

Request:

Specify the POST method, mydb database, and people collection:

```
POST /mydb/people
```

Data: Specify John Doe age 31:

```
{firstName:"John",lastName:"Doe",age:31}
```

Response:

Here is a successful response:

```
{"n":1,"ok":true}
```

Insert multiple documents into a collection

This example inserts multiple documents into a collection.

Request:

Specify the POST method, mydb database, and people collection:

```
POST /mydb/people
```

Data: Specify John Doe age 31 and Jane Doe age 31:

```
[{firstName:"John",lastName:"Doe",age:31},
{firstName:"Jane",lastName:"Doe",age:31}]
```

Response:

Here is a successful response:

```
{"n":2,"ok":true}
```

PUT

The PUT method maps to the MongoDB update command.

Table 5-2. Supported PUT method syntax

Method	Path	Description
PUT	*/databaseName/ collectionName?queryParameters*	Update a document. ***databaseName*** The database name. ***collectionName*** The collection name. ***queryParameters*** The supported Informix *queryParameters* are query, upsert, and multiupdate. These map to the equivalent MongoDB query, insert, and multi query parameters, respectively.

Update a document in a collection

This example updates the value for Larry in an existing collection, from age 49 to 25:

```
[{"_id":{"$oid":"536d20f1559a60e677d7ed1b"},"firstName":"Larry"
,"lastName":"Doe","age":49},{"_id":{"$oid":"536d20f1559a60e677d7ed1c"}
,"firstName":"Bob","lastName":"Doe","age":47}]
```

Request:

Specify the PUT method and query the name Larry:

```
PUT /mydb/people?query={firstName:"Larry"}
```

Data: Specify the MongoDB $set operator with age 25:

```
{"$set":{age:25}}
```

Response:

Here is a successful response:

```
{"n":1,"ok":true}
```

GET

The GET method maps to the MongoDB query command.

Table 5-3. Supported GET method syntax

Method	Path	Description
GET	/	List databases
GET	*/databaseName*	List collections ***databaseName*** The database name.
GET	*/databaseName/collectionName?queryParameters*	Query the collection. ***databaseName*** The database name. ***collectionName*** The collection name. ***queryParameters*** The query parameters. The supported Informix *queryParameters* are batchSize, query, fields, and sort. These map to the equivalent MongoDB batchSize, query, fields, and sort parameters.
GET	*/databaseName/*$cmd?query={*command_document*}	Run the Informix or MongoDB JSON command. ***databaseName*** The database name. ***command_document*** The Informix or MongoDB JSON command document. Specify the command document in the same format that is used by the **db.runCommand()** in the **mongo** shell.

List databases

This example lists all of the databases on the server.

Request:

Specify the GET method and forward slash (/):

```
GET /
```

Data: None.

Response:

Here is a successful response:

```
[ "mydb" , "test" ]
```

List all collections

This example lists all of the collections in a database.

Request:

Specify the GET method and mydb database:

```
GET /mydb
```

Data: None.

Response:

Here is a successful response:

```
["bar"]
```

Query a collection and sort the results in ascending order

This example sorts the query results in ascending order by age.

Request:

Specify the GET method, mydb database, people collection, and query with the sort parameter. The sort parameter specifies ascending order (age:1), and filters id (_id:0) and last name (lastName:0) from the response:

```
GET /mydb/people?sort={age:1}&fields={_id:0,lastName:0}
```

Data: None.

Response:

The first names are displayed in ascending order with the _id and lastName filtered from the response:

```
[{"firstName":"Sherry","age":31},
{"firstName":"John","age":31},
{"firstName":"Bob","age":47},
{"firstName":"Larry","age":49}]
```

Run the collStats command to get statistics about a collection

This example submits the MongoDB collStats command by using the REST API to get statistics about the jsonlog collection.

Here is the MongoDB shell syntax:

```
db.runCommand({collStats:"jsonlog"})
```

Request:

Specify the GET method, mydb database, and the collStats command document as the query:

```
GET /mydb/$cmd?query={collStats:"jsonlog"}
```

Data: None.

Response:

```
[
     {
         "ns":"mydb.jsonlog",
         "count":1000,
         "size":322065,
         "avgObjSize":322,
         "storageSize":323584,
         "numExtents":158,
         "nindexes":1,
         "lastExtentSize":2048,
```

```
            "paddingFactor":0,
            "flags":1,
            "indexSizes":
            {
                "_id_":49152
            },
            "totalIndexSize":49152,
            "ok":1
        }
]
```

DELETE

The DELETE method maps to the MongoDB delete command.

Table 5-4. Supported DELETE method syntax

Method	Path	Description
DELETE	/	Delete all databases.
DELETE	*/databaseName*	Delete a database. ***databaseName*** The database name.
DELETE	*/databaseName/collectionName*	Delete a collection. ***databaseName*** The database name. ***collectionName*** The collection name.
DELETE	*/databaseName/collectionName?queryParameter*	Delete all documents that satisfy the query from a collection. ***databaseName*** The database name. ***collectionName*** The collection name. ***queryParameters*** The query parameters. The supported Informix *queryParameter* is query. This map to the equivalent MongoDB query parameter.

Delete a database

This example deletes a database called mydb.

Request:

Specify the DELETE method and the mydb database:

```
DELETE /mydb
```

Data: None.

Response:

Here is a successful response:

```
{msg: "dropped database", ns: "mydb", ok: true}
```

Delete a collection

This example deletes a collection from a database.

Request:

Specify the DELETE method, mydb database, and bar collection:

```
DELETE /mydb/bar
```

Data: None.

Response:

Here is a successful response:

```
{"msg":"dropped collection", "ns":"mydb.bar", "ok":true}
```

Delete documents from a collection

This example deletes documents from a collection that contains the user "bob".

Request:

Specify the DELETE method, mydb database, people collection, and the query condition:

```
DELETE /mydb/people?query={user:"bob"}
```

Data: None.

Response:

Here is a successful response where *n* indicates the number of documents deleted.

```
{"n":1,"ok":true}
```

Related concepts:

Chapter 6, “Create time series through the wire listener,” on page 6-1

Related tasks:

“Running multiple wire listeners” on page 2-35

Related reference:

“The wire listener configuration file” on page 2-3

Chapter 6. Create time series through the wire listener

You can create and manage time series with the REST API or the MongoDB API through the wire listener. You create time series objects by adding definitions to time series collections. You interact with time series data through a virtual table. For example, you can program sensor devices that do not have client drivers to load time series data directly into the database with HTTP commands from the REST API.

Prerequisites

Before you create a time series, you must understand time series concepts, the properties of your data, and how much storage space your data requires. For an overview of time series concepts and guidance on how to design your time series solution, see Informix TimeSeries solution.

You must also configure the wire listener for the REST API or the MongoDB API.

Restrictions

The following restrictions apply when you create a time series through the wire listener:

- You cannot define hertz or compressed time series.
- You cannot define rolling window containers.
- You cannot load time series data through a loader program. You must load time series data through a virtual table.
- You cannot run time series SQL routines or methods from the time series Java class library. You operate on the data through a virtual table.

Creating a time series

To create a time series through the wire listener:

1. Choose a predefined calendar from the `system.timeseries.calendar` collection or create a calendar by adding a document to the `system.timeseries.calendar` collection.
2. Create a **TimeSeries** row type by adding a document to the `system.timeseries.rowType` collection.
3. Create a container by adding a document to the `system.timeseries.container` collection.
4. Create a time series table with the time series table format syntax.
5. Instantiate the time series by creating a virtual table with the time series virtual table format syntax.
6. Load time series data by inserting documents into the virtual table.

After you create and load a time series, you query and update the data though the virtual table.

Related reference:

"REST API syntax" on page 5-1

Time series collections and table formats

You can add, view, and remove documents from the time series collections with REST API and MongoDB API methods to create and manage your time series. You must use a specific format to create time series tables and virtual tables that are based on time series tables.

For the REST API, use the GET, POST, and DELETE methods to view, insert, or delete data in the time series collections.

For the MongoDB API, use the query, create, or remove methods to view, insert, or delete data in the time series collections.

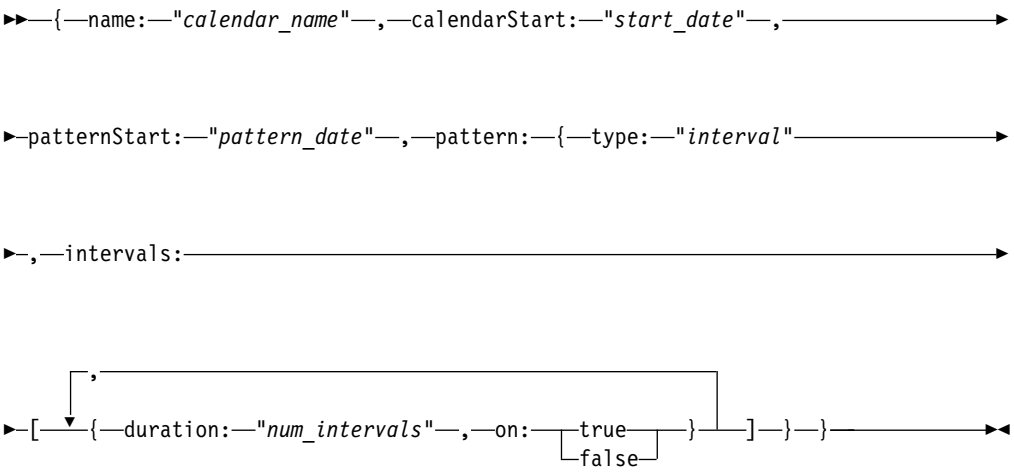
The time series collections are virtual collections that are used to manage the objects that are required to store time series data in a database.

system.timeseries.calendar collection

The `system.timeseries.calendar` collection stores the definitions of predefined and user-defined calendars. A calendar controls the times at which time series data can be stored. The calendar definition embeds the calendar pattern definition. For details and restrictions about calendars, see Calendar data type. For a list of predefined calendars, see Predefined calendars.

Use the following format to add a calendar to the `system.timeseries.calendar` collection.

calendar



name The name of the calendar.

calendarStart
The start date of the calendar.

patternStart
The start date of the calendar pattern.

pattern
The calendar pattern definition.

type The time interval. Valid values for *interval* are: `second`, `minute`, `hour`, `day`, `week`, `month`, `year`.

intervals
The description of when to record data.

duration
The number of intervals, as a positive integer.

on Whether to record data during the interval:

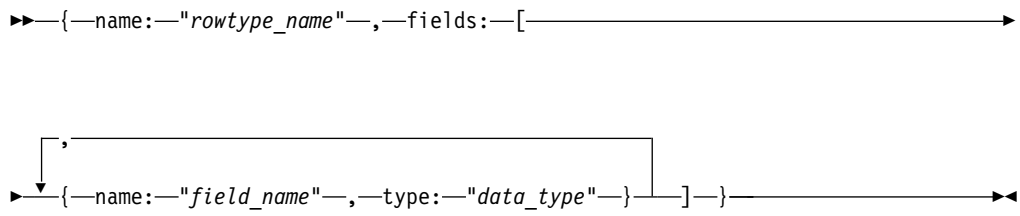
`true` = Recording is on.

`false` = Recording is off.

system.timeseries.rowType collection

The `system.timeseries.rowType` collection stores **TimeSeries** row type definitions. The **TimeSeries** row type defines the structure for the time series data within a single column in the database. For details and restrictions on **TimeSeries** row types, see TimeSeries data type.

Use the following format to add a **TimeSeries** row type to the `system.timeseries.rowType` collection.



name The *rowtype_name* is the name of the **TimeSeries** row type.

fields

name The name of the field in the row data type. The *field_name* must be unique for the row data type. The number of fields in a row type is not restricted.

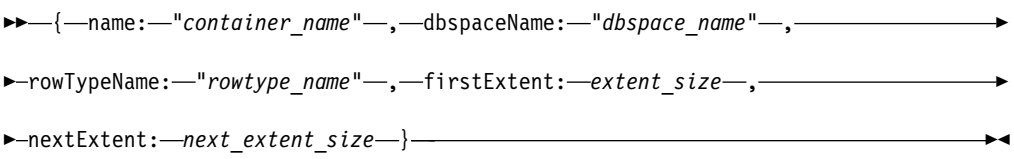
type Must be `datetime year to fraction(5)` for the first field, which contains the time stamp.

The data type of the field. Most data types are valid for fields after the time stamp field.

system.timeseries.container collection

The `system.timeseries.container` collection stores container definitions. Time series data is stored in containers. For details and restrictions on containers, see TSContainerCreate procedure. Rolling window container syntax is not supported.

Use the following format to add a container to the `system.timeseries.container` collection.



name The *container_name* is the name of the container. The container name must be unique.

dbspaceName
The *dbspace_name* is the name of the dbspace for the container.

rowTypeName
The *rowtype_name* is the name of an existing **TimeSeries** row type in the `system.timeseries.rowType` collection.

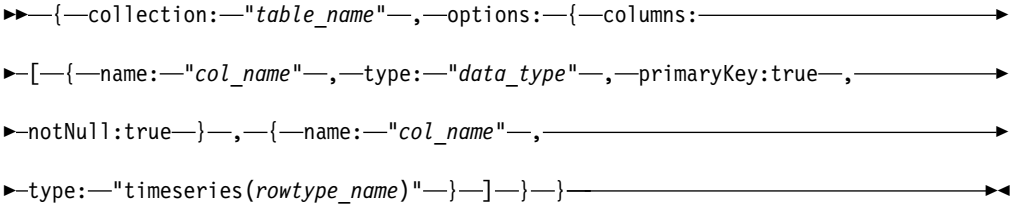
firstExtent
The *extent_size* is a number that represents the first extent size for the container, in KB.

nextExtent
The *next_extent_size* is a number that represents the increments by which the container grows, in KB. The value must be equivalent to at least 4 pages.

Time series table format

A time series table must have a primary key column that does not allow null values. The last column in the time series table must be the **TimeSeries** column. For details and restrictions on time series tables, see Create the database table.

The following format describes the simplest structure of a time series table. You can include other options and columns in a time series table.



collection
The *table_name* is the name of the time series table.

options
The collection definition.

columns
The column definitions.

name The *col_name* is the name of the column.

type The *data_type* is the data type of the column.

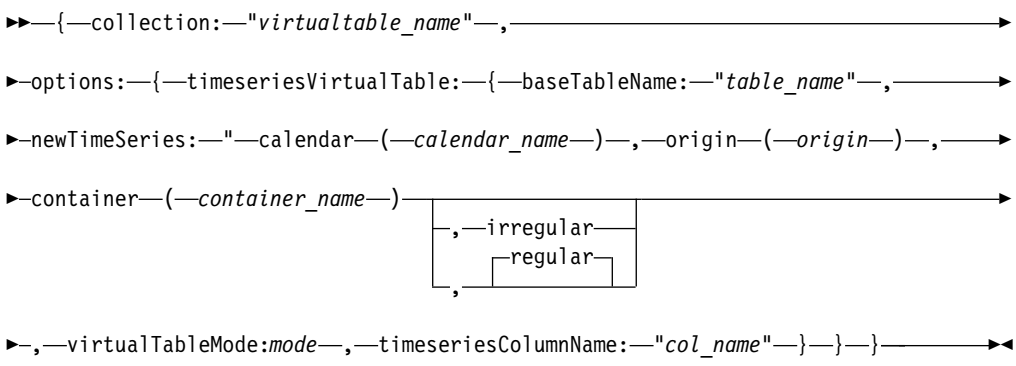
For the **TimeSeries** column, the *rowtype_name* is the name of an existing **TimeSeries** row type in the `system.timeseries.rowType` collection.

primaryKey
`true` = The column is the primary key.

notNull
`true` = The column does not allow null values.

Virtual table format

You use a virtual table that is based on the time series table to insert and query time series data.



collection
The *virtualtable_name* is the name of the virtual table.

options

timeseriesVirtualTable
The definition of the virtual table.

baseTableName
The *table_name* is the name of the time series table.

newTimeseries
The time series definition.

calendar
The *calendar_name* is the name of a calendar in the `system.timeseries.calendar` collection.

origin The *origin* is the first time stamp in the time series. The data type is DATETIME YEAR TO FRACTION(5).

container
The *container_name* is the name of a container in the `system.timeseries.container` collection.

regular
Default. The time series is regular.

irregular
The time series is irregular.

virtualTableMode
The *mode* is the integer value of the TSVTMode parameter that controls the behavior and display of the virtual table

for time series data. For the settings of the TSVTMode parameter, see The TSVTMode parameter.

timeseriesColumnName
The *col_name* is the name of the **TimeSeries** column.

Example: Create a time series through the wire listener

This example shows how to create, load, and query a time series with the MongoDB API or the REST API through the wire listener.

Before you begin

Before you start this example, ensure these tasks are complete:

- Connect to a database in which to create the time series table. You run all methods in the database.
- Configure the wire listener for the MongoDB API or the REST API. For more information, see "Configuring the wire listener for the first time" on page 2-1.
- Define a dbspace that is named **dbspace1**. For more information, see Dbspaces.

About this task

In this example, you create a time series that contains sensor readings about the temperature and humidity in a house. Readings are taken every 10 minutes. The following table lists the time series properties that are used in this example.

Table 6-1. Time series properties used in this example

Time series property	Definition
Timepoint size	10 minutes
When timepoints are valid	Every 10 minutes
Data in the time series	The following data: • Timestamp • A float value that represents temperature • A float value that represents humidity
Time series table	The following columns: • A meter ID column of type INTEGER • A **TimeSeries** data type column
Origin	2014-01-01 00:00:00.00000
Regularity	Regular
Where to store the data	In a container that you create
How to load the data	Through a virtual table
How to access the data	Through a virtual table

Procedure

To create a time series with the MongoDB API **mongo** shell or the REST API:

1. Create a time series calendar. The time series calendar is named **ts_10min**, with a calendar and pattern start date of **2014-01-01 00:00:00**, a calendar pattern that is defined with intervals of minutes, and data is recorded in 10 minute increments after the origin.

   MongoDB API

   Add to the predefined **system.timeseries.calendar** collection.

   ```
   db.system.timeseries.calendar.insert({"name":"ts_10min",
    "calendarStart":"2014-01-01 00:00:00",
    "patternStart":"2014-01-01 00:00:00",
    "pattern":{"type":"minute",
             "intervals":[{"duration":"1","on":"true"},
                       {"duration":"9","on":"false"}]}})
   ```

   REST API

   Request:

   Specify the POST method and the **system.timeseries.calendar** collection:

   ```
   POST /stores_demo/system.timeseries.calendar
   ```

   Data: Specify the calendar attributes:

   ```
   {"name":"ts_10min",
    "calendarStart":"2014-01-01 00:00:00",
    "patternStart":"2014-01-01 00:00:00",
    "pattern":{"type":"minute",
             "intervals":[{"duration":1,"on":true},
                       {"duration":9,"on":false}]}}
   ```

   Response:

   The following response indicates that the operation was successful:

   ```
   [{"ok":true}]
   ```

2. Create a **TimeSeries** row type. The row type is named **reading** and includes fields for timestamp, temperature, and humidity.

   MongoDB API

   Add to the predefined **system.timeseries.rowType** collection.

   ```
   db.system.timeseries.rowType.insert({"name":"reading",
   "fields":
   [{"name":"tstamp","type":"datetime year to fraction(5)"},
           {"name":"temp","type":"float"},
           {"name":"hum","type":"float"}]})
   ```

   REST API

   Request:

   Specify the POST method and the **system.timeseries.rowType** collection:

   ```
   POST /stores_demo/system.timeseries.rowType
   ```

   Data: Specify the row type attributes:

   ```
   {"name":"reading",
   "fields":
   [{"name":"tstamp","type":"datetime year to fraction(5)"},
           {"name":"temp","type":"float"},
           {"name":"hum","type":"float"}]}
   ```

   Response:

   The following response indicates that the operation was successful:

   ```
   [{"ok":true}]
   ```

3. Create a container. The container is named **c_0** and is created in the **dbspace1** dbspace, in the **reading** time series row, with a first extent size of **1000**, and with growth increments of **500**.

   MongoDB API

   Add to the predefined **system.timeseries.container** collection.

   ```
   db.system.timeseries.container.insert({"name":"c_0",
    "dbspaceName":"dbspace1",
    "rowTypeName":"reading",
    "firstExtent":1000,
    "nextExtent":500})
   ```

   REST API

   Request:

   Specify the POST method and the **system.timeseries.container** collection:

   ```
   POST /stores_demo/system.timeseries.container
   ```

   Data: Specify the container attributes:

   ```
   {"name":"c_0",
    "dbspaceName":"dbspace1",
    "rowTypeName":"reading",
    "firstExtent":1000,
    "nextExtent":500}
   ```

   Response:

   The following response indicates that the operation was successful:

   ```
   [{"ok":true}]
   ```

4. Create a time series table. The time series table is named **ts_data1** and includes **id** and **ts** columns.

   MongoDB API

   Create the **ts_data1** time series table:

   ```
   db.runCommand({"create":"ts_data1",
    "columns":[{"name":"id","type":"int","primaryKey":"true",
   "notNull":"true"},
             {"name":"ts","type":"timeseries(reading)"}]})
   ```

   REST API

   Request:

   Specify the GET method:

   ```
   GET /stores_demo/$cmd?query={create:"ts_data1",
      "columns":[{"name":"id","type":"int","primaryKey":true,
   "notNull":true},
              {"name":"ts","type":"timeseries(reading)"}]}
   ```

   Data: None.

   Response:

   The following response indicates that the operation was successful:

   ```
   [{"ok":true}]
   ```

5. Create a virtual table. The virtual table is named **ts_data1_v** and is based on the time series table that is named **ts_data1** and its timeseries column **ts**, using the **ts_10min** calendar, starting on **2014-01-01 00:00:00.00000**, in the time series container **c_0**, with the virtualTableMode parameter set to **0** (default).

   Important: This example contains line breaks for page formatting, however, JSON does not allow line breaks within strings.

MongoDB API

Create the **ts_data1_v** virtual table:

```
db.runCommand({"create":"ts_data1_v",
 "timeseriesVirtualTable":
         {"baseTableName":"ts_data1",
          "newTimeseries":"calendar(ts_10min),origin(2014-01-01
 00:00:00.00000),container(c_0)",
          "virtualTableMode":0,
          "timeseriesColumnName":"ts"}})
```

REST API

Request:

Specify the GET method:

```
GET /stores_demo/$cmd?query={"create":"ts_data1_v",
 "timeseriesVirtualTable":
         {"baseTableName":"ts_data1",
          "newTimeseries":"calendar(ts_10min),
                          origin(2014-01-01 00:00:00.00000),
                          container(c_0)",
                          "virtualTableMode":0,
                          "timeseriesColumnName":"ts"}}
```

Data: None.

Response:

The following response indicates that the operation was successful:

```
[{"ok":true}]
```

6. Load records into the time series by inserting documents into the **ts_data1_v** virtual table.

   Because this time series is regular, you are not required to include the time stamp. The first record is inserted for the origin of the time series, 2014-01-01 00:00:00.00000. The second record has the time stamp 2014-01-01 00:10:00.00000, and the third record has the time stamp 2014-01-01 00:20:00.00000.

   MongoDB API

   Add documents to the **ts_data1_v** virtual table:

   ```
   db.ts_data1_v.insert([{"id":1,"temp":15.0,"hum":20.0},
    {"id":1,"temp":16.2,hum:19.0},{id:1,temp:16.5,hum:22.0}])
   ```

   REST API

   Request:

   Specify the POST method:

   ```
   POST /stores_demo/ts_data1_v
   ```

   Data: Specify the documents to load:

   ```
   [{"id":1,"temp":15.0,"hum":20.0},
   {"id":1,"temp":16.2,"hum":19.0},
   {"id":1,"temp":16.5,"hum":22.0}]
   ```

   Response:

   The following response indicates that the operation was successful:

   ```
   {"ok":true}
   ```

7. Query the time series data by using the **ts_data1_v** virtual table.

   MongoDB API

   Query the **ts_data1_v** virtual table:

```
db.ts_data1_v.find()

Results:
> db.ts_data1_v.find()
{"id":1,"tstamp":ISODate("2014-01-01T06:00:00Z"),"temp":15,"hum":20}
{"id":1,"tstamp":ISODate("2014-01-01T06:10:00Z"),"temp":16.2,"hum":19}
{"id":1,"tstamp":ISODate("2014-01-01T06:20:00Z"),"temp":16.5,"hum":22}
```

REST API

Request:

```
GET /stores_demo/ts_data1_v
```

Data: None.

Response:

The following response indicates that the operation was successful:

```
[{"id":1,"tstamp":{"$date":1388556000000},
"temp":15.0,"hum":20.0},
 {"id":1,"tstamp":{"$date":1388556600000},
"temp":16.2,"hum":19.0},
 {"id":1,"tstamp":{"$date":1388557200000},
"temp":16.5,"hum":22.0}]
```

Example queries of time series data by using the wire listener

These examples show how to query time series data by using the MongoDB API or the REST API.

Before using these examples, you must configure the wire listener for the MongoDB or REST API. For more information, see "Configuring the wire listener for the first time" on page 2-1. These examples are run against the **stores_demo** database. For more information, see dbaccessdemo command: Create demonstration databases. These examples query the **ts_data_v virtual** table that stores the device ID in the **loc_esi_id** column.

List all device IDs

This query returns all unique device IDs.

MongoDB API

Run a **distinct** command on the ts_data_v virtual table:

```
db.ts_data_v.distinct("loc_esi_id")

Results:
["4727354321000111","4727354321046021","4727354321090954",...]
```

REST API

Request:

Specify the GET method on the stores_demo database with the query parameter specified:

```
GET /stores_demo/$cmd?query={"distinct":"ts_data_v",
"key":"loc_esi_id"}
```

Data: None.

Response:

The following response indicates that the operation was successful:

```
[{"values":["4727354321000111","4727354321046021",
"4727354321090954",...],"ok":1.0}]
```

List device IDs that have a value greater than 10

This query returns the list of device IDs that have at least one measured value in the time series that is greater than 10.

MongoDB API

Run a **distinct** command on the ts_data_v table, with $gt value comparison operator specified:

```
db.ts_data_v.distinct("loc_esi_id",{"value":{"$gt":10}})

Results:
["4727354321046021","4727354321132574","4727354321289322",...]
```

REST API

Request:

Specify the GET method with the query condition on the ts_data_v table and the $gt value comparison operator specified:

```
GET /stores_demo/$cmd?query={"distinct":"ts_data_v",
"key":"loc_esi_id","query":{"value":{"$gt":10}}}
```

Data: None.

Response:

The following response indicates that the operation was successful:

```
[{"values":["4727354321046021","4727354321132574",
"4727354321289322",...],"ok":1.0}]
```

Find the data for a specific device ID

This query returns the data for the device with the ID of 4727354321046021.

MongoDB API

Run a **find** command on the ts_data_v virtual table with the loc_esi_id value specified:

```
db.ts_data_v.find({"loc_esi_id":4727354321046021})

Results:
 {"loc_esi_id":"4727354321046021","measure_unit":"KWH",
 "direction":"P","tstamp":ISODate("2010-11-10T06:00:00Z"),
 "value":0.041}
 {"loc_esi_id":"4727354321046021","measure_unit":"KWH",
 "direction":"P","tstamp":ISODate("2010-11-10T06:15:00Z"),
 "value":0.041}
 {"loc_esi_id":"4727354321046021","measure_unit":"KWH",
 "direction":"P","tstamp":ISODate("2010-11-10T06:30:00Z"),
 "value":0.04}
...]
```

REST API

Request:

Specify the GET method on the ts_data_v virtual table, with the loc_esi_id specified on the query operator:

```
GET /stores_demo/ts_data_v?query=
{"loc_esi_id":4727354321046021}
```

Data: None.

Response:

The following response indicates that the operation was successful:

```
[{"loc_esi_id":"4727354321046021","measure_unit":"KWH",
"direction":"P","tstamp":{"$date":1289368800000},"value":0.041},
 {"loc_esi_id":"4727354321046021","measure_unit":"KWH",
"direction":"P","tstamp":{"$date":1289369700000},"value":0.041},
 {"loc_esi_id":"4727354321046021","measure_unit":"KWH",
"direction":"P","tstamp":{"$date":1289370600000},"value":0.040},
 ...]
```

Find and sort data with multiple qualifications

This query finds all data for the device with the ID of 4727354321046021 with a value greater than 10.0 and a direction of P. The query returns the tstamp and value fields, and sorts the results in descending order by the value field.

To query for specific dates when using the REST API, convert the dates to milliseconds since the epoch. For example:

- 2011-01-01 00:00:00 = 1293861600000
- 2011-01-02 00:00:00 = 1293948000000

MongoDB API

Run a **find** command on the ts_data_v table, with the $and boolean logical operator specified:

```
db.ts_data_v.find({"$and":[{"loc_esi_id":4727354321046021},
{"value":{"$gt":10.0}},{"direction":"P"}]},
{"tstamp":1,"value":1}).sort({"value":-1})

Results:
 {"tstamp":ISODate("2011-01-25T16:15:00Z"),"value":14.58}
 {"tstamp":ISODate("2011-01-26T00:45:00Z"),"value":12.948}
 {"tstamp":ISODate("2011-01-26T02:30:00Z"),"value":12.768}
 ...
```

REST API

Request:

Specify the GET method on the ts_data_v table, with the $and boolean logical operator specified:

```
GET /stores_demo/ts_data_v?query={"$and":[{"loc_esi_id":
4727354321046021},{"value":{"$gt":10.0}},{"direction":"P"}]}
&fields={"tstamp":1,"value":1}&sort={"value":-1}
```

Data: None.

Response:

The following response indicates that the operation was successful:

```
[{"tstamp":{"$date":1295972100000},"value":14.580},
 {"tstamp":{"$date":1296002700000},"value":12.948},
 {"tstamp":{"$date":1296009000000},"value":12.768},
 ...]
```

Find all data for a device in a specific date range

This query returns the data from midnight January 1, 2011 to January 2, 2011 for device ID 4727354321000111. The date that is queried is greater than 1293861600000 and less than 1293948000000. The query returns the tstamp and value fields.

MongoDB API

Run a **find** command on the ts_data_v table, with values specified for the $and boolean logical query operator:

```
db.ts_data_v.find({"$and":[{"loc_esi_id":"4727354321000111"},
{"tstamp":{"$gte":ISODate("2011-01-01 00:00:00")}},
{"tstamp":{"$lt":ISODate("2011-01-02 00:00:00")}}]},
{"tstamp":"1","value":"1"})

Results:
 {"tstamp":ISODate("2011-01-01T00:00:00Z"),"value":0.343 }
 {"tstamp":ISODate("2011-01-01T00:15:00Z"),"value":0.349 }
 {"tstamp":ISODate("2011-01-01T00:30:00Z"),"value":1.472 }
...]
```

REST API

Request:

Specify the GET method on the ts_data_v table in the stores_demo database, with values specified for the $and boolean logical query operator:

```
GET /stores_demo/ts_data_v?query={"$and":
[{"loc_esi_id":4727354321000111},{"tstamp":{"$gte":
{"$date":1293861600000}}},{"tstamp":{"$lt":
{"$date":1293948000000}}} ]}&fields={"tstamp":1,"value":1}
```

Data: None.

Response:

The following response indicates that the operation was successful:

```
[{"tstamp":{"$date":1293840000000},"value":0.343},
 {"tstamp":{"$date":1293840900000},"value":0.349},
 {"tstamp":{"$date":1293841800000},"value":1.472},
 ...]
```

Find the latest data point for a specific device

This query sets the sort parameter to order the tstamp field in descending order and sets the limit parameter to 1 to return only the latest value. The device ID is 4727354321000111 and the query returns the tstamp and value fields.

MongoDB API

Run a **find** command on the ts_data_v table, with sort and limit values specified:

```
db.ts_data_v.find({"loc_esi_id":"4727354321000111"},
{"tstamp":"1","value":"1"}).sort({"tstamp":-1}).limit(1)

Results:
 {"tstamp":ISODate("2011-02-08T05:45:00Z"),"value":1.412 }
```

REST API

Request:

Specify the GET method on the ts_data_v table, with sort and limit values specified in the query parameter:

```
GET /stores_demo/ts_data_v?query={"loc_esi_id":4727354321000111}
&fields={"tstamp":1,"value":1}&sort={"tstamp":-1}&limit=1
```

Data: None.

Response:

The following response indicates that the operation was successful:

```
[{"tstamp":{"$date":1297143900000},"value":1.412}]
```

Find the 100th data point for a specific device

This query sets the sort parameter to order the tstamp field in ascending order and sets the skip parameter to 100 to return the 100th value. The device ID is 4727354321000111 and the query returns the tstamp and value field.

MongoDB API

Run the **find** command on the ts_data_v table, with values specified for sort, limit and skip:

```
db.ts_data_v.find({"loc_esi_id":4727354321000111},
{"tstamp":1,"value":1}).sort({"tstamp":1}).limit(1).skip(100)

Results:
 {"tstamp":ISODate("2010-11-11T07:00:00Z"),"value":0.013}
```

REST API

Request:

Specify the GET method on the ts_data_v table, with values specified for sort, limit, and skip in the query parameter:

```
GET /stores_demo/ts_data_v?query={"loc_esi_id":4727354321000111}
&fields={"tstamp":1,"value":1}&sort={"tstamp":1}&limit=1&skip=100
```

Data: None.

Response:

The following response indicates that the operation was successful:

```
[{"tstamp":{"$date":1289458800000},"value":0.013}]
```

Chapter 7. Monitoring collections

You can use the IBM OpenAdmin Tool (OAT) for Informix to monitor collections in the Informix database.

Youc can view collections by using the IBM Informix JSON Plug-in for OpenAdmin Tool (OAT) or by using the IBM Informix Schema Manager Plug-in for OpenAdmin Tool (OAT).

See the OAT help for more information.

Related information:

cdr list trustedhost argument: List trusted hosts (SQL administration API)

Installing the OpenAdmin Tool for Informix with the Client SDK

cdr list shardCollection

onstat -g shard command: Print information about the shard cache

Chapter 8. Troubleshooting Informix JSON compatibility

Several troubleshooting techniques, tools, and resources are available for resolving problems that you encounter with Informix JSON compatibility.

Problem	Solution
How do I start the wire listener?	If the wire listener does not automatically start: 1. Verify that the user was created. For more information, see "Configuring the wire listener for the first time" on page 2-1. 2. Manually start the wire listener. For more information, see "Starting the wire listener" on page 2-33.
How can I debug wire listener problems?	From the wire listener command line, run the `-loglevel` *`level`* command, where *level* is the logging level. Log level options are: • error • warn • info • debug • trace For more information, see "Wire listener command line options" on page 2-32.
Where is the wire listener log file?	**UNIX:** The log file is in `$INFORMIXDIR/jsonListener.log`. **Windows:** The log file is named *`servername`*`_jsonListener.log` and is in your home directory. For example, `C:\Users\ifxjson\ol_informix1210_1_jsonListener.log`.
How can I view all of the current properties for the wire listener properties file?	From the wire listener command line, you can run the `-listProperties` command. This command prints all of the supported properties and their default values. For more information, see "The wire listener configuration file" on page 2-3.
How do I access the wire listener help?	You can view a list of available command line options by running the `-help` command.

Appendix. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix products

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 IBM Informix products. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Keyboard navigation

This product uses standard Microsoft Windows navigation keys.

Related accessibility information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software.

IBM and accessibility

For more information about the IBM commitment to accessibility, see the *IBM Accessibility Center* at http://www.ibm.com/able.

Dotted decimal syntax diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, that element is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 refers to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

***** Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be

repeated. For example, if you hear the line `5.1* data-area`, you know that you can include more than one data area or you can include none. If you hear the lines `3*, 3 HOST`, and `3 STATE`, you know that you can include `HOST`, `STATE`, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write `HOST STATE`, but you cannot write `HOST HOST`.
3. The `*` symbol is equivalent to a loop-back line in a railroad syntax diagram.

`+` Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line `6.1+ data-area`, you must include at least one data area. If you hear the lines `2+`, `2 HOST`, and `2 STATE`, you know that you must include `HOST`, `STATE`, or both. As for the * symbol, you can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the `*` symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. This material may 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 grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (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

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: 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 states 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 Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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 the names and addresses used by an actual business enterprise 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 follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

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 in 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.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks 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.

Adobe, the Adobe logo, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

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

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

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

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Printed in USA

SC27-5556-05