Informix Product Family
Informix
Version 12.10

IBM Informix JSON Compatibility Guide

SC27-5556-03

Informix Product Family
Informix
Version 12.10

IBM Informix JSON Compatibility Guide

SC27-5556-03

Note

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

This edition replaces SC27-5556-02.

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.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 `jsonListener.properties` file" on page 2-3

Table 2. 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 4-9 "Informix query operators" on page 4-18

Table 2. 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 4-14 "Update operators" on page 4-16 "Aggregation framework operators" on page 4-18 "Informix JSON commands" on page 4-9 "The `jsonListener.properties` 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 3. 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 2-20 "Running MongoDB operations on relational tables" on page 2-21

Table 3. 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 `jsonListener.properties` file" on page 2-3

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.

Syntax diagrams

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

Table 4. Syntax Diagram Components

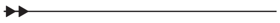
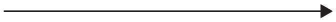
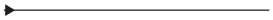
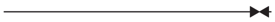
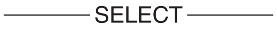
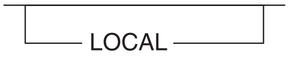
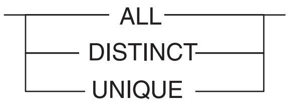
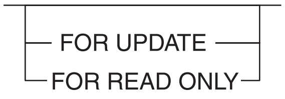
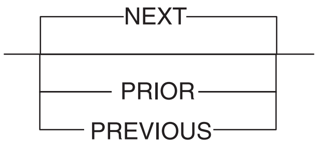
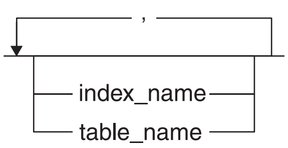
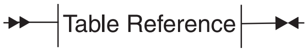
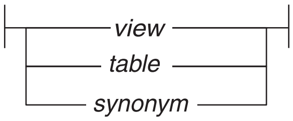
Component represented in PDF	Component represented in HTML	Meaning
	>>----------------------	Statement begins.
	----------------------->	Statement continues on next line.
	>-----------------------	Statement continues from previous line.
	-----------------------><	Statement ends.
	--------SELECT----------	Required item.
	--+-----------------+--- '------LOCAL------'	Optional item.
	---+-----ALL-------+--- +--DISTINCT-----+ '---UNIQUE------'	Required item with choice. Only one item must be present.
	---+------------------+--- +--FOR UPDATE-----+ '--FOR READ ONLY--'	Optional items with choice are shown below the main line, one of which you might specify.
	.---NEXT---------. ----+----------------+--- +---PRIOR--------+ '---PREVIOUS-----'	The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line is used by default.
	.-------,-----------. V | ---+-----------------+--- +---index_name---+ '---table_name---'	Optional items. Several items are allowed; a comma must precede each repetition.
	>>-| Table Reference |-><	Reference to a syntax segment.

Table 4. Syntax Diagram Components (continued)

Component represented in PDF	Component represented in HTML	Meaning

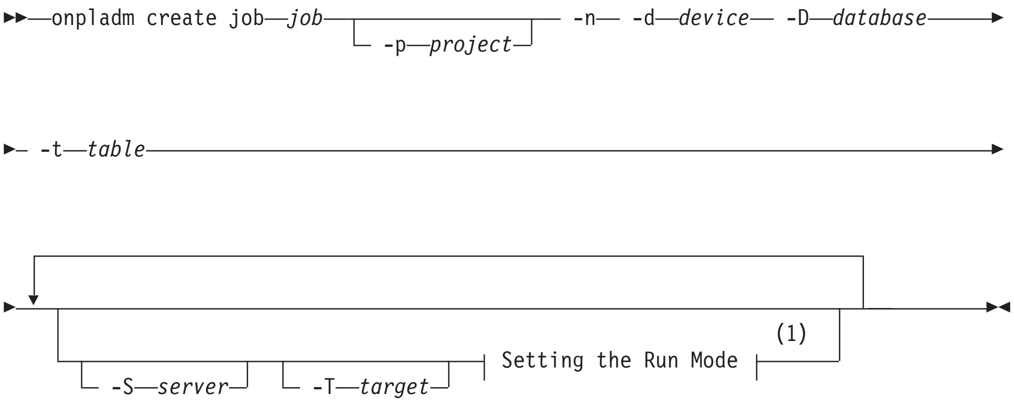
Table Reference

 	
	`Table Reference`
	`|--+-----view--------+--|`
	`   +------table------+`
	`   '----synonym------'`
		Syntax segment.

How to read a command-line syntax diagram

Command-line syntax diagrams use similar elements to those of other syntax diagrams.

Some of the elements are listed in the table in Syntax Diagrams.

Creating a no-conversion job

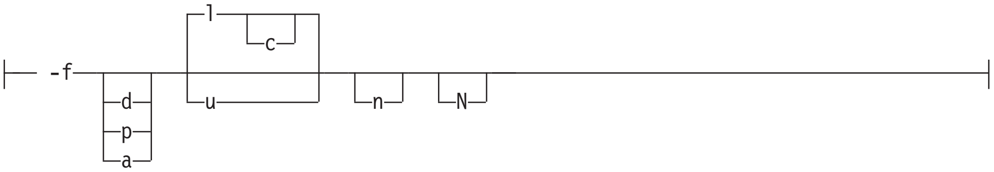


Notes:

1 See page Z-1

This diagram has a segment that is named “Setting the Run Mode,” which according to the diagram footnote is on page Z-1. If this was an actual cross-reference, you would find this segment on the first page of Appendix Z. Instead, this segment is shown in the following segment diagram. Notice that the diagram uses segment start and end components.

Setting the run mode:



To see how to construct a command correctly, start at the upper left of the main diagram. Follow the diagram to the right, including the elements that you want. The elements in this diagram are case-sensitive because they illustrate utility syntax. Other types of syntax, such as SQL, are not case-sensitive.

The Creating a No-Conversion Job diagram illustrates the following steps:

1. Include **onpladm create job** and then the name of the job.
2. Optionally, include **-p** and then the name of the project.
3. Include the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table
4. Optionally, you can include one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to include **-f**, optionally include **d**, **p**, or **a**, and then optionally include **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and punctuation

Keywords are words that are reserved for statements and all commands except system-level commands.

A keyword in a syntax diagram is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

You must also use any punctuation in your statements and commands exactly as shown in the syntax diagrams.

Identifiers and names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples.

You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in other syntax diagrams. A variable in a syntax diagram, an example, or text, is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.

```
►►──SELECT──column_name──FROM──table_name──────────────────────────────────►◄
```

When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

How to provide documentation feedback

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

Use one of the following methods:

- Send email to docinf@us.ibm.com.
- Add comments 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 from all methods is monitored by the team that maintains the user documentation. The feedback methods are reserved for reporting errors and omissions in the documentation. For immediate help with a technical problem, contact IBM Technical 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, created by MongoDB, 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 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

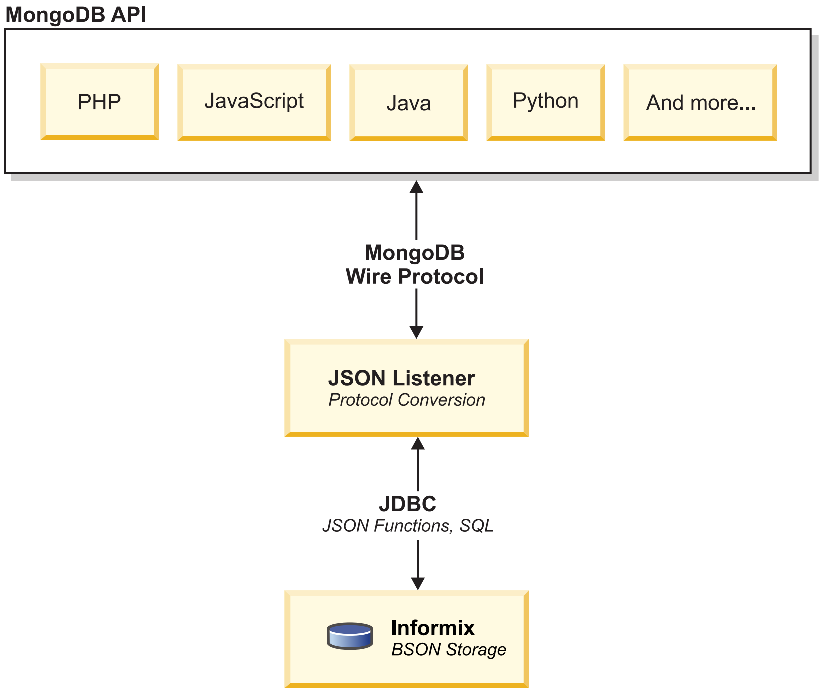
API type	Relational table access	JSON collection access
SQL API	Uses SQL language and standard ODBC, JDBC.NET, OData, and so on.	Uses direct SQL access, dynamic views, and row types.
MongoDB API	Uses MongoDB APIs for Java™, JavaScript, C++, C#, and so on.	Uses MongoDB APIs for Java, JavaScript, C++,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

Authentication of MongoDB clients occurs in the wire listener, not in the Informix server. Privileges are enforced by the wire listener. All communications that are sent to Informix originate from the user that is specified in the **url** parameter, regardless of which user was authenticated. User information and privileges are stored in the system_users collection in each database. MongoDB authentication is done on a per database level, whereas Informix authenticates to the instance.



Software dependencies

This topic describes the software requirements for Informix JSON compatibility.

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

The Informix JSON compatibility support is based on MongoDB version 2.4.

MongoDB API access
You must use IBM Java Runtime Environment (JRE) 1.6 or later versions. Version 1.6 is delivered with the Informix installation.

REST API access
You must use IBM Java Runtime Environment (JRE) 1.7. For more information and downloads, see http://www.ibm.com/developerworks/java/jdk/index.html.

You must use Tomcat version 8, which is included in the Informix installation as a part of `$INFORMIXDIR/bin/nosql_sdk.zip`. For the latest updates to Tomcat version 8, see http://tomcat.apache.org/download-80.cgi.

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, an Informix secondary server has 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.

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 provided as an executable JAR file that is named `$INFORMIXDIR/bin/jsonListener.jar`. The JAR file provides access to the MongoDB API and REST API.

MongoDB API access

You can connect to a JSON collection in 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.

For more information, see "Software dependencies" on page 1-2.

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.

For more information, see "Software dependencies" on page 1-2.

The wire listener configuration file, `jsonListener.properties`, defines every operational characteristic.

When you create a database or a table through the wire listener, automatic location and fragmentation is 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.

Related concepts:

➡ Managing automatic location and fragmentation (Administrator's Guide)

Related reference:

➡ SQL administration API portal: Arguments by privilege groups (Administrator's Reference)

Install the wire listener

You can install the wire listener by choosing the typical or custom installation options.

If you choose to create a server instance as a part of the installation process:

- The required wire listener configuration file `$INFORMIXDIR/etc/jsonListener.properties` is automatically created with default values established for each property, except the **`url`** parameter.
- The user **ifxjson**, which has REPLICATION privilege group access, is created and added to the `jsonListener.properties` file. This user ID is used by the wire listener to connect to Informix.

- The wire listener is automatically started and connected to the MongoDB API and the database server with the default operational instance. If you want to use the REST API, you must modify the **listener.type** parameter and restart the wire listener.

This option reduces the complexity of installation and maintenance of the Informix server and installs the required extensions for using BSON and JSON types.

If you do not create a server during installation:

- You must configure and start a server.
- You must configure the wire listener.

Related concepts:

Overview of database server configuration and administration (Administrator's Guide)

Create a configured server during installation (Installation Guide (UNIX))

Database server configuration after installation (Installation Guide (Windows))

Configuring the wire listener

You must configure the wire listener if you did not create a database server during installation or if you want to use the REST API.

Before you begin

"Install the wire listener" on page 2-1

Procedure

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

     ```
     CREATE USER userID WITH PASSWORD 'password' ACCOUNT unlock PROPERTIES
      USER daemon;
     ```
2. If you want to use the Informix sharding capability, you must grant the user REPLICATION privilege in the SQL Admin API. For example:

   ```
   EXECUTE FUNCTION task('grant admin','userID','replication');
   ```
3. Create a `jsonListener.properties` file in `$INFORMIXDIR/etc`. You can use the `$INFORMIXDIR/etc/jsonListener-example.properties` file as a template. To include parameters in the wire listener, you must uncomment the row and customize the parameter.
   1. Configure the **url** parameter for your environment and uncomment if necessary. You can specify the authorized user ID and password in the **url** parameter of the `jsonListener.properties` file. 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 the user that started the wire listener.
   2. Optional: If you are using the REST API, set the **listener.type** parameter to `listener.type=rest`.

c. Optional: Modify additional parameters for your environment as described in the `jsonListener.properties` file. For more information, see "The `jsonListener.properties` file."

4. 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 wire listener alias by using `127.0.0.1`. For example:

   ```
   lo_informix1210 onsoctcp 127.0.0.1 9090
   ```

   c. In the `jsonListener.properties` 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 2-20

Related reference:

CREATE DEFAULT USER statement (UNIX, Linux) (SQL Syntax)

grant admin argument: Grant privileges to run SQL administration API commands (Administrator's Reference)

The `jsonListener.properties` file

The properties that control the wire listener and the connection between the client and database server are set in the `%INFORMIXDIR%\etc\jsonListener.properties` file.

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

If your properties file is created during installation, or if you are using the `jsonListener-example.properties` template file, all of the property file parameters are commented out by default. To include a parameter in the wire listener, you must uncomment the row for the parameter and customize the settings.

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

Required parameter

`url`

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

The user and password that is specified in the **url** parameter are optional. These credentials are used to connect to the Informix database server for all operations that go through the wire listener. 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 that were specified in the listener start command.

You must specify the **sysmaster** database in the **url** parameter, which is used for administrative purposes by the wire listener. The **url** parameter has this format:

```
jdbc:informix-sqli://hostname/sysmaster:INFORMIXSERVER=server;USER=userid;
PASSWORD=password
```

Where:

hostname
: The host name of your computer. For example, `localhost:9088`.

server
: The name of the database server to connect to.

userid
: This optional attribute specifies the user ID that is used in connections to the Informix database server. If you plan to use the Informix sharding capability, you must specify the user with REPLICATION privilege group access to this parameter.

password
: This optional attribute specifies the password for the user ID.

Optional parameters

authentication.enable

This optional parameter indicates whether to enable user authentication.

Authentication of MongoDB clients occurs in the wire listener, not in the Informix server. Privileges are enforced by the wire listener. All communications that are sent to Informix originate from the user that is specified in the **url** parameter, regardless of which user was authenticated. User information and privileges are stored in the system_users collection in each database. MongoDB authentication is done on a per database level, whereas Informix authenticates to the instance.

false
: Do not allow user authentication. This is the default value.

true
: Allow user authentication. 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.listDatabases.sizeStrategy`
This optional parameter specifies a strategy for calculating the size of your database when the MongoDB listDatabases command is run.

Important: The MongoDB 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 command.listDatabases.sizeStrategy parameter.

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

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

`compute`
Compute the exact size of the database.

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

`estimate`
Estimate the size of the documents sampled. The default value is 1000 (or 0.1%) of the documents.

```
command.listDatabases.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:

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

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

`false`
Use a maximum document size of 256 MB. This is the default value.

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

`database.buffer.enable`

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

`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 preexisting databases.

database.dbspace

Prerequisite: `dbspace.strategy=fixed`
This optional parameter specifies the name of the Informix 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. 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.

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. 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 newly created databases, tables, and indexes.

autolocate
The Informix 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.

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 table. The _id field of the document is used as the input to the algorithm. The available algorithms are:

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).

This 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 investigate using 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.

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 number greater than 0, these fragments are created when the collection is created. The default value is 0.

include
This optional parameter specifies the location of a properties file to reference. The path can be absolute or relative. For more information, see “Running multiple wire listeners” on page 2-18.

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 into collections.

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

false
Do not batch document inserts 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 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 Informix before proceeding with the next operation.

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

true
Queue INSERT statements into larger batches.

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 Informix. The default value is 100.

index.cache.enable
This optional parameter indicates whether to enable index caching on collections.

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

false
Do not cache indexes on collections.

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 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.

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. For example, to allow the headers `foo` and `bar` in a request:

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

The default value is `listener.http.accessControlAllowHeaders=["accept","cursorId","content-type"]`.

`listener.http.accessControlAllowMethods`

This optional parameter, which is part of the response to a preflight request, specifies the 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 is `listener.http.accessControlAllowOrigin={"$regex":".*"}`, which means any origin is allowed to perform a CORS request.

Here are some usage examples:

- In this example, the localhost is granted access:

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

- In this example, access is granted 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:

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

- In this example, access is granted 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. For example, to expose the headers `foo` and `bar` to a client:

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

The default value is `listener.http.accessControlExposeHeaders=["cursorId"]`.

`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 0 seconds.

`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.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.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.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 above the core wire listener pool size to queue 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. The following are special values:

`0` Do not allocate a queue size for tasks. All new sessions are either immediately run on an available or new thread up to the maximum pool size, or are instantly 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.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.

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 64 bytes.

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 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 forward slash (/).

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.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.

`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.

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

`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.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.

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

`pool.semaphore.timeunit`

Prerequisite: `pool.semaphore.timeout=`*`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.timeunit

Prerequisite: `pool.service.interval=`*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.

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 with a thread-safe type map.

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 20.

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

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

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.

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.

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 performs the update to 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.

Tip: Choose an option that is based on type of update statements that you are running. 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. If your document updates are complicated or use document replacement, you can use **client** to process these updates by using the wire listener.

client
Use the wire listener to process updates. This is the default value.

mixed
Attempt to process updates on the database server first, then fallback to the wire listener.

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

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

With the `update.one.enable=false` setting, the MongoDB **db.collection.update** multi-parameter is ignored and all updates are treated as multiple JSON document updates. The performance of updates is improved with the `update.one.enable=false` setting.

true
Allow updates 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.

Related reference:

“Collection methods” on page 4-1

“REST API syntax” on page 5-1

Modifying the wire listener properties file

You can modify the wire listener properties in the `jsonListener.properties` file.

About this task

The `jsonListener.properties` file controls the wire listener and the connection between the client and database server.

Procedure

To modify the wire listener properties:

1. Stop the wire listener.
2. Update the `jsonListener.properties` file.
3. Start the wire listener.

Related concepts:

"Starting the wire listener"

Related tasks:

"Stopping the wire listener" on page 2-19

Related reference:

"The `jsonListener.properties` file" on page 2-3

Starting the wire listener

You can start the wire listener for the REST API and the MongoDB API.

Related reference:

➡ start json listener argument: Start the wire listener (Administrator's Reference)

Starting the MongoDB API wire listener

You can start the MongoDB API wire listener by using the **`start json listener`** SQL administration API **task()** or **admin()** function, or the command line argument.

Before you begin

- 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 create a server instance after the installation process, you must start the wire listener.
- The wire listener configuration file `jsonListener.properties` must exist. If a server instance is created as a part of the installation process, the `jsonListener.properties` is automatically created with default properties, otherwise you must manually create this file.
- If you use the SQL administration API **task()** or **admin()** function:
  - You must be logged in to the **sysadmin** database as user **informix** or another privileged user.
  - The `jsonListener.properties` file must be located in `$INFORMIXDIR/etc`.
- "Software dependencies" on page 1-2

Procedure

Use one of the following methods to start the MongoDB API wire listener for the current database server session:

- Run the SQL administration API **task()** or **admin()** function with the **start json listener** argument. For example:

  ```
  EXECUTE FUNCTION task("start json listener");
  ```

- From the command line, run the following command to start the MongoDB API wire listener:

  ```
  java -jar $INFORMIXDIR/bin/jsonListener.jar
  -config pathname/jsonListener.properties
  -logfile pathname/jsonListener.log -start
  ```

  Important: You must specify the **-config** argument in the **start** command.

Examples

Start the MongoDB API wire listener by using the command line

In this example, the MongoDB API wire listener is started from the command line:

```
java -jar $INFORMIXDIR/bin/jsonListener.jar -config
 $INFORMIXDIR/etc/jsonListener.properties
 -logfile $INFORMIXDIR/jsonListener.log -start
```

Output from starting the MongoDB API wire listener

In this example, output from starting the MongoDB API wire listener is shown:

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

Starting the REST API wire listener

You can start the REST API wire listener by using the command line argument.

Before you begin

"Software dependencies" on page 1-2

Procedure

1. Create a wire listener properties file for REST API that includes the parameter setting `listener.type="rest"`. You can use the `$INFORMIXDIR/etc/jsonListener-example.properties` file as a template.

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

2. From the command line, run the following command to start the REST API wire listener:

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

   Where *pathname* is the location where the `nosql_sdk.zip` file was extracted.

   Important: You must specify the **-config** argument in the **start** command.

Examples

Start the REST API wire listener by using the command line

In this example, the REST API wire listener is started from the command line:

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

Output from starting the REST API wire listener

In this example, output from starting the REST API wire listener is shown:

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

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 properties file to start the MongoDB API and a properties file to start the REST API.

Procedure

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

   Important: The **`url`** parameter must be specified, either in each individual properties file or in the file that is referenced by the **`include`** parameter.
3. Optional: Specify the **`include`** parameter to reference another properties file. The path can be relative or absolute. If you have multiple properties files, you can avoid duplicating parameter settings in the multiple properties files by specifying a subset of shared parameters in a single properties file, and the unique parameters in the individual properties files.
4. 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 separate wire listeners:

1. Create a properties 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 properties file for use with the MongoDB API that is named `mongo.properties`, with the parameter setting `include=shared.properties` included:

   ```
   include=shared.properties
   listener.type=mongo
   listener.port=27017
   ```

3. Create a properties file for use with the REST API that is named `rest.properties`, with the parameter setting `include=shared.properties` included:

   ```
   include=shared.properties
   listener.type=rest
   listener.port=8080
   ```

4. Start the wire listeners by using the command line:

   ```
   java -jar jsonListener.jar -start
   -config json.properties
   -config rest.properties
   ```

Related reference:

"REST API syntax" on page 5-1

Stopping the wire listener

You can stop the wire listener by using the **`stop json listener`** argument with the SQL administration API **task()** or **admin()** function, or the command line argument.

Before you begin

- If you use SQL administration API **task()** or **admin()** function, you must be connected to the **sysadmin** database as user **informix** or another authorized user.

Procedure

Use one of the following methods to stop the wire listener for the current database server session:

- Run the SQL administration API **task()** or **admin()** function with the **`stop json listener`** argument. For example:

  ```
  EXECUTE FUNCTION task("stop json listener");
  ```

- From the command line, run the following command to stop the wire listener. For example:

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

  Important: You must specify the configuration file in the **`stop`** argument.

Related reference:

stop json listener: Stop the wire listener (Administrator's Reference)

Wire listener command line options

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

Syntax

```
►►─java─-jar─jsonListener.jar─┬──────────────────────────────────────────┬──────►
                               └─-config─config_file_name─┬─-start─┬─┘
                                                          └─-stop──┘

►─┬─────────────────────────┬─┬─────────────────┬─┬───────────────────┬─────────►
  └─-logfile─log_file_name─┘ └─-loglevel─level─┘ └─-port─port_number─┘
```

```
►─┬──────────────────┬──┬──────────┬──┬───────────────────┬──────────────►◄
  └─-wait──wait_time─┘  └─-version─┘  └─-buildInformation─┘
```

Argument	Purpose
-config *config_file_name*	Specifies the deployment configuration file to run. This element 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.
-logfile *log_file_name*	Specifies the name of the log file used. If this option is not specified, the log messages are sent to `std.out`.
-loglevel *level*	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.
-port *port_number*	Specifies the port number. If a port is specified on the command line, it overrides the port properties set in the configuration file. The default port is 27017.
-wait *wait_time*	Specifies the amount of time, in seconds, to wait before the wire listener stops. The default is immediate shutdown.
-version	Prints the wire listener version.
-buildInformation	Prints the wire listener build information.

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 `jsonListener.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" on page 2-2

Related reference:

"The jsonListener.properties 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

Chapter 3. JSON data sharding

IBM Informix can horizontally partition data across multiple database servers. Documents from a collection or rows from a table can be distributed 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 distribute data across database servers, you also distribute performance across hardware. As your database grows in size, you can scale up by adding more database servers.

Horizontal partitioning is also known as *sharding*. The database servers that receive sharded data are *shard servers*, and a cluster of shard servers is a *shard cluster*.

Documents or rows that are inserted on a shard server can be copied to other shard servers in a shard cluster. Queries that are performed on a shard server can select data from other 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. This query optimization is another benefit that comes from data sharding.

To use Informix sharding capability, you must complete the following steps:

1. Add existing database servers to a shard cluster. You can create a cluster of shard servers by using MongoDB commands or IBM Informix commands.
2. Define a schema for sharding data. You can create a definition by using MongoDB commands or IBM Informix commands.

Related concepts:

Shard cluster setup (Enterprise Replication Guide)

Related tasks:

"Configuring the wire listener" on page 2-2

Enabling sharding for JSON or relational data

You must enable sharding support before you can shard data.

Before you begin

Verify that the user of the wire listener has REPLICATION privilege group access in the SQL Admin API. If a database server instance is created as a part of the installation process, the user **ifxjson** is created and added to the `$INFORMIXDIR/etc/jsonListener.properties` file, with REPLICATION privilege group access. If a database server instance is created after the installation process, you must add the user with REPLICATION privilege group access to the **`url`** parameter in the `$INFORMIXDIR/etc/jsonListener.properties` file.

Procedure

To enable sharding for JSON or relational data:

1. Specify trusted hosts for each database server that is added to the shard cluster. Use one of the following methods to set each database server's REMOTE_SERVER_CFG configuration parameter, and add trusted-host names as values to each database server's trusted-host file:

   - 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. You must be a Database Server Administrator (DBSA) to run these functions.
2. Set the **`sharding.enable`** parameter to `true` in each database server's `$INFORMIXDIR/etc/jsonListener.properties` file.
3. Set the **`update.client.strategy`** parameter to `deleteInsert` in each database server's `$INFORMIXDIR/etc/jsonListener.properties` file.
4. Restart the wire listener.

Related reference:

cdr add trustedhost argument: Add trusted hosts (SQL administration API) (Administrator's Reference)

cdr remove trustedhost argument: Remove trusted hosts (SQL administration API) (Administrator's Reference)

cdr list trustedhost argument: List trusted hosts (SQL administration API) (Administrator's Reference)

Creating a shard cluster by running the addShard command in the MongoDB shell

The **sh.addShard** MongoDB shell command adds a database server to a shard cluster.

Before you begin

You must verify the following details:

- All database servers that are participating in, or being added to, a shard cluster are listed in each database server's trusted-host file.
- The **`sharding.enable`** parameter is set to `true` in each database server's `$INFORMIXDIR/etc/jsonListener.properties` file.
- The user of the wire listener has REPLICATION privilege group access in the SQL Admin API.

Procedure

To create a shard cluster by running the **addShard** command in the MongoDB shell:

1. Run the **mongo** command to start the MongoDB shell.
2. Run the **sh.addShard** command with the host and port specified for the Informix server that you want to add. The specified port must be an Informix port that runs a SQLI, network-based listener, for example the **onsoctcp** protocol. For example:

   ```
   > sh.addShard("myhost1.ibm.com:9201")
   ```

Example: Adding a single database server to a shard cluster

The following command adds the database server that is at port **9202** of **myhost2.ibm.com** to a shard cluster. The shard-cluster definition changes to include the new server.

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

Related reference:

cdr define shardCollection (Enterprise Replication Guide)

cdr add trustedhost argument: Add trusted hosts (SQL administration API) (Administrator's Reference)

cdr remove trustedhost argument: Remove trusted hosts (SQL administration API) (Administrator's Reference)

cdr list trustedhost argument: List trusted hosts (SQL administration API) (Administrator's Reference)

"Database commands" on page 4-4

Creating a shard cluster by running the addShard command through db.runCommand in the MongoDB shell

The **db.runCommand** command with **addShard** command syntax adds database servers to a shard cluster.

Before you begin

You must verify the following details:

- All database servers that are participating in, or being added to, a shard cluster are listed in each database server's trusted-host file.
- The **`sharding.enable`** parameter is set to `true` in each database server's `$INFORMIXDIR/etc/jsonListener.properties` file.
- The user of the wire listener has REPLICATION privilege group access in the SQL Admin API.

Procedure

To create a shard cluster by running the **addShard** command through **db.runCommand** in the MongoDB shell:

1. Run the **mongo** command. The command starts the MongoDB shell.
2. Run the **db.runCommand** from the MongoDB shell, with **addShard** command syntax.
   - Add a single database server by using the following syntax:

```
►►──db.runCommand({"addShard":─"──┬─local_hostname──────────────┬─:port_number"─})──────────►◄
                                  └─fully_qualified_domain_name─┘
```

Element	Description	Restrictions
local_hostname	The localhost name for a server.	None.
fully_qualified_domain_name	The full domain name for a server.	None.
port_number	The port that is used for communication with the server.	None.

For example:

```
> db.runCommand({"addShard":"myhost1.ibm.com:9201"})
```

   - Add multiple database servers by using the following syntax:

```
                                  ┌─,──────────────────────────────────────────────┐
                                  ▼                                                │
►►─db.runCommand({"addShard":[────"─┬─local_hostname──────────────┬─:port"─┴─]})──────►◄
                                    └─fully_qualified_domain_name─┘
```

Element	Description	Restrictions
local_hostname	The localhost name for a server.	None.
fully_qualified_domain_name	The full domain name for a server.	None.
port	The port that is used for communication with the server.	None.

For example:

```
> db.runCommand({"addShard":["myhost2.ibm.com:9202",
   "myhost3.ibm.com:9203"]})
```

Examples

Add a database server to a shard cluster

This example adds the database server that is at port **9204** of **myhost4.ibm.com** to a shard cluster. The shard-cluster definition changes to include the new server.

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

Add multiple database 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. The shard-cluster definition changes to include the new servers.

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

Related reference:

cdr define shardCollection (Enterprise Replication Guide)

cdr add trustedhost argument: Add trusted hosts (SQL administration API) (Administrator's Reference)

cdr remove trustedhost argument: Remove trusted hosts (SQL administration API) (Administrator's Reference)

cdr list trustedhost argument: List trusted hosts (SQL administration API) (Administrator's Reference)

“Database commands” on page 4-4

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 database servers that are participating in a shard cluster.

Procedure

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

   Syntax:

```
►►─db.runCommand({listShards:1})──────────────────────────────►◄
```

For example, run the following command:

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

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
}
```

Element	Description	Restrictions
ER_group_name	The Enterprise Replication group name of a database server that is a shard-cluster participant. The Enterprise Replication group name for a database server can be found in the database server's `sqlhosts` file. The default location for the `sqlhosts` file is: • UNIX: `$INFORMIXDIR/etc/sqlhosts` • Windows: `%INFORMIXDIR%\etc\sqlhosts.%INFORMIXSERVER%` 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 named **myserver** is **g_myserver**.	None.
host	The host for a shard-cluster participant. The host can be a localhost name or a full domain name.	None.
IP_address	The IP address of the database server that the listener is connected to.	None.
port	The port number that a shard-cluster participant uses to communicate with other shard-cluster participants.	None.
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.	None.

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 concepts:

Installing the OpenAdmin Tool for Informix with the Client SDK (Client Products Installation Guide)

Related reference:

cdr list trustedhost argument: List trusted hosts (SQL administration API) (Administrator's Reference)

cdr list shardCollection (Enterprise Replication Guide)

onstat -g shard command: Print information about the shard cache (Administrator's Reference)

“Database commands” on page 4-4

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.

A definition that uses a hash algorithm to shard data is modified when MongoDB commands are used to add a server to the shard cluster.

A definition that uses an expression to shard data can be modified by running the **changeShardCollection** command. If you add a shard server to a definition, you must first add the server to the shard cluster by running the **db.runCommand** command with **addShard** command syntax.

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

Related reference:

cdr change shardCollection (Enterprise Replication Guide)

cdr delete shardCollection (Enterprise Replication Guide)

Creating a shard-cluster definition that uses a hash algorithm for distributing data across database servers

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:
     ```
     ►►──db.runCommand──({"shardCollection":"database.─┬─collection─┬─",──────────►
                                                       └─table──────┘

     ►─key:{─┬─field──┬─:"hashed"}})──────────────────────────────────────────►◄
             └─column─┘
     ```

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.

Element	Description	Restrictions
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:

```
►►──sh_database_─┬─collection─┬─────────────────────────────────────────────►◄
                 └─table──────┘
```

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:

cdr change shardCollection (Enterprise Replication Guide)

cdr delete shardCollection (Enterprise Replication Guide)

"Database commands" on page 4-4

Creating a shard-cluster definition that uses an expression for distributing data across database servers

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:

```
►►──db.runCommand──({"shardCollection":"database.─┬─collection─┬─",──────────►
                                                  └─table──────┘

►─key:{─┬─column─┬─:1},expressions:{─────────────────────────────────────►
        └─field──┘

   ┌─,──────────────────────────────────┐
   ▼                                    │
►───"──ER_group_name──":──expression──"─┴──────────────────────────────►

►─"──ER_group_name──":"──remainder──"──})─────────────────────────────►◄
```

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:

```
►►──sh_database_─┬─collection─┬──────────────────────────────────────────►◄
                 └─table──────┘
```

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:

cdr add trustedhost argument: Add trusted hosts (SQL administration API) (Administrator's Reference)

cdr remove trustedhost argument: Remove trusted hosts (SQL administration API) (Administrator's Reference)

cdr list trustedhost argument: List trusted hosts (SQL administration API) (Administrator's Reference)

cdr define shardCollection (Enterprise Replication Guide)

cdr change shardCollection (Enterprise Replication Guide)

cdr delete shardCollection (Enterprise Replication Guide)

cdr list shardCollection (Enterprise Replication Guide)

“Database commands” on page 4-4

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.

```
►►──db.runCommand──({"changeShardCollection":"database.─┬─collection─┬─",──────►
                                                         └─table──────┘


                   ┌─,─────────────────────────────────────┐
                   ▼                                       │
►─expressions:{──"──ER_group_name──":"──expression──"─┴──────────────────────────►

►─,"ER_group_name":"remainder"──})──────────────────────────────────────────────►◄
```

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.

Related reference:

cdr define shardCollection (Enterprise Replication Guide)

cdr change shardCollection (Enterprise Replication Guide)

cdr delete shardCollection (Enterprise Replication Guide)

cdr list shardCollection (Enterprise Replication Guide)

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 2.4.3.

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

The collection methods for the mongo shell that are supported by Informix are shown.

The MongoDB collection methods are operations that are run on a JSON collection or a relational table.

Table 4-1. Supported collection methods

Name	JSON collections	Relational tables	Details
aggregate	No	No	
count	Yes	Yes	
createIndex	Yes	Yes	For more information, see “Index creation” on page 4-2.
dataSize	Yes	No	
distinct	Yes	Yes	
drop	Yes	Yes	
dropIndex	Yes	Yes	
dropIndexes	Yes	No	
ensureIndex	Yes	Yes	For more information, see “Index creation” on page 4-2.

Table 4-1. Supported collection methods (continued)

Name	JSON collections	Relational tables	Details
find	Yes	Yes	
findAndModify	Yes	Yes	For relational tables, findAndModify is only supported for tables that have a primary key, a serial column, or a rowid. This command does 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 only if `update.one.enable=true` in the `jsonListener.properties` file. 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/v2.4/.

Related tasks:

"Running MongoDB operations on relational tables" on page 2-21

Related reference:

"The `jsonListener.properties` 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" on page 4-3
- "Index creation for a specific data type by using the Informix extended syntax" on page 4-3
- "Index creation for text, geospatial, and hashed" on page 4-4

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 4-9.

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/v2.4/.

Database commands

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

The MongoDB database commands are run on a database.

User commands

Aggregation commands

Table 4-2. Aggregation commands

Name	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 4-18.
count	Yes	Yes	
distinct	Yes	Yes	
group	No	No	
mapReduce	No	No	

Geospatial commands

Table 4-3. Geospatial commands

Name	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
eval	No	No	
findAndModify	Yes	Yes	For relational tables, the findAndModify command is only supported for tables that have a primary key, a serial column, or a rowid. This command does not support sharded data.
getLastError	Yes	Yes	
getPrevError	No	No	
resetError	No	No	
text	No	No	Text queries are supported by using the $text or $ifxtext query operators, not through the text command.

Database operations

Authentication commands

Table 4-5. Authentication commands

Name	Supported
authenticate	Yes
logout	Yes
getnonce	Yes

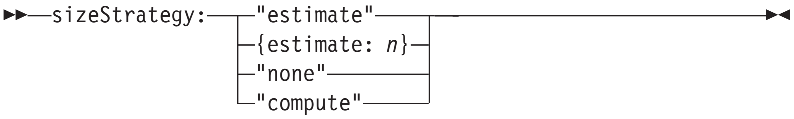
Diagnostic commands

Table 4-6. 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	
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	

Table 4-6. Diagnostic commands (continued)

Name	Supported	Details
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	No	

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.
		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.
		 sizeStrategy:
		"estimate"
		{estimate: *n*}
		"none"
		"compute"
		`estimate`
		Estimate the size of the documents in the collection by using 1000 (or 0.1%) of the documents.
		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 } })`
		`none`
		List the databases but do not compute the size. The database size is listed as 0.
		`db.runCommand({listDatabases:1 ,`
		` sizeStrategy: "none" })`
		`compute`
		Compute the exact size of each database.
		`db.runCommand({listDatabases:1 ,`
		` sizeStrategy: "compute" })`

ping	Yes	
serverStatus	Yes	

Table 4-6. Diagnostic commands (continued)

Name	Supported	Details
top	No	
whatsmyuri	Yes	

Instance administration commands

Table 4-7. 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
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	
logRotate	No	No	
reIndex	No	No	
renameCollection	No	No	
repairDatabase	No	No	
setParameter	No	No	
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 4-8. Replication commands

Name	Supported
isMaster	Yes

Table 4-8. Replication commands (continued)

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

Sharding commands

Table 4-9. 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 by running the addShard command through db.runCommand in the MongoDB shell" on page 3-3.
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	
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/v2.4/.

Related tasks:

"Creating a shard-cluster definition that uses an expression for distributing data across database servers" on page 3-8

"Viewing shard-cluster participants" on page 3-4

"Creating a shard cluster by running the addShard command in the MongoDB shell" on page 3-2

"Creating a shard cluster by running the addShard command through db.runCommand in the MongoDB shell" on page 3-3

"Creating a shard-cluster definition that uses a hash algorithm for distributing data across database servers" on page 3-7

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 4-10
- "importCollection" on page 4-12
- "transaction" on page 4-12

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.

```
►►──createTextIndex:──"──collection_name──"──,──name:──"──indexName──"─────────────►

►─options:──{──────────────────────────────────┬─}──────────────────────────────────►
                │                          (1) │
                └─┤ btx index parameters ├─────┘

►─┬──────────────────────────────┬──────────────────────────────────────────────────►◄
  │           ┌─,──────────┐      │
  │           ▼            │      │
  └─key:──{────"──column──"──┴──}─┘
```

Notes:

1 See bts access method syntax (Database Extensions Guide).

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:─"─collection_name─"─,─file:─"─filepath─"─,──────────►

                  ┌─json─┐
►─format:─┬─┬─"─┴──────┴─"─┬─┬──────────────────────────────────┬─┬──────►
          │ └─"jsonArray"──┘ │                 ┌─,──────┐       │ │
          │                  └─,─fields:─{─"─▼─filter─┴─"─}──┘ │
          │                         ┌─,──────┐                    │
          └─"─csv─"─,─fields:─{─"─▼─filter─┴─"─}──────────────────┘

►─┬─────────────────────────────────┬──────────────────────────────────►◄
  └─,─query:─{─"─query_document─"─}─┘
```

exportCollection

This required parameter specifies the collection name to export.

file

This required parameter specifies the output file path. For example, `file: "/tmp/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:──"──collection_name──"──,──file:──"──filepath──"──,──────────►

                 ┌─json──────┐
►─format:──"─────┼─jsonArray─┼───"───────────────────────────────────────────────►◄
                 └─csv───────┘
```

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"`.

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.

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.

```
►►──transaction:──"──┬──────────────────────┬──"─────────────────────────────►◄
                     ├─enable──┬─────────┬──┤
                     │         ├─commit──┤  │
                     │         └─rollback┘  │
                     ├─disable──────────────┤
                     └─status───────────────┘
```

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 }
```

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 }
```

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 }
```

Configuring authentication

You can configure Informix to use MongoDB authentication.

About this task

The authentication support for Informix JSON is based on MongoDB version 2.4.

Procedure

1. Start the MongoDB wire listener with authentication turned off.

2. For each database, add the users that you want grant access. For example, to grant user bob readWrite access:

   ```
   db.addUser({user:"bob", pwd: "myPass1", roles:["readWrite","sql"]})
   ```

3. Stop the wire listener.
4. Set `authentication.enable=true` in the properties file.
5. Restart the wire listener.

What to do next

After authentication is turned on, each client must authenticate.

Related concepts:

"Starting the wire listener" on page 2-16

Related tasks:

"Stopping the wire listener" on page 2-19

Related reference:

"The `jsonListener.properties` file" on page 2-3

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

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

Query selectors

Array query operators

Table 4-10. Array query operators

MongoDB command	JSON collections	Relational tables	Details
$elemMatch	No	No	
$size	Yes	No	Supported for simple queries only. The operator is only supported when it is the only condition in the query document.

Comparison query operators

Table 4-11. 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.
$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 4-12. Element query operators

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

Evaluation

Table 4-13. Evaluation query operators

MongoDB command	JSON collections	Relational tables	Details
$mod	Yes	Yes	
$regex	Yes	No	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 4-9.
$where	No	No	

Geospatial query operators

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

Table 4-14. Geospatial query operators

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

Table 4-14. Geospatial query operators (continued)

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

JavaScript query operators

The JavaScript query operators are not supported.

Logical query operators

Table 4-15. Logical query operators

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

Projection operators

Comparison query operators

Table 4-16. Comparison query operators

MongoDB command	JSON collections	Relational tables	Details
$	No	No	
$elemMatch	No	No	
$slice	No	No	
$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	
$snapshot	No	No	

For more information about the MongoDB features, see http://docs.mongodb.org/v2.4/.

Update operators

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

Array update operators

Table 4-17. 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 4-18. 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 4-19. Bitwise update operators

MongoDB command	JSON collections	Relational tables	Details
$bit	Yes	No	

Field update operators

Table 4-20. Field update operators

MongoDB command	JSON collections	Relational tables	Details
$inc	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/v2.4/.

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 (SQL Syntax).

In the following example, a wildcard search is run for strings that contain `Informix`:

```
db.collection.find( { "$like" : "%Informix%" )
```

Related reference:

Basic Text Search query syntax (Database Extensions Guide)

Aggregation framework operators

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

Pipeline operators

Table 4-21. 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	

Table 4-21. Pipeline operators (continued)

MongoDB command	JSON collections	Relational tables	Details
$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	No	No	

For more information about the MongoDB features, see http://docs.mongodb.org/v2.4/.

Expression operators

$group operators

Table 4-22. $group operators

MongoDB command	JSON collections	Relational tables
$addToSet	Yes	No
$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/v2.4/.

Example: Accessing MongoDB collections by using SQL

As an alternative to using the MongoDB API, you can use Informix SQL to access BSON data.

Important: This method of accessing MongoDB collections is only available in a projection list of the main query.

In this example, a table named people is created with names and ages inserted by using the mongo 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});
```

In this example, the name and age fields in each of the BSON documents are accessed by using the following query. This query returns a BSON document for the name and age of each person in the table.

```
SELECT data.name, data.age FROM people;
```

In this example, casts to JSON are added to the name and age query, which returns documents in human readable form.

```
> 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.
```

In this example, casts are used to convert the values to their underlying data type. This query retrieves the name as a varchar and the age as an integer for all people younger than 35.

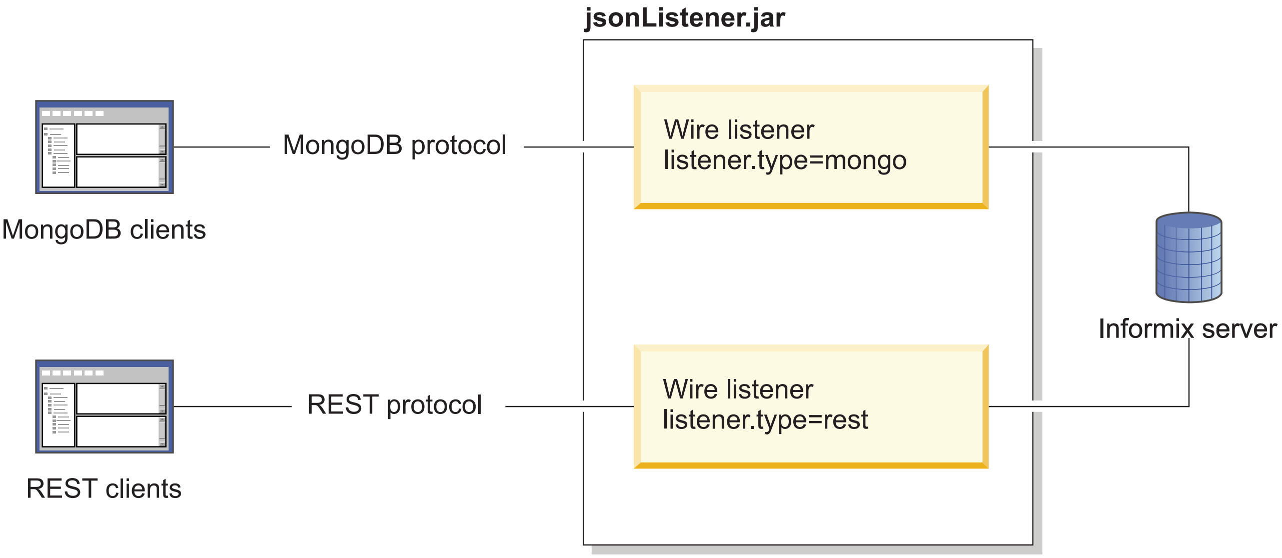
```
> SELECT data.name::varchar as name, data.age::int as age FROM people
 WHERE data.age::int < 35;

name  Anne
age   31

name  Charlie
age   29

2 row(s) retrieved.
```

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, `jsonListener.properties`, which defines the operational characteristics for theMongoDB API and REST API.



REST API syntax

A subset of the HTTP methods are supported by the REST API. These methods are DELETE, GET, POST, and PUT.

- "POST"
- "PUT" on page 5-3
- "GET" on page 5-4
- "DELETE" on page 5-5

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.

Table 5-1. Supported POST method syntax (continued)

Method	Path	Description
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:

```
Response does not contain any data.
```

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", 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:
Because the _id field was not included in the document, the automatically generated _id is included in the response. Here is a successful response:

```
{"id":{"$oid":"537cf433559aeb93c9ab66cd"},"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:

```
{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 /?query={name:"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.

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"]
```

Sort 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}]
```

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}
```

Related concepts:

Chapter 6, "Create time series through the wire listener," on page 6-1

Related tasks:

"Running multiple wire listeners" on page 2-18

Related reference:

"The `jsonListener.properties` 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”
- “system.timeseries.rowType collection” on page 6-3
- “system.timeseries.container collection” on page 6-3
- “Time series table format” on page 6-4
- “Virtual table format” on page 6-5

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:─"calendar_name"─,─calendarStart:─"start_date"─,──────────────────►

►─patternStart:─"pattern_date"─,─pattern:─{─type:─"interval"────────────────►

►─,─intervals:──────────────────────────────────────────────────────────────►

     ┌─,─────────────────────────────────────────────┐
     ▼                                               │
►─[────{─duration:─"num_intervals"─,─on:──┬─true──┬─}─┴──]─}─}────────────►◄
                                          └─false─┘
```

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:—"rowtype_name"—,—fields:—[——————————————————————————►


   ┌,———————————————————————————————————————————┐
   ▼                                            │
►——{—name:—"field_name"—,—type:—"data_type"—}———┴—]—}———————————►◄
```

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:─"container_name"─,─dbspaceName:─"dbspace_name"─,──────────────►

►─rowTypeName:─"rowtype_name"─,─firstExtent:─extent_size─,───────────────►

►─nextExtent:─next_extent_size─}────────────────────────────────────────►◄
```

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:─"table_name"─,─options:─{─columns:──────────────────────►

►─[─{─name:─"col_name"─,─type:─"data_type"─,─primaryKey:true─,───────────►

►─notNull:true─}─,─{─name:─"col_name"─,──────────────────────────────────►

►─type:─"timeseries(rowtype_name)"─}─]─}─}──────────────────────────────►◄
```

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:─"virtualtable_name"─,────────────────────────────────►

►─options:─{─timeseriesVirtualTable:─{─baseTableName:─"table_name"─,──────►

►─newTimeSeries:─"─calendar─(─calendar_name─)─,─origin─(─origin─)─,────►

►─container─(─container_name─)─┬─────────────┬─────────────────────────►
                               ├─,─irregular─┤
                               │   ┌regular┐ │
                               └─,─┴───────┴─┘

►─,─virtualTableMode:mode─,─timeseriesColumnName:─"col_name"─}─}─}──────►◄
```

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 REST API or the MongoDB API through the wire listener.

Before you begin

Before you start this tutorial, complete the following prerequisite tasks:

- 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 REST API or the MongoDB API. See "Configuring the wire listener" on page 2-2.

About this task

In this example, you create a time series that contains sensor readings about the temperature and humidity in your 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

Attention: The example code is formatted with line breaks for usability. Do not include line breaks in the data portion of REST API methods.

Procedure

To create a time series with the REST API or the MongoDB API:

1. Create a time series calendar that is named **ts_10min** by adding the following document to the **system.timeseries.calendar** collection with the REST API POST method or the MongoDB API insert method:

```
{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}]}}
```

2. Create a **TimeSeries** row type that is named **reading** by adding the following document to the **system.timeseries.rowType** collection with the REST API POST method or the MongoDB API insert method:

```
{name:"reading",
fields:[{name:"tstamp", type:"datetime year to fraction(5)"},
        {name:"temp", type:"float"},
        {name:"hum", type:"float"}]}
```

3. Create a container that is named **c_0** in the **dbspace1** dbspace by adding the following document to the **system.timeseries.container** collection with the REST API POST method or the MongoDB API insert method:

```
{name:"c_0",
 dbspaceName:"dbspace1",
 rowTypeName:"reading",
 firstExtent:1000,
 nextExtent:500}
```

4. Create the time series table that is named **ts_data1** by running the REST API POST method or the MongoDB API create method with the following table format:

```
{name:"ts_data1",
 columns:[{name:"id", type:"int", primaryKey:true, notNull:true},
          {name:"ts", type:"timeseries(reading)"}]}
```

5. Create the virtual table that is named **ts_data1_v** by running the REST API POST method or the MongoDB API create method with the following table format:

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

6. Load records into the time series by inserting the following documents into the **ts_data1_v** virtual table with the REST API POST method or the MongoDB API insert method:

```
{ id: 1, temp: 15.0, hum: 20.0}
```

```
{ id: 1, temp: 16.2, hum: 19.0}
```

```
{ id: 1, temp: 16.5, hum: 22.0}
```

   Because this time series is regular, you do not need 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.

Example

The following examples show queries on time series data with REST API methods. You run the following examples against the **stores_demo** database. Run the DB-Access command **dbaccessdemo** to create the **stores_demo** database. For instructions, see dbaccessdemo command: Create demonstration databases.

List all device IDs

The following query returns all device IDs:

```
GET /stores_demo/$cmd?query={distinct:"ts_data_v",key:"loc_esi_id"}
```

List device IDs that are greater than 10

The following query returns the device IDs that are greater than 10:

```
GET /stores_demo/$cmd?
query={distinct:"ts_data_v",key:"loc_esi_id",que
ry:{value:{"$gt":10}}}
```

Find the data for a specific device ID

The following query returns the data for the device with the ID of 4727354321046021:

```
GET /stores_demo/ts_data_v?
query={loc_esi_id:"4727354321046021"}
```

Find and sort data with multiple qualifications

The following query finds all data for a specific device with a value greater than 100.0 and a direction of P, returns the **tstamp** and **value** fields, and sorts the results in descending order by the **value** field:

```
GET /stores_demo/ts_data_v?query={"$and":
[{loc_esi_id:"4727354321046021"},{value:
{"$gt":100.0}},
{direction:"P"}]}&fields={tstamp:1,value:1}&sort= {value:-1}
```

Find all data for a device in a specific date range

To query for specific dates, convert the dates to milliseconds since the epoch. For example:

- 2011-01-01 00:00:00 = 1293861600000
- 2011-01-02 00:00:00 = 1293948000000

The following query returns the data from midnight January 1, 2011 to January 2, 2011 for device ID 4727354321000111:

```
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}
```

Find the latest data point for a specific device

The following 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:

```
GET /stores_demo/ts_data_v?
query={loc_esi_id:"4727354321000111"}&fields ={tstamp:1,value:1}
&sort={tstamp:-1}&limit=1
```

Find the 100th data point for a specific device

The following 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:

```
GET /stores_demo/ts_data_v?
query={loc_esi_id:"4727354321000111"}&fields ={tstamp:1,value:1}
&sort={tstamp:1}&limit=1& skip=100
```

Chapter 7. Monitoring collections

You can use the IBM OpenAdmin Tool (OAT) for Informix to monitor collections in an 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 concepts:

Installing the OpenAdmin Tool for Informix with the Client SDK (Client Products Installation Guide)

Related reference:

cdr list trustedhost argument: List trusted hosts (SQL administration API) (Administrator's Reference)

cdr list shardCollection (Enterprise Replication Guide)

onstat -g shard command: Print information about the shard cache (Administrator's Reference)

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” on page 2-2. 2. Manually start the wire listener. For more information, see “Starting the wire listener” on page 2-16.
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 2-19.
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 `jsonListener.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 `jsonListener.properties` 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 (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

Printed in USA

SC27-5556-03