Progress DataDirect for JDBC for Jira

User's Guide

Release 6.0.0

Copyright

Table of Contents

1

Welcome to the Progress DataDirect for JDBC for Jira Driver

The Progress® DataDirect® for JDBC™ for Jira™ driver (Jira driver) supports SQL read-only access for JDBC applications to Atlassian Jira. To support SQL access to Jira services, the driver creates a relational map of the Jira data model and translates SQL statements to Jira REST API requests. In addition, the driver employs a SQL engine component that provides support to SQL constructs unavailable to Jira services. This functionality offers a number of advantages, including support for reporting data and metadata in a form that JDBC applications are ready to use.

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-atlassian-jira.

For details, see the following topics:

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

Highlights of 6.0.0 Release

- The driver supports SQL read-only access to Atlassian Jira. See Supported SQL statements and extensions on page 79 for details.
- The driver supports JDBC core functions. For details, refer to "JDBC support" in the *Progress DataDirect for JDBC Drivers Reference*.
- The driver supports standard JSON data types and additional data types through data type inference. See Data types on page 19 and getTypeInfo() on page 19 for details.
- * The driver supports Jira custom fields, including many fields added through third-party plug-ins
- The driver supports the handling of large result sets with paging, and the FetchSize on page 52 and WSFetchSize on page 75 connection properties.
- The driver supports SSL data encryption. See Data encryption on page 40 for more information.

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.

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\60\jira.jar
     ```

   - **UNIX/LINUX Example**

     ```
     CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/60/jira.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.

     ```
     jdbc:datadirect:jira:http://mycompany.atlassian.net;User=jsmith;Password=secret;
     ```

     Note: The User and Password properties can also be passed separately from the connection URL, such as your application's logon dialog.

   - **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.

Note: For most connections, specifying the minimum required connection properties is sufficient to begin accessing data; however, you can provide values for optional properties to use additional supported features and improve performance.

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 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 the processes for testing `DriverManager` and data source connections, and the SQL statements supported by the driver.
   - Testing a DriverManager connection: This section discusses how to establish and test a `DriverManager` connection using DataDirect Test.
   - Testing a data source connection: This section discusses how to establish and test a data source connection using DataDirect Test.
   - 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.

Version string information

The DatabaseMetaData.getDriverVersion() method returns a driver version string in the format:

M.m.s.bbbbbb(C*XXXX*.F*YYYYYY*.U*ZZZZZZ*)

where:

M is the major version number.

m is the minor version number.

s is the service pack number.

bbbbbb is the driver build number.

XXXX is the cloud adapter build number.

YYYYYY is the framework build number.

ZZZZZZ is the utl build number.

For example:

```
6.0.0.000002(C0003.F000001.U000002)
      |____| |___| |_____| |_____|
      Driver Cloud Frame    Utl
```

Driver and DataSource classes

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

Driver class:

com.ddtek.jdbc.jira.JiraDriver

DataSource class:

com.ddtek.jdbcx.jira.JiraDataSource

Connection URL examples

After setting the CLASSPATH, the required connection information needs to be passed in the form of a connection URL.

```
jdbc:datadirect:jira:servername;User=username;Password=password_token;
[property=value[;...]];
```

where:

servername

specifies the base URL of the Jira service to which you want to issue requests. For example, `http://mycompany.atlassian.net`.

Note: Specifying an HTTPS URL for this property enables data encryption.

username

specifies the user name that is used to connect to your Jira service.

password

specifies the password or API token used to connect to your Jira service.

property=value

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

The following example shows how to establish a connection to a Jira service:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:jira:http://mycompany.atlassian.net;User=jsmith;Password=secret;");
```

See also

ServerName on page 67
User on page 73
Password on page 61
Connection property descriptions on page 45

Connection properties

The driver includes over 40 connection properties. You can use connection properties to customize the driver for your environment. Connection properties can be used to accomplish different tasks, such as implementing driver functionality and optimizing performance. You can specify connection properties in a connection URL or within a JDBC data source object.

See also

Connection property descriptions on page 45

Introduction to the Jira Data Model

To expose Jira resources to SQL applications, the driver maps the native data model to relational tables and translates SQL statements to REST API requests. See "Mapping Objects to Tables" for more information on relational mapping.

Although the JIRA environment promotes customization, a similar set of resources are used by most Jira services. The following table documents the types of Jira API Resources you are likely to encounter in your environment, along with the relational mapping of the primary and child tables. Refer to the Jira documentation for more information on resources.

Table 1: Jira API Resource Mapping

Jira API Resource	Relational Parent Table(s)	Relational Child Table(s)
Application roles	ApplicationRoles	• ApplicationRolesDefaultGroups • ApplicationRolesGroups
Audit records	Auditing	• AuditingAssociatedItems • AuditingChangedValues
Dashboards	Dashboards	DashboardSharePermissions
Filters	Filters	• FilterSharedUserItems • FilterSharePermissions • FilterSubscriptions
	FilterDefaultColumns	NA
Groups	Groups	GroupLabels
	GroupUsers	NA

Jira API Resource	Relational Parent Table(s)	Relational Child Table(s)
Issue attachment	Attachments	NA
	AttachmentMeta	NA
Issue comment properties	CommentProperty	NA
	CommentPropertyKeys	NA
Issue comments	Comments	NA
Issue fields	Fields	FieldClauseNames
Issue link types	IssueLinkTypes	NA
Issue resolutions	Resolutions	NA
Issue types	IssueTypes	NA
Issue votes	Votes	Voters
Issue watchers	Watchers	NA
Issue worklogs	Worklogs	• WorklogProperties • WorklogPropertyValues

<table>
<tr><th>Jira API Resource</th><th>Relational Parent Table(s)</th><th>Relational Child Table(s)</th></tr>
<tr><td rowspan="5">Issues</td><td>Issues</td><td>• IssueAttachments
• IssueComments
• IssueComponents
• IssueFixVersions
• IssueLabels
• IssueLinks
• IssueSubtasks
• IssueVersions
• IssueWorklogs</td></tr>
<tr><td>IssueChangeLog</td><td>NA</td></tr>
<tr><td>IssueChangeLogItems</td><td>NA</td></tr>
<tr><td>IssueTransitions</td><td>NA</td></tr>
<tr><td colspan="2">Comments:
• Primary table contains a record of all the Issues. Note that issuing unfiltered queries against large data sources may take a long time to process. To limit the number of rows returned, use the following statements:
<code>SELECT TOP</code>
or
<code>SELECT ... LIMIT</code>
• The IssueChangeLog tables require either an ID or a KEY, in the Where clause.</td></tr>
<tr><td rowspan="4">Jira expressions</td><td>Boards</td><td>NA</td></tr>
<tr><td>Epics</td><td>NA</td></tr>
<tr><td>Sprints</td><td>NA</td></tr>
<tr><td colspan="2">Comments:The Sprints table requires either an ID or a KEY, in the Where clause.</td></tr>
<tr><td rowspan="3">Jira settings</td><td>AdvancedSettings</td><td>AdvancedSettingsAllowedValues</td></tr>
<tr><td>ApplicationProperties</td><td>ApplicationPropertiesAllowedValues</td></tr>
<tr><td>Configuration</td><td>NA</td></tr>
<tr><td>Permissions</td><td>MyPermissions</td><td>NA</td></tr>
<tr><td>Projects</td><td>Projects</td><td>ProjectIssueTypes</td></tr>
</table>

Jira API Resource	Relational Parent Table(s)	Relational Child Table(s)
Project components	Components	NA
Project Versions	ProjectVersions	ProjectKey
Project role actors	Roles	RoleActors
Users	UserDetails	• UserApplicationRoles • UserGroups • Users
Workflows	Workflows	NA

Mapping objects to tables

Data mapping describes how elements are mapped between two distinct data models. To support SQL access to a Jira service, Jira REST endpoints must be mapped to a relational schema. The driver automatically generates a relational view of your data the first time you execute a SQL statement. When generating the relational view, the driver decompounds JSON documents stored at endpoints into parent-child tables. The driver handles mapping the following manner:

- Simple and nested objects are flattened and mapped to a parent table
- Arrays are mapped to related child tables

For example, the following JSON document contains nested objects in the `issuetype` object and an array in the `labels` object.

```
{
    "id": "10000",
    "key": "ABC-100",
    "fields": {
        "issuetype": {
            "id": "20003",
            "description": "A problem that impairs the functions of the product.",
            "name": "Bug",
        },
    "labels": [
         "Dev",
         "Severe",
         "v7.1"
       ],
}
{
    "id": "10001",
    "key": "ABC-101",
    "fields": {
        "issuetype": {
            "id": "20022",
            "description": "An user story.",
            "name": "Story",
        },
    "labels": [
         "Dev",
         "Installer"
```

```
            "v8.0"
        ],
}
```

When generating the relational view, the driver decompounds native arrays into separate, but related tables. The mapping of the sample JSON document produced one parent table and two child tables. In the parent table, simple objects, such as `id` and `key`, are flattened into corresponding relational columns. Nested objects are also flattened into relational columns; however, column names are formed by concatenating the name of the parent and nested objects, which are joined by an underscore character. For example, the `FIELDS_ISSUETYPE_NAME` column contains the values of the `name` object that is nested in the `fields` and `issuetype` objects.

The names of parent tables are determined by the Jira REST API resource used to issue requests. For a list of common tables, refer to "Introduction to the Jira Data Model."

The parent table for our example uses the Issues API resource, and therefore, is named `ISSUES`. The table takes the following form:

Table 2: ISSUES

ID (PK)	KEY	FIELDS_ISSUETYPE_ID	FIELDS_ISSUETYPE_DESCRIPTION	FIELDS_ISSUETYPE_NAME
10000	ABC-100	20003	A problem that impairs the functions of the product	Bug
10001	ABC-101	20022	An user story.	Story

The information in the `labels` array normalizes to the `ISSUELABELS` child table. The values of the array are mapped into a single relational column that corresponds to the name of the array. For example, the values for the `labels` array in the JSON sample, such as `Dev` and `Severe`, map to the `ISSUELABEL` column in the `ISSUELABELS` table. A foreign key relationship to the parent table is provided by including the primary key of the parent in the child, in this case, `ISSUES_ID`. The primary key of the child table is a composite key formed by the primary key of the parent table combined with the positional information contained in the `POSITION` column. If the array is nested multiple layers deep, additional positional columns for parent objects are mapped to insure that a unique key is used.

The child table for the `ISSUELABLES` array would take the following form:

Table 3: ISSUELABELS

ISSUES_ID (PK)	POSITION (PK)	ISSUELABEL
10000	0	Dev
10000	1	Severe
10000	2	v7.1
10001	0	Dev
10001	1	Installer
10001	2	v7.1

See also

Introduction to the Jira Data Model on page 14

Data types

The following table lists native 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 4: Jira Data Types

Jira Data Type	JDBC Data Type
BigInt	BIGINT
Boolean	BOOLEAN
Date	DATE
Integer	INTEGER
JSON	JSON
LongVarChar	LONGVARCHAR
Timestamp	TIMESTAMP
VarChar	VARCHAR

getTypeInfo()

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

Table 5: getTypeInfo() Results

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 = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
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 = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = Date AUTO_INCREMENT = false CASE_SENSITIVE = false CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = false LITERAL_PREFIX = DATE ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = DATE MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = NULL 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 = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = JSON AUTO_INCREMENT = false CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = JSON MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
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 = 10 PRECISION = 16777215 SEARCHABLE = 0 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

TYPE_NAME = Timestamp 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 = TIMESTAMP MAXIMUM_SCALE = 9	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 23 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false
TYPE_NAME = VarChar AUTO_INCREMENT = false CASE_SENSITIVE = true CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = false LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = VARCHAR MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32767 SEARCHABLE = 3 SQL_DATA_TYPE = NULL SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = false

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" in the *Progress DataDirect for JDBC Drivers Reference* for information about SQL escape sequences.

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 6: 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	
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	

String Functions	Numeric Functions	Timedate Functions	System Functions
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		

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*.

- 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 the "Troubleshooting" section in the *Reference* 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

Using the Driver

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.

For details, see the following topics:

- Required permissions for Java SE with the standard Security Manager enabled
- Connecting from an application
- Connecting through a proxy server
- IP addresses
- Performance considerations
- Data encryption
- Enabling Debug Record Mode
- Isolation levels
- Statement Pooling
- Connection pooling

Required permissions for Java SE with the standard Security Manager enabled

Using the driver on a Java platform with the standard Security Manager enabled requires certain permissions to be set in the Java SE security policy file `java.policy`. The default location of this file is *`java_install_dir`*`/jre/lib/security`.

Note: Security manager may be enabled by default in certain scenarios, such as running on an application server or in a Web browser applet.

To run an application on a Java platform with the standard Security Manager, use the following command:

```
"java -Djava.security.manager application_class_name"
```

where *`application_class_name`* is the class name of the application.

Refer to your Java documentation for more information about setting permissions in the security policy file.

Permissions for establishing connections

To establish a connection to the database server, the driver must be granted the permissions as shown in the following example:

```
grant codeBase "file:/install_dir/lib/60/-" {
   permission java.net.SocketPermission "*", "connect";
};
```

where:

`install_dir`

is the product installation directory.

Granting access to Java properties

To allow the driver to read the value of various Java properties to perform certain operations, permissions must be granted as shown in the following example:

```
grant codeBase "file:/install_dir/lib/60/-" {
   permission java.util.PropertyPermission "*", "read, write";
};
```

where:

`install_dir`

is the product installation directory.

Granting access to temporary files

Access to the temporary directory specified by the JVM configuration must be granted in the Java SE security policy file to use insensitive scrollable cursors or to perform client-side sorting of DatabaseMetaData result sets. The following example shows permissions that have been granted for the `C:\TEMP` directory:

```
grant codeBase "file:/install_dir/lib/60/-" {
// Permission to create and delete temporary files.
// Adjust the temporary directory for your environment.
   permission java.io.FilePermission "C:\\TEMP\\-", "read,write,delete";
};
```

where:

install_dir

is the product installation directory.

Connecting from an application

After the driver has been installed and defined on your class path, 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).

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/60/jira.jar
```

Windows Example

```
CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC\lib\60\jira.jar
```

UNIX Example

```
CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/60/jira.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:jira:https://mycompany.atlassian.net;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.

Connection URL Syntax

The connection URL takes the following form:

```
jdbc:datadirect:jira:servername;User=username;Password=password_token;
[property=value[;...]];
```

where:

servername

specifies the URL of the Jira service to which you want to issue requests. For example, `http://mycompany.atlassian.net`.

user

specifies the user name that is used to connect to the Jira service.

password_token

specifies the password or API token used to connect to your Jira service.

property=value

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

Connection URL Example

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:jira:http://mycompany.atlassian.net;User=jsmith;Password=secret;");
```

See also

ServerName on page 67
User on page 73
Password on page 61
Connection property descriptions on page 45
Connection URL examples on page 13

Testing the connection

You can also use DataDirect Test to establish and test a `DriverManager` connection. The screen shots in this section were taken on a Windows system.

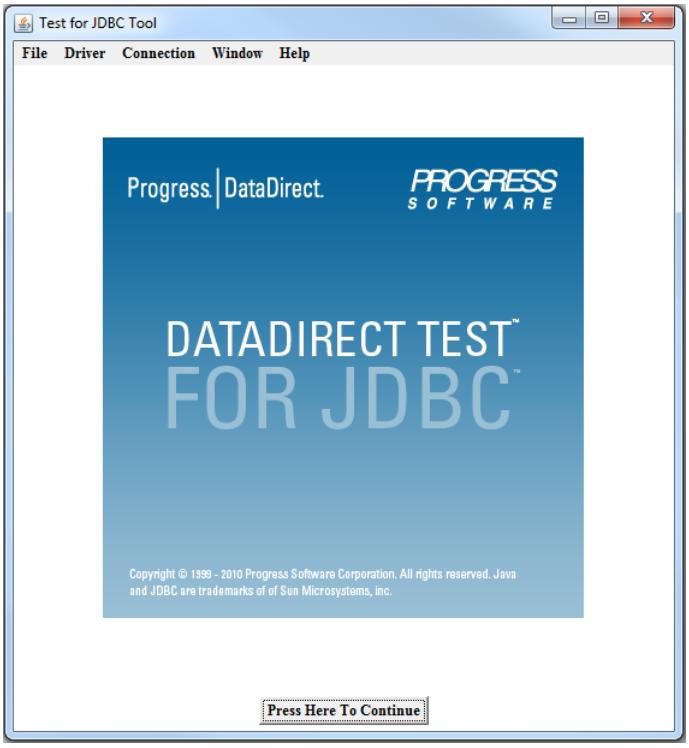
Take the following steps to establish a connection.

1. Navigate to the installation directory. The default location is:
   - Windows systems: `Program Files\Progress\DataDirect\JDBC_60\testforjdbc`
   - UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC_60/testforjdbc`

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

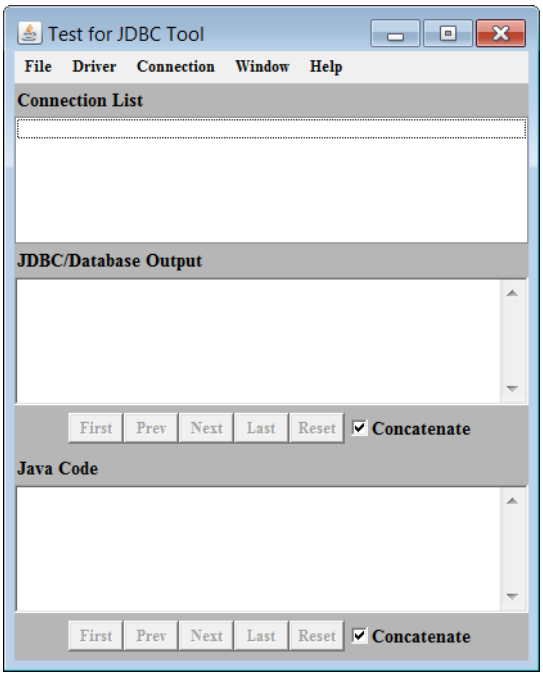
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

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



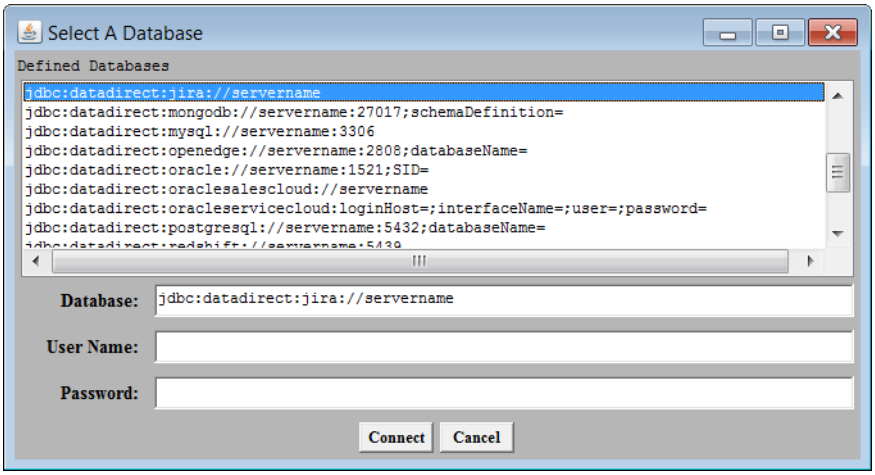
3. Click **Press Here to Continue**.

   The main dialog appears:



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

   The **Select A Database** dialog appears:



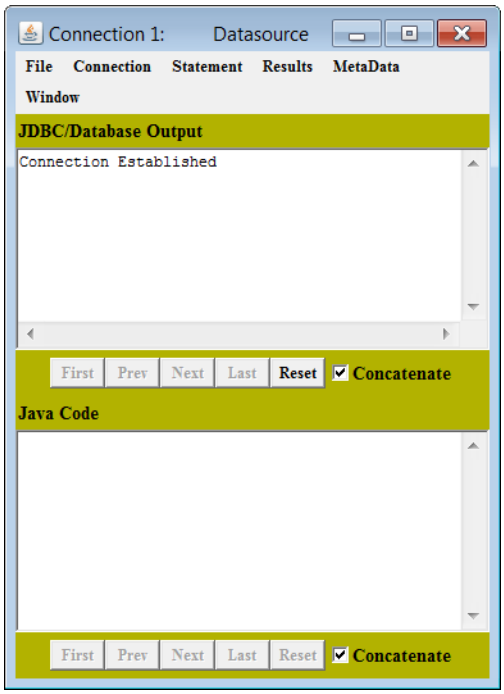
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify the ServerName for your Jira service.

   For example:

   ```
   jdbc:datadirect:jira:http://mycompany.atlassian.net
   ```

7. Enter values for the following fields.
   - In the User Name field, enter your user ID used to access your Jira service.
   - In the Password field, enter your password or API token used to access your Jira service.
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.)



Refer to "DataDirect Test" in the *Progress DataDirect for JDBC Drivers Reference* for more information about using DataDirect Test.

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.

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.

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.jira.JiraDataSource;
```

Next, you will need to set the values and define the data source. For example, the following definition contains the minimum properties required to establish connection:

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").

```
JiraDataSource mds = new JiraDataSource();
mds.setDescription("My Jira Data Source");
mds.setServerName("http://mycompany.atlassian.net");
mds.setUser("jsmith");
mds.setPassword("secret");
```

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 JiraDataSource)
{
JiraDataSource jmds = (JiraDataSource) ds;
System.out.println("description=" + jmds.getDescription());
System.out.println("servername=" + jmds.getServerName());
System.out.println("user=" + jmds.getUser());
System.out.println("password=" + jmds.getPassword());
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.

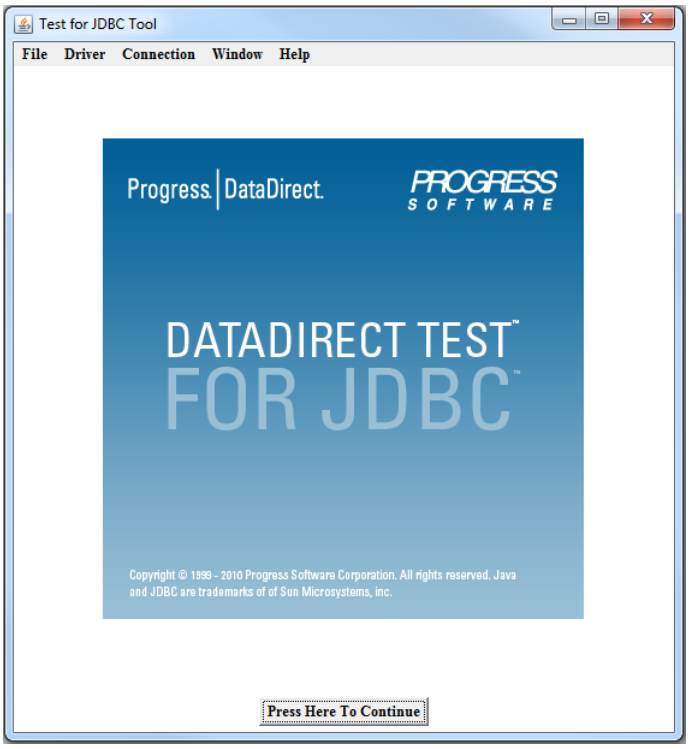
Take the following steps to establish a connection.

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.

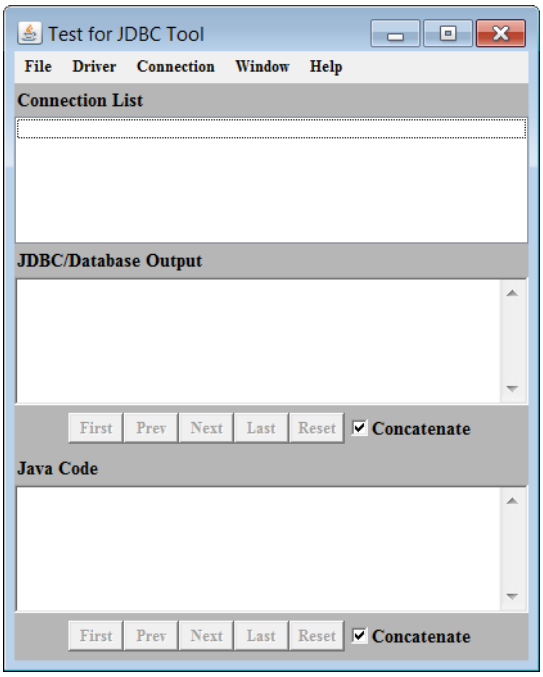
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

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



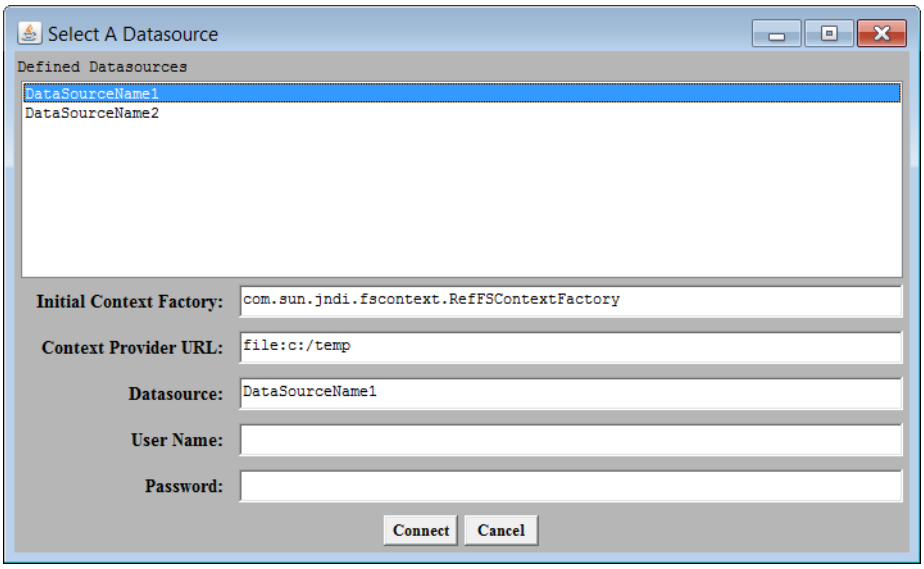
3. Click **Press Here to Continue**.

   The main dialog appears:



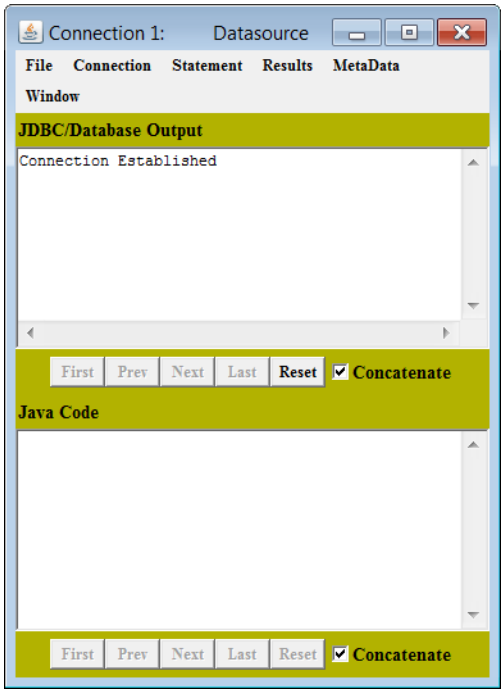
4. From the menu bar, select **Connection > Connect to DB via Data Source**.

   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.



Connecting through a 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 jira.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.

See also

Connection property descriptions on page 45

IP addresses

The driver supports Internet Protocol (IP) addresses in IPv4 and IPv6 format.

The server name specified in the connection URL, or data source, can resolve to an IPv4 or IPv6 address. For example, the following URL can resolve to either type of address:

```
jdbc:datadirect:jira:http://mycompany.atlassian.net;User=jsmith;Password=secret;
```

Note: The server name specifies the base Jira URL to use for logging in, for example, `http://mycompany.atlassian.net`.

Alternately, you can specify addresses using IPv4 or IPv6 format in the server portion of the connection URL. For example, the following connection URL specifies the server using an IPv4 address:

```
jdbc:datadirect:jira://123.456.78.90;User=jsmith;Password=secret;
```

You also can specify addresses in either format using the ServerName data source property. The following example shows a data source definition that specifies the server name using IPv6 format:

```
JiraDataSource mds = new JiraDataSource();
   mds.setDescription("My Jira DataSource");
   mds.setServerName("[ABCD:EF01:2345:6789:ABCD:EF01:2345:6789]");
   ...
```

Note: When specifying IPv6 addresses in a connection URL or data source property, the address must be enclosed by brackets.

In addition to the normal IPv6 format, the drivers support IPv6 alternative formats for compressed and IPv4/IPv6 combination addresses. For example, the following connection URL specifies the server using IPv6 format, but uses the compressed syntax for strings of zero bits:

```
jdbc:datadirect:jira://[2001:DB8:0:0:8:800:200C:417A];User=jsmith;Password=secret
```

Similarly, the following connection URL specifies the server using a combination of IPv4 and IPv6:

```
jdbc:datadirect:jira://[0000:0000:0000:0000:0000:FFFF:123.456.78.90];User=jsmith;
   Password=secret
```

For complete information about IPv6, go to the following URL:

http://tools.ietf.org/html/rfc4291#section-2.2

See also

ServerName on page 67

Performance considerations

InsensitiveResultSetBufferSize: To improve performance when using scroll-insensitive result sets, the driver can cache the result set data in memory instead of writing it to disk. By default, the driver caches 2 MB of insensitive result set data in memory and writes any remaining result set data to disk. Performance can be improved by increasing the amount of memory used by the driver before writing data to disk or by forcing the driver to never write insensitive result set data to disk. The maximum cache size setting is 2 GB.

FetchSize/**WSFetchSize**: The connection options FetchSize and WSFetchSize can be used to adjust the trade-off between throughput and response time. In general, setting larger values for WSFetchSize and FetchSize will improve throughput, but can reduce response time.

For example, if an application attempts to fetch 100,000 rows from the remote data source and WSFetchSize is set to `500`, the driver must make 200 Web service calls to get the 100,000 rows. If, however, WSFetchSize is set to `1000` (the maximum), the driver only needs to make 100 Web service calls to retrieve 100,000 rows. Web service calls are expensive, so generally, minimizing Web service calls increases throughput.

For many applications, throughput is the primary performance measure, but for interactive applications, such as Web applications, response time (how fast the first set of data is returned) is more important than throughput. For example, suppose that you have a Web application that displays data 50 rows to a page and that, on average, you view three or four pages. Response time can be improved by setting FetchSize to 50 (the number of rows displayed on a page) and WSFetchSize to 200. With these settings, the driver fetches all of the rows from the remote data source that you would typically view in a single Web service call and only processes the rows needed to display the first page.

Note that the values specified for the WSFetchSize and FetchSize properties provide only a suggestion as to the number of rows to be processed. The service will not exceed these values, but it will adjust page sizes to balance throughput amongst all its users. This behavior can result in different page sizes for successive queries.

ReadAhead: The ReadAhead property allows you to issue multiple fetch requests in parallel. By increasing this number, you can improve throughput and performance, but it does so with the following restrictions:

- Larger values can increase the load on the server, which may adversely affect performance of other users. If you encounter issues, decrease the value specified for this option.
- Larger values may result in unnecessary requests if your application only requires the first few rows of results. This may be an issue if your service places limits on the number of web requests.

Caution: Due to potential impacts to other users, we strongly recommend specifying only smaller values for the ReadAhead property. For example, in fully optimized environments, which include exceptionally fast connections and low latency, we recommend a setting of no higher than 5. For typical environments, this value is sometimes lower.

WSPoolSize: WSPoolSize determines the maximum number of sessions the driver uses when there are multiple active connections. By increasing this number, you increase the number of sessions the driver uses to distribute calls to the service, thereby improving throughput and performance. For example, if WSPoolSize is set to 1, and you have two open connections, the session must complete a call from one connection before it can begin processing a call from the other connection. However, if WSPoolSize is equal to 2, a second session is opened that allows calls from both connections to be processed simultaneously.

Note: The number specified for WSPoolSize should not exceed the amount of sessions permitted by your service.

Data encryption

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

- *SSL server authentication* requires the server to authenticate itself to the client.
- *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 SSL encryption

SSL encryption is enabled when the URL specified by the ServerName property file uses HTTPS. The following steps outline how to configure SSL encryption.

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:

1. Use the CryptoProtocolVersion property to specify acceptable cryptographic protocol versions (for example, TLSv1.2) supported by your server.
2. 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).
3. To validate certificates sent by the database server, set the ValidateServerCertificate property to `true`.
4. Optionally, 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.
5. If your database server is configured for SSL client authentication, configure your keystore information:
   a) 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).
   b) If any key entry in the keystore file is password-protected, set the KeyPassword property to the key password.

See also

ServerName on page 67
CryptoProtocolVersion on page 50
TrustStore on page 72
TrustStorePassword on page 72
ValidateServerCertificate on page 74
KeyStore on page 57
KeyStorePassword on page 58
KeyPassword on page 57

Configuring 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.

See also

TrustStore on page 72
TrustStorePassword on page 72

Configuring 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.

See also

KeyStore on page 57
KeyStorePassword on page 58
KeyPassword on page 57

Enabling Debug Record Mode

The driver supports a debug record mode that provides a method for troubleshooting issues that occur when accessing data from a Jira service. When Debug Record Mode is enabled, the driver captures and records server requests and responses to a set of files stored in a designated location. Technical Support can then use these files to analyze and reproduce the issue without requiring access to your private data source.

To generate debug record files:

1. Using the DebugRecord property, specify the location where the driver will generate the files used to record server requests and responses.
2. Start the JDBC application and reproduce the issue.
3. Stop the application.

The driver generates a set of files containing the server requests and responses that occurred during the session. After generating the debug files, you can remove the location specified for the DebugRecord property. If you do not remove this value, the driver will overwrite debug files in the specified location the next time you start the application.

Contact Technical Support for assistance analyzing the files and reproducing the issue.

Important: Debug record files may capture security-related headers, such as auth or token headers. Before sending Technical Support debug files, review the content to remove any confidential information that may have been recorded.

See also

DebugRecord on page 51

Isolation levels

Because transactions are not supported, the Jira driver supports only the TRANSACTION_NONE isolation level.

Statement Pooling

Most applications have a certain set of SQL statements that are executed multiple times and a few SQL statements that are executed only once or twice during the life of the application. Similar to connection pooling, *statement pooling* provides performance gains for applications that execute the same SQL statements multiple times over the life of the application.

A *statement pool* is a group of prepared statements that can be reused by an application. If you have an application that repeatedly executes the exact same SQL statements, statement pooling can improve performance because the database server does not have to repeatedly parse and create cursors for the same statement. In addition, the associated network round trips to the database server are avoided.

The drivers have an internal prepared statement pooling mechanism, which allows you to realize the performance benefits of statement pooling when you are not running from within an application server or another application that provides its own statement pooling. You can enable this driver-based internal statement pooling with the MaxPooledStatements connection property.

The DataDirect for JDBC drivers also support the DataDirect Statement Pool Monitor. You can use the Statement Pool Monitor to load statements into and remove statements from the statement pool as well as generate information to help you troubleshoot statement pooling performance. The Statement Pool Monitor is an integrated component of the driver, and you can manage statement pooling directly with DataDirect-specific methods. In addition, the Statement Pool Monitor can be enabled as a Java Management Extensions (JMX) MBean. When enabled as a JMX MBean, the Statement Pool Monitor can be used to manage statement pooling with standard JMX API calls, and it can easily be used by JMX-compliant tools, such as JConsole. To enable the Statement Pool Monitor as a JMX MBean, you must register the Statement Pool Monitor MBean with the RegisterStatementPoolMonitorMBean connection property.

Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.

See also

MaxPooledStatements on page 60
RegisterStatementPoolMonitorMBean on page 66

Connection pooling

Progess DataDirect for JDBC drivers support connection pooling using the DataDirect Connection Pool Manager. Typically, creating a connection is the most performance-expensive operations that an application performs. Connection pooling allows you to reuse connections rather than create a new one every time a driver needs to connect to the database. Further, connection pooling manages connection sharing across multiple user requests to maintain performance and reduce the number of new connections to be created.

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

3

Connection property descriptions

You can use connection properties to customize the driver for your environment. This section lists the connection properties supported by the driver and describes each property. You can use these connection properties with either the JDBC `DriverManager` or a JDBC `DataSource`. 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 setproperty(value). Connection property names are case-insensitive. For example, Password is the same as password.

Note: In a JDBC `DataSource`, string values must be enclosed in double quotation marks, for example, `setUser("abc@defcorp.com")`.

Note: The data type listed for each connection property is the Java data type used for the property value in a JDBC data source.

The following tables provide a summary of supported connection properties by functionality, including their corresponding data source methods, and their default values.

Required properties

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

Table 7: Required Properties

Property	Data Source Method	Default
ServerName on page 67	`setServerName`	None

Property	Data Source Method	Default
Password on page 61	setPassword	None
User on page 73	setUser	None

Data encryption properties

The following table summarizes connection properties which can be used to enable SSL.

Table 8: Data Encryption Properties

Property	Data Source Method	Default
CryptoProtocolVersion on page 50	setCryptoProtocolVersion	None
KeyPassword on page 57	setKeyPassword	None
KeyStore on page 57	setKeyStore	None
KeyStorePassword on page 58	setKeyStorePassword	None
HostNameInCertificate on page 53	setHostNameInCertificate	None
TrustStore on page 72	setTrustStore	None
TrustStorePassword on page 72	setTrustStorePassword	None
ValidateServerCertificate on page 74	setValidateServerCertificate	true

Proxy server properties

The following table summarizes proxy server connection properties.

Table 9: Proxy Server Properties

Property	Data Source Method	Default
ProxyHost on page 63	setProxyHost	Empty string
ProxyPassword on page 63	setProxyPassword	Empty string
ProxyPort on page 64	setProxyPort	0
ProxyUser on page 64	setProxyUser	Empty string

Web service properties

The following table summarizes Web service connection properties, including those related to timeouts.

Table 10: Web Service Properties

Property	Data Source Method	Default
LoginTimeout on page 59	`setLoginTimeout`	`0` (no timeout)
StmtCallLimit on page 70	`setStmtCallLimit`	`0` (no limit)
StmtCallLimitBehavior on page 71	`setStmtCallLimitBehavior`	`errorAlways`
WSFetchSize on page 75	`setWSFetchSize`	`0` (maximum of 1000 rows)
WSPoolSize on page 76	`setWSPoolSize`	`1`
WSRetryCount on page 76	`setWSRetryCount`	`5`
WSTimeout on page 77	`setWSTimeout`	`120` (seconds)

Data type properties

The following table summarizes connection properties which can be used to handle data types.

Table 11: Data Type Properties

Property	Data Source Method	Default
ConvertNull on page 49	`setConvertNull`	`1` (data type check is performed if column value is null)
JDBCBehavior on page 56	`setJDBCBehavior`	`1` (data types described using JDBC 3.0-equivalent data types)

Timeout properties

The following table summarizes timeout connection properties.

Table 12: Timeout Properties

Property	Data Source Method	Default
LoginTimeout on page 59	`setLoginTimeout`	`0` (no timeout)
WSRetryCount on page 76	`setWSRetryCount`	`0` (no retries for timed-out requests)
WSTimeout on page 77	`setWSTimeout`	`120` (seconds)

Statement pooling properties

The following table summarizes statement pooling connection properties.

Table 13: Statement Pooling Properties

Property	Data Source Method	Default
ImportStatementPool on page 54	`setImportStatementPool`	Empty string
MaxPooledStatements on page 60	`setMaxPooledStatements`	`0` (driver's internal prepared statement pooling is not enabled)
RegisterStatementPoolMonitorMBean on page 66	`setRegisterStatementPoolMonitorMBean`	`false`

Additional properties

The following table summarizes additional connection properties.

Table 14: Additional Properties

Property	Data Source Method	Default
DebugRecord on page 51	`setDebugRecord`	None
FetchSize on page 52	`setFetchSize`	`100` (rows)
InsensitiveResultSetBufferSize on page 55	`setInsensitiveResultSetBufferSize`	`2048` (KB of memory)
LogConfigFile on page 59	`setLogConfigFile`	`ddlogging.properties`

PortNumber on page 62	`setPortNumber`	For http connections:
		`80`
		For https connections:
		`443`

ReadAhead on page 65	`setReadAhead`	`0`
SpyAttributes on page 68	`setSpyAttributes`	None

For details, see the following topics:

- ConvertNull
- CryptoProtocolVersion
- DebugRecord
- FetchSize
- HostNameInCertificate
- ImportStatementPool
- InsensitiveResultSetBufferSize

ConvertNull

Purpose

Controls how data conversions are handled for null values.

Valid Values

`0 | 1`

Behavior

If set to `0`, the driver does not perform the data type check if the value of the column is null. This allows null values to be returned even though a conversion between the requested type and the column type is undefined.

If set to `1`, the driver checks the data type being requested against the data type of the table column that stores the data. If a conversion between the requested type and column type is not defined, the driver generates an "unsupported data conversion" exception regardless of whether the column value is NULL.

Data Source Method

`setConvertNull`

Default

`1`

Data Type

int

See Also

Additional properties

CryptoProtocolVersion

Purpose

Specifies a comma-separated list of the cryptographic protocols to use when accessing an HTTPS endpoint. When multiple protocols are specified, the driver uses the highest version supported by the server.

Valid Values

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

where:

`cryptographic_protocol`

is one of the following cryptographic protocols:

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

Caution: To avoid vulnerabilities associated with SSLv3 and SSLv2, good security practices recommend using TLSv1 or higher.

Example

If your server supports TLSv1.1 and TLSv1.2, you can specify acceptable cryptographic protocols with the following key-value pair:

```
CryptoProtocolVersion=TLSv1.1,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.
- When no value has been specified for CryptoProtocolVersion, the cryptographic protocol used depends on the highest protocol version supported by the server and the highest protocol version supported by the JDK. The driver uses the lower version of these two protocols to establish the SSL connection. Refer to the Jira documentation or contact your system administrator for information on which cryptographic protocols are supported.

Data Source Method

setCryptoProtocolVersion

Default

No default value

Data Type

String

See Also

- Data encryption on page 40
- Data encryption properties

DebugRecord

Purpose

Specifies the directory where the driver generates debug record files. When a value is specified, the driver records server requests and responses to set of files stored in this location. These files assist in troubleshooting by providing a method for Technical Support to reproduce and debug issues for Jira services.

Important: Debug record files may capture security-related headers, such as auth or token headers. Before sending Technical Support debug files, review the content to remove any confidential information that may have been recorded.

Valid Values

debug_record_folder

where:

debug_record_folder

is the location of the folder where the debeg record files are to be generated. For example, `C:/Temp/MyDebug Folder`.

Notes

- You must have write access to the specified directory.
- Contact Technical Support for more information.

Data Source Method

setDebugRecord

Default

None

See Also

- Contacting Technical Support on page 25
- Enabling Debug Record Mode on page 43

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 and WSFetchSize 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.
- The values specified for the WSFetchSize and FetchSize properties provide only a suggestion as to the number of rows to be processed. Jira will not exceed these values, but it will adjust page sizes to balance throughput amongst all its users. This behavior can result in different page sizes for successive queries.
- 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 Method

setFetchSize

Default

`100` (rows)

Data Type

Int

See Also

- Additional properties
- Performance considerations on page 39

HostNameInCertificate

Purpose

A host name for certificate validation when accessing an HTTPS endpoint and validation is enabled (`ValidateServerCertificate=1`). 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 a DNSName value does not exist in the SubjectAlternativeName or if the certificate does not have a SubjectAlternativeName, the driver compares the host name with the Common Name (CN) part of the certificate's Subject name. If the values do not match, the connection fails and the driver throws an exception.

Notes

- If the HostNameInCertificate is not specified, the driver automatically uses the value of the ServerName from the URL as the value for validating the certificate.
- 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 method

`setHostNameInCertificate`

Default

Empty string

Data type

String

See also

- Data encryption on page 40
- ValidateServerCertificate on page 74
- Data encryption properties

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.

If the driver cannot locate the specified file when establishing the connection, the connection fails and the driver throws an exception. See "Importing statements into a statement pool" for details about the import 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.

Data Source Method

`setImportStatementPool`

Default

empty string

Data Type

String

See Also

- Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.
- Statement pooling properties

InsensitiveResultSetBufferSize

Purpose

Determines the amount of memory that is used by the driver to cache insensitive result set data.

Valid Values

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

where:

x

is a positive integer that represents the amount of memory.

Behavior

If set to `-1`, the driver caches insensitive result set data in memory. If the size of the result set exceeds available memory, an OutOfMemoryException is generated. With no need to write result set data to disk, the driver processes the data efficiently.

If set to `0`, the driver caches insensitive result set data in memory, up to a maximum of 2 MB. If the size of the result set data exceeds available memory, then the driver pages the result set data to disk, which can have a negative performance effect. Because result set data may be written to disk, the driver may have to reformat the data to write it correctly to disk.

If set to *x*, the driver caches insensitive result set data in memory and uses this value to set the size (in KB) of the memory buffer for caching insensitive result set data. If the size of the result set data exceeds available memory, then the driver pages the result set data to disk, which can have a negative performance effect. Because the result set data may be written to disk, the driver may have to reformat the data to write it correctly to disk. Specifying a buffer size that is a power of 2 results in efficient memory use.

Data Source Method

`setInsensitiveResultSetBufferSize`

Default

2048

Data Type

int

See also

- Additional properties
- Performance considerations on page 39

JDBCBehavior

Purpose

Determines how the driver describes native data types that map to the following JDBC 4.0 data types: NCHAR, NVARCHAR, NLONGVARCHAR, NCLOB, and SQLXML.

Valid Values

0 | 1

Behavior

If set to 0, the driver describes the data types as JDBC 4.0 data types.

If set to 1, the driver describes the data types using JDBC 3.0-equivalent data types, regardless of JVM. This allows your application to continue using JDBC 3.0 types in a Java SE 8 or higher environment.

Data Source Method

setJDBCBehavior

Default

1

Data Type

int

See Also

- Data type properties

KeyPassword

Purpose

Specifies the password that is used to access the individual keys in the keystore file when accessing an HTTPS endpoint and SSL client authentication is enabled on the 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 Method

`setKeyPassword`

Default

None

Data Type

String

See Also

- Data encryption properties

KeyStore

Purpose

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

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.

Valid Values

string

where:

string

is a valid directory of a keystore file.

Notes

- The keystore and truststore files can be the same file.

Data Source Method

`setKeyStore`

Default

None

Data Type

String

See Also

- Data encryption properties

KeyStorePassword

Purpose

Specifies the password that is used to access the keystore file when accessing an HTTPS endpoint 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.

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.

Valid Values

string

where:

string

is a valid password.

Notes

- The keystore and truststore files can be the same file.

Data Source Method

setKeyStorePassword

Default

None

Data Type

String

See Also

- Data encryption properties

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.

Data Source Method

`setLogConfigFile`

Default

`ddlogging.properties`

Data Type

String

See Also

Refer to "Using Java logging" in the *Progress DataDirect for JDBC Drivers Reference*.

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

`0` | *x*

where:

x

is a positive integer that represents a number of seconds.

Behavior

If set to `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 Method

`setLoginTimeout`

Default

`0`

Data Type

int

See Also

- Timeout properties

MaxPooledStatements

Purpose

The maximum number of pooled prepared statements for this connection. Setting MaxStatements to an integer greater than zero (0) enables the driver's internal prepared statement pooling, which is useful when the driver is not running from within an application server or another application that provides its own prepared statement pooling.

Valid Values

`0` | *x*

where

x

is a positive integer that represents a number of pooled prepared statements.

Behavior

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

If set to *x*, the driver enables the DataDirect Statement Pool and uses the specified value to cache a certain number of 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.

Data Source Method

`setMaxPooledStatements`

Default

`0`

Data Type

int

See Also

- Statement pooling properties

Password

Description

Specifies the password or API token used to connect to your Jira service.

Important: Setting the password property using a data source is not recommended. The data source persists all properties, including the Password property, in clear text.

Valid Values

`password_token`

where:

`password_token`

is a valid password or API token. The password is case-sensitive.

Notes

- For information on generating a token, refer to the Jira documentation.

Data Source Method

`setPassword`

Default

None

Data Type

String

See Also

- Required properties

PortNumber

Purpose

Specifies the TCP port of the primary REST service server that is listening for connections.

Valid values

port

where:

port

is the port number.

Data source method

`setPortNumber`

Default

For HTTP connections:

`80`

For HTTPS connections:

`443`

Data type

Int

See Also

- Additional properties

ProxyHost

Description

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. See "Using IP addresses" for details about using these formats.

Data Source Method

`setProxyHost`

Default

Empty string

See Also

- IP addresses on page 38
- Connecting through a proxy server on page 38
- Proxy server properties

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 Method

`setProxyPassword`

Default

Empty string

See Also

- Connecting through a proxy server on page 38
- Proxy server properties

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 Method

`setProxyPort`

Default

`0`

See Also

- Connecting through a proxy server on page 38
- Proxy server properties

ProxyUser

Purpose

Specifies the 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 Method

`setProxyUser`

Default

Empty string

See Also

- Connecting through a proxy server on page 38
- Proxy server properties

ReadAhead

Purpose

Specifies the maximum number of fetch requests the driver issues in parallel. By default, the driver queues the next page when processing the current page. This option allows you to fetch multiple requests simultaneously, thereby improving throughput and performance.

Caution: Due to potential impacts to other users on the network, we strongly recommend specifying only smaller values for this option. For example, in fully optimized environments, which include exceptionally fast connections and low latency, we recommend a setting of no higher than `5`. For typical environments, this value is sometimes lower.

Valid Values

`0` | *x*

where:

x

is the maximum number of fetch requests the driver issues in parallel.

Behavior

If set to `0`, the driver queues the next page while processing the current page.

If set to *x*, the driver executes fetch requests as they are issued until the number of active parallel-requests equals the specified value. When that threshold is met, the driver waits until a the results of a request are processed before requesting the next page of data.

Notes

- Specifying larger values for this option generally improves performance; however, with the following warnings:
  - Larger values can increase the load on the server, which may adversely affect performance of other users. If you encounter issues, decrease the value specified for this option.
  - Larger values may result in unnecessary requests if your application only requires the first few rows of results. This may force the Jira server to use smaller page sizes, thereby negating the benefits.

Data Source Method

`setReadAhead`

Default

`0`

Data Type

int

See Also

- Additional properties

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.

Data Source Method

setRegisterStatementPoolMonitorMBean

Default

`false`

Data Type

Boolean

See Also

- Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for further details.
- Statement pooling properties
- MaxPooledStatements on page 60

ServerName

Purpose

Specifies the base URL of the Jira service to which you want to issue requests.

Valid Values

`string`

where:

`string`

specifies the URL of the Jira service to which you want to issue requests. For example, `https://mycompany.atlassian.net`.

Notes

- Specifying an HTTPS URL for this property enables data encryption. See "Data Encryption" for details.

Data Source Method

`setServerName`

Default

None

Data Type

String

See Also

- Data encryption on page 40
- Required properties

SpyAttributes

Purpose

Enables DataDirect Spy to log detailed information about calls 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).
load=*classname*	Loads the driver specified by *classname*.
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.

Attribute	Description
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.
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 yes.

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`.

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)
```

Data Source Method

`setSpyAttributes`

Default

None

See Also

Refer to "Tracking JDBC Calls with DataDirect Spy" in the *Progress DataDirect for JDBC Drivers Reference* for more information about using DataDirect Spy.

StmtCallLimit

Purpose

Specifies the maximum number of Web service calls the driver can make when executing any single SQL statement or metadata query.

Valid Values

`0` | *x*

where:

x

is a positive integer that defines the maximum number of Web service calls the driver can make when executing any single SQL statement or metadata query.

Behavior

If set to `0`, there is no limit.

If set to x, the driver uses this value to set the maximum number of Web service calls on a single connection that can be made when executing a SQL statement. This limit can be overridden by changing the STMT_CALL_LIMIT session attribute using the ALTER SESSION statement. For example, the following statement sets the statement call limit to 10 Web service calls:

```
ALTER SESSION SET STMT_CALL_LIMIT=10
```

If the Web service call limit is exceeded, the behavior of the driver depends on the value specified for the StmtCallLimitBehavior property.

Data Source Method

`setStmtCallLimit`

Default

`0` (No limit)

Data Type

int

See Also

- Web service properties

StmtCallLimitBehavior

Purpose

Specifies the behavior of the driver when the maximum Web service call limit specified by the StmtCallLimit property is exceeded.

Valid Values

`errorAlways` | `returnResults`

Behavior

If set to `errorAlways`, the driver generates an exception if the maximum Web service call limit is exceed.

If set to `returnResults`, the driver returns any partial results it received prior to the call limit being exceeded. The driver generates a warning that not all of the results were fetched.

Data Source Method

`setStmtCallLimitBehavior`

Default

`errorAlways`

Data Type

String

See Also

- Web service properties

TrustStore

Purpose

The directory that contains the truststore file and the truststore file name to be used when accessing an HTTPS endpoint and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

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 `ValidateServerCertificate=false`.

Valid Values

`string`

where:

`string`

is the directory of the truststore file.

Data Source Method

`setTrustStore`

Default

None

Data Type

String

See Also

- Performance considerations on page 39
- Data encryption properties

TrustStorePassword

Purpose

The password that is used to access the truststore file when accessing an HTTPS endpoint and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

This value overrides the password of the truststore file that is specified by the javax.net.ssl.trustStorePassword Java system property. If this property is not specified, the truststore password is specified by the javax.net.ssl.trustStorePassword Java system property.

This property is ignored if `ValidateServerCertificate=false`.

Valid Values

string

where:

string

is a valid password for the truststore file.

Data Source Method

`setTrustStorePassword`

Default

None

Data Type

String

See Also

- Performance considerations on page 39
- Data encryption properties

User

Purpose

Specifies the user name that is used to connect to your Jira service.

Valid Values

string

where:

`string`

is a valid user name. The user name is case-insensitive.

Data Source Method

`setUser`

Default

None

Data Type

String

See Also

- Required properties

ValidateServerCertificate

Purpose

Determines whether the driver validates the certificate that is sent by the Jira service server when accessing an HTTPS endpoint. When using SSL server authentication, any certificate that is sent by the server must be issued by a trusted Certificate Authority (CA).

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.

Valid values

`true` | `false`

Behavior

If set to `true`, the driver validates the certificate that is sent by the Jira service. Any certificate from the service 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 service 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.

Data source method

`setValidateServerCertificate`

Default

`true`

Data type

Boolean

See also

- Data encryption properties

WSFetchSize

Purpose

Specifies the number of rows of data the driver attempts to fetch for each JDBC call when paging is enabled for an endpoint.

Valid Values

`0` | *x*

where:

x

is a positive integer that defines a number of rows.

Behavior

If set to `0`, the driver attempts to fetch up to a maximum of 1000 rows. This value typically provides the maximum throughput.

If set to *x*, the driver attempts to fetch up to the maximum of the specified number of rows. Setting the value lower than the maximum can reduce the response time for returning the initial data. Consider using a smaller WSFetch size for interactive applications only.

Notes

- WSFetchSize and 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.
- The values specified for the WSFetchSize and FetchSize properties provide only a suggestion as to the number of rows to be processed. Jira will not exceed these values, but it will adjust page sizes to balance throughput amongst all its users. This behavior can result in different page sizes for successive queries.

Data Source Method

`setWSFetchSize`

Default

`0` (up to a maximum of 1000 rows)

Data Type

Int

See Also

- Web service properties
- Performance considerations on page 39

WSPoolSize

Purpose

Specifies the maximum number of sessions the driver uses. This allows the driver to have multiple web service requests active when multiple JDBC connections are open, thereby improving throughput and performance.

Valid Values

x

where:

x

is the number of sessions the driver uses to distribute calls. This value should not exceed the number of sessions permitted by your account.

Notes

- You can improve performance by increasing the number of sessions specified by this property. By increasing the number of sessions the driver uses, you can improve throughput by distributing calls across multiple sessions when multiple connections are active.
- The maximum number of sessions is determined by the setting of WSPoolSize for the connection that initiates the session. For subsequent connections to an active session, the setting is ignored and a warning is returned. To change the maximum number of sessions, close all connections using the driver; then, open a new connection with desired limit specified for this property.

Data Source Method

`setWSPoolSize`

Default

`1`

Data Type

Int

See also

- Web service properties
- Performance considerations on page 39

WSRetryCount

Description

The number of times the driver retries a timed-out Select request. The timeout period is specified by the WSTimeout connection property.

Valid Values

`0` | *x*

where:

x

is a positive integer.

Behavior

If set to `0`, the driver does not retry timed-out requests after the initial unsuccessful attempt.

If set to *x*, the driver retries the timed-out request the specified number of times.

Data Source Method

`setWSRetryCount`

Default

5

Data Type

Int

See Also

- Web service properties

WSTimeout

Purpose

Specifies the time, in seconds, that the driver waits for a response to a Web service request.

Valid Values

`0` | *x*

where:

x

is a positive integer that defines the number of seconds the driver waits for a response to a Web service request.

Behavior

If set to `0`, the driver waits indefinitely for a response; there is no timeout.

If set to *x*, the driver uses the value as the default timeout for any statement created by the connection.

If a Select request times out and WSRetryCount is set to retry timed-out requests, the driver retries the request the specified number of times.

Data Source Method

`setWSTimeout`

Default

`120` (seconds)

Data Type

Int

See Also

- Web service properties

4

Supported SQL statements and extensions

The driver provides support for the SQL statements and the SQL extensions described in this section. SQL extensions are denoted by an (EXT) in the topic title.

For details, see the following topics:

Alter Session (EXT)

Purpose

Changes various attributes of a local or remote session. A local session maintains the state of the overall connection. A remote session maintains the state that pertains to a particular remote data source connection.

Syntax

`ALTER SESSION SET` *`attribute_name=value`*

where:

attribute_name

specifies the name of the attribute to be changed. Attributes apply to either local or remote sessions.

value

specifies the value for that attribute.

The following table lists the local and remote session attributes, and provides descriptions of each.

Table 15: Alter Session Attributes

Attribute Name	Session Type	Description
Current_Schema	Local	Sets the current schema for the local session. The current schema is the schema used when an identifier in a SQL statement is unqualified. The string value must be the name of a schema visible in the local session. For example: `ALTER SESSION SET CURRENT_SCHEMA=`
Stmt_Call_Limit	Local	Sets the maximum number of Web service calls the driver can make in executing a statement. Setting the Stmt_Call_Limit attribute has the same effect as setting the Statement Call Limit connection option. It sets the default Web service call limit used by any statement on the connection. Executing this command on a statement overrides the previously set Statement Call Limit for the connection. The value specified must be a positive integer or `0`. The value `0` means that no call limit exists. For example: `ALTER SESSION SET STMT_CALL_LIMIT=150`

Ws_Call_Count	Remote	Resets the Web service call count of a remote session to the value specified. The value must be `0` or a positive integer. WS_Call_Count represents the total number of Web service calls made to the remote data source instance for the current session. For example:
		`ALTER SESSION SET .WS_CALL_COUNT=0`
		The current value of WS_Call_Count can be obtained by referring to the System_Remote_Sessions system table (see SYSTEM_REMOTE_SESSIONS Catalog Table for details). For example:
		`SELECT * from information_schema.system_remote_sessions WHERE session_id = cursessionid()`

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.

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 91

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 following table lists supported aggregate functions.

Note: Doubly nested aggregates, such as `SUM(COUNT(col1))`, are currently not permitted by the driver.

Table 16: 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. **Note:** The driver does not support `COUNT(DISTINCT *)`. For example, `SELECT COUNT(DISTINCT *) FROM mytable` results in a syntax error.
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

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.

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
```

Note: The `ON` clause in a join expression must evaluate to a true or false value.

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 99

Where clause

Purpose

Specifies the conditions that rows must meet to be retrieved.

Syntax

`WHERE` *`expr1 rel_operator expr2`*

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

SQL expressions on page 91
Subqueries on page 99

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.

See also

SQL expressions on page 91
Subqueries on page 99

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
```

See also

SQL expressions on page 91
Subqueries on page 99

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 91

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
```

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 driver supports both unquoted and quoted identifiers. An unquoted identifier must start with an ASCII alpha character and can be followed by zero

Quoted identifiers must be enclosed in double quotation marks (""). A quoted identifier can contain any Unicode character including the space character. 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.

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 17: Literal Syntax Examples

SQL Type	Literal Syntax	Example
BIGINT	*n* where *n* is any valid integer value in the range of the INTEGER data type	`12 or -34 or 0`
BOOLEAN	Min Value: 0 Max Value: 1	`0` `1`
DATE	DATE'*date*'	`'2010-05-21'`
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`

SQL Type	Literal Syntax	Example
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.

Note: Numeric operators are restricted to numeric types. Numeric operators do not support non-numeric types.

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 18: Arithmetic Operators

Operator	Purpose	Example
+ -	Denotes a positive or negative expression. These are unary operators.	`SELECT * FROM emp WHERE comm = -1`

Operator	Purpose	Example
* /	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 19: 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 driver considers the UNKNOWN result as FALSE.

The following table lists the supported comparison operators.

Table 20: 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`

Operator	Purpose	Example
LIKE	% and _ wildcards can be used to search for a pattern in a column. The percent sign denotes zero, one, or multiple characters, while the underscore denotes a single character. The right-hand side of a LIKE expression must evaluate to a string or binary.	`SELECT * FROM emp WHERE ENAME LIKE 'J%'`

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 21: 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 22: 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 driver supports JDBC core functions. For details, refer to "JDBC support" in the *Progress DataDirect for JDBC Drivers Reference*.

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 23: 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`
Range	Tests for inclusion in a range. `[NOT] BETWEEN`

Condition	Description
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)
```