Progress® DataDirect® for JDBC™ for MongoDB™
User's Guide

Release 6.1.0

Copyright

Visit the following page online to see Progress Software Corporation's current Product Documentation Copyright Notice/Trademark Legend: https://www.progress.com/legal/documentation-copyright.

Updated: 2026/02/27

Table of Contents

Welcome to the Progress DataDirect for JDBC for MongoDB Driver.......9

What's new in this release?....10
Requirements....14
Migrating schema maps and native files to 6.1....14
Installing and setting up the driver....14
Driver and DataSource classes....16
Connection URL examples....16
- No authentication....17
- User ID and password....17
- Kerberos authentication....19
- LDAP authentication....20
- TLS/SSL encryption....21
- Replica set failover....23
- MongoDB Atlas....23
- Microsoft Azure Cosmos DB for MongoDB....25
- Proxy server....26

Data types....27
- Default mapping of columns with inconsistent native data types....28
- getTypeInfo()....29

Mapping objects to tables....34
- Normalized view....35
- Flattened view....44
- Mixed view....44

SQL escape sequences....52
- Supported scalar functions....52
- CAST_TO_NATIVE function escape....54

DataDirect tools....54
Troubleshooting....55
Additional information....55
Contacting Technical Support....55

Tutorials....57

Interactive SQL....57
Tableau....58
DbVisualizer....59
- Adding a driver....59
- Connecting and executing SQL statements....60

Configuring and connecting 63

Setting the classpath 64
Connecting using the JDBC Driver Manager......64
Passing the connection URL......64
Generating connection URLs with the Configuration Manager......66
Testing connections and queries 67
Connecting using data sources......67
How data sources are implemented......68
Creating data sources......68
Calling a data source in an application......69
Testing a data source connection......69
Refreshing the relational map......72
Authentication......73
User ID and password authentication......74
LDAP authentication......75
Kerberos authentication......75
Data Encryption......80
Configuring TLS/SSL Encryption......80
Configuring TLS/SSL Server Authentication......81
Configuring TLS/SSL Client Authentication......82
FIPS (Federal Information Processing Standard)......83
Proxy server......83
MongoDB Atlas clusters......84
Replica set failover for write operations......85
Performance considerations......86

Additional features and functionality 87

MongoDB sharding......87
Identifiers......88
MongoDB views 90
Local views 90
MinKey and MaxKey values......91

Connection property descriptions......93

ArrayNormalizationThreshold......102
AuthenticationDatabase......103
AuthenticationMethod......104
ColumnDiscoverySampleSize......105
CreateMap......105
CryptoProtocolVersion......106
DatabaseName......107
EnableDNSLookup......108

EncryptionMethod....109
FetchSize....110
FlattenArrayBase....111
HostNameInCertificate....112
ImportStatementPool....113
InitializationString....114
JSONColumns....114
KeyPassword....115
Keystore....116
KeystorePassword....117
KeywordConflictSuffix....117
LeadingUnderscoreReplacement....118
LegacyVirtualKeys....119
LogConfigFile....120
LoginConfigName....120
LoginTimeout....121
MaxPooledStatements....122
MinVarcharSize....123
NetworkMessageCompressors....124
Password....125
PortNumber....125
ProxyHost....126
ProxyPassword....127
ProxyPort....127
ProxyUser....128
QualifyNormalizedNames....129
ReadOnly....130
ReadPreference....131
RefreshSchema....132
RegisterStatementPoolMonitorMBean....132
ReplicaSetName....133
ResultMemorySize....134
SchemaFilter....135
SchemaFormat....136
SchemaMap....137
ServerName....139
ServicePrincipalName....139
SpecialCharBehavior....140
SpyAttributes....141
StringTruncationMethodForWrites....143
TimestampFormat....144
TransactionMode....145
Truststore....146
TruststorePassword....146
UppercaseIdentifiers....147

User....................148
ValidateServerCertificate....................149
VarcharThreshold....................150

Supported SQL statements and extensions....................151

Create View....................152
Delete....................153
Drop View....................154
Insert....................155
Refresh Map (EXT)....................156
Reload Map (EXT)....................157
Select....................157
    Select Clause....................159
Update....................168
SQL Expressions....................169
    Column Names....................170
    Literals....................170
    Operators....................172
    Functions....................176
    Conditions....................176
Subqueries....................177
    IN Predicate....................177
    EXISTS Predicate....................178
    UNIQUE Predicate....................178
    Correlated Subqueries....................178
Custom function escapes....................179
    CAST_TO_NATIVE function escape....................180

1

Welcome to the Progress DataDirect for JDBC for MongoDB Driver

The Progress® DataDirect® *for* JDBC™ for MongoDB™ driver supports SQL to select data from MongoDB data sources. Some insert, update, and delete capabilities are also supported. The driver translates the SQL statements provided by an application, enabling you to leverage your knowledge of SQL. Additionally, the driver creates a relational schema of your native MongoDB data to support SQL access to MongoDB's flexible schema data model. The relational schema created by the driver is written to a configuration file that can be shared among client machines accessing a common data store.

Once the driver has generated a relational schema, the relationships among objects can be reported through the following metadata methods: getColumns, getImportedKeys, getExportedKeys, getPrimaryKeys, getTypeInfo, getCrossReference, getBestRowIdentifier, getIndexInfo, getSchemas, getCatalogs, and getTables. Furthermore, when performing joins, the driver leverages data relationships in MongoDB collections, minimizing the amount of data that needs to be fetched over the network.

The documentation for the driver also includes the *Progress DataDirect for JDBC Drivers Reference*. The reference provides general reference information for all DataDirect drivers for JDBC, including content on troubleshooting, supported SQL escapes, and DataDirect tools.

For the complete documentation set, visit the Progress DataDirect Connectors Documentation Hub: https://docs.progress.com/category/datadirect-mongodb

For details, see the following topics:

- What's new in this release?
- Requirements
- Migrating schema maps and native files to 6.1
- Installing and setting up the driver

- Driver and DataSource classes
- Connection URL examples
- Data types
- Mapping objects to tables
- SQL escape sequences
- DataDirect tools
- Troubleshooting
- Additional information
- Contacting Technical Support

What's new in this release?

Support and certification

Visit the following web pages for the latest support and certification information.

- Release Notes: https://www.progress.com/datadirect-connectors/whats-new#jdbc
- DataDirect Product Compatibility Guide: https://docs.progress.com/bundle/datadirect-product-compatibility/resource/datadirect-product-compatibility.pdf

Changes Since the 6.1.0 Release

- **Driver Enhancements**
  - The driver has been enhanced to comply with FIPS standards for data encryption. As part of this enhancement, the driver was tested with FIPS 140-3 enabled using a Red Hat OpenJDK 21 on a Red Hat Universal Base Image 9 instance. See FIPS (Federal Information Processing Standard) on page 83 for details.
  - The driver has been enhanced to support the TLSv1.3 cryptographic protocol. See CryptoProtocolVersion on page 106 for details.
  - The driver has been enhanced to allow you to configure whether the driver inserts String values that exceed the column length defined in the relational schema. You can configure this behavior using the new StringTruncationMethodForWrites property. See StringTruncationMethodForWrites on page 143 for details.
  - The driver has been enhanced to allow you to increase the default length of VARCHAR columns to increase the size of data values you can insert with the driver. You can configure the new MinVarcharSize property to specify minimum default length of fields mapped to VARCHAR. See MinVarcharSize on page 123 for details.
  - The driver has been enhanced to allow you to specify the format in which the driver renders the MongoDB composite timestamp. This allows you to specify the format of the MongoDB composite timestamp that is most appropriate for your use case. You can configure this driver behavior using the new TimestampFormat property. See TimestampFormat on page 144 for details.
  - The driver has been enhanced to support replacing the internal mapping files at connection. When the CreateMap property is set to `ForceNew`, the driver replaces the schema map files specified by the SchemaMap property with newly generated files at the same location. In addition, if you are upgrading

from driver version 6.0 to 6.1, the driver retains a copy of the 6.0 version of the schema map files as a backup when generating the 6.1 version of the files. See CreateMap on page 105 for details.

- The driver has been enhanced to support data compression for all messages passed between the client and server. Data compression can significantly reduce network traffic, which, in turn, can lower data transfer costs for cloud services. See NetworkMessageCompressors on page 124 for details.
- The driver has been enhanced to support MongoDB views created in the database. MongoDB views are discovered during the sampling process and mapped using the same schema format used for your collections. See MongoDB views on page 90 for details.
- The driver has been enhanced to support generating legacy virtual keys for newly-discovered nested objects. For driver versions earlier than 6.1, the driver used legacy virtual keys (unique identifiers) as a foreign key to associate the child table back to the parent. When the new LegacyVirtualKeys property is set to `true`, the driver generates legacy virtual keys in the *object_name*`_GENERATED_ID` column in new and migrated schemas. This functionality allows you to maintain consistent column naming for driver generated virtual key columns when migrating normalized schema maps from the 6.0 format. See LegacyVirtualKeys on page 119 for details.
- The driver has been enhanced to support connections to Microsoft Azure Cosmos DB for MongoDB. See Microsoft Azure Cosmos DB for MongoDB on page 25 for details.
- The driver has been enhanced to support the JavaScript and Regex data types. See Data types on page 27 for details.
- The new JSONColumns property allows you to determine whether the driver exposes complex columns as JSON values in addition to their normalized mapping. See JSONColumns on page 114 for details.
- The new FlattenArrayBase property allows you to specifies the starting ordinal value appended to column names for flattened arrays. When flattening arrays, column names are appended with an underscore and the ordinal value (*<array_name>_<oridinal_location>*). This property allows you to determine whether the first ordinal value in the series as either a `0` or a `1`. See FlattenArrayBase on page 111 for details.
- The driver has been enhanced with the new ArrayNormalizationThreshold connection property. ArrayNormalizationThreshold allows you to specify the length of arrays (in elements) at which the driver begins to normalize arrays to child tables when generating a flattened view. This provides you with a method to control the size and focus of your parent table when encountering large arrays or nested arrays. See ArrayNormalizationThreshold on page 102 for details.
- The driver has added support for the MinKey and MaxKey special values. See MinKey and MaxKey values on page 91 for details.
- The driver has been enhanced to support LDAP (Lightweight Directory Access Protocol) authentication. See LDAP authentication on page 75 for details.
- The driver has been enhanced to support the SCRAM-SHA-256 authentication method for user ID and password authentication. When user ID and password authentication is enabled (`AuthenticationMethod=UserIDPassword`), the driver detects and uses the most secure method supported by the server. See AuthenticationMethod on page 104 for details.

- **Changed behavior**
  - The connection property `SpyAttributes` has been updated to exclude the attribute `load=classname`, which was previously used to load the driver specified by the given class name. See SpyAttributes on page 141 for details.
  - The driver no longer compresses messages passed between the client and the server by default. Additional research has uncovered that network compression might not provide the best performance in all environments. As a result, the default behavior of the NetworkMessageCompressors property has been changed to `none`. See NetworkMessageCompressors on page 124 for details.

- The SchemaDefinition property has been added as an alias for the SchemaMap property. This change allows users to continue using their configurations for earlier versions of the driver until they are able update their connection settings. Note that SchemaDefinition is a deprecated property and may not be supported in future versions of the driver. See SchemaMap on page 137 for details.

Changes for the 6.1.0 Release

- **Driver Enhancements**
  - The normalization algorithm has been upgraded to improve sampling performance and optimize the generation of tables in the relational view. In addition, the new SchemaFormat connection property allows you to determine to which relational view the driver maps data, including normalized, mixed, and flattened views. See Mapping objects to tables on page 34 for details.
  - The driver supports migrating schema maps and internal files created with the 6.0 version of the driver so that they can be used by the 6.1 driver. By migrating these files, you can continue to execute the same SQL statements that you did with the 6.0 driver, while still leveraging the advantages of the 6.1 driver. See Migrating schema maps and native files to 6.1 on page 14 for details.
  - The new SchemaFilter connection property allows you to specify the database and collections pairs for which you want the driver to fetch metadata. Using this property can significantly improve connection times by limiting the collections for which metadata is fetched to only those that are required by your application. See SchemaFilter on page 135 for details.
  - The new QualifyNormalizedNames property allows you to determine whether names of relational child-tables normalized from arrays, objects, subdocuments are prefixed with the collection name and names of any parent objects. See QualifyNormalizedNames on page 129 for details.
  - The new SpecialCharBehavior property allows you to determines how the driver handles the mapping of native identifiers containing characters that would require them to be quoted in SQL statements. This property provides a method to choose to continue using identifiers that require quotation marks or for the driver to modify affected identifier names so that quotation marks are not required. See SpecialCharBehavior on page 140 for details.
  - The driver has been enhanced to support replica set failover for write operations. This feature can be enabled by specifying your replica set name using the new ReplicaSetName connection property. See Replica set failover for write operations on page 85 and ReplicaSetName on page 133 for details.
  - The driver has been enhanced to support connections to MongoDB Atlas clusters through a domain. Instead of specifying connection information for individual nodes, the driver allows you to specify the name of the domain using the ServerName property. The driver then uses a DNS lookup to discover the member nodes in the cluster to which it can connect. See MongoDB Atlas clusters on page 84 for details.
  - The new EnableDNSLookup property specifies whether the driver performs a DNS lookup to discover member nodes of a cluster when connecting. When enabled (the default), the driver will discover and connect to the nodes of a cluster when a domain is specified by the ServerName property. Alternatively, if not connecting to cluster, you can experience improved connection times by disabling this property. See EnableDNSLookup on page 108 for details.
  - The driver now includes the MongoDB Configuration Manager for quick configuration and testing of your driver in a web browser. The tool allows you to:
    - Generate and edit connection URLs
    - Test connect your connection URLs
    - Execute SQL commands for testing

    For details, see Generating connection URLs with the Configuration Manager on page 66 and Testing connections and queries on page 67.

- The driver has been enhanced to use proxy server settings defined in the JVM system properties by default. If no proxy settings are defined in the connection string or data source, the driver will attempt to use the values of the `http.proxyHost` and `http.proxyPort` JVM system properties to connect to the database.
- Interactive SQL is now installed with the product. Interactive SQL is a command-line interface that supports connecting your driver to a data source, executing SQL statements and retrieving results in a terminal. This tool provides a method to quickly test your drivers in an environment that does not support GUIs. See Interactive SQL on page 57 for details.
- The driver has been enhanced to allow you to limit sampling to only new collections when refreshing the schema map. This provides for quicker processing times if you only want to map new collections or if existing collections are unchanged. You can limit sampling to only new collections by specifying the `NEW` option in a `Refresh Map` statement. See Refresh Map (EXT) on page 156 for details.
- The driver has been enhanced to support the native Decimal128 data type, which maps to the Decimal JDBC type by default. See Data types on page 27 for details.

- **Changed Behavior**
  - The default mapping behavior of the driver has been changed from generating a normalized view of data to one that is a mixture of normalized and flattened views. The mixed view reduces the number of tables generated while providing a more intuitive data model. You can configure the mapping behavior using the new SchemaFormat property. For details see Mixed view on page 44 and SchemaFormat on page 136 for details.
  - The Schema Tool and manual customizations of the schema map are not currently supported. However, you can modify schema using the mapping connection properties. See Connection property descriptions on page 93 for details.
  - The purpose of the AuthenticationDatabase property has been changed. The property is now used for user id and password authentication (`AuthenticationMethod=userIdPassword`) to specify the database in which the user id was created. This allows you to explicitly select a set of credentials when the same user ID was created on multiple databases.

    As part of this change, AuthenticationDatabase is ignored when using Kerberos authentication. The driver now uses `$external` as the authentication database for all Kerberos-enabled connections. See AuthenticationDatabase on page 103 for details.
  - CreateDB connection property has been replaced by the CreateMap property. The valid values and behavior are identical for both properties. See CreateMap on page 105 for details.
  - The ConfigOptions connection property has been deprecated. As a result, the driver has been enhanced to support setting each of the configuration options formerly supported by ConfigOptions as standalone connection properties. See Connection property descriptions on page 93 for details.
  - The following Config Options are no longer supported:
    - DefaultVarcharSize
    - MaxVarcharSize
    - MinVarcharSize

    To determine the default length of VARCHAR fields, the driver multiplies the largest discovered field by 1.5. For example, if the largest detected field has a length of 100 characters, the driver sets the default length to 150 characters.
  - Supports has ended for versions earlier than MongoDB 3.6.

Requirements

The driver is compatible with JDBC 2.0, 3.0, and 4.0.

The driver requires a Java Virtual Machine (JVM) that is Java SE 8 or higher, including Oracle JDK, OpenJDK, and IBM SDK (Java) distributions.

Migrating schema maps and native files to 6.1

The driver supports migrating schema maps and internal files created with the 6.0 version of the driver so that they can be used by the 6.1 driver. By migrating these files, you can continue to execute the same SQL statements that you did with the 6.0 driver, while leveraging the advantages of the 6.1 driver. Note that new objects discovered by the driver are mapped using the behavior of the 6.1 driver by default. See "Mapping objects to tables" for details.

Note: You can configure the driver to generate legacy virtual keys for new objects by setting LegacyVirtualKeys to `true`. See "LegacyVirtualKeys" for details.

To migrate an existing 6.0 schema map and internal files, using the SchemaMap property, specify the location of your configuration file used for the 6.0 driver and set SchemaFormat to `NormalizeAll`. At connection, the driver modifies your configuration and internal files so that they are compatible with the 6.1 driver. In addition, the driver creates copies of the original files in the same directory, if you want to continue using the 6.0 driver while you complete your upgrade. The copies of the original files take the following form:

```
<existing_file>-<file_version>.<extension>
```

For example, version 2 of a file named `myserver.config` would be renamed:

```
myserver-2.config
```

Note: The versions of the configuration files and internal files are not synched. Therefore, the files will likely have different numbers appended to the them for the 6.0 versions of the files.

Using migrated schema maps and internal files have the following limitation:

- The driver supports only normalized relational views (`SchemaFormat=NormalizeAll`) for migrated schema maps and internal files.

See also

Mapping objects to tables on page 34
LegacyVirtualKeys on page 119

Installing and setting up the driver

This section provides you with an overview of the steps required to install and set-up the driver. After completing this procedure, you will be able to begin accessing data with your application.

To begin accessing data with the driver:

1. Install the driver:
   a) After downloading the product, unzip the installer files to a temporary directory.
   b) From the installer directory, run the appropriate installer file to start the installer.
      - **Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.exe`
      - **Non-Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.jar`

   c) Follow the prompts to complete installation.

   The installer program supports multiple installation methods, including command-line and silent installations. For detailed instructions, refer to the *Progress DataDirect for JDBC Drivers Installation Guide*.

2. Set your system CLASSPATH to include the driver `.jar` file. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate JDBC drivers on your computer. The following examples demonstrate setting the CLASSPATH from a command line using the default installation directory.
   - **Windows Example**

     ```
     CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC\lib\61\mongodb.jar
     ```

   - **UNIX/LINUX Example**

     ```
     CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/61/mongodb.jar
     ```

3. Configure your driver using one of the following methods:
   - **Connection URL**: You can begin using the driver immediately by passing a connection URL with your application or tool. The following example demonstrates the minimal connection properties required to connect with user ID and password authentication:

     ```
     jdbc:datadirect:mongodb://myserver:27017;AuthenticationDatabase=mydb2;
       DatabaseName=mydb;User=jsmith;Password=secret
     ```

     You can also generate a connection string using the Progress DataDirect MongoDB Configuration Manager. For details, see Generating connection URLs with the Configuration Manager.
   - **Data sources**: The driver also supports connecting using JDBC data sources. A JDBC data source is a Java object, specifically a DataSource object, that defines connection information required for a JDBC driver to connect to the database. See Connecting using data sources for more information.

4. Set the values for any optional properties that you want to configure. For additional information on optional features and functionality, see the following resources:
   - Connection URL Examples provides connection string examples that can be used to configure common functionality and features. You can modify and combine these examples to create a string that best suites your environment.
   - Connection property descriptions provides a complete list of supported properties by functionality.
   - Performance considerations describes connection properties that affect performance, along with recommended settings.

5. Connect to your service and begin accessing data with your applications, BI tools, database tools, and more. To help you get started, the following resources guide you through accessing data with some common tools:

- Progress DataDirect MongoDB Configuration Manager: The MongoDB Configuration Manager is a browser-based tool that allows you to quickly generate connection URLs, test connections, and execute test queries.
- DataDirect Test: DataDirect Test allows you to test connect, execute SQL statements, and practice using the JDBC API right out of the box.
- Interactive SQL: Interactive SQL supports a command line interface that allows you to connect to a data source, execute SQL statements and retrieve results for display on a terminal. This tool provides a method of quickly testing your driver in an environment that does not support GUIs.
- Tableau: Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data.
- DbVisualizer: DB Visualizer is a database tool that allows you to connect and execute SQL statements against your data.
- Supported SQL statements and extensions: This section describes the syntax used for SQL statements supported by the driver. You can modify and use the provided examples for your application or tool.

This completes the deployment of the driver.

Driver and DataSource classes

The following are the `Driver` and `DataSource` classes used by the driver:

Driver class:

com.ddtek.jdbc.mongodb.MongoDBDriver

DataSource class:

com.ddtek.jdbcx.mongodb.MongoDBDataSource

Connection URL examples

After setting the CLASSPATH, the connection information needs to be passed in the form of a connection URL. This section provides examples of connection strings configured to use common features and functionality. You can modify and/or combine these examples to create a connection string for your environment.

Note:

- You can also use the DataDirect Configuration Manager tool to generate and test connection URLs. For more information, see "Generating connection URLs with the Configuration Manager."
- Connection property names are case-insensitive. For example, `Password` is the same as `password`.
- For connection properties that support string values, use the following escape sequence to specify values containing leading or trailing spaces and curly brackets: `{`*`value`*`}`. For example: `User={hello }` or `Password={{hello}}`.

See also

Generating connection URLs with the Configuration Manager on page 66

No authentication

This string includes the properties used to connect to a server that does not require authentication.

```
jdbc:datadirect:mongodb://host:port;AuthenticationMethod=None;DatabaseName=database;
  [property=value[;...]];
```

where:

host

specifies the name or the IP address of the MongoDB server to which you want to connect. For example, `myserver`.

port

specifies the port number of the server listener. The default is `27017`.

database

(optional) specifies the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries.

Important: This value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, this value is case-sensitive.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

The following example connection string includes the properties required for connecting without authentication.

```
Connection conn = DriverManager.getConnection

("jdbc:datadirect:mongodb://myserver:27017;AuthenticationMethod=None;DatabaseName=mydb;");
```

See also

User ID and password authentication on page 74
Connection property descriptions on page 93

User ID and password

This string includes the properties used to connect with basic authentication.

```
jdbc:datadirect:mongodb://host:port;AuthenticationDatabase=auth_db;DatabaseName=database;

     User=username;Password=password;[property=value[;...]];
```

where:

host

specifies the name or the IP address of the MongoDB server to which you want to connect. For example, `myserver`.

port

specifies the port number of the server listener. The default is `27017`.

auth_db

(recommended) specifies the database in which your user ID was created. This value allows you to explicitly select a set of credentials and permissions when the same user ID was created in multiple databases on the server.

Note: We recommend specifying a value for this option when using user ID and password authentication (`AuthenticationMethod=userIdPassword`) to ensure that the correct permissions are used for your connection.

database

(optional) specifies the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries.

Important: This value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, this value is case-sensitive.

user

specifies the user name that is used to connect to the MongoDB database. For example, `jsmith`.

password

specifies the password used to connect to your MongoDB database.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following example connection string includes the properties required for connecting with user ID and password authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:mongodb://myserver:27017;AuthenticationDatabase=mydb2;
         DatabaseName=mydb;User=jsmith;Password=secret;");
```

See also

User ID and password authentication on page 74
Connection property descriptions on page 93

Kerberos authentication

This string includes the properties used to connect with Kerberos authentication.

```
jdbc:datadirect:mongodb://host:port;AuthenticationMethod=kerberos;DatabaseName=database;
ServicePrincipalName=principal_name;User=username;[property=value[;...]];
```

where:

host

specifies the name or the IP address of the MongoDB server to which you want to connect. For example, `myserver`.

port

specifies the port number of the server listener. The default is `27017`.

database

(optional) specifies the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries.

Important: This value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, this value is case-sensitive.

---

principal_name

(optional) if the default value built by the driver does not match the service principal name registered with the KDC, specify the three-part service principal name. By default, the driver builds the ServicePrincipalName by concatenating the service name `mongodb`, the fully qualified domain name (FQDN) as specified with the HostName property, and the default realm name as specified in the Kerberos configuration file.

user

(optional) specifies the user name that is used to connect to the MongoDB database. For example, `jsmith`.

Note: The User property is not required to be stored in the connection string. This value can also be passed separately by the application.

---

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

The following example connection string includes the properties required for connecting with Kerberos authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:mongodb://myserver:27017;AuthenticationMethod=kerberos;
         DatabaseName=mydb;user=jsmith;");
```

See also

Connection property descriptions on page 93
Kerberos authentication requirements on page 78

LDAP authentication

This string includes the properties used to connect with LDAP (Lightweight Directory Access Protocol) authentication.

```
jdbc:datadirect:mongodb://host:port;AuthenticationMethod=plain;DatabaseName=database;
User=username;Password=password;[property=value[;...]];
```

where:

host

specifies the name or IP address of the MongoDB server to which you want to connect. For example, `myserver`.

port

specifies the port number of the server listener. The default is `27017`.

database

(optional) specifies the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries.

Important: This value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, this value is case-sensitive.

user

(optional) specifies the user name that is used to connect to the MongoDB database. For example, `jsmith`.

password

specifies the password used to connect to your MongoDB database.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Important: When LDAP authentication is enabled, credentials are passed in clear text. Therefore, you should use LDAP authentication only on servers that are configured for TLS/SSL encryption.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following example connection string includes the properties required for connecting with LDAP authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:mongodb://myserver:27017;AuthenticationMethod=plain;
         DatabaseName=mydb;user=jsmith;password=secret;");
```

See also

Connection property descriptions on page 93
LDAP authentication on page 75

TLS/SSL encryption

This string includes the properties used to connect with TLS/SSL encryption and user ID and password authentication. For more information, see "TLS/SSL data encryption."

```
jdbc:datadirect:mongodb://host:port;CryptoProtocolVersion=protocol;DatabaseName=database;
EncryptionMethod=[SSL |
requestSSL];HostNameInCertificate=host_name;KeyPassword=key_password;
KeyStore=key_store;TrustStore=trust_store;TrustStorePassword=ts_password;
ValidateServerCertificate=vsc_value;User=username;Password=password;[property=value[;...]];
```

where:

host

specifies the name or the IP address of the MongoDB server to which you want to connect.

For example, `myserver`.

port

specifies the port number of the server listener. The default is `27017`.

protocol

(optional) specifies a cryptographic protocol or comma-separated list of cryptographic protocols that can be used when SSL is enabled. For example, `TLSv1.3,TLSv1.2`.

database

(optional) specifies the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries.

Important: This value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, this value is case-sensitive.

host_name

(optional) specifies the host name to be used to validate the certificate. The HostNameInCertificate property provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.

key_password

(optional) specifies the password of the keystore file, if your database server is configured for SSL client authentication. Alternatively, you can set the corresponding Java system property javax.net.ssl.keyStorePassword, in lieu of this property.

key_store

(optional) specifies the location of the keystore file, if your database server is configured for SSL client authentication. Alternatively, you can set the corresponding Java system property, javax.net.ssl.keyStore, in lieu of this property.

trust_store

(optional) specifies the location of the truststore file used for SSL server authentication. Alternatively, you can set the corresponding Java system property, javax.net.ssl.trustStore, in lieu of this property.

ts_password

(optional) specifies the password of the truststore file used for SSL server authentication. Alternatively, you can set the corresponding Java system property, javax.net.ssl.trustStorePassword, in lieu of this property.

vsc_value

(optional) specify `true` to validate certificates sent by the database server.

user

(optional) specifies the user name that is used to connect to the MongoDB database. For example, `jsmith`.

password

(optional) specifies the password used to connect to your MongoDB database.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Note: The User and password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following example connection string includes the minimum properties required for connecting with data encryption, and user ID and password authentication.

```
Connection conn = DriverManager.getConnection

("jdbc:datadirect:mongodb://myserver:27017;DatabaseName=mydb;EncryptionMethod=SSL;User=jsmith;

          Password=secret;");
```

See also

Data Encryption on page 80
Connection property descriptions on page 93

Replica set failover

This string includes the properties used to enable replica set failover for write operations with no authentication.

```
jdbc:datadirect:mongodb://host:port;AuthenticationMethod=None;DatabaseName=database;
     ReplicaSetName=replica_set;[property=value[;...]];
```

where:

host

specifies the name or the IP address of the MongoDB server to which you want to connect.

For example, `myserver`.

port

specifies the port number of the server listener. The default is `27017`.

database

(optional) specifies the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries. Required for User/ID password authentication.

Important: This value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, this value is case-sensitive.

replica_set

specifies the name of the replica set against which you want to execute write operations.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

The following example connection string includes the properties used to enable replica set failover for write operations with no authentication:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:mongodb://MongodbServer:27017;AuthenticationMethod=None;
         DatabaseName=Mongodb1;ReplicaSetName=MyReplicaSet;");
```

See also

Replica set failover for write operations on page 85
Connection property descriptions on page 93

MongoDB Atlas

This string includes the properties used to connect to a MongoDB Atlas cluster.

```
jdbc:datadirect:mongodb://host:27017;AuthenticationDatabase=auth_db;
  DatabaseName=database;EncryptionMethod=SSL;User=username;Password=password;
  [property=value[;...]];
```

where:

host

specifies the domain name of the MongoDB Atlas cluster to which you want to connect. At connection, the driver performs a DNS lookup to identify the member nodes of the domain, then connects to an available node.

port

specifies the port number of the server listener. The default is `27017`.

auth_db

(recommended) specifies the database in which your user ID was created. This value allows you to explicitly select a set of credentials and permissions when the same user ID was created in multiple databases on the server.

Note: We recommend specifying a value for this option when using user ID and password authentication (`AuthenticationMethod=userIdPassword`) to ensure that the correct permissions are used for your connection.

---

database

(optional) specifies the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries. Required for User/ID password authentication.

Important: This value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, this value is case-sensitive.

---

user

specifies the user name that is used to connect to the MongoDB database. For example, `jsmith`.

password

specifies the password used to connect to your MongoDB database.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following example connection string includes the properties required for connecting to a MongoDB Atlas cluster.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:mongodb://myaltas.mongodb.net:27017;AuthenticationDatabase=mydb2;
         DatabaseName=mydb;EncryptionMethod=SSL;User=jsmith;Password=secret;");
```

See also

Connection property descriptions on page 93
MongoDB Atlas clusters on page 84

Microsoft Azure Cosmos DB for MongoDB

This string includes the properties used to connect to a Microsoft Azure Cosmos DB for MongoDB. Connection strings for Azure Cosmos DB for MongoDB are similar to standard MongoDB connections, but require TLS encryption and authentication to be enabled.

```
jdbc:datadirect:mongodb://host:port;DatabaseName=database;EncryptionMethod=SSL;
ValidateServerCertificate=false;User=username;Password=password;[property=value[;...]];
```

where:

host

specifies the name or the IP address of the Azure Cosmos DB for MongoDB server to which you want to connect. For example, `myCosmosdbServer`.

port

specifies the port number of the server listener. This value is typically `10255` for Azure Cosmos DB for MongoDB connections.

database

(optional) specifies the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries.

Important: This value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, this value is case-sensitive.

user

(optional) specifies the your Azure Cosmos DB account name. For example, `jsmith`.

password

(optional) specifies the password used to connect to your Azure Cosmos DB account.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following example connection string includes the basic properties used for connecting to a Azure Cosmos DB database.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:mongodb://myCosmosdbServer:10255;DatabaseName=mydb;
     EncryptionMethod=SSL;ValidateServerCertificate=false;User=jsmith;Password=secret;");
```

See also

Connection property descriptions on page 93

Proxy server

This string includes the properties used to connect through a proxy server with basic authentication.

```
jdbc:datadirect:mongodb://host:port;AuthenticationDatabase=auth_db;
DatabaseName=database;ProxyHost=proxy_host;ProxyPassword=proxy_password;
ProxyPort=proxy_port;ProxyUser=proxy_user;User=username;Password=password;
[property=value[;...]];
```

where:

host

specifies the name or the IP address of the MongoDB server to which you want to connect. For example, `myserver`.

port

specifies the port number of the server listener. The default is `27017`.

auth_db

(recommended) specifies the database in which your user ID was created. This value allows you to explicitly select a set of credentials and permissions when the same user ID was created in multiple databases on the server.

Note: We recommend specifying a value for this option when using user ID and password authentication (`AuthenticationMethod=userIdPassword`) to ensure that the correct permissions are used for your connection.

database

(optional) specifies the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries.

Important: This value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, this value is case-sensitive.

proxy_host

specifies the proxy server to use for the first connection.

proxy_password

specifies the password needed to connect to a proxy server for the first connection.

proxy_port

specifies the port number where the proxy server is listening for requests for the first connection. The default is `0`.

proxy_user

specifies the user name needed to connect to a proxy server for the first connection.

user

specifies the user name that is used to connect to the MongoDB database. For example, `jsmith`.

password

specifies the password used to connect to your MongoDB database.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following example connection string includes the properties required for using a proxy server with user ID and password authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:mongodb://myserver:27017;AuthenticationDatabase=mydb2;
    ProxyHost=pserver;ProxyPassword=psecret;ProxyPort=808;ProxyUser=PUser;
    DatabaseName=mydb;User=jsmith;Password=secret;");
```

See also

Connection property descriptions on page 93

Data types

The following table lists data types supported by the driver and how they are mapped to JDBC data types.

See "getTypeInfo()" for getTypeInfo() results of data types supported by the driver.

Table 1: MongoDB Data Types

MongoDB Data Type	JDBC Data Type
ARRAY	LONGVARCHAR
BIGINT	BIGINT

MongoDB Data Type	JDBC Data Type
BINDATA	VARBINARY
BOOLEAN	BOOLEAN
CHAR	CHAR
DATE	TIMESTAMP
DECIMAL128	DECIMAL
DOUBLE	DOUBLE
INTEGER	INTEGER
JAVASCRIPT	VARCHAR
LONGVARCHAR	LONGVARCHAR
OBJECT	LONGVARCHAR
OBJECTID	VARCHAR
REGEX[1]	VARCHAR
STRING	VARCHAR

Default mapping of columns with inconsistent native data types

Due to the flexibility of the MongoDB schema data model, your native data sources may not enforce consistent data types. To ensure data integrity when mapping to a relational model, the driver describes columns with inconsistent native data types as a single SQL data type. The driver determines which SQL type to use based on the combination of native types detected when sampling data. The following table lists combinations of MongoDB data types and their default mapping for JDBC.

Note: In some cases, a value with a specific native type can be concealed or obscured because the driver describes a field with inconsistent native types as a column with a single SQL type. The `CAST_TO_NATIVE` function escape allows users to send a value as it is defined in the MongoDB database, rather than how it is described in the relational model of the data. Currently, `CAST_TO_NATIVE` can only be used with the ObjectID type in `SELECT` statement filters and literal `INSERT` values. See "CAST_TO_NATIVE Function Escape" for details.

Table 2: Default Mapping for Columns Containing Inconsistent MongoDB Data Types

MongoDB Data Types	JDBC Data Type
BIGINT and INTEGER	BIGINT (-5)

[1] The driver supports only the literal format for the Regex data type. Therefore, when executing an Insert or Select statement, Regex values must be specified using the literal format.

MongoDB Data Types	JDBC Data Type
DOUBLE and INTEGER	DOUBLE (8)
All other combinations	VARCHAR (12) or LONGVARCHAR (-1)[2]

See also

CAST_TO_NATIVE function escape on page 54

getTypeInfo()

The DatabaseMetaData.getTypeInfo() method returns information about data types. The following table provides getTypeInfo() results for supported data types.

Table 3: getTypeInfo() Results

TYPE_NAME = ARRAY AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = ARRAY MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777204 SEARCHABLE = 0 SQL_DATA_TYPE = -1 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = BIGINT AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = BIGINT MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = 25 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE

[2] When the driver discovers a column with 4000 characters or less, the column is mapped as VARCHAR and the precision is 4000 characters. When the driver discovers a column with more than 4000 characters, the column is mapped as LONGVARCHAR and precision is 16 MB.

TYPE_NAME = BINDATA AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = BINDATA MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777204 SEARCHABLE = 3 SQL_DATA_TYPE = 61 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = BOOLEAN AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 16 (BOOLEAN) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = BOOLEAN MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 3 SQL_DATA_TYPE = 16 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = CHAR AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = CHAR MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = 1 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = DATE AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = TIMESTAMP ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATE MAXIMUM_SCALE = 3	MINIMUM_SCALE = 3 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 23 SEARCHABLE = 3 SQL_DATA_TYPE = 93 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = DECIMAL128 AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = DECIMAL128 MAXIMUM_SCALE = 34	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 68 SEARCHABLE = 3 SQL_DATA_TYPE = 3 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = DOUBLE AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = DOUBLE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 53 SEARCHABLE = 3 SQL_DATA_TYPE = 8 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE

TYPE_NAME = INTEGER AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = INTEGER MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = 4 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = JAVASCRIPT AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = JAVASCRIPT MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777204 SEARCHABLE = 3 SQL_DATA_TYPE = 12 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = LONGVARCHAR AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LONGVARCHAR MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777204 SEARCHABLE = 0 SQL_DATA_TYPE = -1 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = OBJECT AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = OBJECT MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777204 SEARCHABLE = 0 SQL_DATA_TYPE = -1 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = OBJECTID AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = OBJECTID MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 24 SEARCHABLE = 3 SQL_DATA_TYPE = 12 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = REGEX AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = REGEX MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777204 SEARCHABLE = 3 SQL_DATA_TYPE = 12 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = STRING AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = STRING MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 4000 SEARCHABLE = 3 SQL_DATA_TYPE = 12 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

Mapping objects to tables

Data mapping describes how elements are mapped between two distinct data models. To support SQL access to a MongoDB database, the MongoDB database must be mapped to a relational schema. The driver creates the relational schema by mapping MongoDB data in a normalized, flattened, or mixed view of the data.

- **Normalized view**: MongoDB collections are normalized into sets of parent-child tables. The child tables correspond to complex types, such as arrays and embedded documents (or subdocuments), and have a foreign key relationship to the parent table. For more information, see "Normalized view."
- **Flattened view**: MongoDB collections are flattened into single, relational tables. All fields, including those that may comprise complex types, are organized as columns within the relational table. For more information, see "Flattened view."
- **Mixed view**: The driver generates an optimized view of the data in which MongoDB collections are mapped to a normalized view that flattens certain objects into columns in the parent table. For example, subdocuments are mapped as columns in the parent table, while most arrays and nested subdocuments are mapped to separate child tables. See "Mixed view" for more information.

The driver generates the relational view upon the initial connection to a data source. By default, the driver generates a mixed view of relational data. You can determine the relational view using the SchemaFormat property.

At connection, the relational schema is written to the schema map configuration file. This configuration file may be shared by multiple users. Note that users must have appropriate privileges to the exposed collections to access them.

Note: The names of relational tables demonstrated in this section are generated based on the default behavior of the driver. You can modify how the driver generates table names using the QualifyNormalizedNames property.

See also

SchemaFormat on page 136
QualifyNormalizedNames on page 129

Normalized view

When your native data is normalized, each MongoDB collection is decompounded into a set of parent-child tables. Fields containing simple types are mapped as columns in a parent table, while complex types, such as subdocuments and arrays, are mapped as child tables. Child tables share a foreign key relationship with their parent table.

For example, the collection `residents` contains the array `vehicles` and fields in the `address` document (or object). The collection's JSON structure can be rendered as follows:

```
{"_id": "ajx363",
 "name": "Sydney Smith",
 "address": {"street": "101 Main Street", "city": "Raleigh", "state": "NC"},
 "county": "Wake",
 "vehicles": ["car", "boat"]
}

{"_id": "tzn525",
 "name": "Cora Welch",
 "address": {"street": "191 First Street", "city": "Chapel Hill", "state": "NC"},
 "county": "Orange",
 "vehicles": ["scooter", "truck"]
}
```

The collection has been decompounded into separate but related tables. The normalization of the `residents` collection yields one parent table and two child tables. In the parent table, the `_id` field is mapped as a primary key column, and fields with other simple types are mapped as relational columns. The parent table adopts the name `RESIDENTS` and takes the following form:

Table 4: RESIDENTS

_ID (PK)	NAME	COUNTY
ajx363	Sydney Smith	Wake
tzn525	Cora Welch	Orange

The information in the `vehicles` array is mapped to the following child table:

Table 5: RESIDENTS_VEHICLES

RESIDENTS_ID (FK)	POSITION	VEHICLES
ajx363	0	car
ajx363	1	boat
tzn525	0	scooter
tzn525	1	truck

The mapping of the `vehicles` array to the `RESIDENTS_VEHICLES` table is handled as follows:

- The table name, `RESIDENTS_VEHICLES`, is derived from the name of the parent table and the array.
- To establish a foreign key relationship with the parent table, a column is generated based on the `_ID` field using the following naming convention:

  ```
  <parent_table>_ID
  ```

  For example, `RESIDENTS_ID`.
- The primary key of the child table is a composite key formed by the primary key of the parent table (`RESIDENTS_ID`) combined with the positional information contained in the `POSITION` column.
- Fields in the object are mapped to columns in the child table. For example, the `vehicles` field becomes the `VEHICLES` column.

The information in the `address` object maps to the following child table:

Table 6: RESIDENTS_ADDRESS

RESIDENTS_ID (PK & FK)	STREET	CITY	STATE
ajx363	101 Main Street	Raleigh	NC
tzn525	191 First Street	Chapel Hill	NC

The mapping of the `address` object to the `RESIDENTS ADDRESS` table is handled as follows:

- The table name, `RESIDENTS_ADDRESS`, is derived from the parent table and the name of the object.
- To establish a foreign key relationship with the parent table, a column is generated based on the `_ID` field using the following naming convention:

  ```
  <parent_table>_ID
  ```

  For example, `RESIDENTS_ID`.
- Fields in the object are mapped to columns in the child table. For example, the `city` field becomes the `CITY` column.

See also

ColumnDiscoverySampleSize on page 105

Nested complex types (normalized view)

The following examples illustrate a number of ways the normalization of complex types are handled.

A subdocument nested in a subdocument

In this example, the subdocument `location` contains the subdocuments `city` and `state` in the `employee` collection:

```
{"_id": "pdn313",
 "name": "Charlotte",
 "manager": {"name": "Robert", "department": "Development",
            "location": {"city": "Tulsa", "state": "OK"}}
}

{"_id": "gkx136",
 "name": "Benjamin",
 "manager": {"name": "Michael", "department": "Quality Assurance",
            "location": {"city": "Dallas", "state": "TX"}}
}
```

The information in the `employee` collection would be mapped to the `EMPLOYEE` parent table, and the `EMPLOYEE_MANAGER` and the `EMPLOYEE_LOCATION` child tables. The `EMPLOYEE` table would be derived from the simple types `_id` and `name` as follows:

Table 7: EMPLOYEE

_ID	NAME
pdn313	Charlotte
gkx136	Benjamin

The information from the `manager` object is mapped to the following child table:

Table 8: EMPLOYEE_MANAGER

	NAME	DEPARTMENT
pdn313	Robert	Development
gkx136	Michael	Quality Assurance

The mapping of the `manager` object to the `EMPLOYEE_MANAGER` table is handled as follows:

- The table name, `EMPLOYEE_MANAGER`, is derived from the name of the parent table and the object.
- To establish a foreign key relationship with the parent table, a column is generated based on the `_ID` field using the following naming convention:

  ```
  <parent_table>_ID
  ```

  For example, `EMPLOYEE_ID`.
- The nested subdocuments are mapped to columns in the child table. For example, the `name` field is mapped to the `NAME` column.

The information in the `location` object is mapped to the following child table:

Table 9: EMPLOYEE_LOCATION

EMPLOYEE_ID (PK & FK)	CITY	STATE
pdn313	Tulsa	OK
gkx136	Dallas	TX

The mapping of the `location` object to the `EMPLOYEE_LOCATION` table is handled as follows:

- The table name, `EMPLOYEE_LOCATION`, is derived from the name of the parent table and the object.
- The `id_` field is mapped as `EMPLOYEE_ID` to establish a foreign key relationship to the parent table.
- Nested subdocuments are mapped to columns. In this example, the `city` and `state` subdocuments, which indicate the location of an employee's manager, are mapped as `CITY` and `STATE` columns.

Subdocuments nested in an array

In the collection `contacts`, the `addresses` array contains the subdocuments: `label`, `street`, `city`, and `state`.

```
{"_id": "ajx456",
 "name": "Andrea",
 "addresses": [
              {"label": "Home", "street": "101 Main St", "zip": "27513"},
              {"label": "Work", "street": "303 Main St", "zip": "27560"}
              ]
}
```

The information in the `contacts` collection would be mapped to the `CONTACT` parent table, and the `CONTACTS_ADDRESSES` child table. The `CONTACTS` table would be derived from the simple types `_id` and `name` as follows:

Table 10: CONTACTS

_ID	NAME
ajx456	Andrea

The information in the `addresses` array maps to the following child table:

Table 11: CONTACTS_ADDRESSES

CONTACTS_ID (FK)	POSITION (PK)	LABEL	STREET	ZIP
ajx456	0	Home	101 Main St	27513
ajx456	1	Work	303 Main St	27560

The mapping of the `addresses` array to the `CONTACTS_ADDRESSES` tables is handled as follows:

- The table name, `CONTACTS_ADDRESSES`, is derived from the name of the parent table and array.
- To establish a foreign key relationship with the parent table, a column is generated based on the `_ID` field using the following naming convention:

  `<parent_table>_ID`

For example, CONTACTS_ID.

- The primary key of the child table is a composite key formed by the primary key of the parent table (CONTACTS_ID) combined with the positional information contained in the POSITION column.
- Subdocuments in an array are mapped as columns. For example, the label field is mapped to the LABEL column.

An array nested in a document

In a separate scenario, the manager document contains the emails array in the employee collection. The collection has the following JSON structure:

```
{"_id": "ajp211",
 "name": "Mark",
 "manager": {"name": "Cynthia",
            "emails": ["cynthia@email.com", "watsonc@email.com"]}
}
{"_id": "mpc393",
 "name": "Deborah",
 "manager": {"name": "Cynthia",
            "emails": ["cynthia@email.com", "watsonc@email.com"]}
}
{"_id": "dlm215",
 "name": "Jason",
 "manager": {"name": "Chris",
            "emails": ["chris@email.com", "cwright@email.com"]}
}
```

The information in the employee collection would be mapped to the EMPLOYEE parent table, and the EMPLOYEE_EMAILS and EMPLOYEE_MANAGER child tables. The EMPLOYEE table would be derived from the simple types _id and name as follows:

Table 12: EMPLOYEE

_ID	NAME
ajp211	Mark
mpc393	Deborah
dlm215	Jason

The information in the emails array is mapped to the following child table:

Table 13: EMPLOYEE_EMAILS

EMPLOYEE_ID (FK)	POSITION	EMAILS
ajp211	0	cynthia@email.com
ajp211	1	watsonc@email.com
mpc393	0	cynthia@email.com
mpc393	1	watsonc@email.com
dlm215	0	chris@email.com
dlm215	1	cwright@email.com

The mapping of the nested `emails` array to the `EMPLOYEE_EMAILS` table is handled as follows:

- The table name, `EMPLOYEE_EMAILS`, is derived from the name of the parent table and the array.
- To establish a foreign key relationship with the parent table, a column is generated based on the `_ID` field using the following naming convention:

  `<parent_table>_ID`

  For example, `EMPLOYEE_ID`.
- The primary key of the child table is a composite key formed by the primary key of the parent table (`EMPLOYEE_ID`) combined with the positional information contained in the `POSITION` column.
- Nested arrays are mapped as columns. For example, the `emails` array is mapped to the `EMAILS` column. Note that the emails in the following child table are those of the managers, but correspond to the employee identification number.

The information in the `names` subdocument is mapped to the following child table:

Table 14: EMPLOYEE_MANAGER

EMPLOYEE_ID (PK & FK)	NAME
ajp211	Cynthia
mpc393	Cynthia
dlm215	Chris

The mapping of the nested names subdocument to the EMPLOYEE_MANAGER table is handled as follows:

- The table name, EMPLOYEE_MANAGER, is derived from the name of the parent table and the object.
- To establish a foreign key relationship with the parent table, a column is generated based on the _ID field using the following naming convention:

  *<parent_table>*_ID

  For example, EMPLOYEE_ID.
- The nested name subdocument (the name of the manager) is mapped as a column. Note that the names in the following child table are those of the managers, but correspond to the employee identification number.

An array nested in an array

In the collection offices, the departments array contains an employees array.

```
{"_id": "au32",
 "city": "Bedford",
 "departments": [{"name": "Development",
                  "employees": [
                                {"first": "Leslie", "last": "Jacobs"},
                                {"first": "Emma", "last": "Alves"},
                                {"first": "Brad", "last": "Jones"}
                               ]
                 },
                 {"name": "Human Resources",
                  "employees": [
                                {"first": "Joseph", "last": "Lu"},
                                {"first": "Margaret", "last": "Baker"},
                                {"first": "Chetna", "last": "Campbell"}
                               ]
                 },
                 {"name": "IT",
                  "employees": [
                                {"first": "Caroline", "last": "Evans"},
                                {"first": "Markus", "last": "Campanella"},
                                {"first": "Jennifer", "last": "Bradley"}
                               ]
                 }]
 }
{"_id": "xn44",
 "city": "Morrisville",
 "departments": [{"name": "Development",
                  "employees": [
                                {"first": "Charles", "last": "Scott"},
                                {"first": "Mary", "last": "Gonzales"},
                                {"first": "Phil", "last": "McEnroe"}
                               ]
                 },
                 {"name": "Operations",
                  "employees": [
                                {"first": "Rachel", "last": "Cullingford"},
                                {"first": "Lance", "last": "Friedman"},
                                {"first": "Amanda", "last": "Giachetti"}
                               ]
                 },
                 {"name": "IT",
                  "employees": [
                                {"first": "Ingrid", "last": "Burkis"},
                                {"first": "Catherine", "last": "Wheeler"},
                                {"first": "Jacob", "last": "Williams"}
                               ]
                 }]
}
```

The information in the `offices` collection would be mapped to the `OFFICES` parent table, and the `OFFICES_DEPARTMENTS` and `OFFICES_EMPLOYEES` child tables. The `OFFICES` table would be derived from the simple types `_id` and `city` as follows:

Table 15: OFFICES

_ID (PK)	CITY
au32	Bedford
xn44	Morrisville

The information in the `departments` nested array is mapped to the following child table:

Table 16: OFFICES_DEPARTMENTS

OFFICES_ID (FK)	POSITION (PK)	NAME
au32	0	Development
au32	1	Human Resources
au32	2	IT
xn44	0	Development
xn44	1	Operations
xn44	2	IT

The mapping of the `departments` nested array to the `OFFICES_DEPARTMENTS` table is handled as follows:

- The table name, `OFFICES_DEPARTMENTS`, is derived from the name of the parent table and the array.
- To establish a foreign key relationship with the parent table, a column is generated based on the `_ID` field using the following naming convention:

  `<parent_table>_ID`

  For example, `OFFICES_ID`.
- The primary key of the child table is a composite key formed by the primary key of the parent table (`OFFICES_ID`) combined with the positional information contained in the `POSITION` column.
- Nested arrays are mapped as columns. For example, the `name` array is mapped to the `NAME` column.

The information in the `employees` array is mapped to the following child table:

Table 17: OFFICES_EMPLOYEES

	OFFICES_ DEPARTMENTS _POSITION	POSITION (PK)	FIRST	LAST
au32	0	0	Leslie	Jacobs
au32	0	1	Emma	Alves
au32	0	2	Brad	Jones
au32	1	0	Joseph	Lu
au32	1	1	Margaret	Baker
au32	1	2	Chetna	Campbell
au32	2	0	Caroline	Evans
au32	2	1	Markus	Campanella
au32	2	2	Jennifer	Bradley
xn44	0	0	Charles	Scott
xn44	0	1	Mary	Gonzales
xn44	0	2	Phil	McEnroe
xn44	1	0	Rachel	Cullingford
xn44	1	1	Lance	Friedman
xn44	1	2	Amanda	Giachetti
xn44	2	0	Ingrid	Burkis
xn44	2	1	Catherine	Wheeler
xn44	2	2	Jacob	Williams

The mapping of the nested `employees` array to the `OFFICES_EMPLOYEES` table is handled as follows:

- The table name, `OFFICES_EMPLOYEES`, is derived from the name of the parent table and the array.
- The `OFFICES_ID` and `OFFICES_DEPARTMENTS_POSITION` columns establish a foreign key relationship to the parent array based on the primary key from the `DEPARTMENTS` table.
- The primary key of the child table is a composite key formed by the primary key of the parent table (`OFFICES_ID`) combined with the positional information contained in the `POSITION` column.
- Nested arrays are mapped as columns. For example, the `first` array is mapped to the `FIRST` column.

Flattened view

The following example shows how the source data is flattened.

A data source has a collection called `employee` with the following structure:

```
{"_id": "pdn313",
 "name": "Charlotte",
 "manager": {"name": "Robert",
             "emails": ["bob@email.com", "robert@email.com"]
            }
}
```

The `employee` collection would be flattened into a relational table with the following structure:

_ID	NAME	MANAGER_NAME	MANAGER_EMAILS_1	MANAGER_EMAILS_2
pdn313	Charlotte	Robert	bob@email.com	robert@email.com

All fields are retained as separate columns in the resulting relational table. Simple types such as `_id` and `name` correspond directly to columns. In turn, subdocuments are flattened into columns using the `<objectname>_<fieldname>` pattern. Next, the `emails` array is flattened into two columns, using an extended version of the `<arrayname>_<arrayindex>` pattern: `<objectname>_<arrayname>_<arrayindex>`.

Note: As subdocuments or arrays are discovered at deeper and deeper levels, the `<objectname>_<fieldname>` or `<arrayname>_<arrayindex>` pattern is extended, for example, `<objectname>_<objectname>_<fieldname>`.

Note: To avoid creating very large tables, arrays containing twelve or more elements are normalized into child tables by default. You can configure this behavior using the ArrayNormalizationThreshold property.

Mixed view

By default, the driver generates a mixed-normalized view, which changes the composition of the relational tables and which objects are mapped to child tables.

In the following example, the collection `residents` contains the array `vehicles` and fields in the `address` document (or object). The collection's JSON structure can be rendered as follows:

```
{"_id": "ajx363",
 "name": "Sydney Smith",
 "address": {"street": "101 Main Street", "city": "Raleigh", "state": "NC"},
 "county": "Wake",
 "vehicles": ["car", "boat"]
}

{"_id": "tzn525",
 "name": "Cora Welch",
 "address": {"street": "191 First Street", "city": "Chapel Hill", "state": "NC"},
 "county": "Orange",
 "vehicles": ["scooter", "truck"]
}
```

In the mixed view, fields containing simple types are mapped to the parent table and nested complex types and arrays are mapped as child tables. Subdocuments with simple types are appended to the parent table. Therefore, normalization of the `residents` collection, produces a single table. The resulting table takes the following form:

Table 18: RESIDENTS

_ID (PK)	NAME	ADDRESS_ STREET	ADDRESS_ CITY	ADDRESS_ STATE	COUNTY	VEHICLES_ 1	VEHICLES_ 2
ajx363	Sydney Smith	101 Main Street	Raleigh	NC	Wake	car	boat
tzn525	Cora Welch	191 First Street	Chapel Hill	NC	Orange	scooter	truck

The mapping of the `residents` collection to the `RESIDENTS` table is handled as follows:

- Simple types are mapped to columns in the parent table as columns. For example, the `name` object is mapped to `NAME` column.
- Fields for the subdocument `address` are mapped as columns to the parent table. Column names for fields generated from a subdocument take the following form:

  <subdocument_name>_<field_name>

  For example, for the field `street` in the subdocument `address`, the resulting column name would be `ADDRESS_STREET`.
- Fields in the vehicles array are mapped as columns to the parent table. Column names for fields generated from an array take the following form:

  <array_name>_<oridinal_location>

  Important: In the mixed view, if the number of values in an array are uniform across the collection and are less than or equal to twelve per element, the array values are flattened into columns in the parent table (for example, `VEHICLES_1`, `VEHICLES_2`). If the number of values are not uniform or exceed twelve per element, the driver maps the array to a child table.

  For example, for the second value of the array `vehicles`, the resulting column name would be `VEHICLE_2`.

Nested complex types (mixed view)

The following examples illustrate a number of ways the driver maps complex types in a mixed representation of the relational schema.

A subdocument nested in a subdocument

In this example, the subdocument `location` contains the subdocuments `city` and `state` in the `employee` collection:

```
{"_id": "pdn313",
 "name": "Charlotte",
 "manager": {"name": "Robert", "department": "Development",
 "location": {"city": "Tulsa", "state": "OK"}}
}

{"_id": "gkx136",
 "name": "Benjamin",
 "manager": {"name": "Michael", "department": "Quality Assurance",
             "location": {"city": "Dallas", "state": "TX"}}
}
```

The information from the `employee` object is mapped to the following table:

Table 19: EMPLOYEE

_ID	NAME	MANAGER_ NAME	MANAGER_ DEPARTMENT	MANAGER_ LOCATION_CITY	MANAGER_ LOCATION_STATE
pdn313	Charlotte	Robert	Development	Tulsa	OK
gkx136	Benjamin	Michael	Quality Assurance	Dallas	TX

The mapping of the `employee` collection to the `EMPLOYEE` table is handled as follows:

- Simple types are mapped to columns in the parent table as columns. For example, the `name` object is mapped to `NAMES` column.
- Subdocuments nested in subdocuments are mapped as columns to the parent table using the following naming convention:

  <subdocument_name>_<nested_subdocument_name>_<field_name>

  For example, for the `location` subdocument nested in the `manager` subdocument, the `city` field would map to a column name of `MANAGER_LOCATION_CITY`.

Subdocuments nested in an array

In the collection `contacts`, the `addresses` array contains the subdocuments: `label`, `street`, and `zip`.

```
{"_id": "ajx456",
 "name": "Andrea",
 "addresses": [
    {"label": "Home", "street": "101 Main St", "zip": "27513"},
    {"label": "Work", "street": "303 Main St", "zip": "27560"}
              ]
}
```

In this example, the information from the `contacts` collection is mapped to the `CONTACTS` parent table and `CONTACTS_ADDRESSES` child table. The `CONTACTS` parent table is derived from the `_id` and `name` simple types, which are mapped to the `_ID` and `NAMES` column. The resulting parent table would take the following form:

Table 20: CONTACTS

_ID	NAME
ajx456	Andrea

The information from the ADDRESSES array is mapped to the following child table:

Table 21: CONTACTS_ADDRESSSES

CONTACTS_ID (FK)	POSITION (PK)	LABEL	STREET	ZIP
ajx456	0	Home	101 Main St	27513
ajx456	1	Work	303 Main St	27560

The mapping of the parent array, addresses, to the CONTACTS_ADDRESSES table is handled as follows:

- The table name, CONTACTS_ADDRESSES, is derived from the name of the array.
- To establish a foreign key relationship with the parent table, a column is generated based on the _ID field using the following naming convention:

  ```
  <parent_table>_ID
  ```

  For example, CONTACTS_ID.
- The primary key of the child table is a composite key formed by the primary key of the parent table (CONTACTS_ID) combined with the positional information contained in the POSITION column.
- Fields in the array are mapped to columns in the child table. For example, the label field becomes the LABEL column.

An array nested in a document

In this scenario, the manager document contains the emails array in the employee collection. The collection has the following JSON structure:

```
{"_id": "ajp211",
 "name": "Mark",
 "manager": {"name": "Cynthia",
              "emails": ["cynthia@email.com", "watsonc@email.com"]}
}
{"_id": "mpc393",
 "name": "Deborah",
 "manager": {"name": "Cynthia",
              "emails": ["cynthia@email.com", "watsonc@email.com"]}
}
{"_id": "dlm215",
 "name": "Jason",
 "manager": {"name": "Chris",
              "emails": ["chris@email.com"]}
}
```

The information from the employee collection is mapped to the EMPLOYEE parent table and the EMPLOYEE_EMAILS child table.

Important: In the mixed view, if the number of values in an array are uniform across the collection and are less than or equal to twelve per element, the array values are flattened into columns in the parent table (for example, EMAILS_1, EMAILS_2, EMAILS_3). If the number of values are not uniform or exceed twelve per element, the driver maps the array to a separate child table, as demonstrated by this example.

The resulting parent table takes the following form:

Table 22: EMPLOYEE

_ID	NAME	MANAGER_NAME
ajp211	Mark	Cynthia
mpc393	Deborah	Cynthia
dlm215	Jason	Chris

Mapping for the EMPLOYEE parent table is handled as follows:

- Simple types are mapped to columns in the parent table as columns. For example, the name object is mapped to NAME column.
- Fields for the subdocument manager are mapped as columns to the parent table. Column names for fields generated from a subdocument take the following form:

  <subdocument_name>_<field_name>

  For example, for the field name in the subdocument manager, the resulting column name would be MANAGER_NAME.

The information from the emails nested array is mapped to the following child table:

Table 23: EMPLOYEE_EMAILS

EMPLOYEE_ID (FK)	POSITION (PK)	EMAILS
ajp211	0	cynthia@email.com
ajp211	1	watsonc@email.com
mpc393	0	cynthia@email.com
mpc393	1	watsonc@email.com
dlm215	0	chris@email.com

The mapping of the emails array to the EMPLOYEE_EMAILS table is handled as follows:

- The table name is now derived from the name of the array. For example, EMAILS.

- To establish a foreign key relationship with the parent table, a column is generated based on the `_ID` field using the following naming convention:

  `<parent_table>_ID`

  For example, `EMPLOYEE_ID`.

- The primary key of the child table is a composite key formed by the primary key of the parent table (`MANAGER_ID`) combined with the positional information contained in the `POSITION` column.
- Nested arrays are mapped to the column. In this example, the `emails` array is mapped to the `EMAILS` column.

An array nested in an array

In the collection `offices`, the `departments` array contains an `employees` array.

```
{"_id": "au32",
 "city": "Bedford",
 "departments": [{"name": "Development",
                  "employees": [
                                {"first": "Leslie", "last": "Jacobs"},
                                {"first": "Emma", "last": "Alves"},
                                {"first": "Brad", "last": "Jones"}
                               ]
                 },
                 {"name": "Human Resources",
                  "employees": [
                                 {"first": "Joseph", "last": "Lu"},
                                 {"first": "Margaret", "last": "Baker"},
                                 {"first": "Chetna", "last": "Campbell"}
                                ]
                 },
                 {"name": "IT",
                  "employees": [
                                 {"first": "Caroline", "last": "Evans"},
                                 {"first": "Markus", "last": "Campanella"},
                                 {"first": "Jennifer", "last": "Bradley"}
                                 ]
                 }]
}
{"_id": "xn44",
 "city": "Morrisville",
 "departments": [{"name": "Development",
                  "employees": [
                                {"first": "Charles", "last": "Scott"},
                                {"first": "Mary", "last": "Gonzales"},
                                {"first": "Phil", "last": "McEnroe"}
                               ]
                },
                {"name": "Operations",
                 "employees": [
                                {"first": "Rachel", "last": "Cullingford"},
                                {"first": "Lance", "last": "Friedman"},
                                {"first": "Amanda", "last": "Giachetti"}
                               ]
                },
                {"name": "IT",
                 "employees": [
                                {"first": "Ingrid", "last": "Burkis"},
                                {"first": "Catherine", "last": "Wheeler"},
                                {"first": "Jacob", "last": "Williams"}
                               ]
                 }]
}
```

The information from the offices collection is mapped to the OFFICES parent table and the OFFICES_DEPARTMENTS and OFFICES_EMPLOYEES child tables. The OFFICES parent table is derived from the _id and city simple types. The _id field is mapped as the primary key column _ID, and the city field is mapped as the relational column CITY. The resulting OFFICES parent table takes the following form:

Table 24: OFFICES

_ID (PK)	CITY
au32	Bedford
xn44	Morrisville

The information in the departments parent array is mapped to the following child table:

Table 25: OFFICES_DEPARTMENTS

OFFICES_ID (FK)	POSITION (PK)	NAME
au32	0	Development
au32	1	Human Resources
au32	2	IT
xn44	0	Development
xn44	1	Operations
xn44	2	IT

The mapping of the departments parent array to the OFFICES_DEPARTMENTS child table is handled as follows:

- The table name is now derived from the name of the parent table and the array. For example, OFFICES_DEPARTMENTS.
- To establish a foreign key relationship with the parent table, a column is generated based on the _ID field using the following naming convention:

  ```
  <parent_table>_ID
  ```

  For example, OFFICES_ID.
- The primary key of the child table is a composite key formed by the primary key of the parent table (OFFICES_ID) combined with the positional information contained in the POSITION column.
- The fields in the parent array are mapped to columns. For example, the name array is mapped to the NAME column.

The information in the employees nested array is mapped to the following child table:

Table 26: OFFICES_EMPLOYEES

OFFICES_ID (FK)	OFFICES_DEPARTMENTS _POSITION	POSITION (PK)	FIRST	LAST
au32	0	0	Leslie	Jacobs
au32	0	1	Emma	Alves
au32	0	2	Brad	Jones
au32	1	0	Joseph	Lu
au32	1	1	Margaret	Baker
au32	1	2	Chetna	Campbell
au32	2	0	Caroline	Evans
au32	2	1	Markus	Campanella
au32	2	2	Jennifer	Bradley
xn44	0	0	Charles	Scott
xn44	0	1	Mary	Gonzales
xn44	0	2	Phil	McEnroe
xn44	1	0	Rachel	Cullingford
xn44	1	1	Lance	Friedman
xn44	1	2	Amanda	Giachetti
xn44	2	0	Ingrid	Burkis
xn44	2	1	Catherine	Wheeler
xn44	2	2	Jacob	Williams

The mapping of the `departments` array to the `OFFICES_DEPARTMENTS` child table is handled as follows:

- The table name is derived from the name of the parent table and the array. For example, `OFFICES_EMPLOYEES`.
- The foreign key, `OFFICES_ID`, reflects the name of the parent table.
- A foreign key relationship with the parent array is established using the `<parent_array>_POSITION` column. For example, `OFFICES_DEPARTMENTS_POSITION`.
- The primary key of the child table is a composite key formed by the primary key of the parent (`OFFICES_ID`), the positional information of the parent of the array (`OFFICES_DEPARTMENTS_POSITION`), and the positional information contained in the `POSITION` column.

- The fields in the nested array are mapped to columns in the table. For example, the `first` array is mapped to the `FIRST` column.

SQL escape sequences

The driver supports the following SQL escape sequences.

- Date, Time, and Timestamp Escape Sequences
- Scalar Functions
- Outer Join Escape Sequences
- LIKE Escape Character Sequence for Wildcards

Refer to SQL Escape Sequences for JDBC in the *Progress DataDirect for JDBC Drivers Reference* for information about SQL escape sequences supported by the driver.

Supported scalar functions

The driver supports the scalar functions in the following table. Note that your database system may not support all these functions. Refer to the documentation for your database system to find out which functions are supported by your database.

In addition, you can also determine the supported scalar functions by using DatabaseMetaData methods.

You can use scalar functions in SQL statements with the following syntax:

`{fn` *`scalar-function`*`}`

where:

`scalar-function`

is a scalar function supported by the drivers, as listed in the following table.

Example:

```
SELECT id, name FROM emp WHERE name LIKE {fn UCASE('Smith')}
```

Refer to "Scalar functions" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

Table 27: Supported Scalar Functions

String Functions	Numeric Functions	Timedate Functions	System Functions
ASCII	ABS	CURDATE	CURSESSIONID
BIT_LENGTH	ACOS	CURRENT_DATE	DATABASE
CHAR	ASIN	CURRENT_TIME	IDENTITY
CHAR_LENGTH	ATAN	CURRENT_TIMESTAMP	USER
CHARACTER_LENGTH	ATAN2	CURTIME	

String Functions	Numeric Functions	Timedate Functions	System Functions
CONCAT	BITAND	DATEDIFF	
DIFFERENCE	BITOR	DATE_ADD	
HEXTORAW	BITXOR	DATE_SUB	
INSERT	CEILING	DAY	
LCASE	COS	DAYNAME	
LEFT	COT	DAYOFMONTH	
LENGTH	DEGREES	DAYOFWEEK	
LOCATE	EXP	DAYOFYEAR	
LOCATE_2	FLOOR	EXTRACT	
LOWER	LOG	HOUR	
LTRIM	LOG10	MINUTE	
OCTET_LENGTH	MOD	MONTH	
RAWTOHEX	PI	MONTHNAME	
REPEAT	POWER	NOW	
REPLACE	RADIANS	QUARTER	
RIGHT	RAND	SECOND	
RTRIM	ROUND	SECONDS_SINCE_MIDNIGHT	
SOUNDEX	ROUNDMAGIC	TIMESTAMPADD	
SPACE	SIGN	TIMESTAMPDIFF	
SUBSTR	SIN	TO_CHAR	
SUBSTRING	SQRT	WEEK	
UCASE	TAN	YEAR	
UPPER	TRUNCATE		

CAST_TO_NATIVE function escape

The MongoDB driver supports the custom function escape `CAST_TO_NATIVE`. The `CAST_TO_NATIVE` function escape can be used in a filter to select or insert a value of a specific native type.

In some cases, a value with a specific native type can be concealed or obscured because the driver describes a field with inconsistent native types as a column with a single SQL type. For example, the driver maps a MongoDB `_id` field with the native types String and `ObjectId` to a relational `_ID` column with the VARCHAR type. In this context, the `CAST_TO_NATIVE` function escape allows users to send a value as it is defined in the MongoDB database, rather than how it is described in the relational model of the data.

Currently, `CAST_TO_NATIVE` can only be used with the `ObjectID` type in SELECT statement filters and literal INSERT values.

Syntax

```
{fn CAST_TO_NATIVE('value','native_type')}
```

where:

value

is the value to be selected or inserted.

native_type

is the native type that in part defines the value.

Example Select

The following statement selects records from `Account` where the `_ID` field has an `ObjectID` value of `507f1f77bcf86cd799439011`.

```
SELECT * FROM Account WHERE _ID = {fn
CAST_TO_NATIVE('507f1f77bcf86cd799439011','objectid')}
```

Example Insert

The following statement inserts the `ObjectID` value `507f1f77bcf86cd799439011` into the `_ID` field.

```
INSERT INTO Account(_ID) VALUES ({fn
CAST_TO_NATIVE('507f1f77bcf86cd799439011','objectid')})
```

DataDirect tools

Progress DataDirect for JDBC drivers install the set of tools described in this section. For detailed instructions on using these tools, refer to the corresponding topics in the *Progress DataDirect for JDBC Drivers Reference* at the following links:

- DataDirect Test allows you to test your JDBC driver and learn the JDBC API.
- DataDirect Connection Pool Manager allows you to pool connections when accessing databases. When your applications use connection pooling, connections are reused rather than created each time a connection is requested. Because establishing a connection is among the most costly operations an application may

perform, using Connection Pool Manager to implement connection pooling can significantly improve performance.

- Statement Pool Monitor loads statements into and remove statements from the statement pool as well as generate information to help you troubleshoot statement pooling performance.
- DataDirect Spy logs detailed information about calls your driver makes that can be used for troubleshooting.

Troubleshooting

The *Progress DataDirect for JDBC Drivers Reference* provides information on troubleshooting problems should they occur. Refer to Troubleshooting for details.

Additional information

In addition to the content provided in this guide, the documentation set also contains detailed conceptual and reference information that applies to all the drivers. For more information in these topics, refer the *Progress DataDirect for JDBC Drivers Reference* or use the links below to view some common topics:

- "JDBC support" describes support for JDBC interfaces and methods for the Progress DataDirect for JDBC drivers.
- "JDBC extensions" describes the JDBC extensions provided by the com.ddtek.jdbc.extensions package.
- "SQL escape sequences for JDBC" provides an overview of SQL escape sequences for JDBC. In addition, it documents the scalar functions that you use in SQL statements.
- "Security best practices for JDBC applications" describes the security best practices you should employ when developing and deploying your application with the driver.

Contacting Technical Support

Progress DataDirect offers a variety of options to meet your support needs. Please visit our Web site for more details and for contact information:

https://www.progress.com/support

The Progress DataDirect Web site provides the latest support information through our global service network. The SupportLink program provides access to support contact details, tools, patches, and valuable information, including a list of FAQs for each product. In addition, you can search our Knowledgebase for technical bulletins and other information.

When you contact us for assistance, please provide the following information:

- Your number or the serial number that corresponds to the product for which you are seeking support, or a case number if you have been provided one for your issue. If you do not have a SupportLink contract, the SupportLink representative assisting you will connect you with our Sales team.
- Your name, phone number, email address, and organization. For a first-time call, you may be asked for full information, including location.
- The Progress DataDirect product and the version that you are using.

- The type and version of the operating system where you have installed your product.
- Any database, database version, third-party software, or other environment information required to understand the problem.
- A brief description of the problem, including, but not limited to, any error messages you have received, what steps you followed prior to the initial occurrence of the problem, any trace logs capturing the issue, and so on. Depending on the complexity of the problem, you may be asked to submit an example or reproducible application so that the issue can be re-created.
- A description of what you have attempted to resolve the issue. If you have researched your issue on Web search engines, our Knowledgebase, or have tested additional configurations, applications, or other vendor products, you will want to carefully note everything you have already attempted.
- A simple assessment of how the severity of the issue is impacting your organization.

2

Tutorials

The following sections guide you through using the driver to access your data with some common third-party applications. For information on installing your driver and setting the CLASSPATH, see "Installing and setting-up the driver."

For details, see the following topics:

- Interactive SQL
- Tableau
- DbVisualizer

Interactive SQL

After you have installed your driver, you can use the driver to access your data with the Interactive SQL tool. Interactive SQL supports a command line interface that allows you to connect to a data source, execute SQL statements and retrieve results for display on a terminal.

To execute commands with Interactive SQL:

1. Start the ISQL tool. From a command line, enter the following:

   ```
   java -jar mongodb.jar --isql
   ```

2. Enter connection properties one at a time by typing `property=value`, then pressing **Enter**. For example, to configure the ServerName property:

   ```
   ServerName=myserver
   ```

3. After specifying values for your properties, type `connect`, then press **Enter**. If successful, the tool will return a confirmation message.

   Note: If you are unable to connect, you can review the URL by entering the `SHOW URL` command.

4. At the `ISQL>` prompt, issue a SQL command to query or modify the data source; then, press **Enter**. For example:

   ```
   SELECT * FROM INFORMATION_SCHEMA.SYSTEM_TABLES;
   ```

   Note: SQL commands must be terminated by a semi-colon.

   Note: In addition to SQL commands, the tool supports a set of proprietary commands. Type `Help` at the prompt for a list of supported commands and syntax.

   The results of the command are displayed in the terminal.

5. After you are finished executing queries and commands, you can disconnect from the data source by typing the following; then, pressing **Enter**:

   ```
   DISCONNECT
   ```

6. To end the session, type `exit`; then, press **ENTER**.

Tableau

After you have installed your driver and defined it on the CLASSPATH, you can use the driver to access your data with Tableau. Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data. By using the driver with Tableau, you can improve performance when retrieving data while leveraging the driver's relational mapping tools.

To use the driver to access data with Tableau:

1. Navigate to the `\lib\xx` subdirectory of the Progress DataDirect installation directory; then, locate the `jar` file for your driver:

   ```
   mongodb.jar
   ```

2. Copy the `.jar` file for your driver into the following directory:

   Windows: `C:\Program Files\Tableau\Drivers`

   Linux: `/opt/tableau/tableau_driver/jdbc`

3. Open Tableau. From the **Connect** menu, select **Other Databases (JDBC)**.
4. In the **Other Databases (JDBC)** dialog, provide values for the following fields; then, click **Sign In**.
   - **URL**: Copy and paste your connection URL into this field. The following example demonstrates how to connect using user ID and password authentication.

     ```
     jdbc:datadirect:mongodb://myserver:27017;AuthenticationDatabase=mydb2;DatabaseName=mydb
     ```

     Note: See User ID and password authentication on page 74 for details.

   - **Dialect**: Select **SQL92** (the default) from the drop-down box.
   - **Username**: If required by the authentication method being used, enter the user name. Alternatively, this value can be specified with the `User` property in the connection string.
   - **Password**: If required by the authentication method being used, enter the password. Alternatively, this value can be specified with the `Password` property in the connection string.
5. The **Data Source** window appears. In the **Schema** field, select the schema for the service you want to use.
6. In the **Table** field, the tables stored in the selected schema are now exposed and available for selection.

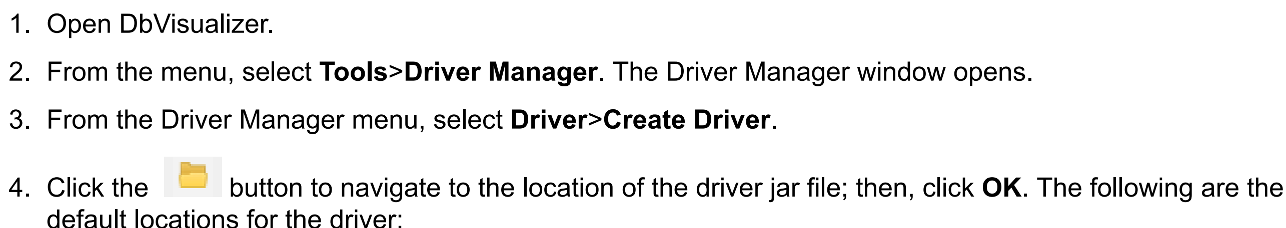
You have successfully accessed your data and are now ready to create reports with Tableau. For detailed information, refer to the Tableau product documentation at: https://www.tableau.com/support/help.

DbVisualizer

After you have installed your driver and defined it on the CLASSPATH, you can use the driver to access your data with the third-party DbVisualizer tool. The following topics guide you through using DbVisualizer to add your driver, connect, and execute SQL statements.

Adding a driver

To add a driver with DbVisualizer:

1. Open DbVisualizer.
2. From the menu, select **Tools**>**Driver Manager**. The Driver Manager window opens.
3. From the Driver Manager menu, select **Driver**>**Create Driver**.
4. Click the  button to navigate to the location of the driver jar file; then, click **OK**. The following are the default locations for the driver:

   Windows

   ```
   C:\Program Files\Progress\DataDirect\JDBC\lib\61\mongodb.jar
   ```

   Linux

   ```
   /opt/Progress/DataDirect/JDBC/lib/61/mongodb.jar
   ```

5. Provide values for the following fields; then, close the Driver Manager window.

- **Name**: Type an alias for your driver. For example:

  ```
  MongoDB
  ```

- **URL Format**: Optionally, specify the format of the connection string for your driver. For example:

  ```
  jdbc:datadirect:mongodb://myserver:27017
  ```

- **Driver Class**: From the drop down menu, select the driver class for your driver. For example:

  ```
  com.ddtek.jdbc.mongodb.MongoDBDriver
  ```

You can now use your driver with DbVisualizer. Proceed to "Connecting and executing SQL statements" for information on connecting and executing SQL statements.

See also

Connecting and executing SQL statements on page 60

Connecting and executing SQL statements

To use the driver to access data with DbVisualizer:

1. Open DbVisualizer.
2. From the menu, select **Database**>**New Connection**. When prompted to use the Connection Wizard, click **OK**.
3. Provide the following information when prompted; then, click **Next** to proceed:
   - **Connection alias**: Type the name to be used when referring to this connection.
   - **Driver**: Select the alias that you provided for your driver from the drop-down menu.
4. Provide values for the following fields; then, click **Finish**.
   - **Database URL**: Copy and paste your connection URL into this field. The following example demonstrates how to connect using user ID and password authentication.

     Note: You can also generate connection strings using MongoDB Configuration Manager. For more information, see Generating connection URLs with the Configuration Manager on page 66.

     ```
     jdbc:datadirect:mongodb://myserver:27017;AuthenticationDatabase=mydb2;
          DatabaseName=mydb;user=jsmith;password=secret;
     ```

     Note: See User ID and password authentication on page 74 for details.

5. To execute SQL statements, select **SQL Commander**>**New SQL Commander**. A SQL Commander tab opens.
6. Select values for the following fields:
   - **Database Connection**: Select connection alias you provided for the connection from the drop-down menu.

- **Schema**: Select the schema you want to execute queries against from the drop-down menu.

7. In the SQL Commander tab, enter SQL commands you want to execute; then select **SQL Commander**>**Execute**. For example:

   To select all of the rows from the `INFORMATION_SCHEMA.SYSTEM_TABLES` table:

   ```
   SELECT * FROM INFORMATION_SCHEMA.SYSTEM_TABLES
   ```

You have successfully accessed your data with DbVisualizer.

3

Configuring and connecting

This section provides information on how to connect to your data store using either the JDBC Driver Manager or DataDirect JDBC data sources, as well as information on how to implement and use functionality supported by the driver.

After the driver has been installed and defined on your classpath, you can connect from your application to your data in either of the following ways.

- Using the JDBC `DriverManager` by specifying the connection URL in the `DriverManager.getConnection()` method.
- Creating a JDBC data source that can be accessed through the Java Naming Directory Interface (JNDI).

For details, see the following topics:

- Setting the classpath
- Connecting using the JDBC Driver Manager
- Connecting using data sources
- Refreshing the relational map
- Authentication
- Data Encryption
- Proxy server
- MongoDB Atlas clusters
- Replica set failover for write operations
- Performance considerations

Setting the classpath

The driver must be defined on your CLASSPATH before you can connect. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate JDBC drivers on your computer. If the driver is not defined on your CLASSPATH, you will receive a `class not found` exception when trying to load the driver. Set your system CLASSPATH to include the driver jar file as shown, where *install_dir* is the path to your product installation directory.

```
install_dir/lib/61/mongodb.jar
```

Windows Example

```
CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC\lib\61\mongodb.jar
```

UNIX Example

```
CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/61/mongodb.jar
```

Connecting using the JDBC Driver Manager

One way to connect to a service is through the JDBC DriverManager using the `DriverManager.getConnection()` method. As the following example shows, this method specifies a string containing a connection URL.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:mongodb://myserver:27017;AuthenticationDatabase=mydb2;
        DatabaseName=mydb;user=jsmith;password=secret;");
```

Passing the connection URL

After setting the CLASSPATH, the required connection information needs to be passed in the form of a connection URL. The following example includes the properties required for connecting with user ID and password authentication.

Connection URL Syntax

The connection URL takes the following form:

```
jdbc:datadirect:mongodb://host:port;AuthenticationDatabase=auth_db;
     DatabaseName=database;User=username;Password=password;[property=value[;...]];
```

where:

host

specifies the name or the IP address of the MongoDB server to which you want to connect.

For example, `myserver`.

port

specifies the port number of the server listener. The default is `27017`.

auth_db

(recommended) specifies the database in which your user ID was created. This value allows you to explicitly select a set of credentials and permissions when the same user ID was created in multiple databases on the server.

Note: We recommend specifying a value for this option when using user ID and password authentication (`AuthenticationMethod=userIdPassword`) to ensure that the correct permissions are used for your connection.

database

(optional) specifies the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries.

Important: This value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, this value is case-sensitive.

user

(optional) specifies the user name that is used to connect to the MongoDB database. For example, `jsmith`.

password

(optional) specifies the password used to connect to your MongoDB database.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following example connection string includes the properties required for connecting with user ID and password authentication.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:mongodb://myserver:27017;AuthenticationDatabase=mydb2;
        DatabaseName=mydb;user=jsmith;password=secret;");
```

See also

Connection property descriptions on page 93
Connection URL examples on page 16

Generating connection URLs with the Configuration Manager

The driver includes a browser-based tool, Progress DataDirect MongoDB Configuration Manager, that allows you to generate connection URLs, test connections, and execute test queries. This section will guide you through generating and testing a connection URL that can be used by your application.

To generate a connection URL:

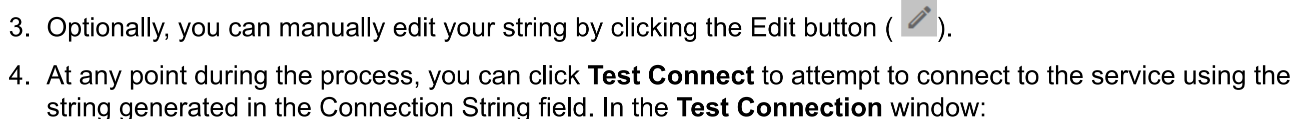
1. Open the MongoDB Configuration Manager by double-clicking the driver jar file. Or, in a command line, navigate to the directory containing your driver jar file; then, execute the following command:

   ```
   java -jar mongodb.jar
   ```

   The MongoDB Configuration Manager opens in your default web browser.

2. From the browser window, provide values of the connection properties you want to configure in the corresponding fields. A connection URL will generate in the Connection String field as you provide settings. To view more properties, select the tabs at the top of the page. See "Connection URL examples" for a list of required properties and properties used for different configurations.

   Note: If you do not specify a value for an optional property, the property will be omitted from the string and the default value will be used.

3. Optionally, you can manually edit your string by clicking the Edit button ().

4. At any point during the process, you can click **Test Connect** to attempt to connect to the service using the string generated in the Connection String field. In the **Test Connection** window:

   a) Provide values for any fields required by your service.

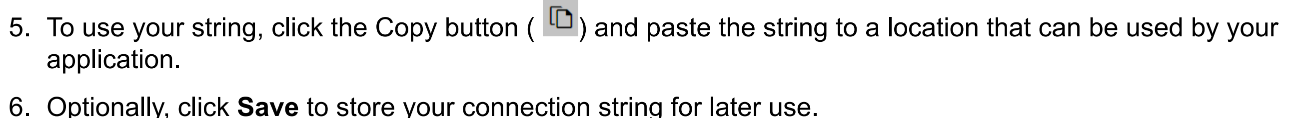
   b) Optionally, in the Test Query field, enter any SQL queries you want to execute during the test. For example:

   ```
   SELECT * FROM INFORMATION_SCHEMA.SYSTEM_TABLES
   ```

   c) Click **Execute**.

   If successful, the window displays a confirmation message and, if a query was specified, the results of the query.

   Note: The information you enter in the logon dialog box during a test connect is not saved.

5. To use your string, click the Copy button () and paste the string to a location that can be used by your application.

6. Optionally, click **Save** to store your connection string for later use.

See also

Connection URL examples on page 16

Testing connections and queries

You can quickly test a connection string and queries using Progress DataDirect MongoDB Configuration Manager.

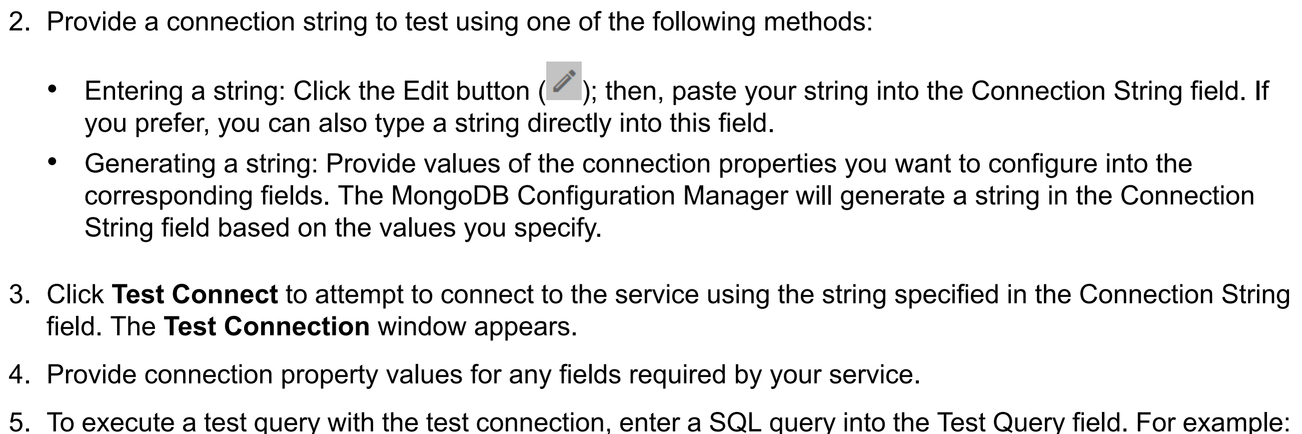
To test your connection and query:

1. Open the MongoDB Configuration Manager by double-clicking on the driver jar file. Or, in a command line, navigate to the directory containing your driver jar file; then, execute the following command:

   ```
   java -jar mongodb.jar
   ```

   The MongoDB Configuration Manager opens in your default web browser.

2. Provide a connection string to test using one of the following methods:

   - Entering a string: Click the Edit button (); then, paste your string into the Connection String field. If you prefer, you can also type a string directly into this field.
   - Generating a string: Provide values of the connection properties you want to configure into the corresponding fields. The MongoDB Configuration Manager will generate a string in the Connection String field based on the values you specify.

3. Click **Test Connect** to attempt to connect to the service using the string specified in the Connection String field. The **Test Connection** window appears.
4. Provide connection property values for any fields required by your service.
5. To execute a test query with the test connection, enter a SQL query into the Test Query field. For example:

   ```
   SELECT * FROM INFORMATION_SCHEMA.SYSTEM_TABLES
   ```

6. Click **Execute**.

   If successful, the window displays a confirmation message and, if a query was specified, the results of the query.

Connecting using data sources

A *JDBC data source* is a Java object, specifically a `DataSource` object, that defines connection information required for a JDBC driver to connect to the database. Each JDBC driver vendor provides their own data source implementation for this purpose. A Progress DataDirect data source is Progress DataDirect's implementation of a `DataSource` object that provides the connection information needed for the driver to connect to a database.

Because data sources work with the Java Naming Directory Interface (JNDI) naming service, data sources can be created and managed separately from the applications that use them. Because the connection information is defined outside of the application, the effort to reconfigure your infrastructure when a change is made is minimized. For example, if the database is moved to another database server, the administrator need only change the relevant properties of the `DataSource` object. The applications using the database do not need to change because they only refer to the name of the data source.

How data sources are implemented

Data sources are implemented through a `DataSource` class. A data source class implements the following interfaces.

- `javax.sql.DataSource`
- `javax.sql.ConnectionPoolDataSource` (allows applications to use connection pooling)

Refer to "Connection Pool Manager" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

See also

Driver and DataSource classes on page 16

Creating data sources

The following example files provide details on creating and using Progress DataDirect data sources with the Java Naming Directory Interface (JNDI), where *`install_dir`* is the product installation directory.

- *`install_dir`*`/Examples/JNDI/JNDI_LDAP_Example.java` can be used to create a JDBC data source and save it in your LDAP directory using the JNDI Provider for LDAP.
- *`install_dir`*`/Examples/JNDI/JNDI_FILESYSTEM_Example.java` can be used to create a JDBC data source and save it in your local file system using the File System JNDI Provider.

See "Example data source" for an example data source definition for the example files.

To connect using a JNDI data source, the driver needs to access a JNDI data store to persist the data source information. For a JNDI file system implementation, you must download the File System Service Provider from the Oracle Technology Network Java SE Support downloads page, unzip the files to an appropriate location, and add the `fscontext.jar` and `providerutil.jar` files to your CLASSPATH. These steps are not required for LDAP implementations because the LDAP Service Provider is included with supported versions of Java SE.

See also

Example data source on page 68

Example data source

To configure a data source using the example files, you will need to create a data source definition. The content required to create a data source definition is divided into three sections.

First, you will need to import the data source class. For example:

```
import com.ddtek.jdbcx.mongodb.MongoDBDataSource;
```

Next, you will need to set the values and define the data source. For example, the following definition contains the minimum properties required for a connection using user and password authentication.

Note:

- Setting the password using a data source is generally not recommended. The data source persists all properties, including the Password property, in clear text.
- In a JDBC data source, string values must be enclosed in double quotation marks, for example, `setUser("abc@defcorp.com")`.

```
MongoDBDataSource mds = new MongoDBSource();
mds.setDescription("My MongoDB Data Source");
mds.setAuthenticationMethod("userIdPassword");
mds.setDatabaseName("mydb");
mds.setServerName("myserver");
```

Finally, you will need to configure the example application to print out the data source attributes. Note that this code is specific to the driver and should only be used in the example application. For example, you would add the following section for the minimum properties required to establish a connection:

```
if (ds instanceof MongoDBDataSource)
{
MongoDBDataSource jmds = (MongoDBDataSource) ds;
System.out.println("Description=" + jmds.getDescription());
System.out.println("AuthenticationMethod=" + jmds.getAuthenticationMethod());
System.out.println("DatabaseName=" + jmds.getDatabaseURL());
System.out.println("ServerName=" + jmds.getServerNameURL());

...
System.out.println();
}
```

Calling a data source in an application

Applications can call a Progress DataDirect data source using a logical name to retrieve the `javax.sql.DataSource` object. This object loads the specified driver and can be used to establish a connection to the database.

Once the data source has been registered with JNDI, it can be used by your JDBC application as shown in the following code example.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("EmployeeDB");
Connection con = ds.getConnection("domino", "spark");
```

In this example, the JNDI environment is first initialized. Next, the initial naming context is used to find the logical name of the data source (`EmployeeDB`). The `Context.lookup()` method returns a reference to a Java object, which is narrowed to a `javax.sql.DataSource` object. Then, the `DataSource.getConnection()` method is called to establish a connection.

Testing a data source connection

You can use DataDirect Test™ to establish and test a data source connection. The screen shots in this section were taken on a Windows system.

Take the following steps to establish a connection.

1. Navigate to the installation directory. The default location is:

- Windows systems: `Program Files\Progress\DataDirect\JDBC\testforjdbc`
- UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC/testforjdbc`

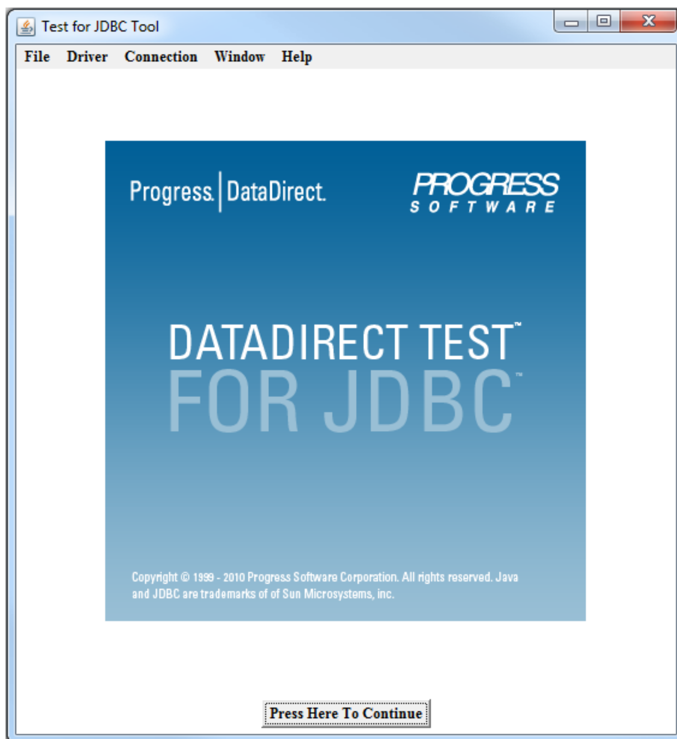
Note: For UNIX/Linux, if you do not have access to `/opt`, your home directory will be used in its place.

---

2. From the `testforjdbc` folder, run the platform-specific tool:

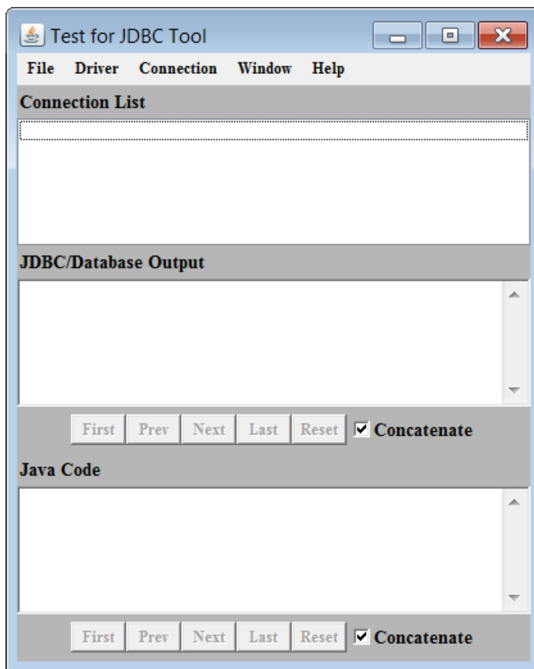
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

   The **Test for JDBC Tool** window appears:



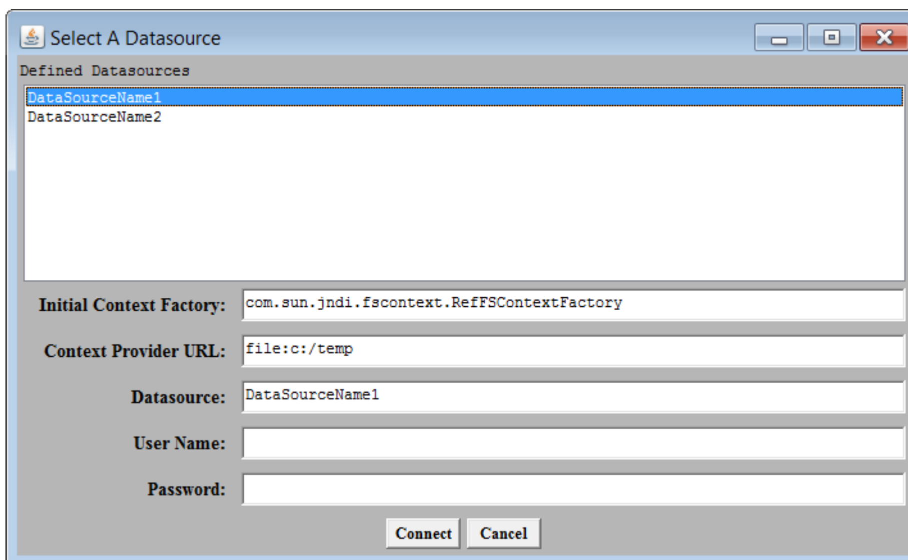
3. Click **Press Here to Continue**.

   The main dialog appears:



4. From the menu bar, select **Connection > Connect to DB via Data Source**.

   The **Select A Database** dialog appears:



5. Select a datasource template from the **Defined Datasources** field.
6. Provide the following information:

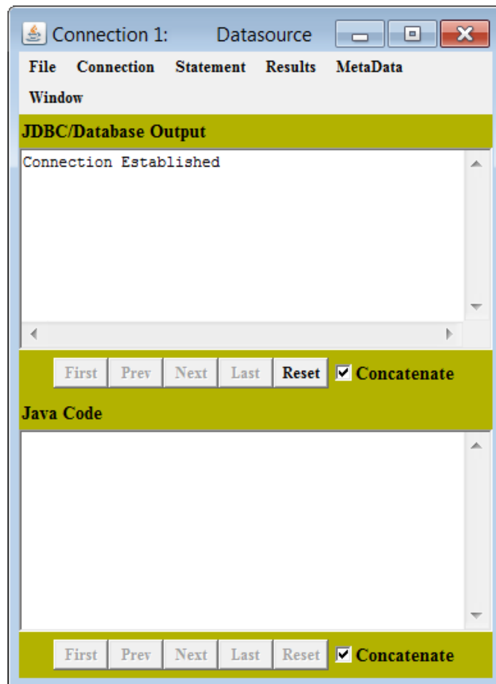
   a) In the **Initial Context Factory**, specify the location of the initial context provider for your application.

   b) In the **Context Provider URL**, specify the location of the context provider for your application.

   c) In the **Datasource** field, specify the name of your datasource.

7. If you are using user ID/password authentication, enter your user ID and password in the corresponding fields.
8. Click **Connect**.

If the connection information is entered correctly, the **JDBC/Database Output** window reports that a connection has been established. If a connection is not established, the window reports an error.



Refreshing the relational map

When new native objects are added to your database, the driver must refresh the schema map to discover and map these objects to the relational view. You can refresh the schema map using the following methods:

- **At Connection**: When the RefreshSchema connection property is set to `true`, the driver refreshes the map each time it connects to your data source.
- **Configuration Manager**: In the Configuration Manager, you can refresh your schema map by clicking the **Test Connect**, entering the `REFRESH MAP` statement in the Test Query field, then clicking **Execute**. Note that you must provide your connection information in the appropriate fields to successfully test connect. For more information on testing connections with the Configuration Manager, see "Testing connections and queries."
- **Refresh Map SQL Extension**: Your application can refresh the schema map by executing the Refresh Map statement. For details and syntax, see "REFRESH MAP (EXT)."
- **Command-line (Interactive SQL script)**: You can refresh the schema map from a command-line using the Interactive SQL tool that is embedded in your driver `.jar` file. For the Interactive SQL method, we recommend writing a simple script to invoke the tool to refresh the schema map. You can execute a script using the following command:

  ```
  java -jar mongodb.jar --isql <path>/<script_name>.isql
  ```

  See the following section for information on creating an Interactive SQL script.

Using an Interactive SQL script to refresh the relational map

An Interactive SQL script is a simple text file that uses the *file_name*.isql naming convention. The file is comprised of the connection properties used to make your connection, plus a set of required commands. For properties with sensitive values, such as passwords, you can use environment and system variables, or specify that the tool prompts the user for those values. See the following example for examples of the supported syntax.

Note: The driver uses the schema map specified by the SchemaMap property to obtain the values for connection properties stored in the file. If you don't specify a value for SchemaMap, you will need to manually specify these connection properties in the script file. Note that any values specified in the script file will override those in the SchemaMap file.

Sample Interactive SQL Script

```
# You can set connection property values one at a time, using variables,
# literal values, or prompts.

# Takes SchemaMap value from the system property,
# for example: -Dexample.refresh_map.schemamap=<path>
SCHEMAMAP=${example.refresh_map.schemamap}

# Takes the user value from the environment varaiable.
USER=${env.refresh_map.user}

# Prompts the user for ask for the password value when running the script.
PASSWORD?

# The following are required commands
CONNECT
REFRESH MAP;
EXIT
```

See also

Testing connections and queries on page 67
Refresh Map (EXT) on page 156

Authentication

The driver supports the following authentication methods:

- *User ID and password authentication* authenticates using the specified user IDs and passwords.
- *LDAP authentication* authenticates using user ID and password information centrally maintained in LDAP entries.
- *Kerberos authentication* authenticates by using Kerberos authentication protocol.

By default, the driver does not attempt to authenticate (AuthenticationMethod=UserIDPassword).

See also

AuthenticationMethod on page 104

User ID and password authentication

To configure the driver to use basic authentication:

- Set the ServerName property to the name or the IP address of the MongoDB server to which you want to connect. For example, `myserver`.
- Set the AuthenticationMethod property to `userIdPassword` (the default).
- Optionally, set the DatabaseName property to specify the name of the database to which you are connecting.
- Set the User property to specify your user ID.
- Set the Password property to specify your password.
- Optionally, set the AuthenticationDatabase property to specify the database in which your user ID was created. This value allows you to explicitly select a set of credentials and permissions when the same user ID was created in multiple databases on the server.

  Note: We recommend specifying a value for the AuthenticationDatabase to ensure the correct permissions are used for your connection. If you do not specify a value for this property, the driver attempts to use your user ID with the database specified by the Databasename property. If your user ID was not created in the specified database, the driver will attempt to connect to the Admin database using your user ID and password.

- Optionally, specify values for any additional properties you want to configure.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following examples demonstrate a session with user ID and password authentication enabled.

For a connection URL:

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:mongodb://myserver:27017;AuthenticationDatabase=mydb2;
       DatabaseName=mydb;user=jsmith;password=secret;");
```

For a data source:

```
MongoDBDataSource mds = new MongoDBDataSource();
mds.setDescription("My MongoDB Data Source");
mds.setAuthenticationDatabase("mydb2");
mds.setDatabaseName("mydb");
mds.setServerName("myserver");
```

See also

Connection property descriptions on page 93

LDAP authentication

LDAP (Lightweight Directory Access Protocol) is a directory information service that allows you to centrally store information and share it across an IP network. In an LDAP service, information is stored in objects called entries, which can contain a variety of data—including authentication information. LDAP entries are often used to store authentication information because data storage is centralized, thereby simplifying maintenance when changes occur.

To configure the driver to use LDAP authentication:

- Set the ServerName property to specify the name or IP address of the MongoDB server to which you want to connect. For example, `myserver`.
- Set the AuthenticationMethod property to `plain`.
- Optionally, set the DatabaseName property to specify the name of the database to which you are connecting.
- Set the User property to specify your user ID.
- Set the Password property to specify your password.
- Optionally, specify values for any additional properties you want to configure.

Important: When LDAP authentication is enabled, credentials are passed in clear text. Therefore, you should use LDAP authentication only on servers that are configured for TLS/SSL encryption.

Note: The User and Password properties are not required to be stored in the connection string. They can also be passed separately by the application.

The following examples demonstrate a session with LDAP authentication enabled.

For a connection URL:

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:mongodb://myserver:27017;AuthenticationMethod=plain;
       DatabaseName=mydb;user=jsmith;password=secret;");
```

For a data source:

```
MongoDBDataSource mds = new MongoDBDataSource();
mds.setDescription("My MongoDB Data Source");
mds.setAuthenticationMethod("plain");
mds.setDatabaseName("mydb");
mds.setServerName("myserver");
```

See also

Connection property descriptions on page 93

Kerberos authentication

Your Kerberos environment should be fully configured before you configure the driver for Kerberos authentication. You should refer to your MongoDB and Java documentation for instructions on configuring Kerberos. For a Windows Active Directory implementation, you should also consult your Windows documentation. For a non-Active Directory implementation (on a Windows or non-Windows operating system), you should consult MIT Kerberos documentation.

Important: A properly configured Kerberos environment must include a means of obtaining a Kerberos Ticket Granting Ticket (TGT). For a Windows Active Directory implementation, Active Directory automatically obtains the TGT. However, for a non-Active Directory implementation, the means of obtaining the TGT must be automated or handled manually.

Once your Kerberos environment has been configured, take the following steps to configure the driver.

1. Use one of the following methods to integrate the JAAS configuration file into your Kerberos environment. (See "The JAAS Login Configuration File" for details.)

   Note: The `install_dir/lib/JDBCDriverLogin.conf` file is the JAAS login configuration file installed with the driver. You can use this file or another file as your JAAS login configuration file.

   Note: Regardless of operating system, forward slashes must be used when designating the path of the JAAS login configuration file.

   - Specify a login configuration file directly in your application with the `java.security.auth.login.config` system property. For example:

     ```
     System.setProperty("java.security.auth.login.config","install_dir/lib/JDBCDriverLogin.conf");
     ```

   - Set up a default configuration. Modify the Java security properties file to indicate the URL of the login configuration file with the `login.config.url.n` property where `n` is an integer connoting separate, consecutive login configuration files. When more than one login configuration file is specified, then the files are read and concatenated into a single configuration.

   a) Open the Java security properties file. The security properties file is the `java.security` file in the `/jre/lib/security` directory of your Java installation.

   b) Find the line `# Default login configuration file` in the security properties file.

   c) Below the `# Default login configuration file` line, add the URL of the login configuration file as the value for a `login.config.url.n` property. For example:

     ```
     # Default login configuration file
     login.config.url.1=file:${user.home}/.java.login.config
     login.config.url.2=file:install_dir/lib/JDBCDriverLogin.conf
     ```

2. Ensure your JAAS login configuration file includes an entry with authentication technology that the driver can use to establish a Kerberos connection. (See "The JAAS Login Configuration File" for details.)

   Note: The JAAS login configuration file installed with the driver (`install_dir/lib/JDBCDriverLogin.conf`) includes a default entry with the name `JDBC_DRIVER_01`. This entry specifies the Kerberos authentication technology used with an Oracle JVM.

   The following examples show that the authentication technology used in a Kerberos environment depends on your JVM.

   Oracle JVM

   ```
   JDBC_DRIVER_01 {
     com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;
   };
   ```

IBM JVM

```
JDBC_DRIVER_01 {
  com.ibm.security.auth.module.Krb5LoginModule required useDefaultCcache=true;
};
```

3. Set the driver's AuthenticationMethod connection property to `kerberos`.
4. Set the ServicePrincipalName connection property if the default value built by the driver does not match the service principal name registered with the KDC.

   By default, the driver builds the ServicePrincipalName by concatenating the service name `mongodb`, the fully qualified domain name (FQDN) as specified with the ServerName property, and the default realm name as specified in the Kerberos configuration file. If this value does not match the service principal name registered with the KDC, then the value of the service principal name registered with the KDC should be specified for the ServicePrincipalName property.

   The ServicePrincipalName takes the following form.

   `Service_Name/Fully_Qualified_Domain_Name@REALM_NAME`

   See "ServicePrincipalName" for details on the composition of the service principal name.

5. Set the AuthenticationDatabase connection property if user principal names stored by MongoDB are stored in a database other than the default `$external` database. (See "AuthenticationDatabase" for details.)
6. Set the LoginConfigName connection property if the name of the JAAS login configuration file entry is different from the driver default `JDBC_DRIVER_01`. (See "The JAAS Login Configuration File" and "LoginConfigName" for details.)

   `JDBC_DRIVER_01` is the default entry name for the JAAS login configuration file (`JDBCDriverLogin.conf`) installed with the driver. When configuring your Kerberos environment, your network or system administrator may have used a different entry name. Check with your administrator to verify the correct entry name.

7. Set the User connection property as appropriate. (See "User" for details.)

   In most circumstances, there is no need to set the User connection property. By default, the driver uses the user principal name in the Kerberos Ticket Granting Ticket (TGT) as the value for the User property.

8. Set the DatabaseName connection property as appropriate. (See "DatabaseName" for details.)

   If authentication has not been enabled, client applications will have access to all databases on the server. If authentication has been enabled, a client application will only have access to the database specified by the DatabaseName property, assuming it has the required permissions. However, an application with clusterAdmin privileges will have access to all databases on the server even when authentication is enabled.

   Even when authentication has not been enabled, DatabaseName is strongly recommended because its value functions as the default qualifier for unqualified tables in SQL queries.

The following examples demonstrate the required properties for a session using Kerberos authentication. Properties not specified in the example use the default setting.

For a connection URL:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:mongodb://myserver:27017;AuthenticationMethod=kerberos;
         DatabaseName=mydb;");
```

For a data source:

```
MongoDBDataSource mds = new MongoDBDataSource();
mds.setDescription("My MongoDB Data Source");
mds.setAuthenticationMethod("kerberos");
mds.setDatabaseName("mydb");
mds.setServerName("myserver");
```

See also

AuthenticationMethod on page 104
ServicePrincipalName on page 139
LoginConfigName on page 120
User on page 148
DatabaseName on page 107

Kerberos authentication requirements

Verify that your environment meets the requirements listed in the following table before you configure the driver for Kerberos authentication.

Note: For Windows Active Directory, the domain controller must administer both the database server and the client.

Table 28: Kerberos Configuration Requirements

Component	Requirements
Database server	The database server must be running MongoDB 2.4 or higher.
Kerberos server	The Kerberos server is the machine where the user IDs for authentication are administered. The Kerberos server is also the location of the Kerberos key distribution center (KDC). Network authentication must be provided by one of the following methods. • Windows Active Directory • MIT Kerberos 1.5 or higher
Client	Java Virtual Machine (JVM) that is Java SE 8 or higher, including Oracle JDK, OpenJDK, and IBM SDK (Java) distributions.

The JAAS login configuration file

The Java Authentication and Authorization Service (JAAS) login configuration file contains one or more entries that specify authentication technologies to be used by applications. To establish Kerberos connections with the driver, the JAAS login configuration file must include an entry specifically for the driver. In addition, the login configuration file must be referenced either by setting the `java.security.auth.login.config` system property or by setting up a default configuration using the Java security properties file.

Setting up a default configuration

To set up a default configuration, you must modify the Java security properties file to indicate the URL of the login configuration file with the `login.config.url.n` property where *n* is an integer connoting separate, consecutive login configuration files. When more than one login configuration file is specified, then the files are read and concatenated into a single configuration. The following steps summarize how to modify the security properties file.

1. Open the Java security properties file. The security properties file is the `java.security` file in the `/jre/lib/security` directory of your Java installation.
2. Find the line `# Default login configuration file` in the security properties file.
3. Below the `# Default login configuration file` line, add the URL of the login configuration file as the value for a `login.config.url.n` property. For example:

```
# Default login configuration file
login.config.url.1=file:${user.home}/.java.login.config
login.config.url.2=file:install_dir/lib/JDBCDriverLogin.conf
```

JAAS login configuration file entry for the driver

You can create your own JAAS login configuration file, or you can use the `JDBCDriverLogin.conf` file installed in the `/lib` directory of the product installation directory. In either case, the login configuration file must include an entry that specifies the Kerberos authentication technology to be used by the driver.

JAAS login configuration file entries begin with an entry name followed by one or more LoginModule items. Each LoginModule item contains information that is passed to the LoginModule. A login configuration file entry takes the following form.

```
entry_name {
  login_module flag_value module_options
};
```

where:

entry_name

is the name of the login configuration file entry. The driver's LoginConfigName connection property can be used to specify the name of this entry. `JDBC_DRIVER_01` is the default entry name for the `JDBCDriverLogin.conf` file installed with the driver.

login_module

is the fully qualified class name of the authentication technology used with the driver.

flag_value

specifies whether the success of the module is `required`, `requisite`, `sufficient`, or `optional`.

module_options

specifies available options for the LoginModule. These options vary depending on the LoginModule being used.

The following examples show that the LoginModule used for a Kerberos implementation depends on your JVM.

Oracle JVM

```
JDBC_DRIVER_01 {
  com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;
};
```

IBM JVM

```
JDBC_DRIVER_01 {
  com.ibm.security.auth.module.Krb5LoginModule required useDefaultCcache=true;
};
```

Refer to Java Authentication and Authorization Service documentation for information about the JAAS login configuration file and implementing authentication technologies.

See also

Kerberos authentication on page 75
LogConfigFile on page 120

Data Encryption

TLS/SSL works by allowing the client and server to send each other encrypted data that only they can decrypt. TLS/SSL negotiates the terms of the encryption in a sequence of events known as the *handshake*. The handshake involves the following types of authentication:

- *TLS/SSL server authentication* requires the server to authenticate itself to the client.
- *TLS/SSL client authentication* is optional and requires the client to authenticate itself to the server after the server has authenticated itself to the client.

Configuring TLS/SSL Encryption

The driver supports TLS/SSL encryption for all supported MongoDB databases.

Note: Connection hangs can occur when the driver is configured for SSL and the database server does not support SSL. You may want to set a login timeout using the LoginTimeout property to avoid problems when connecting to a server that does not support SSL.

To configure SSL encryption:

Important: The driver complies with FIPS when FIPS mode is enabled with the client JVM. See "FIPS (Federal Information Processing Standard)" for more information.

- Set the ServerName property to the name or the IP address of the MongoDB server to which you want to connect. For example, `myserver`.
- Set the PortNumber property to specify the port number of the server listener. The default is `27017`.
- Set the EncryptionMethod property to `SSL`.
- (Optional) Set the CryptoProtocolVersion property to specify acceptable cryptographic protocol versions (for example, TLSv1.3) supported by your server.

- (Optional) Specify the location and password of the truststore file used for SSL server authentication. Either set the TrustStore and TrustStorePassword properties or their corresponding Java system properties (`javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword`, respectively).
- (Optional) To validate certificates sent by the database server, set the ValidateServerCertificate property to `true`.
- (Optional) Set the HostNameInCertificate property to a host name to be used to validate the certificate. The HostNameInCertificate property provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.
- (Optional) If your database server is configured for SSL client authentication, configure your keystore information:
  - Specify the location and password of the keystore file. Either set the KeyStore and KeyStorePassword properties or their corresponding Java system properties (`javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword`, respectively).
  - If any key entry in the keystore file is password-protected, set the KeyPassword property to the key password.

The following examples demonstrate the required properties for a session using TLS/SSL encryption with no authentication.

For a connection URL:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:mongodb://myserver:27017;AuthenticationMethod=None;DatabaseName=mydb;

   EncryptionMethod=SSL");
```

For a data source:

```
MongoDBDataSource mds = new MongoDBDataSource();
mds.setDescription("My MongoDB Data Source");
mds.setAuthenticationMethod ("None")
mds.setDatabaseName("mydb");
mds.setEncryptionMethod("SSL");
mds.setServerName("myserver");
```

See also

Connection property descriptions on page 93
Configuring TLS/SSL Server Authentication on page 81
Configuring TLS/SSL Client Authentication on page 82

Configuring TLS/SSL Server Authentication

When the client makes a connection request, the server presents its public certificate for the client to accept or deny. The client checks the issuer of the certificate against a list of trusted Certificate Authorities (CAs) that resides in an encrypted file on the client known as a *truststore*. Optionally, the client may check the subject (owner) of the certificate. If the certificate matches a trusted CA in the truststore (and the certificate's subject matches the value that the application expects), an encrypted connection is established between the client and server. If the certificate does not match, the connection fails and the driver throws an exception.

To check the issuer of the certificate against the contents of the truststore, the driver must be able to locate the truststore and unlock the truststore with the appropriate password. You can specify truststore information in either of the following ways:

- Specify values for the Java system properties javax.net.ssl.trustStore and javax.net.ssl.trustStorePassword. For example:

  ```
  java -Djavax.net.ssl.trustStore=C:\Certificates\MyTruststore
       -Djavax.net.ssl.trustStorePassword=MyTruststorePassword
  ```

  This method sets values for all TLS/SSL sockets created in the JVM.

- Specify values for the connection properties TrustStore and TrustStorePassword in the connection URL. For example:

  `TrustStore=C:\Certficates\MyTruststore`

  and

  `TrustStorePassword=MyTruststorePassword`

  Any values specified by the TrustStore and TrustStorePassword properties override values specified by the Java system properties. This allows you to choose which truststore file you want to use for a particular connection.

Alternatively, you can configure the drivers to trust any certificate sent by the server, even if the issuer is not a trusted CA. Allowing a driver to trust any certificate sent from the server is useful in test environments because it eliminates the need to specify truststore information on each client in the test environment. If the driver is configured to trust any certificate sent from the server, the issuer information in the certificate is ignored.

Configuring TLS/SSL Client Authentication

If the server is configured for TLS/SSL client authentication, the server asks the client to verify its identity after the server has proved its identity. Similar to TLS/SSL server authentication, the client sends a public certificate to the server to accept or deny. The client stores its public certificate in an encrypted file known as a *keystore*.

The driver must be able to locate the keystore and unlock the keystore with the appropriate keystore password. Depending on the type of keystore used, the driver also may need to unlock the keystore entry with a password to gain access to the certificate and its private key.

The drivers can use the following types of keystores:

- Java Keystore (JKS) contains a collection of certificates. Each entry is identified by an alias. The value of each entry is a certificate and the certificate’s private key. Each keystore entry can have the same password as the keystore password or a different password. If a keystore entry has a password different than the keystore password, the driver must provide this password to unlock the entry and gain access to the certificate and its private key.
- PKCS #12 keystores. To gain access to the certificate and its private key, the driver must provide the keystore password. The file extension of the keystore must be .pfx or .p12.

You can specify this information in either of the following ways:

- Specify values for the Java system properties javax.net.ssl.keyStore and javax.net.ssl.keyStorePassword. For example:

  ```
  java -Djavax.net.ssl.keyStore=C:\Certificates\MyKeystore
       -Djavax.net.ssl.keyStorePassword=MyKeystorePassword
  ```

  This method sets values for all TLS/SSL sockets created in the JVM.

Note: If the keystore specified by the javax.net.ssl.keyStore Java system property is a JKS and the keystore entry has a password different than the keystore password, the KeyPassword connection property must specify the password of the keystore entry (for example, `KeyPassword=MyKeyPassword`).

- Specify values for the connection properties KeyStore and KeyStorePassword in the connection URL. For example:

  ```
  KeyStore=C:\Certficates\MyKeyStore
  ```

  and

  ```
  KeyStorePassword=MyKeystorePassword
  ```

Note: If the keystore specified by the KeyStore connection property is a JKS and the keystore entry has a password different than the keystore password, the KeyPassword connection property must specify the password of the keystore entry (for example, `KeyPassword=MyKeyPassword`).

Any values specified by the KeyStore and KeyStorePassword properties override values specified by the Java system properties. This allows you to choose which keystore file you want to use for a particular connection.

FIPS (Federal Information Processing Standard)

The Federal Information Processing Standard (or FIPS) is a cryptography standard created by the U.S. government. FIPS specifications require certain secure algorithms, cryptographic modules, and random number generation. The driver is FIPS compliant for data encryption when FIPS is enabled for the JVM on the client machine.

The following applies when the driver is running in a FIPS environment:

- The driver complies with 140-3 and 140-2 standards.
- The driver uses PKCS #11 providers to access keystores.

The driver was tested with FIPS 140-3 enabled using Red Hat OpenJDK 21 on a Red Hat Universal Base Image 9 instance.

Proxy server

In some environments, your application may need to connect through a proxy server, for example, if your application accesses an external resource such as a Web service. At a minimum, your application needs to provide the following connection information when you invoke the JVM if the application connects through a proxy server:

- Server name or IP address of the proxy server
- Port number on which the proxy server is listening for HTTP/HTTPS requests

In addition, if authentication is required, your application may need to provide a valid user ID and password for the proxy server. Consult with your system administrator for the required information.

For example, the following command invokes the JVM while specifying a proxy server named `pserver`, a port of `8888`, and provides a user ID and password for authentication:

```
java -Dhttp.proxyHost=pserver -Dhttp.proxyPort=8888 -Dhttp.proxyUser=smith
-Dhttp.proxyPassword=secret -cp mongodb.jar com.acme.myapp.Main
```

Alternatively, you can use the ProxyHost, ProxyPort, ProxyUser, and ProxyPassword connection properties. See "Connection property descriptions" for details about these properties.

MongoDB Atlas clusters

MongoDB Atlas is a hosted solution for accessing your MongoDB data in the cloud. Because MongoDB Atlas is a clustered implementation, you configure the driver to connect through the domain, instead of directly to a server. At connection, the driver performs a DNS lookup to discover the member nodes, then connects to an available node in the cluster.

Note: In addition to MongoDB Atlas, you can connect to other clustered environments by specifying the domain name using the ServerName property. However, other settings for that data source, such as authentication, are likely to differ from those described in this topic. Note that the EnableDNSLookup property must be set to `true` (the default) to connect to a clustered environment.

To connect to a MongoDB Atlas cluster:

- Set ServerName to specify the domain name of your MongoDB Atlas cluster to which you want to connect. For example, `myaltas.mongodb.net`.
- Optionally, set DatabaseName to specify the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries.
- Set PortNumber to specify the port number of the server listener. The default is `27017`.
- Set AuthenticationMethod to `UserIDPassword` (the default).
- Set EncryptionMethod to `SSL`.
- Set User to specify the user name that is used to connect to the MongoDB database. For example, `jsmith`.
- Set Password to specify the password used to connect to your MongoDB database.
- Optionally, set the AuthenticationDatabase property to specify the database in which your user ID was created. This value allows you to explicitly select a set of credentials and permissions when the same user ID was created in multiple databases on the server.

  Note: We recommend specifying a value for the AuthenticationDatabase to ensure the correct permissions are used for your connection. If you do not specify a value for this property, the driver attempts to use your user ID with the database specified by the Databasename property. If your user ID was not created in the specified database, the driver will attempt to connect to the Admin database using your user ID and password.

The following examples include the properties required for connecting to a MongoDB Atlas cluster.

Connection URL

```
jdbc:datadirect:mongodb://myaltas.mongodb.net:27017;AuthenticationDatabase=mydb2;
     DatabaseName=mydb;EncryptionMethod=SSL;User=jsmith;Password=secret;
```

Data Source

```
MongoDBDataSource mds = new MongoDBSource();
mds.setDescription("My MongoDB Atlas Data Source");
mds.setAuthenticationDatabase("mydb2");
mds.setDatabaseName("mydb");
mds.setEncryptionMethod("SSL");
mds.setPortNumber("27017");
mds.setServerName("myaltas.mongodb.net");
```

See also

Connection property descriptions on page 93

Replica set failover for write operations

Write operations for replica sets are performed exclusively through connections to the primary member. However, sometimes the primary member can become unavailable, for example, due to hardware failure or traffic overload. If this occurs, the other members elect a secondary member to assume the role of the primary. The ability to promote the secondary member during outages ensures uninterrupted availability of write operations and access to the most current version of data.

When replicate set failover is enabled, the driver handles this behavior by attempting to connect to the primary member for each write operation. If the connection fails, the driver repeats the discovery process until it finds the new primary member or the maximum number of retries have been attempted. You can enable replica set failover for write operations by specifying the name of your replica set using the ReplicaSetName property.

Note: When a value is specified for Replica Set Name, write operations will not fail when connected to secondary member for read operations. The driver will always attempt to connect to the primary member for write operations.

To configure replica set name failover for write operations:

- Set ServerName to specify the name or the IP address of the MongoDB server to which you want to connect. For example, `mymongodbserver`.
- Set DatabaseName to specify the name of the database to which you want to connect. This value is used as the default qualifier for unqualified table names in SQL queries. Required for User/ID password authentication.
- Set PortNumber to specify the port number of the server listener. The default is `27017`.
- Set ReplicaSetName to specify the name of the replica set against which you want to execute write operations.

The following examples include the properties required for connecting with no authentication, replica set failover for write operations enabled, and replica set failover for read operations set to primary preferred .

Connection URL

```
jdbc:datadirect:mongodb://MongodbServer:27017;AuthenticationMethod=None;
     DatabaseName=Mongodb1;ReplicaSetName=MyReplicaSet
```

Data Source

```
MongoDBDataSource mds = new MongoDBSource();
mds.setDescription("My Replica Set Data Source");
mds.setAuthenticationMethod("None")
mds.setDatabaseName("Mongodb1");
mds.setPortNumber("27017");
```

```
mds.setReplicaSetName("MyReplicaSet");
mds.setServerName("MongodbServer");
```

See also

Connection property descriptions on page 93

Performance considerations

EncryptionMethod: Data encryption may adversely affect performance because of the additional overhead (mainly CPU usage) required to encrypt and decrypt data.

FetchSize: FetchSize can be used to adjust the trade-off between throughput and response time. In general, setting larger values for FetchSize will improve throughput, but can reduce response time. You should set FetchSize to suit your environment. Smaller fetch sizes can improve the initial response time of the query. Larger fetch sizes improve overall fetch times at the cost of additional memory.

MaxPooledStatements: To improve performance, the driver's own internal prepared statement pooling should be enabled when the driver does not run from within an application server or from within another application that does not provide its own prepared statement pooling. When the driver's internal prepared statement pooling is enabled, the driver caches a certain number of prepared statements created by an application. For example, if the MaxPooledStatements property is set to `20`, the driver caches the last 20 prepared statements created by the application. If the value set for this property is greater than the number of prepared statements used by the application, all prepared statements are cached.

ReadPreference: ReadPreference allows you to specify your preference for which replica set members are read when executing queries. Executing queries against primary members (read-write databases) returns the most recent version of the data, but increases the workload of the primary members and may negatively affect performance. If your application does not require the most recent version of data, consider setting this connection property to read from secondary members (read-only databases) to improve performance.

ResultMemorySize: ResultMemorySize can affect performance in two main ways. First, if the size of the result set is larger than the value specified for ResultMemorySize, the driver writes a portion of the result set to disk. Since writing to disk is an expensive operation, performance losses will be incurred. Second, when you remove any limit on the size of an intermediate result set by setting ResultMemorySize to `0`, you can realize performance gains for result sets that easily fit within the JVM's free heap space. However, the same setting can diminish performance for result sets that barely fit within the JVM's free heap space.

JVM Heap Size: JVM heap size can be used to address memory and performance concerns. By increasing the max Java heap size, you increase the amount of data the driver accumulates in memory. This can reduce the likelihood of out-of-memory errors and improve performance by ensuring that result sets fit easily within the JVM's free heap space. In addition, when a limit is imposed by setting ResultMemorySize to `-1` or *x*, increasing the max Java heap size can improve performance by reducing the need to write to disk, or removing it altogether.

SchemaFilter: SchemaFilter provides a method to limit the database and collection pairs for which the driver fetches metadata. If your application only needs to access a subset of the collections you have access to, you can significantly improve connection times by providing a value for this property.

See also

Connection property descriptions on page 93

4

Additional features and functionality

The following section describes additionally supported features and functionality that are specific to the driver.

For details, see the following topics:

- MongoDB sharding
- Identifiers
- MongoDB views
- Local views
- MinKey and MaxKey values

MongoDB sharding

MongoDB employs sharding as a horizontal scaling solution for supporting large sets of data. It is designed to provide scalable throughput and storage capacity. To accomplish this, sharding shares logical databases across multiple independent replica sets (clustered servers), or shards. By distributing data across servers, operations are delegated only to the servers that store data relevant to the task. This increases the availability of servers and CPU capacity, resulting in increased throughput. If operations exceed the available processing capacity of the clusters, additional servers can be added to the cluster, which reduces the number of operations performed by each server and can improve performance. A similar principle applies to storage capacity, where servers can be added to accommodate increased storage requirements.

Caution: Although the driver connects to a MongoDB sharded cluster transparently, it is critical that the primary key column have unique values for every row (or document) in the table. The values for the default primary key of _ID are generated by a MongoDB database; however, in a sharded cluster, these values are not guaranteed to be unique across shards unless specifically configured in the MongoDB cluster. If duplicate identifiers are mapped to a relational view, write operations can produce undesired results.

Identifiers

Identifiers are used to refer to objects exposed by the driver, such as tables and columns. The driver supports both quoted and unquoted identifiers for naming objects. The maximum length of both quoted and unquoted identifiers is 128 characters. Quoted identifiers must be enclosed in double quotation marks (""). A quoted identifier can contain any Unicode character, including the space character, and is case-sensitive. The driver recognizes the Unicode escape sequence `\uxxxx` as a Unicode character. You can specify a double quotation mark in a quoted identifier by escaping it with a double quotation mark. Unquoted identifiers must start with an ASCII alpha character and can be followed by zero or more ASCII alpha or numeric characters.

By default, the driver exposes native objects as unquoted, uppercase identifiers when creating the relational schema of your native data. Since native objects are case sensitive in MongoDB, the driver avoids naming conflicts by appending identifiers which have the same name but different cases with an underscore separator and integer (for example, `_1`). If one of the conflicting names contains only uppercase characters, that name will remain unaltered. For example, if the collections `Test`, `TEST`, and `test` are found, the driver will expose the collections as tables in the following manner:

Table 29: Name Conflict Resolution Example

Collection Name	Table Name
Test	TEST_1
TEST	TEST
test	TEST_2

The driver allows you to change default behavior related to identifiers and manage identifiers directly through the use of the connection properties described in the following sections.

UppercaseIdentifiers property

The UppercaseIdentifiers property determines whether native objects are mapped as unquoted, uppercase identifiers (the default) or quoted, mixed case identifiers that correspond directly with native object names. Therefore, as an alternative to the driver's default behavior, you can use UppercaseIdentifiers to retain the names of native objects in the relational view of your data. When UppercaseIdentifiers is set to `false`, the driver maps the names of native objects as quoted identifiers, maintaining the case of native object names in the relational view of native data. If these identifiers are called in a SQL statement, the statement must enclose the identifiers in double quotation marks and they must exactly match the case of the identifier name. For example, when `UppercaseIdentifiers=false`, you would use the following statement to query the Account table:

```
SELECT "id", "name" FROM "Account"
```

The setting for UppercaseIdentifiers also affects the use of catalog functions. When object names are passed as arguments to catalog functions, the case of the value must match the case of the name in the database. If `UppercaseIdentifiers=true` (the default) when the schema map was created, the value passed to the catalog function must be uppercase because unquoted identifiers are automatically converted to uppercase by the driver. If `UppercaseIdentifiers=false` when the schema map was created, the value passed to the catalog function must match the case of the name as it was defined. In addition, when `UppercaseIdentifiers=false`, object names in results returned from catalog functions are returned in the case that they are stored in the database.

KeywordConflictSuffix property

You can use the KeywordConflictSuffix property to avoid naming conflicts when the name of an object corresponds to the name of a SQL engine keyword. KeywordConflictSuffix specifies a string of up to five alphanumeric characters that the driver appends to any object or field name that conflicts with a SQL engine keyword. For example, if you specify `KeywordConflictSuffix=TAB`, the driver maps the Case object to CASETAB.

LeadingUnderscoreReplacement property

The LeadingUnderscoreReplacement property allows you to replace leading underscores with a string when leading underscores are used in identifiers for collections and fields. For example, MongoDB collections automatically include the _id field. By specifying `LeadingUnderscoreReplacement=XX`, the _id field becomes the XXID column in the relational view of the data. In addition, any other fields or collections with a leading underscore would be modified in the same manner.

SpecialCharBehavior property

The SpecialCharBehavior property allows you to determine how the driver handles the mapping of native identifiers containing characters that would require them to be quoted in SQL statements. This property provides a method to choose to continue using identifiers that require quotation marks or for the driver to modify affected identifier names so that quotation marks are not required. By default, the driver removes any characters that are not part of a legal, unquoted SQL identifier. In practice, this removes any characters that are not letters, digits, or underscores. For example, if the native name were `Cost of Customer Acquisition`, the mapped name would be `CostofCustomerAcquisition`. The default behavior eliminates the need to quote these identifiers, but changes the identifier's name when mapping it to the relational view.

FlattenArrayBase property

When arrays are flattened into columns, the driver appends an underscore and the ordinal position to the column name (`<array_name>_<oridinal_location>`). You can specify the starting ordinal value, either a `0` or `1`, using the FlattenArrayBase property. For example, if you specified a value of `0`, the first three names of columns of an array named vehicles would be `VEHICLES_0`, `VEHICLE_1`, `VEHICLE_0`. The default starting ordinal value is `1`.

JSONColumns property

The JSONColumns property determines whether the driver exposes complex columns as JSON values in addition to their normalized mapping. Exposing complex columns as JSON values can make certain operations more convenient, such as when tools use this data as a single object for communication. In rare instances where variations in data structure might cause this data to not be sampled, enabling this property also provides a method in which it can still be read as a JSON value. When enabled (`JSONColumns=1`), the driver will also expose complex columns values as JSON values, which can be returned if needed. When this property is disable (`JSONColumns=0`), the driver will only map complex values according to the standard normalization rules.

See also

UppercaseIdentifiers on page 147
KeywordConflictSuffix on page 117
LeadingUnderscoreReplacement on page 118
SpecialCharBehavior on page 140
FlattenArrayBase on page 111
JSONColumns on page 114

MongoDB views

MongoDB databases allow you to create read-only views from existing collections or other views. MongoDB views are queryable objects that are typically used to join collections, add computed fields to collections, or exclude sensitive information. Similar to collection discovery, the driver detects MongoDB views when sampling and mapping data from the server. Data from the view is mapped to relational views using the same method as the rest of your collections (normalized, flattened, or mixed).

Be aware of the following behaviors when using MongoDB views with the driver:

- Relational tables mapped from MongoDB views are read-only
- Nested objects in a view are normalized to child tables in accordance to the driver's relational mapping behavior (normalized, flattened, or mixed). This is the same behavior used when mapping collections to the relational view.

Local views

You can create local views with the Create View statement. A view is like a named query. The view can also refer to any combination of other views.

Note: Local views are objects in the driver’s in-memory SQL Engine, not a native MongoDB collection. For information on native MongoDB views, see MongoDB views on page 90.

See "Supported SQL statements and extensions" for details on the Create View statement and other SQL statements supported by the driver.

See also

Supported SQL statements and extensions on page 151

MinKey and MaxKey values

The driver supports the MinKey and Maxkey special values and, by default, maps them to the `VARCHAR` JDBC Data type during relational mapping. Note that MinKey and MaxKey values are static and return the same data whenever they are queried. The following are the respective values returned by the driver:

- MinKey: `{"$MinKey":1}`
- MaxKey: `{"$MaxKey":1}`

MinKey and MaxKey are special values used internally by MongoDB that do not have a counterpart in SQL. Therefore, they cannot be used in Where clauses for comparisons.

Note: The DatabaseMetaData.getTypeInfo() method does not return results for the MinKey and MaxKey data types.

5

Connection property descriptions

You can use connection properties to customize the driver for your environment. This section organizes connection properties according to functionality. You can use connection properties with either the JDBC `DriverManager` or a JDBC data source. For a `DriverManager` connection, a property is expressed as a key value pair and takes the form *property=value*. For a data source connection, a property is expressed as a JDBC method and takes the form `set`*property*`(`*value*`)`.

Note:

- In a JDBC data source, string values must be enclosed in double quotation marks, for example, `setUser("abc@defcorp.com")`.
- The data type listed for each connection property is the Java data type used for the property value in a JDBC data source.
- Connection property names are case-insensitive. For example, `Password` is the same as `password`.
- For connection properties that support string values, use the following escape sequence to specify values containing leading or trailing spaces and curly brackets: `{`*value*`}`. For example: `User={hello }` or `Password={{hello}}`.

The following tables describe the connection properties by functionality:

- Basic connection properties
- User ID and password connection properties
- Kerberos authentication properties
- LDAP authentication properties
- TLS/SSL encryption properties

- Mapping properties
- Proxy server properties
- Timeout properties
- Statement pooling properties
- Additional properties

Basic connection properties

The following table summarizes connection properties which are required to connect to a MongoDB data source.

Table 30: Basic Properties

Property	Data Source Method	Default
DatabaseName on page 107	`getDatabaseName()` `setDatabaseName(String)`	No default value
PortNumber on page 125	`getPortNumber()` `setPortNumber(Integer)`	`27107`
ServerName on page 139	`getServerName()` `setServerName(String)`	No default value

User ID and password authentication connection properties

The following table summarizes connection properties used for user ID and password authentication.

Table 31: User ID and password authentication properties

Property	Data Source Method	Default
AuthenticationDatabase on page 103	`getAuthenticationDatabase()` `setAuthenticationDatabase(String)`	No default value
AuthenticationMethod on page 104	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`UserIDPassword`
Password on page 125	`getPassword()` `setPassword(String)`	No default value
User on page 148	`getUser()` `setUser(String)`	No default value

Kerberos authentication properties

The following table summarizes connection properties used for Kerberos authentication.

Table 32: URL parameter authentication properties

Property	Data Source Method	Default
AuthenticationMethod on page 104	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`UserIDPassword`
ServicePrincipalName on page 139	`getServicePrincipalName()` `setServicePrincipalName(String)`	No default value
User on page 148	`getUser()` `setUser(String)`	No default value

LDAP authentication properties

The following table summarizes connection properties used for LDAP authentication.

Table 33: URL parameter authentication properties

Property	Data Source Method	Default
AuthenticationMethod on page 104	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`UserIDPassword`
Password on page 125	`getPassword()` `setPassword(String)`	No default value
User on page 148	`getUser()` `setUser(String)`	No default value

TLS/SSL encryption properties

The following table summarizes properties used for configuring TLS/SSL encryption.

Table 34: Data encryption properties

Property	Data Source Method	Default
CryptoProtocolVersion on page 106	`getCryptoProtocolVersion()` `setCryptoProtocolVersion(String)`	No default value
EncryptionMethod on page 109	`getEncryptionMethod()` `setEncryptionMethod(String)`	`NoEncryption`

Property	Data Source Method	Default
HostNameInCertificate on page 112	`getHostNameInCertificate()` `setHostNameInCertificate(String)`	No default value
KeyPassword on page 115	`getKeyPassword()` `setKeyPassword(String)`	No default value
Keystore on page 116	`getKeystore()` `setKeystore(String)`	No default value
KeystorePassword on page 117	`getKeystorePassword()` `setKeystorePassword(String)`	No default value
Truststore on page 146	`getTruststore()` `setTruststore(String)`	No default value
TruststorePassword on page 146	`getTruststorePassword()` `setTruststorePassword(String)`	No default value
ValidateServerCertificate on page 149	`getValidateServerCert()` `setValidateServerCert(Boolean)`	No default value

Mapping properties

The following table summarizes connection properties involved in mapping the MongoDB data model to a SQL model.

Table 35: Mapping properties

Property	Data Source Method	Default
ArrayNormalizationThreshold on page 102	`getArrayNormThreshold()` `setArrayNormThreshold(Integer)`	`12`
CreateMap on page 105	`getCreateMap()` `setCreateMap(String)`	`NotExist`
ColumnDiscoverySampleSize on page 105	`getColumnDiscoverySampleSize()` `setColumnDiscoverySampleSize(Integer)`	`1000`

Property	Data Source Method	Default
FlattenArrayBase on page 111	getFlattenArrayBase() setFlattenArrayBase(Integer)	1
JSONColumns on page 114	getJsonColumns() setJsonColumns(Boolean)	false
KeywordConflictSuffix on page 117	getKeywordConflictSuffix() setKeywordConflictSuffix(String)	No default value
LeadingUnderscoreReplacement on page 118	getLeadingUnderscoreReplacement() setLeadingUnderscoreReplacement(String)	None. When no value is specified, a leading underscore is used in identifiers.
LegacyVirtualKeys on page 119	getLegacyVirtualKeys() setLegacyVirtualKeys(Boolean)	false
QualifyNormalizedNames on page 129	getQualifyNormalizedNames() setQualifyNormalizedNames(String)	No
RefreshSchema on page 132	getRefreshSchema() setRefreshSchema(Boolean)	false
SchemaFormat on page 136	getSchemaFormat() setSchemaFormat(String)	Mixed
SchemaMap on page 137	getSchemaMap() setSchemaMap(String)	Default value depends on environment
SpecialCharBehavior on page 140	getSpecialCharBehavior() setSpecialCharBehavior(String)	Include
UppercaseIdentifiers on page 147	getUpperCaseIdentifiers() setUpperCaseIdentifiers(Boolean)	true

Proxy server properties

The following table summarizes proxy server connection properties.

Table 36: Proxy Server Properties

Property	Data Source Method	Default
ProxyHost on page 126	`getProxyHost()` `setProxyHost(String)`	No default value
ProxyPassword on page 127	`getProxyPassword()` `setProxyPassword(String)`	No default value
ProxyPort on page 127	`getProxyPort()` `setProxyPort(Integer)`	`0` which means that the default value is determined by the ServerName property. For HTTP URLs: `80` For HTTPS URLs: `443`
ProxyUser on page 128	`getProxyUser()` `setProxyUser(String)`	No default value

Timeout properties

The following table summarizes timeout connection properties.

Table 37: Timeout Properties

Property	Data Source Method	Default
LoginTimeout on page 121	`getLoginTimeout()` `setLoginTimeout(Integer)`	`0` (no timeout)

Statement pooling properties

The following table summarizes statement pooling connection properties.

Table 38: Statement Pooling Properties

Property	Data Source Method	Default
ImportStatementPool on page 113	`getImportStatementPool()` `setImportStatementPool(String)`	No default value
MaxPooledStatements on page 122	`getMaxPooledStatements()` `setMaxPooledStatements(Integer)`	`0`
RegisterStatementPoolMonitorMBean on page 132	`getRegisterStatementPoolMonitorMbean()` `setRegisterStatementPoolMonitorMbean(Boolean)`	`false`

Additional properties

The following table summarizes additional connection properties.

Table 39: Additional Properties

Property	Data Source Method	Default
EnableDNSLookup on page 108	`getEnableDnsLookup()` `setEnableDnsLookup(Boolean)`	`true`
FetchSize on page 110	`getFetchSize()` `setFetchSize(Integer)`	`100` (rows)
InitializationString on page 114	`getInitializationString()` `setInitializationString(String)`	No default value
LogConfigFile on page 120	`getLogConfigFile()` `setLogConfigFile(String)`	`ddlogging.properties`
LoginConfigName on page 120	`getLoginConfigName()` `setLoginConfigName(String)`	`JDBC_DRIVER_01`
MinVarcharSize on page 123	`public Integer getMinVarcharSize()` `public void setMinVarcharSize(Integer)`	`1`
NetworkMessageCompressors on page 124	`getNetworkMessageCompressors()` `setNetworkMessageCompressors(String)`	`none`
ReadOnly on page 130	`getReadOnly()` `setReadOnly(Boolean)`	`false`
ReadPreference on page 131	`getReadPreference()` `setReadPreference(String)`	`Primary`
ReplicaSetName on page 133	`getReplicaSetName()` `setReplicaSetName(String)`	No default value
ResultMemorySize on page 134	`getResultMemorySize()` `setResultMemorySize(Integer)`	`-1`

Property	Data Source Method	Default
SpyAttributes on page 141	`getSpyAttributes()` `setSpyAttributes(String)`	No default value
SchemaFilter on page 135	`getSchemaFilter()` `setSchemaFilter(String)`	No default value
StringTruncationMethodForWrites on page 143	`getStringTruncationMethodForWrites()` `setStringTruncationMethodForWrites(String)`	`Error`
TimestampFormat on page 144	`getTimestampFormat()` `setTimestampFormat(String)`	`bigint`
TransactionMode on page 145	`getTransactionMode()` `setTransactionMode(String)`	`NoTransactions`
VarcharThreshold on page 150	`getVarcharThreshold()` `setVarcharThreshold(Integer)`	`4000` (characters)

For details, see the following topics:

- ArrayNormalizationThreshold
- AuthenticationDatabase
- AuthenticationMethod
- ColumnDiscoverySampleSize
- CreateMap
- CryptoProtocolVersion
- DatabaseName
- EnableDNSLookup
- EncryptionMethod
- FetchSize
- FlattenArrayBase
- HostNameInCertificate
- ImportStatementPool
- InitializationString
- JSONColumns

- KeyPassword
- Keystore
- KeystorePassword
- KeywordConflictSuffix
- LeadingUnderscoreReplacement
- LegacyVirtualKeys
- LogConfigFile
- LoginConfigName
- LoginTimeout
- MaxPooledStatements
- MinVarcharSize
- NetworkMessageCompressors
- Password
- PortNumber
- ProxyHost
- ProxyPassword
- ProxyPort
- ProxyUser
- QualifyNormalizedNames
- ReadOnly
- ReadPreference
- RefreshSchema
- RegisterStatementPoolMonitorMBean
- ReplicaSetName
- ResultMemorySize
- SchemaFilter
- SchemaFormat
- SchemaMap
- ServerName
- ServicePrincipalName
- SpecialCharBehavior
- SpyAttributes
- StringTruncationMethodForWrites

- TimestampFormat
- TransactionMode
- Truststore
- TruststorePassword
- UppercaseIdentifiers
- User
- ValidateServerCertificate
- VarcharThreshold

ArrayNormalizationThreshold

Purpose

Specifies the length of arrays (in elements) at which the driver begins to normalize elements in arrays to child tables when generating a flattened relational view (`SchemaFormat=Flatten`). In the flattened relational view, elements in arrays are typically flattened into columns in the parent table. This behavior can create tables with a high-number of columns when encountering large arrays or arrays nested in arrays. To avoid the memory and performance issues associated with handling very large tables, you can limit the size of the parent table using this property.

Valid Values

`0` | *x*

where:

x

is the length of arrays (in elements) at which the driver begins to normalize elements in arrays to child tables when generating a flattened view.

Behavior

If set to `0`, all elements in arrays are flattened as columns to the parent table.

If set to *x*, arrays containing a number of elements equal to or greater than the specified value are normalized to child tables when generating a flattened view. In arrays comprising of fewer elements than the specified amount, the elements are flattened into columns in the parent table.

Notes

- The behavior of this property also applies to nested arrays. Therefore, if the length of a nested array exceeds the value specified for this property, a separate child table will be generated for the elements of the nested array.

Data Source Methods

```
public Integer getArrayNormThreshold()
```

```
public void setArrayNormThreshold(Integer)
```

Default Value

`12`

Data Type

Integer

AuthenticationDatabase

Purpose

Specifies the database in which your user ID was created. In MongoDB, you can create the same user ID in different databases with unique permissions. This property ensures that the driver authenticates with the correct version of the user ID when using user ID and password authentication (`AuthenticationMethod=userIdPassword`).

Valid Values

`string`

where:

`string`

is the name of the database in which your user ID was created.

Notes

- AuthenticationDatabase is used only for user ID and password authentication (`AuthenticationMethod=userIdPassword`). To ensure the correct permissions are used for your connection, it's recommended that you specify a value for AuthenticationDatabase when user ID and password authentication is enabled.
- If you do not specify a value for this property, the driver attempts to use your user ID with the database specified by the Databasename property. If your user ID was not created in the specified database, the driver will attempt to connect to the Admin database using your user ID and password.

Data Source Methods

```
public String getAuthenticationDatabase()
```

```
public void setAuthenticationDatabase(String)
```

Default Value

No default value

Data Type

String

AuthenticationMethod

Purpose

Determines which authentication mechanism the driver uses when establishing a connection.

Valid Values

`None` | `UserIDPassword` | `Plain` | `Kerberos`

Behavior

If set to `None`, the driver does not attempt to authenticate.

If set to `UserIdPassword`,the driver uses user ID/password authentication. The driver detects and uses the most secure method supported by the server, starting with SCRAM-SHA-256. You must also provide a value for the User and Password properties.

If set to `Plain`, the driver uses LDAP authentication. You must also provide a value for the User and Password properties.

Important: When LDAP authentication is enabled, credentials are passed in clear text. Therefore, you should use LDAP authentication only on servers that are configured for TLS/SSL encryption.

If set to `Kerberos`, the driver uses Kerberos authentication.

Notes

- When `AuthenticationMethod=userIdPassword`, it is recommended that you specify the database in which your user ID was created using the AuthenticationDatabase property. Since you can create the same user ID in different databases in MongoDB, this ensures that you are attempting to connect to the user ID with the correct permissions.
- When authentication is enabled, the client application will have access only to databases as dictated by the roles and privileges of the authenticated user.

Data Source Methods

```
public String getAuthenticationMethod()
```

```
public void setAuthenticationMethod(String)
```

Default Value

`UserIDPassword`

Data Type

String

See also

DatabaseName on page 107
Authentication on page 73

ColumnDiscoverySampleSize

Purpose

Specifies the number of rows the driver fetches per collection when sampling data to detect columns and gather column statistics. The information collected in these samples is used when creating a schema map. Larger fetch sizes return samples that are more representative of your data, but at the expense of slower performance.

Valid Values

x

where:

x

specifies the number of rows the driver fetches per collection.

Notes

- The setting for this property is written to the configuration file when generating the schema map. If you attempt a subsequent connection using the configuration file and specify a different value for this property, the driver will return an error.

Data Source Methods

`public Integer getColumnDiscoverySampleSize()`

`public void setColumnDiscoverySampleSize(Integer)`

Default Value

`1000`

Data Type

Integer

CreateMap

Purpose

Determines whether the driver creates the internal files required for a relational map of the native data when establishing a connection.

Valid Values

`No` | `ForceNew` | `NotExist`

Behavior

If set to `No`, the driver uses the pre-existing group of schema map files specified by the SchemaMap. If the files do not exist, the connection fails.

If set to `ForceNew`, the driver replaces the schema map files specified by SchemaMap with a newly generated group at the same location.

Note: The 6.1 driver uses an improved normalization schema to expose the JSON structure of MongoDB data as a relational view. The new schema map files use the 6.1 normalization format, which is not expected to be backwards compatible with SQL queries written for the 6.0 schema format. Queries will need to be revised to accommodate for the differences.

If set to `NotExist`, the driver uses the pre-existing group of schema map files specified by SchemaMap. If the files do not exist, the driver creates them.

Notes

- When this property is set to `ForceNew`, the driver creates a backup of the 6.0 schema map files specified by SchemaMap. At connection, if the driver detects a schema map file using the 6.0 format, it will rename the files using the following naming convention:

  ```
  <original_file_prefix>.<timestamp>.backup
  ```

  The driver will then create a set of 6.1 schema map files using the location and or name specified by the SchemaMap. If no file name or prefix is specified, the driver will use the default file name.

- You can refresh the collections and fields exposed by the driver in an existing relational view of your data with the SQL extension Refresh Map. Refresh Map runs a discovery against your native data and adds newly discovered collections and newly discovered fields in existing collections to the schema map. Note that this process is additive only and will not delete any pre-existing collections or fields, even if they are no longer available for sampling in the MongoDB database.

Data Source Methods

```
public String getCreateMap()
public void setCreateMap(String)
```

Default Value

`NotExist`

Data Type

String

See also

SchemaMap on page 137

CryptoProtocolVersion

Purpose

Specifies a cryptographic protocol or comma-separated list of cryptographic protocols that can be used when SSL is enabled.

Valid Values

`cryptographic_protocol` `[[,`*`cryptographic_protocol`*`]...]`

where:

cryptographic_protocol

is one of the following cryptographic protocols:

`TLSv1.3` | `TLSv1.2` | `TLSv1.1` | `TLSv1` | `SSLv3` | `SSLv2`

Note: The protocols available depend on your Java version. Most modern implementations have disabled all but TLSv1.3 and TLSv1.2.

Caution: To avoid vulnerabilities associated with older protocols, best security practices recommend using TLSv1.2 or higher.

Example

If your server supports TLSv1.3 and TLSv1.2, you can specify acceptable cryptographic protocols with the following key-value pair:

```
CryptoProtocolVersion=TLSv1.3,TLSv1.2
```

Notes

- When multiple protocols are specified, the driver uses the highest version supported by the server. If none of the specified protocols are supported by the server, the connection fails and the driver returns an error.
- The default may be set in the Java system property `https.protocols`, which is often set on the Java command line with the `-D`*`property`*`=` option. For example: `-Dhttps.protocols=TLSv1.3,TLSv1.2`

Data Source Methods

```
public String getCryptoProtocolVersion()
```

```
public void setCryptoProtocolVersion(String)
```

Default Value

No default value

Data Type

String

See also

Data Encryption on page 80

DatabaseName

Purpose

Specifies the name of the database to which you are connecting. This value is used as the default qualifier for unqualified table names in SQL queries.

Valid Values

database_name

where:

database_name

is the name of a valid database.

Important: The value is case-insensitive if you have access privileges to query the list of databases on the server. If you do not have access, the value is case-sensitive.

Notes

If authentication has not been enabled, client applications will have access to all databases on the server. If authentication has been enabled, a client application will only have access to the database specified by the DatabaseName property assuming it has the required permissions. However, an application with clusterAdmin privileges will have access to all databases on the server even when authentication is enabled.

Data Source Methods

`public String getDatabaseName()`

`public void setDatabaseName(String)`

Default Value

No default value

Data Type

String

EnableDNSLookup

Purpose

Specifies whether the driver performs a DNS lookup to discover member nodes of a cluster when connecting. If the driver discovers cluster nodes in the specified domain, the driver will attempt to connect to an available node.

Valid Values

`true` | `false`

Behavior

If set to `true`, at connection, the driver performs a DNS lookup on the host specified by the ServerName property. When a domain is specified by ServerName, the driver will attempt to discover the nodes of a cluster, then connect to an available node. If you are not connecting to a clustered environment, the driver will connect to the server specified by ServerName.

If set to `false`, the driver does not attempt to perform a DNS lookup at connection. Instead, it will connect directly to the host specified by the ServerName property. Specify this value if you are not connecting to clustered environment.

Notes

- Setting EnableDNSLookup to `true` does not prohibit you from connecting to a non-clustered environment; however, the driver will still perform the lookup at connection. To improve connection time, you can disable the lookup (`EnableDNSLookup=false`) if you are not connecting to a clustered environment.

Data Source Methods

`public Boolean getEnableDnsLookup()`

`public void setEnableDnsLookup(Boolean)`

Default Value

`true`

Data Type

Boolean

EncryptionMethod

Purpose

Determines whether data is encrypted and decrypted when transmitted over the network between the driver and database server.

Valid Values

`NoEncryption` | `SSL`

Behavior

If set to `noEncryption`, data is not encrypted or decrypted.

If set to `SSL`, data is encrypted using SSL. If the database server does not support SSL, the connection fails and the driver throws an exception.

Notes

When SSL is enabled, the following properties also apply:

- CryptoProtocolVersion
- HostNameInCertificate
- KeyPassword (for SSL client authentication)
- KeyStore (for SSL client authentication)
- KeyStorePassword (for SSL client authentication)
- TrustStore
- TrustStorePassword
- ValidateServerCertificate

Data Source Methods

```
public String getEncryptionMethod()
```

```
public void setEncryptionMethod(String)
```

Default Value

`NoEncryption`

Data Type

String

See also

CryptoProtocolVersion on page 106
HostNameInCertificate on page 112
KeyPassword on page 115
Keystore on page 116
KeystorePassword on page 117
Truststore on page 146
TruststorePassword on page 146
ValidateServerCertificate on page 149
Data Encryption on page 80
Performance considerations on page 86

FetchSize

Purpose

Specifies the maximum number of rows that the driver processes before returning data to the application when executing a Select. This value provides a suggestion to the driver as to the number of rows it should internally process before returning control to the application. The driver may fetch fewer rows to conserve memory when processing exceptionally wide rows.

Valid Values

`0` | *x*

where:

x

is a positive integer indicating the number of rows that should be processed.

Behavior

If set to `0`, the driver processes all the rows of the result before returning control to the application. When large data sets are being processed, setting FetchSize to `0` can diminish performance and increase the likelihood of out-of-memory errors.

If set to *x*, the driver limits the number of rows that may be processed for each fetch request before returning control to the application.

Notes

- To optimize throughput and conserve memory, the driver uses an internal algorithm to determine how many rows should be processed based on the width of rows in the result set. Therefore, the driver may process fewer rows than specified by FetchSize when the result set contains exceptionally wide rows. Alternatively, the driver processes the number of rows specified by FetchSize when the result set contains rows of unexceptional width.
- FetchSize can be used to adjust the trade-off between throughput and response time. Smaller fetch sizes can improve the initial response time of the query. Larger fetch sizes can improve overall response times at the cost of additional memory.
- You can use FetchSize to reduce demands on memory and decrease the likelihood of out-of-memory errors. Simply, decrease FetchSize to reduce the number of rows the driver is required to process before returning data to the application.

Data Source Methods

```
public Integer getFetchSize()
```

```
public void setFetchSize(Integer)
```

Default Value

`100`

Data Type

Integer

See also

Performance considerations on page 86

FlattenArrayBase

Purpose

Specifies the starting ordinal value appended to column names for flattened arrays. When flattening arrays, column names are appended with an underscore and the ordinal value (`<array_name>_<oridinal_location>`).

Valid Values

`0` | `1`

Behavior

If set to `0`, the first column name in the array will be appended with an _0. For example, the first three columns generated in the vehicles array would be the following in the Mixed relational view: `VEHICLES_0`, `VEHICLES_1`, `VEHICLES_2`.

If set to `1`, the first column name in the array will be appended with an _1. For example, the first three columns generated in the vehicles array would be the following in the Mixed relational view: `VEHICLES_1`, `VEHICLES_2`, `VEHICLES_3`.

Data Source Methods

`public Integer getFlattenArrayBase()`

`public void setFlattenArrayBase(Integer)`

Default Value

`1`

Data Type

Integer

HostNameInCertificate

Purpose

Specifies a host name for certificate validation when SSL encryption is enabled and validation is enabled (`ValidateServerCertificate=true`). This property is optional and provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.

Valid Values

`host_name`

where:

`host_name`

is a valid host name.

Behavior

If *`host_name`* is specified, the driver compares the specified host name to the DNSName value of the SubjectAlternativeName in the certificate. If the certificate does not have a SubjectAlternativeName, the driver compares the host name with the Common Name (CN) part of the certificate. If the values do not match, the connection fails and the driver throws an exception.

Notes

- If SSL encryption or certificate validation is not enabled, this property is ignored.
- If SSL encryption and validation is enabled and this property is unspecified, the driver uses the server name that is specified in the connection URL or data source of the connection to validate the certificate.

Data Source Methods

`public String getHostNameInCertificate()`

`public void setHostNameInCertificate(String)`

Default Value

No default value

Data Type

String

See also

ValidateServerCertificate on page 149
Data Encryption on page 80

ImportStatementPool

Purpose

Specifies the path and file name of the file to be used to load the contents of the statement pool. When this property is specified, statements are imported into the statement pool from the specified file.

Valid Values

`String`

where:

`String`

is the path and file name of the file to be used to load the contents of the statement pool.

Notes

- If the driver cannot locate the specified file when establishing the connection, the connection fails and the driver throws an exception.
- For more information, refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

```
public String getImportStatementPool()
```

```
public void setImportStatementPool(String)
```

Default Value

No default value

Data Type

String

InitializationString

Purpose

Specifies one or multiple SQL commands to be executed by the driver after it has established a connection and has performed all initialization for the connection. If the execution of a SQL command fails, the connection attempt also fails and the driver throws an exception indicating which SQL command or commands failed.

Valid Values

command[[;*command*]...]

where:

command

is a SQL command.

Notes

Multiple commands must be separated by semicolons. In addition, if this property is specified in a connection URL, the entire value must be enclosed in parentheses when multiple commands are specified.

Data Source Methods

`public String getInitializationString()`

`public void setInitializationString(String)`

Default Value

No default value

Data Type

String

JSONColumns

Purpose

Determines whether the driver exposes documents and arrays embedded within a collection as JSON formatted fields, in addition to exposing individual collection and array elements as fields, when mapping to a flattened view (`SchemaFormat=Flatten`). Exposing documents and arrays as JSON values can make certain operations more convenient, such as when tools use this data as a single object for communication. In addition, in rare instances where variations in data structure might cause this data to not be sampled, enabling this property provides a method in which the data can still be read as a JSON value.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver exposes documents and arrays embedded within a collection as JSON formatted fields. In addition, the individual collections and arrays are also exposed as fields in accordance to the flattened view.

If set to `false`, the driver exposes documents and arrays embedded within a collection according to the relational view specified by the SchemaFormat property.

Notes

- Querying JSON values can be an expensive operation that could negatively impact performance; therefore, you should only query JSON values when necessary.

Data Source Methods

`public Boolean getJsonColumns()`

`public void setJsonColumns(Boolean)`

Default Value

`false`

Data Type

Boolean

KeyPassword

Purpose

Specifies the password that is used to access the individual keys in the keystore file when SSL is enabled (`EncryptionMethond=SSL`) and SSL client authentication is enabled on the database server. This property is useful when individual keys in the keystore file have a different password than the keystore file.

Valid Values

`string`

where:

`string`

is a valid password.

Data Source Methods

`public String getKeyPassword()`

`public void setKeyPassword(String)`

Default Value

No default value

Data Type

String

See also

Data Encryption on page 80

Keystore

Purpose

Specifies the directory of the keystore file to be used when SSL is enabled and SSL client authentication is enabled on the server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request.

Valid Values

`keystore_directory`

where:

`keystore_directory`

is a valid directory of a keystore file.

Notes

- This value overrides the directory of the keystore file that is specified by the `javax.net.ssl.keyStore` Java system property. If this property is not specified, the keystore directory is specified by the `javax.net.ssl.keyStore` Java system property.
- The keystore and truststore files can be the same file.

Data Source Methods

```
public String getKeystore()
```

```
public void setKeystore(String)
```

Default Value

No default value

Data Type

String

See also

Data Encryption on page 80

KeystorePassword

Purpose

Specifies the password that is used to access the keystore file when SSL is enabled and SSL client authentication is enabled on the server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request.

Valid Values

string

where:

string

is a valid password.

Notes

- This value overrides the password of the keystore file that is specified by the `javax.net.ssl.keyStorePassword` Java system property. If this property is not specified, the keystore password is specified by the `javax.net.ssl.keyStorePassword` Java system property.
- The keystore and truststore files can be the same file; therefore, they may have the same password.

Data Source Methods

```
public String getKeystorePassword()
```

```
public void setKeystorePassword(String)
```

Default Value

No default value

Data Type

String

See also

Data Encryption on page 80

KeywordConflictSuffix

Purpose

Specifies a string of up to 5 alphanumeric characters that the driver appends to any object or field name that conflicts with a SQL engine keyword.

Valid Values

String

where:

String

is a string of up to 5 alphanumeric characters.

Example

A field called `CASE` exists in the data schema. To avoid a naming conflict with the SQL engine keyword CASE, you could set `KeywordConflictSuffix=TAB`. In this scenario, the driver maps the `CASE` field to the `CASETAB` column.

Notes

- The setting for this property is written to the configuration file when generating the schema map. If you attempt a subsequent connection using the configuration file and specify a different value for this property, the driver will return an error.

Data Source Methods

```
public String getKeywordConflictSuffix()
```

```
public void setKeywordConflictSuffix(String)
```

Default Value

No default value

Data Type

String

LeadingUnderscoreReplacement

Purpose

Specifies the string of characters that replace leading underscores used in identifiers for documents and arrays.

Valid Values

`string`

where:

`string`

is comprised of any Unicode character or group of characters, including spaces.

Example

MongoDB collections automatically include the _id field. By specifying `LeadingUnderscoreReplacement=XX`, the `_id` field becomes the `XXID` column in the relational view of the data. In addition, any other fields with a leading underscore would be modified in the same manner.

Notes

- The setting for this property is written to the configuration file when generating the schema map. If you attempt a subsequent connection using the configuration file and specify a different value for this property, the driver will return an error.

Data Source Methods

`public String getLeadingUnderscoreReplacement()`

`public void setLeadingUnderscoreReplacement(String)`

Default Value

None. When no value is specified, a leading underscore is used in identifiers.

Data Type

String

See also

Identifiers on page 88

LegacyVirtualKeys

Purpose

Specifies whether the driver generates legacy virtual keys for newly-discovered nested objects when mapping the relational view of data.

For versions earlier than 6.1, the driver used the naming convention `*object_name*_GENERATED_ID` for the unique virtual key column, which was used as a foreign key to associate the child table back to the parent table. Starting in version 6.1, the driver uses the `POSITION` column for this purpose.

By default (`LegacyVirtualKeys=false`), normalized schemas migrated from the 6.0 format retain the legacy key column for existing objects, while the driver generates `POSITION` columns for newly-discovered nested objects. To use a consistent naming convention for virtual key columns in migrated schemas, set this property to `true`.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver generates legacy virtual keys and `POSITION` columns when mapping newly-discovered nested objects for both new and migrated schemas. Virtual keys are populated in the *object_name*`_GENERATED_ID` column.

If set to `false`, the driver uses the `POSITION` column when mapping newly-discovered objects for both new and migrated schemas. Note that migrated schemas will continue to use the *object_name*`_GENERATED_ID` column for existing objects, but newly-discovered objects will have only the `POSTION` column.

Data Source Methods

`public Boolean getLegacyVirtualKeys()`

```
public void setLegacyVirtualKeys(Boolean)
```

Default Value

`false`

Data Type

Boolean

LogConfigFile

Purpose

Specifies the file name, and optionally, the path of the properties file used to initialize driver logging.

Valid Values

`String`

where:

`String`

is the relative or fully qualified path of the properties file to load to initialize driver logging. If you do not specify a path, the driver looks for this file in the current working directory. If the specified file does not exist, the driver continues searching for an appropriate properties file as described in "Using Java Logging" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

```
public String getLogConfigFile()
```

```
public void setLogConfigFile(String)
```

Default Value

`ddlogging.properties`

Data Type

String

LoginConfigName

Purpose

Specifies the name of the entry in the JAAS login configuration file that contains the authentication technology used by the driver to establish a Kerberos connection. The LoginModule-specific items found in the entry are passed on to the LoginModule.

Valid Values

entry_name

where:

entry_name

is the name of the entry that contains the authentication techonology used with the driver.

Example

In the following example, `JDBC_DRIVER_01` is the entry name while the authentication technology and related settings are found in the brackets.

```
JDBC_DRIVER_01{
  com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;
  };
```

Data Source Methods

`public String getLoginConfigName()`

`public void setLoginConfigName(String)`

Default Value

`JDBC_DRIVER_01`

Data Type

String

See also

Kerberos authentication on page 75

LoginTimeout

Purpose

The amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request.

Valid Values

`-1` | `0` | *x*

where:

x

is a positive integer that specifies a number of seconds.

Behavior

If set to `-1` | `0`, the driver does not time out a connection request.

If set to *x*, the driver waits for the specified number of seconds before returning control to the application and throwing a timeout exception.

Data Source Methods

```
public Integer getLoginTimeout()
```

```
public void setLoginTimeout(Integer)
```

Default Value

`0`

Data Type

Integer

MaxPooledStatements

Purpose

Specifies the maximum number of prepared statements to be pooled for each connection and enables the driver's internal prepared statement pooling when set to an integer greater than zero (0). The driver's internal prepared statement pooling provides performance benefits when the driver is not running from within an application server or another application that provides its own statement pooling.

Valid Values

`0` | *x*

where:

x

is a positive integer that represents a number of prepared statements to be cached.

Behavior

If set to `0`, the driver's internal prepared statement pooling is not enabled.

If set to *x*, the driver's internal prepared statement pooling is enabled and the driver uses the specified value to cache up to that many prepared statements created by an application. If the value set for this property is greater than the number of prepared statements that are used by the application, all prepared statements that are created by the application are cached. Because CallableStatement is a sub-class of PreparedStatement, CallableStatements also are cached.

Example

If the value of this property is set to `20`, the driver caches the last 20 prepared statements that are created by the application.

Notes

When you enable statement pooling, your applications can access the Statement Pool Monitor directly with DataDirect-specific methods. However, you can also enable the Statement Pool Monitor as a JMX MBean. To enable the Statement Pool Monitor as an MBean, statement pooling must be enabled with MaxPooledStatements and the Statement Pool Monitor MBean must be registered using the RegisterStatementPoolMonitorMBean connection property.

Data Source Methods

```
public Integer getMaxPooledStatements()
```

```
public void setMaxPooledStatements(Integer)
```

Default Value

`0`

Data Type

Integer

See also

RegisterStatementPoolMonitorMBean on page 132
Performance considerations on page 86

MinVarcharSize

Purpose

Specifies the minimum default length, in characters, of fields that are mapped as VARCHAR. The default length of VARCHAR columns is 1.5 times the largest data value the driver samples from the column. When the default length is less than the value specified for this property, the driver increases the default length to the value set for this property. Setting this property to a larger value than the calculated default allows for larger data values to be inserted by the driver.

Valid Values

x

where:

x

is the maximum size of the default length in characters given to columns that are mapped as VARCHAR.

Example

For example, MinVarcharSize is set to `1000`, but the driver calculates a default length to be 500 based on the values sampled in your MongoDB STRING data column. Since the calculated value would be less than the setting of MinVarcharSize, the length is set to 1000. Conversely, in another column, the driver samples larger values and calculates a default length of 3000. Because the calculated value exceeds that of the MinVarcharSize property, the length would be set to 3000.

Notes

- The StringTruncationMethodForWrites property determines the behavior of the driver when inserting String values that exceed the column length defined in the relational schema.

Data Source Methods

```
public Integer getMinVarcharSize()
```

```
public void setMinVarcharSize(Integer)
```

Default Value

`1`

Data Type

Integer

NetworkMessageCompressors

Purpose

Specifies whether the driver attempts to use data compression for all messages passed between the client and server. Data compression can significantly reduce network traffic, which, in turn, can lower data transfer costs for cloud services.

Valid Values

`none` | `zlib`

Behavior

If set to `none`, the driver passes messages between the client and server without attempting to use compression.

If set to `zlib`, messages passed between the client and the server are compressed using the zlib compression algorithm. If the server is not configured to use this type of compression for network messages, the driver falls back to passing messages without using compression.

Data Source Methods

```
public String getNetworkMessageCompressors()
```

```
public void setNetworkMessageCompressors(String)
```

Default Value

`none`

Data Type

String

Password

Purpose

A password that is used to connect to the server.

Important: Setting the password using a data source is not recommended. The data source persists all properties, including password, in clear text.

Behavior

password

where:

password

is a valid password. The password is case-sensitive.

Notes

- When `AuthenticationMethod=kerberos`, the driver ignores the Password property.

Data Source Methods

```
public String getPassword()
```

```
public void setPassword(String)
```

Default Value

No default value

Data Type

String

See also

Authentication on page 73

PortNumber

Purpose

Specifies the port number of the server listener.

Valid Values

port_number

where:

port_number

is the port number of the server listener. Check with your database administrator for the correct number.

Data Source Methods

`public Integer getPortNumber()`

`public void setPortNumber(Integer)`

Default Value

`27017`

Data Type

Integer

ProxyHost

Purpose

Identifies a proxy server to use for the first connection.

Valid Values

server_name | *IP_address*

where:

server_name

is the name of the proxy server, which may be qualified with the domain name.

IP_address

is an IP address, specified in either IPv4 or IPv6 format, or a combination of the two.

Data Source Methods

`public String getProxyHost()`

`public void setProxyHost(String)`

Default Value

No default value

Data Type

String

See also

Proxy server on page 26
ProxyPassword on page 127
ProxyPort on page 127
ProxyUser on page 128

ProxyPassword

Purpose

Specifies the password needed to connect to a proxy server for the first connection.

Valid Values

password

where:

password

is a valid password for that server. Contact your system administrator to obtain a valid password.

Data Source Methods

```
public String getProxyPassword()
```

```
public void setProxyPassword(String)
```

Default Value

No default value

Data Type

String

See also

Proxy server on page 26
ProxyPort on page 127
ProxyUser on page 128
ProxyHost on page 126

ProxyPort

Purpose

Specifies the port number where the proxy server is listening for HTTP or HTTPS requests for the first connection.

Valid Values

port

where:

port

is the port number on which the proxy server is listening. Contact your system administrator to obtain the correct port.

Data Source Methods

```
public Integer getProxyPort()
```

```
public void setProxyPort(Integer)
```

Default Value

`0` which means that the default value is determined by whether the value specified for the ProxyHost property is an HTTP or HTTPS URL.

For HTTP: `80`

For HTTPS: `443`

Data Type

Integer

See also

Proxy server on page 26
ProxyUser on page 128
ProxyHost on page 126
ProxyPassword on page 127

ProxyUser

Purpose

Specifies the user name needed to connect to a proxy server for the first connection.

Valid Values

`user_name`

where:

`user_name`

is a valid user ID for the proxy server.

Data Source Methods

```
public String getProxyUser()
```

```
public void setProxyUser(String)
```

Default Value

No default value

Data Type

String

See also

Proxy server on page 26

ProxyHost on page 126
ProxyPassword on page 127
ProxyPort on page 127

QualifyNormalizedNames

Purpose

Determines whether the names of relational child-tables normalized from arrays, objects, and subdocuments are prefixed with the collection name and any parent objects.

Valid Values

No | Table | FullPath

Behavior

If set to No, the relational table name is derived solely from the column name of the array, object, or subdocument.

If set to Table, the relational table name is prepended with the name of the collection. For example, if the name of the collection was named Books and the array column was named Chapters, then the relational table name would be BOOKS_CHAPTERS.

If set to FullPath, the relational table name is prepended with the names of all objects in which the array, object, or subdocument is nested. For example, if the collection was BOOKS with an array Chapters that contained an array Pages, then the resulting relational table name would be BOOKS_CHAPTERS_PAGES.

Notes

- If a naming conflict occurs, the driver appends an underscore separator and integer (for example, _1) to the table name.
- The value of this property also controls the name of the foreign key column in the child table. For example, if this property is set to Table, the foreign key column name would be _ID prepended with the parent object name. Therefore, a parent object of BOOKS would result in a foreign key column named BOOKS_ID in the child table.

Data Source Methods

```
public String getQualifyNormalizedNames()
```

```
public void setQualifyNormalizedNames(String)
```

Default Value

Table

Data Type

String

ReadOnly

Purpose

Specifies whether the connection has read-only access to the data source.

Valid Values

`true` | `false`

Behavior

If set to `true`, the connection has read-only access. The following commands are the only commands that you can use when a connection is read-only:

- Explain Plan
- Select (except Select Into)
- Set Schema

The driver returns an error if any other command is used.

If set to `false`, the connection is opened for read/write access, and you can use all commands supported by the product.

Caution:Before disabling the ReadOnly connection property, it is critical to confirm that all values in the primary key column are unique. Executing write operations against data with duplicate primary keys can produce unpredictable and undesirable results.

Notes

You can use use the JDBC connection method setReadOnly to set a read-only state for a connection.

Data Source Methods

```
public Boolean getReadOnly()
```

```
public void setReadOnly(Boolean)
```

Default Value

`true`

Data Type

Boolean

ReadPreference

Purpose

Specifies a preference for the type of member (server node) of a replica set to which the driver attempts to connect. Connections to the primary member (read-write server nodes) return the most recent version of the data when executing Select queries, but increase the workload of the primary member and may negatively affect performance. To reduce the demand on the primary member, secondary members (read only server nodes) can be used at the expense of reading stale data.

Valid Values

`None` | `Primary` | `PrimaryPreferred` | `Secondary` | `SecondaryPreferred`

Behavior

If set to `none`, the driver attempts to connect to only the member authorized by the application.

If set to `primary`, the driver attempts to connect to only the primary member of a replica set. If the primary member is unavailable, the connection fails.

If set to `primaryPreferred`, the driver attempts to connect to the primary member first; but if it is unavailable, the driver attempts to connect to secondary members.

If set to `secondary`, the driver attempts to connect to only secondary members of a replica set. If the secondary members of the replica set are unavailable, the connection fails.

If set to `secondaryPreferred`, the driver attempts to connect to secondary members first; but if they are unavailable, the driver attempts to connect to the primary member.

Notes

- When connected to secondary members (read-only) of a replica set, the driver will return an error when attempting to execute write operations. You can work around this limitation by providing a value for the ReplicaSetName property to enable failover for write operations.
- If the ReadPreference property is configured to connect to a secondary member, the replica set that the driver is connecting to must contain a secondary member. If no secondary member exists, the driver will return an error.
- This property only affects the member to which the driver connects and only during the connection process. For example, if you specified a value of secondary, the driver will not attempt to identify a new secondary member if the secondary member it is connected to is promoted to the primary member.

Data Source Methods

```
public String getReadPreference()

public void setReadPreference(String)
```

Default Value

`Primary`

Data Type

String

See also

ReplicaSetName on page 133
Performance considerations on page 86

RefreshSchema

Purpose

Specifies whether the driver adds newly discovered objects to the relational map when connecting.

Valid Values

`true | false`

Behavior

If set to `true`, the driver adds newly discovered objects to the relational view of your data. At connection, the driver compares the relational map to a new sample of the data from the server. Any new objects that are detected are mapped to the relational view.

If set to `false`, the driver does not refresh the relational map when connecting to the server.

Notes

This property is equivalent to executing the Refresh Map statement.

Data Source Methods

```
public Boolean getRefreshSchema()
```

```
public void setRefreshSchema(Boolean)
```

Default Value

`false`

Data Type

Boolean

See also

CreateMap on page 105

RegisterStatementPoolMonitorMBean

Purpose

Registers the Statement Pool Monitor as a JMX MBean when statement pooling has been enabled with MaxPooledStatements. This allows you to manage statement pooling with standard JMX API calls and to use JMX-compliant tools, such as JConsole.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver registers an MBean for the statement pool monitor for each statement pool. This gives applications access to the Statement Pool Monitor through JMX when statement pooling is enabled.

If set to `false`, the driver does not register an MBean for the Statement Pool Monitor for any statement pool.

Notes

- Registering the MBean exports a reference to the Statement Pool Monitor. The exported reference can prevent garbage collection on connections if the connections are not properly closed. When garbage collection does not take place on these connections, out of memory errors can occur.
- For more information, refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

`public Boolean getRegisterStatementPoolMonitorMbean()`

`public void setRegisterStatementPoolMonitorMbean(Boolean)`

Default Value

`false`

Data Type

Boolean

ReplicaSetName

Purpose

Specifies the name of the replica set against which the driver executes write operations. Providing a value for this property enables replica set failover for write operations.

Valid Values

`replica_set_name`

where:

`replica_set_name`

is the name of the replica set against which the driver performs write operations.

Notes

- When a value is specified for this option, the driver attempts to establish a connection to the primary node when a write operation is executed. If the primary node is unavailable, the driver repeats the discovery process until it finds the newly elected one or until the maximum number of retry attempts is met.

Data Source Methods

```
public String getReplicaSetName()
```

```
public void setReplicaSetName(String)
```

Default Value

No default value

Data Type

String

ResultMemorySize

Purpose

Specifies the maximum size, in megabytes, of an intermediate result set that the driver holds in memory. When this threshold is reached, the driver writes a portion of the result set to disk in temporary files.

Valid Values

`-1` | `0` | *x*

where:

x

is the maximum size, in MB, of a intermediate result set that the driver holds in memory.

Behavior

If set to `-1`, the maximum size of an intermediate result that the driver holds in memory is a percentage of the max Java heap size. When this threshold is reached, the driver writes a portion of the result set to disk.

If set to `0`, the driver holds the entire intermediate result set in memory regardless of size. No portion of the result set is written to disk. Setting ResultMemorySize to `0` can improve performance for result sets that easily fit within the JVM's free heap space, but can diminish performance for result sets that barely fit within the JVM's free heap space.

If set to *x*, the driver holds intermediate results in memory that are no longer than the size specified. When this threshold is reached, the driver writes a portion of the result set to disk.

Notes

- By default, ResultMemorySize is set to `-1`. When set to `-1`, the maximum size of an intermediate result that the driver holds in memory is a percentage of the max Java heap size. When processing large sets of data, out-of-memory errors can occur when the size of the result set exceeds the available memory allocated to the JVM. In this scenario, you can tune ResultMemorySize to suit your environment. To begin, set ResultMemorySize equal to the max Java heap size divided by 4. Proceed by decreasing the value until out-of-memory errors are eliminated. As a result, the maximum size of an intermediate result set the driver holds in memory will be reduced, and some portion of the result set will be written to disk. Be away that while writing to disk reduces the risk of out-of-memory errors, it also negatively impacts performance. For optimal performance, decrease this value only to a size necessary to avoid errors.
- You can adjust the max Java heap size to address memory and performance concerns. By increasing the max Java heap size, you increase the amount of data the driver accumulates in memory. This can reduce

the likelihood of out-of-memory errors and improve performance by ensuring that result sets fit easily within the JVM's free heap space. In addition, when a limit is imposed by setting ResultMemorySize to `-1`, increasing the max Java heap size can improve performance by reducing the need to write to disk, or removing it altogether.

- The FetchSize connection property can also be used to reduce demands on memory and decreasing the likelihood of out-of-memory errors.

Data Source Methods

`public Integer getResultMemorySize()`

`public void setResultMemorySize(Integer)`

Default Value

`-1`

Data Type

Integer

See also

FetchSize on page 110
Performance considerations on page 86

SchemaFilter

Purpose

Specifies a comma-separated list of database and collection pairs for which you want the driver to fetch metadata. This property can significantly improve connection times by limiting the collections for which metadata is fetched to only those that are required by your application. If you do not specify a value, the driver fetches metadata for all the collections in every database that your account has access to, which can adversely impact performance during the initial connection to a database.

Valid Values

database_name:*collection_name*[[,*database_name*:`*collection_name*]...]

where:

database_name

is a literal value or regular expression for the database that contains collections for which the driver fetches metadata.

collection_name

is a literal value or regular expression of the collection for which the driver fetches metadata.

A schema or table name value can be:

- A literal name that does not contain either a comma (,) or a colon (:)
- A literal name that contains a comma (,) or a colon (:) that is bounded by a slash (/) at the beginning and end. For example, a collection named `sales:2019` would be represented by `/sales:2019/`

- A regular expression bounded by a slash (/) at the beginning and end, such as `/sales.*\d/`
- An asterisk (*), which represents all databases or collections within the corresponding schema(s).

Example

Literal values: The following example returns metadata for only the `november` and `march` collections in the `oem_sales` database.

```
SchemaFilter=oem_sales:november,oem_sales:march
```

Wildcard values: If you want the driver to fetch metadata for all the tables in a schema, replace the value for the table or schema name with an * (wildcard) character. For example, the following returns metadata for all the tables in the oem_sales and for tables named customers in all databases.

```
SchemaFilter=oem_sales:*,*:customers
```

Partial wildcard values: You can also use the asterisk to specify partial values for databases and collections. For example, the following returns metadata for all tables that end with region that are in schemas that begin with sales.

```
SchemaFilter=/sales.*/:/.*region/
```

Regular expressions: The following returns metadata for tables named tax in all databases that start with year which end with a number.

```
SchemaFilter=*/year.*\d/:tax
```

Notes

- The setting for this property is written to the configuration file when generating the schema map. If you attempt a subsequent connection using the configuration file and specify a different value for this property, the driver will return an error.

Data Source Methods

```
public String getSchemaFilter()
```

```
public void setSchemaFilter(String)
```

Default Value

No default value

Data Type

String

See also

Performance considerations on page 86

SchemaFormat

Purpose

Specifies to which model of the relational view the driver maps data.

Valid Values

`NormalizeAll` | `Flatten` | `Mixed`

Behavior

If set to `NormalizeAll`, the driver generates a normalized relational view of your data, mapping subdocuments and arrays as child tables with foreign key relationships to parent tables.

If set to `Flatten`, the driver generates a flattened relational view of your data, mapping subdocuments and arrays as columns within a single, comprehensive relational table for each collection.

If set to `Mixed`, the driver generates a normalized view of data that flattens certain objects into columns in the parent table. For example, subdocuments are mapped as columns in the parent table, while arrays and nested subdocuments are mapped to separate child tables. Refer to the "Mapping objects to tables" for details on how the mixed schema format normalizes or flattens subdocuments and arrays.

Notes

- The setting for this property is written to the configuration file when generating the schema map. If you attempt a subsequent connection using the configuration file and specify a different value for this property, the driver will return an error.

Data Source Methods

`public String getSchemaFormat()`

`public void setSchemaFormat(String)`

Default Value

`Mixed`

Data Type

String

See also

Mapping objects to tables on page 34

SchemaMap

Purpose

Specifies the fully qualified path of the configuration file where the relational map of native data is written. The driver looks for this file when connecting to a MongoDB server. The driver generates the files based on the setting of the CreateMap property.

Valid Values

`string`

where:

`string`

is the absolute path and filename of the configuration file, including the `.config` extension. For example, if specifying a value of:

```
C:\\Users\\Default\\AppData\\Local\\Progress\\DataDirect\\MongoDB_Schema\\MyServer.config
```

the driver either creates or looks for the configuration file Myserver.config in the following directory:

```
C:\Users\Default\AppData\Local\Progress\DataDirect\MongoDB_Schema\
```

Notes

- When connecting to a server, the driver looks for the schema map configuration file. If the configuration file does not exist, the driver creates a schema map using the name and location you have provided. If you do not provide a name and location for the schema map, the driver creates a schema map using default values.
- The driver uses the path specified in this connection property to store additional internal files.
- The SchemaDefinition property is an alias for the SchemaMap property.

Data Source Methods

```
public String getSchemaMap()
```

```
public void setSchemaMap(String)
```

Default Value

The default is determined by the environment. The driver attempts to create the files in a subdirectory of the first available directory in the following order:

- Windows
  - DD_HOME environment variable
  - dd.home system property
  - LOCALAPPDATA environment variable
  - APPDATA environment variable
  - user.home system property

For Windows, the file path takes the following format:

```
<available_location>\Progress\DataDirect\MongoDB_Schema\<user_name>.config
```

- UNIX/Linux
  - DD_HOME environment variable
  - dd.home system property
  - user.home system property

For UNIX/Linux, the file path takes the following format:

```
<available_location>/progress/datadirect/MongoDB_schema/<user_name>.config
```

Data Type

String

ServerName

Purpose

Specifies either the IP address in IPv4 or IPv6 format, the server name (if your network supports named servers) of the primary database server, or the domain name of a clustered environment.

Valid Values

string

where:

string

is a valid IP address or server name.

Data Source Methods

`public String getServerName()`

`public void setServerName(String)`

Default Value

None

Data Type

String

ServicePrincipalName

Purpose

Specifies the three-part service principal name registered with the key distribution center (KDC) in a Kerberos configuration.

Valid Values

Service_Name/Fully_Qualified_Domain_Name@REALM_NAME

where:

Service_Name

is the name of the service hosting the instance. The default value is `mongodb`.

Fully_Qualified_Domain_Name

is the fully qualified domain name (FQDN) of the host machine. By default, the driver uses the value specified by the `ServerName` property. This value must match the FQDN registered with the KDC. The FQDN consists of a host name and a domain name. For the example `myserver.test.com`, `myserver` is the host name and `test.com` is the domain name.

REALM_NAME

is the name of the Kerberos realm. By default, the driver uses the default realm specified in the Kerberos configuration file. This part of the value must be specified in upper-case characters, for example, `EXAMPLE.COM`. For Windows Active Directory, the Kerberos realm name is the Windows domain name.

Example

The following is an example of a valid service principal name.

```
mongodb/myserver.test.com@EXAMPLE.COM
```

Notes

- By default, the driver builds the `ServicePrincipalName` by concatenating the service name `mongodb`, the FQDN as specified with the `ServerName` property, and the default realm name as specified in the Kerberos configuration file. If this value does not match the service principal name registered with the KDC, then the value of the service principal name registered with the KDC should be specified for the `ServicePrincipalName` property.
- In a Kerberos configuration, an IP address cannot be used as a FQDN.
- If `AuthenticationMethod` is set to `UserIdPassword`, the value of the `ServicePrincipalName` property is ignored.

Data Source Methods

`public String getServicePrincipalName()`

`public void setServicePrincipalName(String)`

Default Value

No default value

Data Type

String

See also

ServerName on page 139

SpecialCharBehavior

Purpose

Determines how the driver handles the mapping of native identifiers containing characters that would require them to be quoted in SQL statements.

Valid Values

`Strip` | `Include` | `Replace`

Behavior

If set to `Strip`, the driver removes any characters that are not part of a legal, unquoted SQL identifier. In practice, this removes any characters that are not letters, digits, or underscores. For example, if the native name were `Cost of Customer Acquisition`, the mapped name would be `CostofCustomerAcquisition`.

If set to `Replace`, the driver replaces any characters that would cause the identifier to be quoted with underscores. For example, if the native name was `Yearly Cost Percentage`, the mapped name would be `Yearly_Cost_Percentage`.

If set to `Include`, the driver does not modify native identifiers; therefore, identifiers containing characters that are not letters, digits, or underscores would need to be quoted. For example:

```
SELECT "Long & (Very) Unusual Field/Name" FROM "Oddly-Named+Table"
```

Data Source Methods

```
public String getSpecialCharBehavior()
```

```
public void setSpecialCharBehavior(String)
```

Default Value

`Include`

Data Type

String

SpyAttributes

Purpose

Enables DataDirect Spy to log detailed information about calls that are issued by the driver on behalf of the application. DataDirect Spy is not enabled by default.

Valid Values

(*spy_attribute*[;*spy_attribute*]...)

where:

spy_attribute

is any valid DataDirect spy attribute.

Behavior

Attribute	Description
`linelimit=`*numberofchars*	Sets the maximum number of characters that DataDirect Spy logs on a single line. The default is 0 (no maximum limit).

Attribute	Description
log=(file)*filename*	Directs logging to the file specified by *filename*. For Windows, if coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes.
log=(filePrefix)*file_prefix*	Directs logging to a file prefixed by *file_prefix*. The log file is named *file_prefixX*.log where: *X* is an integer that increments by 1 for each connection on which the prefix is specified. For example, if the attribute log=(filePrefix)C:\\temp\\spy_ is specified on multiple connections, the following logs are created: C:\temp\spy_1.log C:\temp\spy_2.log C:\temp\spy_3.log ... If coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: log=(filePrefix)C:\\temp\\spy_;logIS=yes;logTName=yes.
log=System.out	Directs logging to the Java output standard, System.out.
logIS={yes | no | nosingleread}	Specifies whether DataDirect Spy logs activity on InputStream and Reader objects. When logIS=nosingleread, logging on InputStream and Reader objects is active; however, logging of the single-byte read InputStream.read or single-character Reader.read is suppressed to prevent generating large log files that contain single-byte or single character read messages. The default is no.
logLobs={yes | no}	Specifies whether DataDirect Spy logs activity on BLOB and CLOB objects.

Attribute	Description
`logTName={yes|no}`	Specifies whether DataDirect Spy logs the name of the current thread. The default is `no`.
`timestamp={yes|no}`	Specifies whether a timestamp is included on each line of the DataDirect Spy log. The default is `no`.

Example

The following value instructs the driver to log all JDBC activity to a file using a maximum of 80 characters for each line.

```
(log=(file)/tmp/spy.log;linelimit=80)
```

Notes

- If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(file)C:\\temp\\spy.log`.
- If a log file name does not include the `.log` extension, the driver automatically appends it. For example, a file named `spy.jsp` is renamed to `spy.jsp.log` by the driver.
- For more information, refer to "Tracking JDBC calls with DataDirect Spy" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

```
public String getSpyAttributes()
```

```
public void setSpyAttributes(String)
```

Default Value

No default value

Data Type

String

StringTruncationMethodForWrites

Purpose

Determines the behavior of the driver when attempting to insert String values that exceed the column length defined in the relational schema.

Valid Values

`Keep` | `Error`

Behavior

If set to `keep`, the driver inserts String values that exceed the column length defined in the relational schema without truncating the value.

If set to `error` the driver returns an error when attempting to insert a String value that exceeds the column length defined in the relational schema.

Notes

- For STRING columns, the driver determines the column size to be 50% greater than the longest sampled value in the column. Setting this property to `keep` allows for inserts with values larger than those identified through sampling to succeed.

Data Source Methods

```
public String getStringTruncationMethodForWrites()
```

```
public void setStringTruncationMethodForWrites(String)
```

Default Value

`Error`

Data Type

String

TimestampFormat

Purpose

Specifies the format in which the driver renders the MongoDB composite timestamp. This property allows you to specify the format of the MongoDB composite timestamp that is most appropriate for your use case.

Valid Values

`hex` | `bigint` | `bson` | `json` | `text`

Behavior

If set to `hex`, the driver renders the bytes comprising the timestamp value as a string of hexadecimal digits, echoing the storage format. For example, `000000001407AC1A`.

If set to `bigint`, the driver renders the timestamp as a 64-bit integer. This setting is optimal if timestamps are used for ordering the result set in chronological order. For example, `1921918923461099520`.

If set to, `bson`, the driver renders the value in the format used for serializing as BSON, as is used in the MongoDB shell. The format is the expression:

```
Timestamp(<t>, <i>)
```

where `<t>` is the number of seconds since the Unix epoch, and `<i>` is the sequence number. For example, `Timestamp(447481620, 0)`.

If set to json, the driver renders the value in the format used for serializing as JSON. The format is:

{"t":<t>, "i":<i>}

where <t> is the number of seconds since the Unix epoch, and <y> is the sequence number. For example, {"t":447481620, "i":0}.

If set to text, the driver renders the timestamp as a pair of values. The first is the date and time portion. The second is the sequence number. For example, 1984-03-07T04:27:00, 0.

Data Source Methods

public String getTimestampFormat()

public void setTimestampFormat(String)

Default Value

bigint

Data Type

String

TransactionMode

Purpose

Specifies how the driver handles manual transactions.

Valid Values

NoTransactions | Ignore

Behavior

If set to NoTransactions, the data source and the driver do not support transactions. Metadata indicates that the driver does not support transactions.

If set to Ignore, the data source does not support transactions and the driver always operates in auto-commit mode. Calls to set the driver to manual commit mode and to commit transactions are ignored. Calls to rollback a transaction cause the driver to throw an exception indicating that no transaction is started. Metadata indicates that the driver supports transactions and the ReadUncommitted transaction isolation level.

Data Source Methods

public String getTransactionMode()

public void setTransactionMode(String)

Default Value

NoTransactions

Data Type

String

Truststore

Purpose

Specifies the directory of the truststore file to be used when SSL is enabled and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

Valid Values

string

where:

string

is the directory of the truststore file.

Notes

- This value overrides the directory of the truststore file that is specified by the javax.net.ssl.trustStore Java system property. If this property is not specified, the truststore directory is specified by the javax.net.ssl.trustStore java system property.
- This property is ignored if ValidServerCertificate=`false`.

Data Source Methods

```
public String getTruststore()
public void setTruststore(String)
```

Default Value

No default value

Data Type

String

See also

ValidateServerCertificate on page 149
Data Encryption on page 80

TruststorePassword

Valid Values

string

where:

string

is a valid password for the truststore file.

Behavior

Specifies the password that is used to access the truststore file when SSL is enabled and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

Notes

- This value overrides the directory of the truststore file that is specified by the javax.net.ssl.trustStore Java system property. If this property is not specified, the truststore directory is specified by the javax.net.ssl.trustStore java system property.
- This property is ignored if ValidServerCertificate=`false`.

Data Source Methods

```
public String getTruststorePassword()
```

```
public void setTruststorePassword(String)
```

Default Value

No default value

Data Type

String

See also

ValidateServerCertificate on page 149
Data Encryption on page 80

UppercaseIdentifiers

Purpose

Specifies whether the driver maps all identifier names to uppercase. By default, the driver maps all identifier names to uppercase.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver maps identifiers to uppercase.

If set to `false`, the driver maps identifiers to the mixed case name of the object being mapped. If mixed case identifiers are used, SQL statements must enclose those identifiers in double quotes and the case of the identifier must exactly match the case of the identifier name. In addition, object names in results returned from catalog functions are returned in the case that they are stored in the database.

Example

If `UppercaseIdentifiers=false`, to query the Account table you specify:

```
SELECT "id", "name" FROM "Account"
```

Notes

- Do not change the value of Uppercase Identifiers unless the data source you are connecting to has objects with names that differ only by case.
- The setting for this property is written to the configuration file when generating the schema map. If you attempt a subsequent connection using the configuration file and specify a different value for this property, the driver will return an error.

Data Source Methods

`public Boolean getUpperCaseIdentifiers()`

`public void setUpperCaseIdentifiers(Boolean)`

Default Value

`true`

Data Type

Boolean

See also

Identifiers on page 88

User

Purpose

Specifies the user ID for user ID/password authentication or the principal name for Kerberos authentication.

Valid Values

`userid` | *`user_principal_name`*

where:

`userid`

is a valid user ID with permissions to access the database using user ID/password authentication.

`user_principal_name`

is a valid Kerberos user principal name with permissions to access the database using Kerberos authentication.

Notes

When `AuthenticationMethod=kerberos`, the User property does not have to be specified. The driver uses the user principal name in the Kerberos Ticket Granting Ticket (TGT) as the value for the User property. Any value specified must be a valid Kerberos user principal name used in the Kerberos authentication protocol.

Data Source Methods

`public String getUser()`

`public void setUser(String)`

Default Value

No default value

Data Type

String

See also

Authentication on page 73

ValidateServerCertificate

Purpose

Determines whether the driver validates the certificate that is sent by the MongoDB server when SSL encryption is enabled. When using SSL server authentication, any certificate that is sent by the server must be issued by a trusted Certificate Authority (CA).

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver validates the certificate that is sent by the server. Any certificate from the server must be issued by a trusted CA in the truststore file. If the HostNameInCertificate property is specified, the driver also validates the certificate using a host name. The HostNameInCertificate property is optional and provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.

If set to `false`, the driver does not validate the certificate that is sent by the server. The driver ignores any truststore information that is specified by the TrustStore and TrustStorePassword properties or Java system properties.

Notes

- Truststore information is specified using the TrustStore and TrustStorePassword properties or by using Java system properties.
- Allowing the driver to trust any certificate that is returned from the server even if the issuer is not a trusted CA is useful in test environments because it eliminates the need to specify truststore information on each client in the test environment.

Data Source Methods

```
public Boolean getValidateServerCert()
```

```
public void setValidateServerCert(Boolean)
```

Default Value

`true`

Data Type

Boolean

See also

Data Encryption on page 80

VarcharThreshold

Purpose

Specifies the threshold at which the driver describes character columns as type LONGVARCHAR. This property allows you to fetch columns that would otherwise exceed the upper limit of the VARCHAR type for some third-party applications, such as SQL Server Linked Server.

Valid Values

x

where:

x

is the maximum size in characters of columns the driver will describe as VARCHAR.

Notes

- For STRING columns, the driver determines the column size to be 50% greater than the longest sampled value in the column. This behavior allows for values larger than those identified through sampling. If the adjusted column size exceeds the value specified for this property, the driver describes the column as LONGVARCHAR when calling SQLDescribeCol and SQLColumns.

Data Source Methods

```
public Integer getVarcharThreshold()
```

```
public void setVarcharThreshold(Integer)
```

Default Value

`4000`

Data Type

Integer

6

Supported SQL statements and extensions

The MongoDB driver provides support for the SQL statements and the SQL extensions described in this chapter. SQL extensions are denoted by an (EXT) in the topic title.

For details, see the following topics:

- Create View
- Delete
- Drop View
- Insert
- Refresh Map (EXT)
- Reload Map (EXT)
- Select
- Update
- SQL Expressions
- Subqueries
- Custom function escapes

Create View

Purpose

The Create View statement creates a new local view. A view is analogous to a named query. The view's query can refer to any combination of remote and local tables as well as other views. Views are read-only; they cannot be updated.

Note: This statement does not affect native MongoDB views.

Syntax

```
CREATE VIEW view_name[(view_column,...)] AS
SELECT ... FROM ... [WHERE Expression]
  [ORDER BY order_expression [, ...]]
  [LIMIT limit [OFFSET offset]];
```

where:

view_name

specifies the name of the view.

view_column

specifies the column associated with the view. Multiple column names must be separated by commas.

The other commands used for Create View are the same as those used for Select (see "Select").

Notes

- A view can be thought of as a virtual table. A Select statement is stored in the database; however, the data accessible through a view is not stored in the database. The result set of the Select statement forms the virtual table returned by the view. You can use this virtual table by referring to the view name in SQL statements the same way you refer to a table. A view is used to perform any or all of these functions:
  - Restrict a user to specific rows in a table.
  - Restrict a user to specific columns.
  - Join columns from multiple tables so that they function like a single table.
  - Aggregate information instead of supplying details. For example, the sum of a column, or the maximum or minimum value from a column can be presented.
- Views are created by defining the Select statement that retrieves the data to be presented by the view.
- The Select statement in a View definition must return columns with distinct names. If the names of two columns in the Select statement are the same, use a column alias to distinguish between them. Alternatively, you can define a list of new columns for a view.

Example A

This example creates a view named myOpportunities that selects data from three database tables to present a virtual table of data.

```
CREATE VIEW myOpportunities AS
SELECT a.name AS AccountName,
       o.name AS OpportunityName,
       o.amount AS Amount,
       o.description AS Description
FROM Opportunity o INNER JOIN Account a
       ON o.AccountId = a.id
       INNER JOIN User u
       ON o.OwnerId = u.id
WHERE u.name = 'MyName'
       AND o.isClosed = 'false'
ORDER BY Amount desc
```

You can then refer to the myOpportunities view in statements just as you would refer to a table. For example:

```
SELECT * FROM myOpportunities;
```

Example B

The myOpportunities view contains a detailed description for each opportunity, which may not be needed when only a summary is required. A view can be built that selects only specific myOpportunities columns as shown in the following example:

```
CREATE VIEW myOpps_NoDesc as
SELECT AccountName,
       OpportunityName,
       Amount
FROM myOpportunities
```

The view selects the name column from both the opportunity and account tables. These columns are assigned the alias OpportunityName and AccountName, respectively.

See also

Select on page 157
Local views on page 90

Delete

Purpose

The Delete statement is used to delete rows from a table.

Syntax

`DELETE FROM` *`table_name`* `[WHERE` *`search_condition`*`]`

where:

table_name

specifies the name of the table from which you want to delete rows.

search_condition

is an expression that identifies which rows to delete from the table.

Notes

- The Where clause determines which rows are to be deleted. Without a Where clause, all rows of the table are deleted, but the table is left intact. See "Where Clause" for information about the syntax of Where clauses. Where clauses can contain subqueries.

Example A

This example shows a Delete statement on the emp table.

```
DELETE FROM emp WHERE emp_id = 'E10001'
```

Each Delete statement removes every record that meets the conditions in the Where clause. In this case, every record having the employee ID `E10001` is deleted. Because employee IDs are unique in the employee table, at most, one record is deleted.

Example B

This example shows using a subquery in a Delete clause.

```
DELETE FROM emp WHERE dept_id = (SELECT dept_id FROM dept WHERE dept_name =
'Marketing')
```

The records of all employees who belong to the department named Marketing are deleted.

Notes

- Insert, Update, and Delete are supported for MongoDB for simple types at the root level.
- To enable Insert, Update, and Delete, set the ReadOnly connection option to `false`.

See also

Where Clause on page 163

Drop View

Purpose

The Drop View statement drops a local view.

Note: This statement does not affect native MongoDB views.

Syntax

```
DROP VIEW view_name [IF EXISTS] [RESTRICT | CASCADE]
```

where:

view_name

specifies the name of a view.

`IF EXISTS`

specifies that an error is not to be returned if the view does not exist.

`RESTRICT`

is in effect by default, meaning that the drop fails if any other view refers to this view.

`CASCADE`

silently drops all dependent local views.

See also

Local views on page 90

Insert

Purpose

The Insert statement is used to add new rows to a local table. You can specify either of the following options:

- List of values to be inserted as a new row
- Select statement that copies data from another table to be inserted as a set of new rows

Syntax

```
INSERT INTO table_name [(column_name[,column_name]...)] {VALUES (expression
[,expression]...) | select_statement}
```

table_name

is the name of the table in which you want to insert rows.

column_name

is optional and specifies an existing column. Multiple column names (a column list) must be separated by commas. A column list provides the name and order of the columns, the values of which are specified in the Values clause. If you omit a *column_name* or a column list, the value expressions must provide values for all columns defined in the table and must be in the same order that the columns are defined for the table. Table columns that do not appear in the column list are populated with the default value, or with NULL if no default value is specified.

expression

is the list of expressions that provides the values for the columns of the new record. Typically, the expressions are constant values for the columns. Character string values must be enclosed in single quotation marks ('). See "Literals" for more information.

select_statement

is a query that returns values for each *column_name* value specified in the column list. Using a Select statement instead of a list of value expressions lets you select a set of rows from one table and insert it into another table using a single Insert statement. The Select statement is evaluated

before any values are inserted. This query cannot be made on the table into which values are inserted. See "Select" for information about Select statements.

Notes

- Insert, Update, and Delete are supported for MongoDB for simple types at the root level.
- To enable Insert, Update, and Delete, set the ReadOnly connection option to `false`.

See also

Literals on page 170
Select on page 157

Refresh Map (EXT)

Purpose

The REFRESH MAP statement adds newly discovered objects to your relational view of native data. It also incorporates any configuration changes made to your relational view by reloading the schema map and associated files.

Syntax

```
REFRESH MAP [NEW]
```

where

`NEW`

limits the sampling performed by the driver to only those collections that are newly discovered. When executing REFRESH MAP, this can offer significant performance gains if you only want to sample new collections or the existing collections are unchanged. If NEW is not specified in the statement, all collections are resampled.

Notes

- Newly discovered objects are mapped using the same relational view specified by the SchemaFormat property during your initial connection.
- REFRESH MAP is an expensive query since it involves the discovery of native data. However, by specifying NEW in the statement, you can reduce the overhead associated with this query by sampling only new collections.

See also

SchemaFormat on page 136

Reload Map (EXT)

Purpose

The RELOAD MAP statement reloads the schema map and associated files. This statement allows you to update your relational view of native data while the driver is connected to the data store.

Syntax

```
RELOAD MAP
```

Notes

- RELOAD MAP does not discover changes made to the native data store.

Select

Purpose

Use the Select statement to fetch results from one or more tables.

Syntax

```
SELECT select_clause from_clause
[where_clause]
[groupby_clause]
[having_clause]
[{UNION [ALL | DISTINCT] |
 {MINUS [DISTINCT] | EXCEPT [DISTINCT]} |
 INTERSECT [DISTINCT]} select_statement]
[limit_clause]
```

where:

select_clause

specifies the columns from which results are to be returned by the query. See "Select Clause" for a complete explanation.

from_clause

specifies one or more tables on which the other clauses in the query operate. See "From Clause" for a complete explanation.

where_clause

is optional and restricts the results that are returned by the query. See "Where Clause" for a complete explanation.

groupby_clause

is optional and allows query results to be aggregated in terms of groups. See "Group By Clause" for a complete explanation.

having_clause

is optional and specifies conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). See "Having Clause" for a complete explanation.

UNION

is an optional operator that combines the results of the left and right Select statements into a single result. See "Union Operator" for a complete explanation.

INTERSECT

is an optional operator that returns a single result by keeping any distinct values from the results of the left and right Select statements. See "Intersect Operator" for a complete explanation.

EXCEPT | MINUS

are synonymous optional operators that returns a single result by taking the results of the left Select statement and removing the results of the right Select statement. See "Except and Minus Operators" for a complete explanation.

orderby_clause

is optional and sorts the results that are returned by the query. See "Order By Clause" for a complete explanation.

limit_clause

is optional and places an upper bound on the number of rows returned in the result. See "Limit Clause" for a complete explanation.

See also

Select Clause on page 159
From Clause on page 161
Where Clause on page 163
Group By Clause on page 163
Having Clause on page 164
Union Operator on page 165
Intersect Operator on page 166
Except and Minus Operators on page 166
Order By Clause on page 167
Limit Clause on page 168

Select Clause

Purpose

Use the Select clause to specify with a list of column expressions that identify columns of values that you want to retrieve or an asterisk (*) to retrieve the value of all columns.

Syntax

```
SELECT [{LIMIT offsetnumber | TOP number}] [ALL | DISTINCT] {* | column_expression
[[AS] column_alias] [,column_expression [[AS] column_alias], ...]}
```

where:

`LIMIT` *`offset number`*

creates the result set for the Select statement first and then discards the first number of rows specified by *`offset`* and returns the number of remaining rows specified by *`number`*. To not discard any of the rows, specify `0` for *`offset`*, for example, `LIMIT 0` *`number`*. To discard the first *`offset`* number of rows and return all the remaining rows, specify `0` for *`number`*, for example, `LIMIT` *`offset`*`0`.

`TOP` *`number`*

is equivalent to `LIMIT 0`*`number`*.

`column_expression`

can be simply a column name (for example, `last_name`). More complex expressions may include mathematical operations or string manipulation (for example, `salary * 1.05`). See "SQL Expressions" for details. *`column_expression`* can also include aggregate functions. See "Aggregate Functions" for details.

`column_alias`

can be used to give the column a descriptive name. For example, to assign the alias department to the column dep:

```
SELECT dep AS department FROM emp
```

`DISTINCT`

eliminates duplicate rows from the result of a query. This operator can precede the first column expression. For example:

```
SELECT DISTINCT dep FROM emp
```

Notes

- Separate multiple column expressions with commas (for example, `SELECT last_name, first_name, hire_date`).
- Column names can be prefixed with the table name or table alias. For example, `SELECT emp.last_name or e.last_name`, where `e` is the alias for the table `emp`.
- NULL values are not treated as distinct from each other. The default behavior is that all result rows be returned, which can be made explicit with the keyword `ALL`.

See also

SQL Expressions on page 169
Aggregate Functions on page 160

Aggregate Functions

Aggregate functions can also be a part of a Select clause. Aggregate functions return a single value from a set of rows. An aggregate can be used with a column name (for example, `AVG(salary)`) or in combination with a more complex column expression (for example, `AVG(salary * 1.07)`). The column expression can be preceded by the `DISTINCT` operator. The `DISTINCT` operator eliminates duplicate values from an aggregate expression.

The following table lists supported aggregate functions.

Table 40: Aggregate Functions

Aggregate	Returns
AVG	The average of the values in a numeric column expression. For example, `AVG(salary)` returns the average of all salary column values.
COUNT	The number of values in any field expression. For example, `COUNT(name)` returns the number of name values. When using `COUNT` with a field name, `COUNT` returns the number of non-NULL column values. A special example is `COUNT(*)`, which returns the number of rows in the set, including rows with NULL values.
MAX	The maximum value in any column expression. For example, `MAX(salary)` returns the maximum salary column value.
MIN	The minimum value in any column expression. For example, `MIN(salary)` returns the minimum salary column value.
SUM	The total of the values in a numeric column expression. For example, `SUM(salary)` returns the sum of all salary column values.

Example A

In the following example, only distinct last name values are counted. The default behavior is that all duplicate values be returned, which can be made explicit with `ALL`.

```
COUNT (DISTINCT last_name)
```

Example B

The following example uses the `COUNT`, `MAX`, and `AVG` aggregate functions:

```
SELECT
     COUNT(amount) AS numOpportunities,
     MAX(amount) AS maxAmount,
     AVG(amount) AS avgAmount
FROM opportunity o INNER JOIN user u
     ON o.ownerId = u.id
WHERE o.isClosed = 'false' AND
     u.name = 'MyName'
```

From Clause

Purpose

The From clause indicates the tables to be used in the Select statement.

Syntax

```
FROM table_name [table_alias] [,...]
```

where:

table_name

is the name of a table or a subquery. Multiple tables define an implicit inner join among those tables. Multiple table names must be separated by a comma. For example:

```
SELECT * FROM emp, dep
```

Subqueries can be used instead of table names. Subqueries must be enclosed in parentheses. See "Subquery in a From Clause" for an example.

table_alias

is a name used to refer to a table in the rest of the Select statement. When you specify an alias for a table, you can prefix all column names of that table with the table alias.

Example

The following example specifies two table aliases, `e` for `emp` and `d` for `dep`:

```
SELECT e.name, d.deptName
FROM emp e, dep d
WHERE e.deptId = d.id
```

table_alias is a name used to refer to a table in the rest of the Select statement. When you specify an alias for a table, you can prefix all column names of that table with the table alias. For example, given the table specification:

```
FROM emp E
```

you may refer to the last_name field as E.last_name. Table aliases must be used if the Select statement joins a table to itself. For example:

```
SELECT * FROM emp E, emp F WHERE E.mgr_id = F.emp_id
```

The equal sign (=) includes only matching rows in the results.

See also

Subquery in a From Clause on page 162

Outer Join Escape Sequences

Purpose

The SQL-92 left, right, and full outer join syntax is supported.

Syntax

```
{oj outer-join}
```

where *outer-join* is

```
table-reference {LEFT | RIGHT | FULL} OUTER JOIN {table-reference | outer-join} ON
search-condition
```

where *table-reference* is a database table name, and *search-condition* is the join condition you want to use for the tables.

```
Example: SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status FROM {oj
 Customers LEFT OUTER JOIN Orders ON Customers.CustID=Orders.CustID} WHERE
Orders.Status='OPEN'
```

The following outer join escape sequences are supported by MongoDB databases:

- Left outer joins
- Right outer joins
- Full outer joins
- Nested outer joins

Join in a From Clause

Purpose

You can use a Join as a way to associate multiple tables within a Select statement. Joins may be either explicit or implicit. For example, the following is the example from the previous section restated as an explicit inner join:

```
SELECT * FROM emp INNER JOIN dep ON id=empId
SELECT e.name, d.deptName
FROM emp e INNER JOIN dep d ON e.deptId = d.id;
```

whereas the following is the same statement as an implicit inner join:

```
SELECT * FROM emp, dep WHERE emp.deptID=dep.id
```

Syntax

```
FROM table_name {RIGHT OUTER | INNER | LEFT OUTER | CROSS | FULL OUTER} JOIN table.key
 ON search-condition
```

Example

In this example, two tables are joined using `LEFT OUTER JOIN`. `T1`, the first table named includes nonmatching rows.

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.key = T2.key
```

If you use a `CROSS JOIN`, no `ON` expression is allowed for the join.

Subquery in a From Clause

Subqueries can be used in the From clause in place of table references (*table_name*).

Example

```
SELECT * FROM (SELECT * FROM emp WHERE sal > 10000) new_emp, dept WHERE
new_emp.deptno = dept.deptno
```

See also

Subqueries on page 177

Where Clause

Purpose

Specifies the conditions that rows must meet to be retrieved.

Syntax

WHERE *expr1rel_operatorexpr2*

where:

expr1

is either a column name, literal, or expression.

expr2

is either a column name, literal, expression, or subquery. Subqueries must be enclosed in parentheses.

rel_operator

is the relational operator that links the two expressions.

Example

The following Select statement retrieves the first and last names of employees that make at least $20,000.

```
SELECT last_name, first_name FROM emp WHERE salary >= 20000
```

See also

Subqueries on page 177
SQL Expressions on page 169

Group By Clause

Purpose

Specifies the names of one or more columns by which the returned values are grouped. This clause is used to return a set of aggregate values.

Syntax

GROUP BY *column_expression* [,...]

where:

column_expression

is either a column name or a SQL expression. Multiple values must be separated by a comma. If *column_expression* is a column name, it must match one of the column names specified in the Select clause. Also, the Group By clause must include all non-aggregate columns specified in the Select list.

Example

The following example totals the salaries in each department:

```
SELECT dept_id, sum(salary) FROM emp GROUP BY dept_id
```

This statement returns one row for each distinct department ID. Each row contains the department ID and the sum of the salaries of the employees in the department.

Notes

The driver uses the MongoDB aggregation framework whenever possible. In some instances, MongoDB may aggregate data in unexpected ways. For example, if you use the GROUP BY clause to return zip codes and the database contains the zip code 15237 as both a string and an integer, then two rows will be returned for 15237 (one for the string representation and another for the integer representation).

See also

Subqueries on page 177
SQL Expressions on page 169

Having Clause

Purpose

Specifies conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). This clause is valid only if you have already defined a Group By clause.

Syntax

```
HAVING expr1 rel_operator expr2
```

where:

expr1 | *expr2*

can be column names, constant values, or expressions. These expressions do not have to match a column expression in the Select clause. See "SQL Expressions" for details regarding SQL expressions.

rel_operator

is the relational operator that links the two expressions.

Example

The following example returns only the departments that have salaries totaling more than $200,000:

```
SELECT dept_id, sum(salary) FROM emp GROUP BY dept_id HAVING sum(salary) > 200000
```

Notes

The driver uses the MongoDB aggregation framework whenever possible. In some instances, MongoDB may aggregate data in unexpected ways. For example, if you use the HAVING clause to return zip codes and the database contains the zip code 15237 as both a string and an integer, then two rows will be returned for 15237 (one for the string representation and another for the integer representation).

See also

Subqueries on page 177

SQL Expressions on page 169

Union Operator

Purpose

Combines the results of two Select statements into a single result. The single result is all the returned rows from both Select statements. By default, duplicate rows are not returned. To return duplicate rows, use the All keyword (`UNION ALL`).

Syntax

```
select_statement
UNION [ALL | DISTINCT] | {MINUS [DISTINCT] | EXCEPT [DISTINCT]} | INTERSECT
[DISTINCT]select_statement
```

Notes

- When using the Union operator, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
UNION
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
UNION
SELECT salary, last_name FROM raises
```

Intersect Operator

Purpose

Intersect operator returns a single result set. The result set contains rows that are returned by both Select statements. Duplicates are returned unless the Distinct operator is added.

Syntax

```
select_statement
INTERSECT [DISTINCT]
select_statement
```

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

- When using the Intersect operator, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
INTERSECT [DISTINCT]
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
INTERSECT
SELECT salary, last_name FROM raises
```

Except and Minus Operators

Purpose

Return the rows from the left Select statement that are not included in the result of the right Select statement.

Syntax

```
select_statement
{EXCEPT [DISTINCT] | MINUS [DISTINCT]}
select_statement
```

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

- When using one of these operators, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
EXCEPT
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
EXCEPT
SELECT salary, last_name FROM raises
```

Order By Clause

Purpose

The Order By clause specifies how the rows are to be sorted.

Syntax

`ORDER BY` *`sort_expression`* `[DESC | ASC] [,...]`

where:

`sort_expression`

is either the name of a column, a column alias, a SQL expression, or the positioned number of the column or expression in the select list to use.

The default is to perform an ascending (`ASC`) sort.

Example

To sort by `last_name` and then by `first_name`, you could use either of the following Select statements:

```
SELECT emp_id, last_name, first_name FROM emp

ORDER BY last_name, first_name
```

or

```
SELECT emp_id, last_name, first_name FROM emp
ORDER BY 2,3
```

In the second example, `last_name` is the second item in the Select list, so `ORDER BY 2,3` sorts by `last_name` and then by `first_name`.

See also

SQL Expressions on page 169

Limit Clause

Purpose

Places an upper bound on the number of rows returned in the result.

Syntax

```
LIMIT number_of_rows [OFFSET offset_number]
```

where:

number_of_rows

specifies a maximum number of rows in the result. A negative number indicates no upper bound.

`OFFSET`

specifies how many rows to skip at the beginning of the result set. *offset_number* is the number of rows to skip.

Notes

- In a compound query, the Limit clause can appear only on the final Select statement. The limit is applied to the entire query, not to the individual Select statement to which it is attached.

Example

The following example returns a maximum of 20 rows.

```
SELECT last_name, first_name FROM emp WHERE salary > 20000 ORDER BY dept_idc LIMIT
20
```

Update

Purpose

An Update statement changes the value of columns in the selected rows of a table.

Syntax

```
UPDATE table_name SET column_name = expression
[, column_name = expression] [WHERE conditions]
```

table_name

is the name of the table for which you want to update values.

column_name

is the name of a column, the value of which is to be changed. Multiple column values can be changed in a single statement.

expression

is the new value for the column. The expression can be a constant value or a subquery that returns a single value. Subqueries must be enclosed in parentheses.

Example A

The following example changes every record that meets the conditions in the Where clause. In this case, the salary and exempt status are changed for all employees having the employee ID `E10001`. Because employee IDs are unique in the emp table, only one record is updated.

```
UPDATE emp SET salary=32000, exempt=1

WHERE emp_id = 'E10001'
```

Example B

The following example uses a subquery. In this example, the salary is changed to the average salary in the company for the employee having employee ID `E10001`.

```
UPDATE emp SET salary = (SELECT avg(salary) FROM emp)

WHERE emp_id = 'E10001'
```

Notes

- Insert, Update, and Delete are supported for MongoDB for simple types at the root level.
- To enable Insert, Update, and Delete, set the ReadOnly connection option to `false`.
- A Where clause can be used to restrict which rows are updated.

See also

Subqueries on page 177
Where Clause on page 163

SQL Expressions

An expression is a combination of one or more values, operators, and SQL functions that evaluate to a value. You can use expressions in the Where, and Having of Select statements; and in the Set clauses of Update statements.

Expressions enable you to use mathematical operations as well as character string manipulation operators to form complex queries.

The MongoDB driver supports both unquoted and quoted identifiers. An unquoted identifier must start with an ASCII alpha character and can be followed by zero or more ASCII alphanumeric characters. Unquoted identifiers are converted to uppercase before being used.

Quoted identifiers must be enclosed in double quotation marks (""). A quoted identifier can contain any Unicode character including the space character. The MongoDB driver recognizes the Unicode escape sequence \uxxxx as a Unicode character. You can specify a double quotation mark in a quoted identifier by escaping it with a double quotation mark.

The maximum length of both quoted and unquoted identifiers is 128 characters.

Valid expression elements are:

- Column names
- Literals
- Operators
- Functions

Column Names

The most common expression is a simple column name. You can combine a column name with other expression elements.

Literals

Literals are fixed data values. For example, in the expression `PRICE * 1.05`, the value 1.05 is a constant. Literals are classified into types, including the following:

- Binary
- Character string
- Date
- Floating point
- Integer
- Numeric
- Time
- Timestamp

The following table describes the literal format for supported SQL data types.

Table 41: Literal Syntax Examples

SQL Type	Literal Syntax	Example
BIGINT	*n* where *n* is any valid integer value in the range of the BIGINT data type	`12 or -34 or 0`
BOOLEAN	Min Value: 0 Max Value: 1	`0` `1`
DATE	DATE'*date*'	`'2010-05-21'`

SQL Type	Literal Syntax	Example
DATETIME	TIMESTAMP'*ts*'	`'2010-05-21 18:33:05.025'`

DECIMAL	*n.f*
	where:
	n
	is the integral part
	f
	is the fractional part
		`0.25`
		`3.1415`
		`-7.48`

DOUBLE	*n.f*E*x* where: *n* is the integral part *f* is the fractional part *x* is the exponent	`1.2E0` or `2.5E40` or `-3.45E2` or `5.67E-4`
INTEGER	*n* where *n* is a valid integer value in the range of the INTEGER data type	`12` or `-34` or `0`
LONGVARBINARY	'*hex_value*'	`'000482ff'`
LONGVARCHAR	'*value*'	`'This is a string literal'`
TIME	TIME'*time*'	`'2010-05-21 18:33:05.025'`
VARCHAR	'*value*'	`'This is a string literal'`

Character String Literals

Text specifies a character string literal. A character string literal must be enclosed in single quotation marks. To represent one single quotation mark within a literal, you must enter two single quotation marks. When the data in the fields is returned to the client, trailing blanks are stripped.

A character string literal can have a maximum length of 32 KB, that is, (32*1024) bytes.

Example

```
'Hello'
'Jim''s friend is Joe'
```

Numeric Literals

Unquoted numeric values are treated as numeric literals. If the unquoted numeric value contains a decimal point or exponent, it is treated as a real literal; otherwise, it is treated as an integer literal.

Example

```
+1894.1204
```

Binary Literals

Binary literals are represented with single quotation marks. The valid characters in a binary literal are 0-9, a-f, and A-F.

Example

```
'00af123d'
```

Date/Time Literals

Date and time literal values are enclosed in single quotion marks ('*value*').

- The format for a Date literal is DATE'*date*'.
- The format for a Time literal is TIME'*time*'.
- The format for a Timestamp literal is TIMESTAMP'*ts*'.

Integer Literals

Integer literals are represented by a string of numbers that are not enclosed in quotation marks and do not contain decimal points.

Notes

- Integer constants must be whole numbers; they cannot contain decimals.
- Integer literals can start with sign characters (+/-).

Example

```
1994 or -2
```

Operators

This section describes the operators that can be used in SQL expressions.

Unary Operator

A unary operator operates on only one operand.

operator operand

Binary Operator

A binary operator operates on two operands.

operand1 operator operand2

If an operator is given a null operand, the result is always null. The only operator that does not follow this rule is concatenation (||).

Arithmetic Operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of this operation is also a numeric value. The + and - operators are also supported in date/time fields to allow date arithmetic. The following table lists the supported arithmetic operators.

Table 42: Arithmetic Operators

Operator	Purpose	Example
+ -	Denotes a positive or negative expression. These are unary operators.	`SELECT * FROM emp WHERE comm = -1`
* /	Multiplies, divides. These are binary operators.	`UPDATE emp SET sal = sal + sal * 0.10`
+ -	Adds, subtracts. These are binary operators.	`SELECT sal + comm FROM emp WHERE empno > 100`

Concatenation Operator

The concatenation operator manipulates character strings. The following table lists the only supported concatenation operator.

Table 43: Concatenation Operator

Operator	Purpose	Example
||	Concatenates character strings.	`SELECT 'Name is' || ename FROM emp`

The result of concatenating two character strings is the data type VARCHAR.

Comparison Operators

Comparison operators compare one expression to another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN (if one of the operands is NULL). The MongoDB driver considers the UNKNOWN result as FALSE.

The following table lists the supported comparison operators.

Table 44: Comparison Operators

Operator	Purpose	Example
=	Equality test.	`SELECT * FROM emp WHERE sal = 1500`
!=<>	Inequality test.	`SELECT * FROM emp WHERE sal != 1500`
><	“Greater than" and "less than" tests.	`SELECT * FROM emp WHERE sal > 1500 SELECT * FROM emp WHERE sal < 1500`
>=<=	“Greater than or equal to" and "less than or equal to" tests.	`SELECT * FROM emp WHERE sal >= 1500 SELECT * FROM emp WHERE sal <= 1500`

ESCAPE clause in LIKE operator
LIKE ’pattern string’ ESCAPE ’c’
	The Escape clause is supported in the LIKE predicate to indicate the escape character. Escape characters are used in the pattern string to indicate that any wildcard character that is after the escape character in the pattern string should be treated as a regular character.
	The default escape character is backslash (\).
		`SELECT * FROM emp WHERE ENAME LIKE 'J%_%' ESCAPE '\'`
		This matches all records with names that start with letter 'J' and have the '_' character in them.
		`SELECT * FROM emp WHERE ENAME LIKE 'JOE_JOHN' ESCAPE '\'`
		This matches only records with name ’JOE_JOHN’.

[NOT] IN	“Equal to any member of" test.	`SELECT * FROM emp WHERE job IN ('CLERK','ANALYST') SELECT * FROM emp WHERE sal IN (SELECT sal FROM emp WHERE deptno = 30)`
[NOT] BETWEEN x AND y	"Greater than or equal to x" and "less than or equal to y."	`SELECT * FROM emp WHERE sal BETWEEN 2000 AND 3000`
EXISTS	Tests for existence of rows in a subquery.	`SELECT empno, ename, deptno FROM emp e WHERE EXISTS (SELECT deptno FROM dept WHERE e.deptno = dept.deptno)`
IS [NOT] NULL	Tests whether the value of the column or expression is NULL.	`SELECT * FROM emp WHERE ename IS NOT NULL SELECT * FROM emp WHERE ename IS NULL`

Logical Operators

A logical operator combines the results of two component conditions to produce a single result or to invert the result of a single condition. The following table lists the supported logical operators.

Table 45: Logical Operators

Operator	Purpose	Example
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	`SELECT * FROM emp WHERE NOT (job IS NULL)` `SELECT * FROM emp WHERE NOT (sal BETWEEN 1000 AND 2000)`
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE; otherwise, returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' AND deptno = 10`
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE; otherwise, returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' OR deptno = 10`

Example

In the Where clause of the following Select statement, the `AND` logical operator is used to ensure that managers earning more than $1000 a month are returned in the result:

```
SELECT * FROM emp WHERE jobtitle = manager AND sal > 1000
```

Operator Precedence

As expressions become more complex, the order in which the expressions are evaluated becomes important. The following table shows the order in which the operators are evaluated. The operators in the first line are evaluated first, then those in the second line, and so on. Operators in the same line are evaluated left to right in the expression.You can change the order of precedence by using parentheses. Enclosing expressions in parentheses forces them to be evaluated together.

Table 46: Operator Precedence

Precedence	Operator
1	+ (Positive), - (Negative)
2	*(Multiply), / (Division)
3	+ (Add), - (Subtract)
4	|| (Concatenate)
5	=, >, <, >=, <=, <>, != (Comparison operators)
6	NOT, IN, LIKE
7	AND
8	OR

Example A

The query in the following example returns employee records for which the department number is 1 or 2 and the salary is greater than $1000:

```
SELECT * FROM emp WHERE (deptno = 1 OR deptno = 2) AND sal > 1000
```

Because parenthetical expressions are forced to be evaluated first, the OR operation takes precedence over AND.

Example B

In the following example, the query returns records for all the employees in department 1, but only employees whose salary is greater than $1000 in department 2.

```
SELECT * FROM emp WHERE deptno = 1 OR deptno = 2 AND sal > 1000
```

The AND operator takes precedence over OR, so that the search condition in the example is equivalent to the expression `deptno = 1 OR (deptno = 2 AND sal > 1000).`

Functions

The MongoDB driver supports a number of functions that you can use in expressions, as listed and described in "Supported scalar functions."

See also

Supported scalar functions on page 52

Conditions

A condition specifies a combination of one or more expressions and logical operators that evaluates to either TRUE, FALSE, or UNKNOWN. You can use a condition in the Where clause of the Delete, Select, and Update statements; and in the Having clauses of Select statements. The following describes supported conditions.

Table 47: Conditions

Condition	Description
Simple comparison	Specifies a comparison with expressions or subquery results. `= , !=, <>, < , >, <=, <=`
Group comparison	Specifies a comparison with any or all members in a list or subquery. `[= , !=, <>, < , >, <=, <=] [ANY, ALL, SOME]`
Membership	Tests for membership in a list or subquery. `[NOT] IN`

Condition	Description
Range	Tests for inclusion in a range. `[NOT] BETWEEN`
NULL	Tests for nulls. `IS NULL, IS NOT NULL`
EXISTS	Tests for existence of rows in a subquery. `[NOT] EXISTS`
LIKE	Specifies a test involving pattern matching. `[NOT] LIKE`
Compound	Specifies a combination of other conditions. `CONDITION [AND/OR] CONDITION`

Subqueries

A query is an operation that retrieves data from one or more tables or views. In this reference, a top-level query is called a Select statement, and a query nested within a Select statement is called a subquery.

A subquery is a query expression that appears in the body of another expression such as a Select, an Update, or a Delete statement. In the following example, the second Select statement is a subquery:

```
SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM dept)
```

IN Predicate

Purpose

The In predicate specifies a set of values against which to compare a result set. If the values are being compared against a subquery, only a single column result set is returned.

Syntax

```
value [NOT] IN (value1, value2,...)
```

OR

```
value [NOT] IN (subquery)
```

Example

```
SELECT * FROM emp WHERE deptno IN

(SELECT deptno FROM dept WHERE dname <> 'Sales')
```

EXISTS Predicate

Purpose

The Exists predicate is true only if the cardinality of the subquery is greater than 0; otherwise, it is false.

Syntax

```
EXISTS (subquery)
```

Example

```
SELECT empno, ename, deptno FROM emp e WHERE EXISTS
(SELECT deptno FROM dept WHERE e.deptno = dept.deptno)
```

UNIQUE Predicate

Purpose

The Unique predicate is used to determine whether duplicate rows exist in a virtual table (one returned from a subquery).

Syntax

```
UNIQUE (subquery)
```

Example

```
SELECT * FROM dept d WHERE UNIQUE
(SELECT deptno FROM emp e WHERE e.deptno = d.deptno)
```

Correlated Subqueries

Purpose

A correlated subquery is a subquery that references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a Select, Update, or Delete statement.

A correlated subquery answers a multiple-part question in which the answer depends on the value in each row processed by the parent statement. For example, you can use a correlated subquery to determine which employees earn more than the average salaries for their departments. In this case, the correlated subquery specifically computes the average salary for each department.

Syntax

```
SELECT select_list
   FROM table1 t_alias1
   WHERE expr rel_operator
   (SELECT column_list
       FROM table2 t_alias2
```

```
        WHERE t_alias1.columnrel_operatort_alias2.column)
UPDATE table1 t_alias1
   SET column =
   (SELECT expr
        FROM table2 t_alias2
         WHERE t_alias1.column = t_alias2.column)
DELETE FROM table1 t_alias1
   WHERE column rel_operator
   (SELECT expr
        FROM table2 t_alias2
        WHERE t_alias1.column = t_alias2.column)
```

Notes

- Correlated column names in correlated subqueries must be explicitly qualified with the table name of the parent.

Example A

The following statement returns data about employees whose salaries exceed their department average. This statement assigns an alias to `emp`, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT deptno, ename, sal FROM emp x WHERE sal >
  (SELECT AVG(sal) FROM emp WHERE x.deptno = deptno)
  ORDER BY deptno
```

Example B

This is an example of a correlated subquery that returns row values:

```
SELECT * FROM dept "outer" WHERE 'manager' IN
  (SELECT managername FROM emp
  WHERE "outer".deptno = emp.deptno)
```

Example C

This is an example of finding the department number (`deptno`) with multiple employees:

```
SELECT * FROM dept main WHERE 1 <
  (SELECT COUNT(*) FROM emp WHERE deptno = main.deptno)
```

Example D

This is an example of correlating a table with itself:

```
SELECT deptno, ename, sal FROM emp x WHERE sal >
  (SELECT AVG(sal) FROM emp WHERE x.deptno = deptno)
```

Custom function escapes

The MongoDB driver supports the custom function escape CAST_TO_NATIVE, as described in the following topic.

CAST_TO_NATIVE function escape

Purpose

The CAST_TO_NATIVE function escape can be used in a filter to select or insert a value of a specific native type.

In some cases, a value with a specific native type can be concealed or obscured because the driver describes a field with inconsistent native types as a column with a single SQL type. For example, the driver maps a MongoDB `_id` field with the native types String and ObjectId to a relational `_ID` column with the SQL_WVARCHAR type. In this context, the CAST_TO_NATIVE function escape allows users to send a value as it is defined in the MongoDB database, rather than how it is described in the relational model of the data.

Currently, CAST_TO_NATIVE can only be used with the ObjectID type in SELECT statement filters and literal INSERT values.

Syntax

```
{fn CAST_TO_NATIVE('value','native_type')}
```

where:

value

is the value to be selected or inserted.

native_type

is the native type that in part defines the value.

Example Select

The following statement selects records from `Account` where the `_ID` field has an ObjectID value of `507f1f77bcf86cd799439011`.

```
SELECT * FROM Account WHERE _ID = {fn
CAST_TO_NATIVE('507f1f77bcf86cd799439011','objectid')}
```

Example Insert

The following statement inserts the ObjectID value `507f1f77bcf86cd799439011` into the `_ID` field.

```
INSERT INTO Account(_ID) VALUES ({fn
CAST_TO_NATIVE('507f1f77bcf86cd799439011','objectid')})
```