H2 Database Engine

Version 1.1.118 (2009-09-04)

Table of Contents

Quickstart

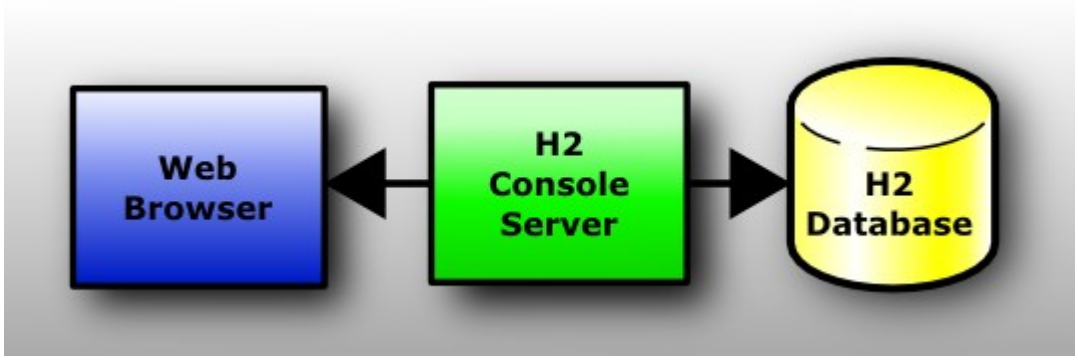
Embedding H2 in an Application

This database can be used in embedded mode, or in server mode. To use it in embedded mode, you need to:

- Add the h2*.jar to the classpath (H2 does not have any dependencies)
- Use the JDBC driver class: org.h2.Driver
- The database URL jdbc:h2:~/test opens the database 'test' in your user home directory
- A new database is automatically created

The H2 Console Application

The Console lets you access a SQL database using a browser interface.



If you don't have Windows XP, or if something does not work as expected, please see the detailed description in the Tutorial .

Step-by-Step

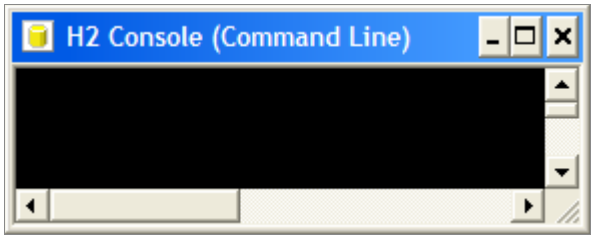
Installation

Install the software using the Windows Installer (if you did not yet do that).

Start the Console

Click [Start], [All Programs], [H2], and [H2 Console (Command Line)]:

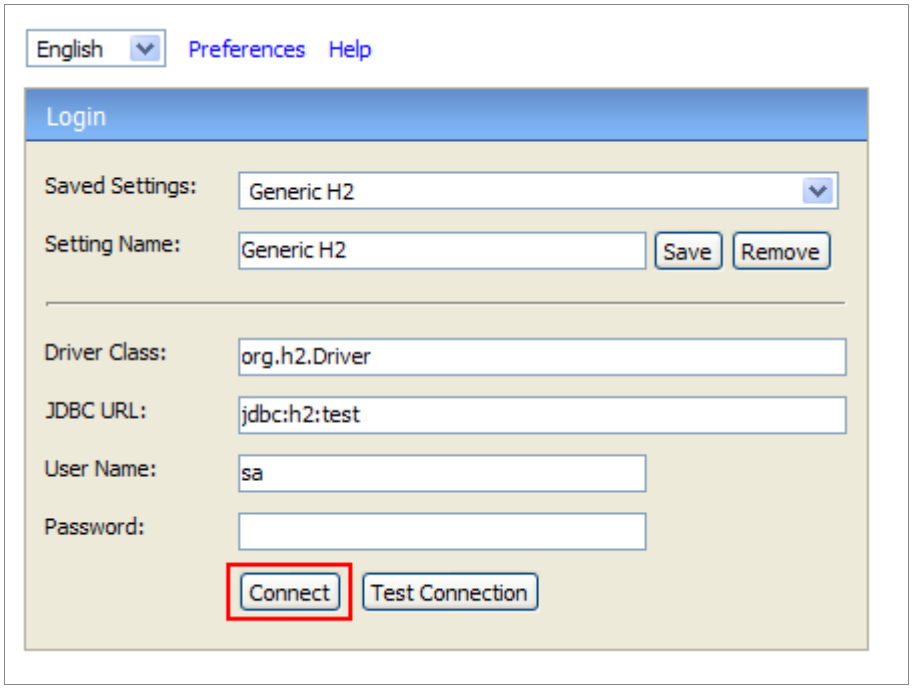
A new console window appears:



Also, a new browser page should open with the URL http://localhost:8082 . You may get a security warning from the firewall. If you don't want other computers in the network to access the database on your machine, you can let the firewall block these connections. Only local connections are required at this time.

Login

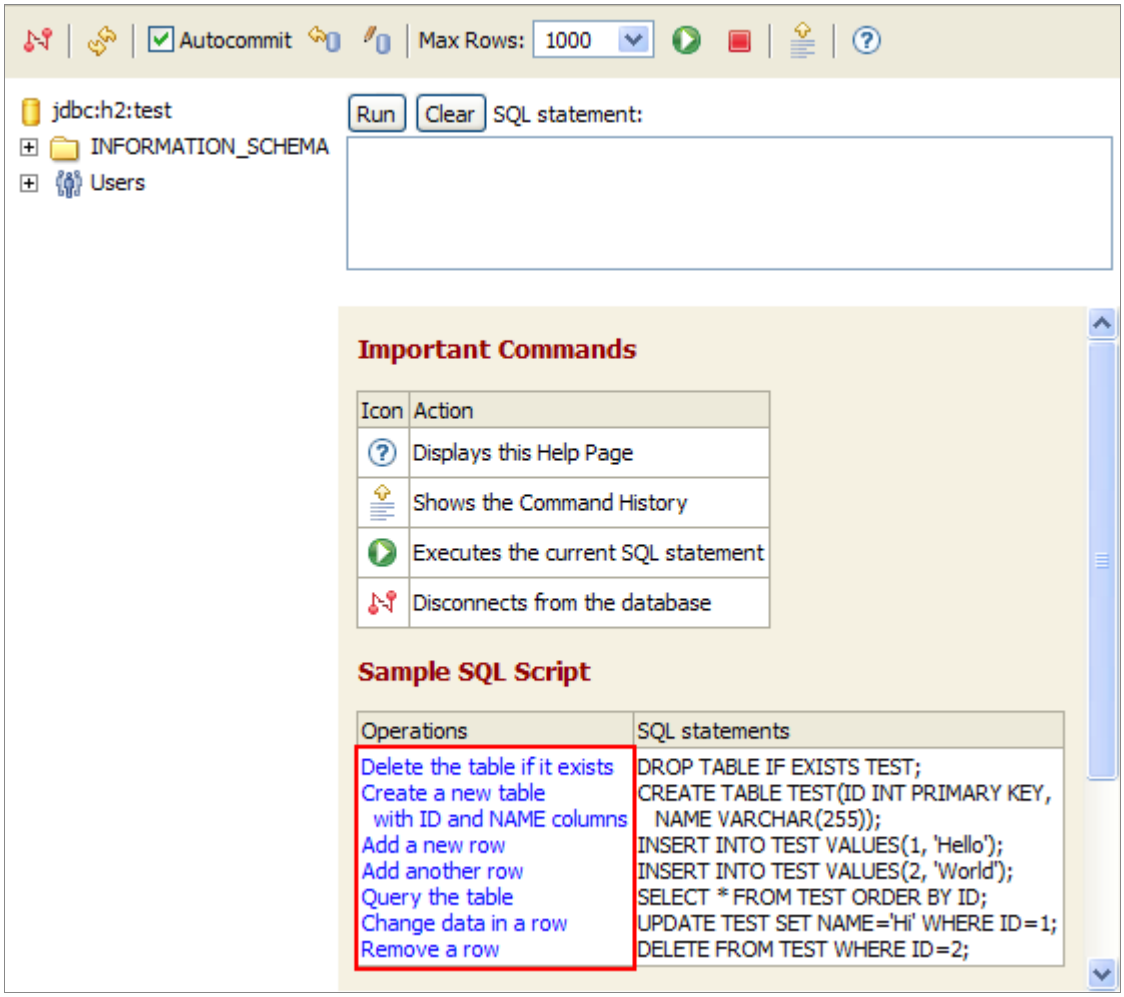
Select [Generic H2] and click [Connect]:



You are now logged in.

Sample

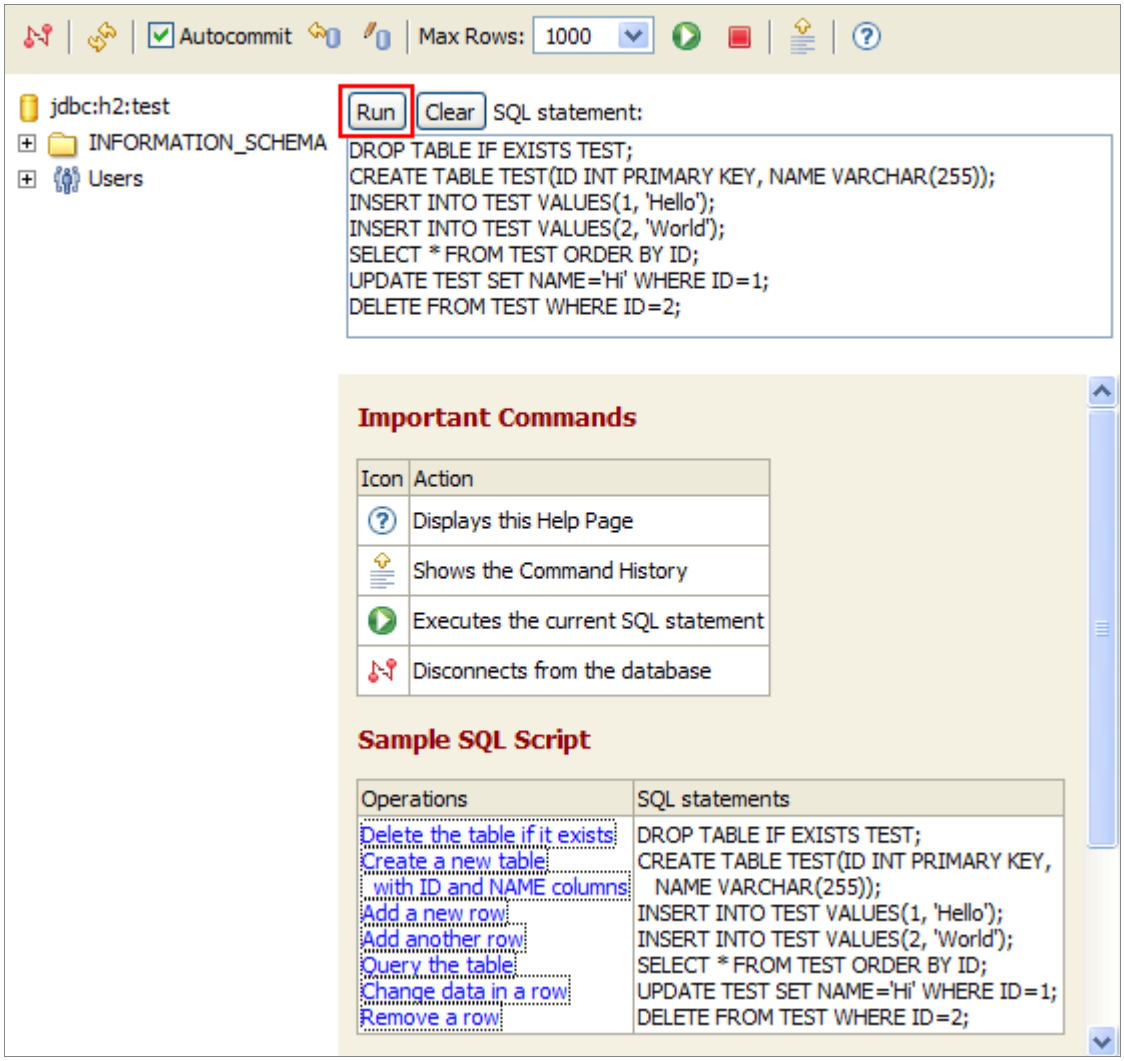
Click on the [Sample SQL Script]:



The SQL commands appear in the command area.

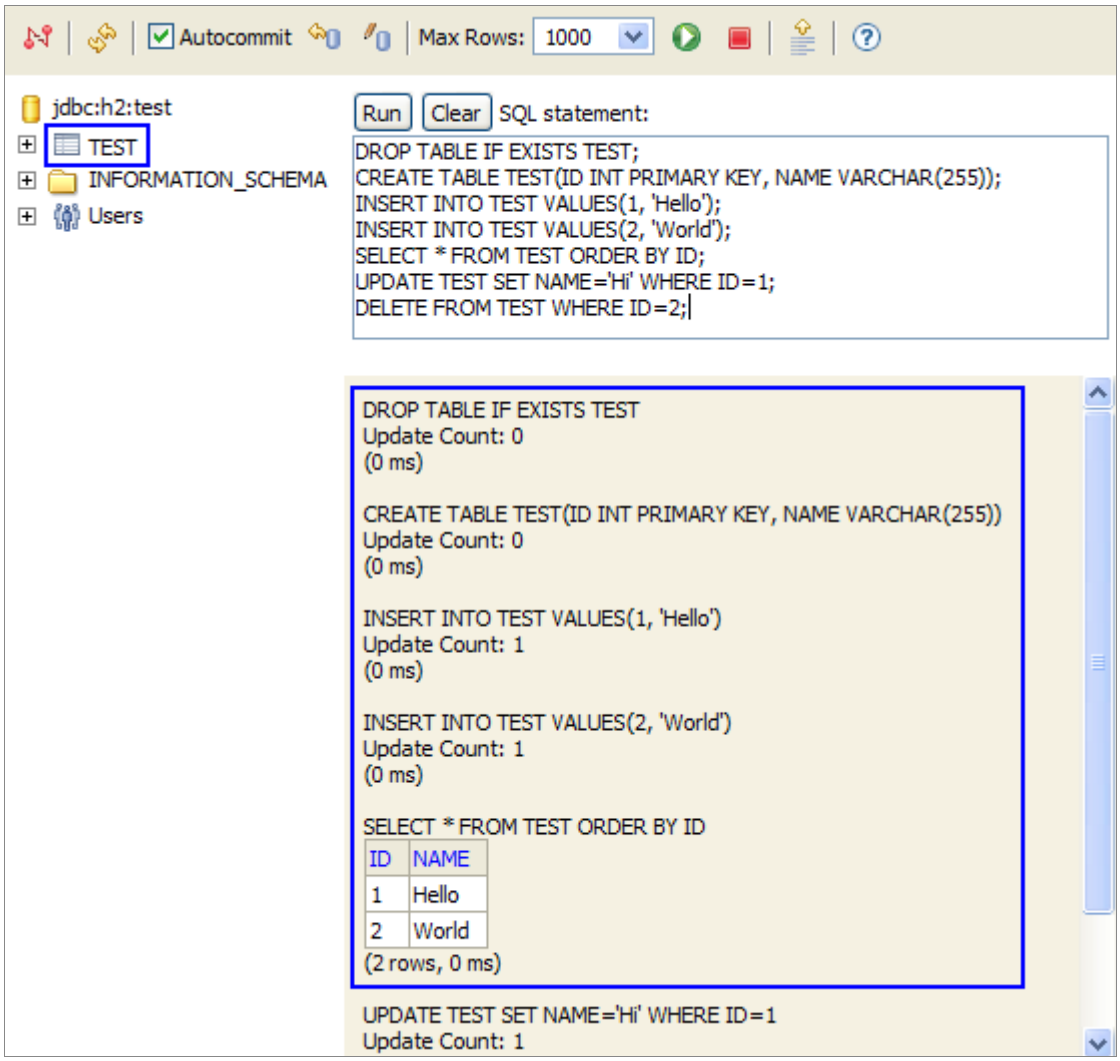
Execute

Click [Run]



On the left side, a new entry TEST is added below the database icon. The operations and results of the statements are shown

below the script.



Disconnect

Click on [Disconnect]:

to close the connection.

End

Close the console window. For more information, see the Tutorial .

Installation

Requirements

To run the database, the following minimum software stack is known to work:

- Windows XP, Mac OS X, or Linux
- Recommended Windows file system: NTFS (FAT32 supports files up to 4 GB)
- Sun JDK 1.5 or newer
- Mozilla Firefox 1.5 or newer

Supported Platforms

As this database is written in Java, it can run on many different platforms. It is tested with Java 1.5 and 1.6 but can also be compiled to native code using GCJ. The source code does not use features of Java 1.6. Currently, the database is developed and tested on Windows XP and Mac OS X using the Sun JDK 1.5, but it also works in many other operating systems and using other Java runtime environments.

Installing the Software

To install the software, run the installer or unzip it to a directory of your choice.

Directory Structure

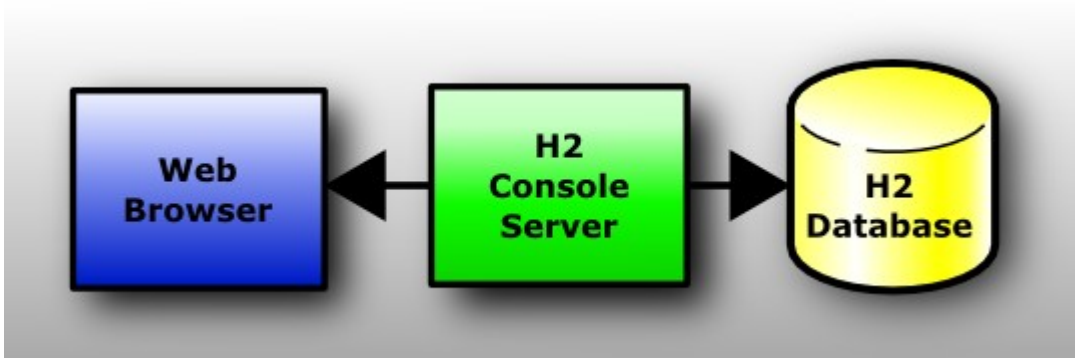
After installing, you should get the following directory structure:

Directory	Contents
bin	JAR and batch files
docs	Documentation
docs/html	HTML pages
docs/javadoc	Javadoc files
ext	External dependencies (downloaded when building)
service	Tools to run the database as a Windows Service
src	Source files

Tutorial

Starting and Using the H2 Console

The H2 Console application lets you access a SQL database using a browser interface. This can be a H2 database, or another database that supports the JDBC API.



This is a client / server application, so both a server and a client (a browser) are required to run it.

Depending on your platform and environment, there are multiple ways to start the application:

OS	Start
Windows	Click [Start], [All Programs], [H2], and [H2 Console (Command Line)] When using the Sun JDK 1.5, a window with the title 'H2 Console ' should appear. When using the Sun JDK 1.6, an icon will be added to the system tray:  If you don't get the window and the system tray icon, then maybe Java is not installed correctly (in this case, try another way to start the application). A browser window should open and point to the Login page at http://localhost:8082 .
Windows	Open a file browser, navigate to h2/bin, and double click on h2.bat. A console window appears. If there is a problem, you will see an error message in this window. A browser window will open and point to the Login page (URL: http://localhost:8082).
Any	Double click on the h2*.jar file. This only works if the .jar suffix is associated with java.
Any	Open a console window, navigate to the directory 'h2/bin' and type: `java -cp h2*.jar org.h2.tools.Server`

Firewall

If you start the server, you may get a security warning from the firewall (if you have installed one). If you don't want other computers in the network to access the application on your machine, you can let the firewall block those connections. The connection from the local machine will still work. Only if you want other computers to access the database on this computer, you need allow remote connections in the firewall.

It has been reported that when using Kaspersky 7.0 with firewall, the H2 Console is very slow when connecting over the IP address. A workaround is to connect using localhost, however this only works on the local machine.

A small firewall is already built into the server: other computers may not connect to the server by default. To change this, go to 'Preferences' and select 'Allow connections from other computers'.

Testing Java

To find out which version of Java is installed, open a command prompt and type:

```
java -version
```

If you get an error message, you may need to add the Java binary directory to the path environment variable.

Error Message 'Port may be in use'

You can only start one instance of the H2 Console, otherwise you will get the following error message: The Web server could not be started. Possible cause: another server is already running... . It is possible to start multiple console applications on the same computer (using different ports), but this is usually not required as the console supports multiple concurrent connections.

Using another Port

If the port is in use by another application, you may want to start the H2 Console on a different port. This can be done by changing the port in the file .h2.server.properties. This file is stored in the user directory (for Windows, this is usually in "Documents and Settings/<username>"). The relevant entry is webPort.

Connecting to the Server using a Browser

If the server started successfully, you can connect to it using a web browser. JavaScript needs to be enabled. If you started the server on the same computer as the browser, open the URL http://localhost:8082 . If you want to connect to the application from another computer, you need to provide the IP address of the server, for example: http://192.168.0.2:8082 . If you enabled SSL on the server side, the URL needs to start with https:// .

Multiple Concurrent Sessions

Multiple concurrent browser sessions are supported. As that the database objects reside on the server, the amount of concurrent work is limited by the memory available to the server application.

Login

At the login page, you need to provide connection information to connect to a database. Set the JDBC driver class of your database, the JDBC URL, user name and password. If you are done, click [Connect].

You can save and reuse previously saved settings. The settings are stored in a properties file (see Settings of the H2 Console).

Error Messages

Error messages in are shown in red. You can show/hide the stack trace of the exception by clicking on the message.

Adding Database Drivers

Additional database drivers can be registered by adding the Jar file location of the driver to the environment variables H2DRIVERS or CLASSPATH. Example (Windows): to add the database driver library C:\Programs\hsqldb\lib\hsqldb.jar, set the environment variable H2DRIVERS to C:\Programs\hsqldb\lib\hsqldb.jar.

Multiple drivers can be set; each entry needs to be separated with a ';' (Windows) or ':' (other operating systems). Spaces in the path names are supported. The settings must not be quoted.

Using the H2 Console

The H2 Console application has three main panels: the toolbar on top, the tree on the left, and the query / result panel on the right. The database objects (for example, tables) are listed on the left panel. Type in a SQL command on the query panel and click 'Run'. The result of the command appears just below the command.

Inserting Table Names or Column Names

The table name and column names can be inserted in the script by clicking them in the tree. If you click on a table while the query is empty, a 'SELECT * FROM ...' is added as well. While typing a query, the table that was used is automatically expanded in the tree. For, example if you type 'SELECT * FROM TEST T WHERE T.' then the table TEST is automatically expanded in the tree.

Disconnecting and Stopping the Application

To log out of the database, click 'Disconnect' in the toolbar panel. However, the server is still running and ready to accept new sessions.

To stop the server, right click on the system tray icon and select [Exit]. If you don't have the system tray icon, navigate to [Preferences] and click [Shutdown], press [Ctrl]+[C] in the console where the server was started (Windows), or close the console window.

Settings of the H2 Console

The settings of the H2 Console are stored in a configuration file called .h2.server.properties in you user home directory. For Windows installations, the user home directory is usually C:\Documents and Settings\[username] . The configuration file contains the settings of the application and is automatically created when the H2 Console is first started.

Connecting to a Database using JDBC

To connect to a database, a Java application first needs to load the database driver, and then get a connection. A simple way to do that is using the following code:

```
import java.sql.*;
public class Test {
    public static void main(String[] a)
            throws Exception {
        Class.forName("org.h2.Driver");
        Connection conn = DriverManager.
            getConnection("jdbc:h2:~/test", "sa", "");
        // add application code here
        conn.close();
    }
}
```

This code first loads the driver (Class.forName(...)) and then opens a connection (using DriverManager.getConnection()). The driver name is "org.h2.Driver" . The database URL always needs to start with jdbc:h2: to be recognized by this database. The second parameter in the getConnection() call is the user name ('sa' for System Administrator in this example). The third parameter is the password. In this database, user names are not case sensitive, but passwords are.

Creating New Databases

By default, if the database specified in the URL does not yet exist, a new (empty) database is created automatically. The user that created the database automatically becomes the administrator of this database.

Using the Server

H2 currently supports three server: a web server (for the H2 Console), a TCP server (for client/server connections) and an PG server (for PostgreSQL clients). The servers can be started in different ways, one is using the server tool.

Starting the Server Tool from Command Line

To start the server tool from the command line with the default settings, run:

```
java -cp h2*.jar org.h2.tools.Server
```

This will start the server tool with the default options. To get the list of options and default values, run:

```
java -cp h2*.jar org.h2.tools.Server -?
```

There are options available to use other ports, and start or not start parts. For details, see the API documentation of the server tool.

Connecting to the TCP Server

To remotely connect to a database using the TCP server, use the following driver and database URL:

- JDBC driver class: org.h2.Driver
- Database URL: jdbc:h2:tcp://localhost/~/test

For details about the database URL, see also in Features.

Starting the TCP Server within an Application

Servers can also be started and stopped from within an application. Sample code:

```
import org.h2.tools.Server;
...
// start the TCP Server
Server server = Server.createTcpServer(args).start();
...
// stop the TCP Server
server.stop();
```

Stopping a TCP Server from Another Process

The TCP server can be stopped from another process. To stop the server from the command line, run:

```
java org.h2.tools.Server -tcpShutdown tcp://localhost:9092
```

To stop the server from a user application, use the following code:

```
org.h2.tools.Server.shutdownTcpServer("tcp://localhost:9094");
```

This function will only stop the TCP server. If other server were started in the same process, they will continue to run. To avoid recovery when the databases are opened the next time, all connections to the databases should be closed before calling this method. To stop a remote server, remote connections must be enabled on the server. Shutting down a TCP server can be protected using the option -tcpPassword (the same password must be used to start and stop the TCP server).

Using Hibernate

This database supports Hibernate version 3.1 and newer. You can use the HSQLDB Dialect, or the native H2 Dialect. Unfortunately the H2 Dialect included in Hibernate is buggy. A patch for Hibernate has been submitted. The dialect for the newest version of Hibernate is also available at src/tools/org/hibernate/dialect/H2Dialect.java.txt. You can rename it to H2Dialect.java and include this as a patch in your application.

Using TopLink and Glassfish

To use H2 with Glassfish (or Sun AS), set the Datasource Classname to org.h2.jdbcx.JdbcDataSource . You can set this in the GUI at Application Server - Resources - JDBC - Connection Pools, or by editing the file sun-resources.xml : at element jdbc-connection-pool , set the attribute datasource-classname to org.h2.jdbcx.JdbcDataSource .

The H2 database is compatible with HSQLDB and PostgreSQL. To take advantage of H2 specific features, use the H2Platform . The source code of this platform is included in H2 at src/tools/oracle/toplink/essentials/platform/database/DatabasePlatform.java.txt . You will need to copy this file to your application, and rename it to .java. To enable it, change the following setting in persistence.xml:

```
<property
   name="toplink.target-database"
   value="oracle.toplink.essentials.platform.database.H2Platform"/>
```

In old versions of Glassfish, the property name is toplink.platform.class.name .

Using Databases in Web Applications

There are multiple ways to access a database from within web applications. Here are some examples if you use Tomcat or JBoss.

Embedded Mode

The (currently) simplest solution is to use the database in the embedded mode, that means open a connection in your application when it starts (a good solution is using a Servlet Listener, see below), or when a session starts. A database can be accessed from multiple sessions and applications at the same time, as long as they run in the same process. Most Servlet Containers (for example Tomcat) are just using one process, so this is not a problem (unless you run Tomcat in clustered mode). Tomcat uses multiple threads and multiple classloaders. If multiple applications access the same database at the same

time, you need to put the database jar in the shared/lib or server/lib directory. It is a good idea to open the database when the web application starts, and close it when the web application stops. If using multiple applications, only one (any) of them needs to do that. In the application, an idea is to use one connection per Session, or even one connection per request (action). Those connections should be closed after use if possible (but it's not that bad if they don't get closed).

Server Mode

The server mode is similar, but it allows you to run the server in another process.

Using a Servlet Listener to Start and Stop a Database

Add the h2*.jar file to your web application, and add the following snippet to your web.xml file (between the 'context-param' and the 'filter' section):

```
<listener>
    <listener-class>org.h2.server.web.DbStarter</listener-class>
</listener>
```

For details on how to access the database, see the file DbStarter.java. By default the DbStarter listener opens an embedded connection using the database URL 'jdbc:h2:~/test', user name 'sa', and password 'sa'. If you want to use this connection within your servlet, you can access as follows:

```
Connection conn = getServletContext().getAttribute("connection");
```

The DbStarter can also start the TCP server, however this is disabled by default. To enable it, use the parameter db.tcpServer in the file web.xml. Here is the complete list of options. These options need to be placed between the 'description' tag and the 'listener' / 'filter' tags:

```
<context-param>
    <param-name>db.url</param-name>
    <param-value>jdbc:h2:~/test</param-value>
</context-param>
<context-param>
    <param-name>db.user</param-name>
    <param-value>sa</param-value>
</context-param>
<context-param>
    <param-name>db.password</param-name>
    <param-value>sa</param-value>
</context-param>
<context-param>
    <param-name>db.tcpServer</param-name>
    <param-value>-tcpAllowOthers</param-value>
</context-param>
```

When the web application is stopped, the database connection will be closed automatically. If the TCP server is started within the DbStarter, it will also be stopped automatically.

Using the H2 Console Servlet

The H2 Console is a standalone application and includes its own web server, but it can be used as a servlet as well. To do that, include the the h2 jar file in your application, and add the following configuration to your web.xml:

```
<servlet>
    <servlet-name>H2Console</servlet-name>
    <servlet-class>org.h2.server.web.WebServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>H2Console</servlet-name>
    <url-pattern>/console/*</url-pattern>
```

```
</servlet-mapping>
```

For details, see also src/tools/WEB-INF/web.xml .

To create a web application with just the H2 Console, run the following command:

```
build warConsole
```

CSV (Comma Separated Values) Support

The CSV file support can be used inside the database using the functions CSVREAD and CSVWRITE, or it can be used outside the database as a standalone tool.

Writing a CSV File from Within a Database

The built-in function CSVWRITE can be used to create a CSV file from a query. Example:

```
CREATE TABLE TEST(ID INT, NAME VARCHAR);
INSERT INTO TEST VALUES(1, 'Hello'), (2, 'World');
CALL CSVWRITE('test.csv', 'SELECT * FROM TEST');
```

Reading a CSV File from Within a Database

A CSV file can be read using the function CSVREAD. Example:

```
SELECT * FROM CSVREAD('test.csv');
```

Writing a CSV File from a Java Application

The CSV tool can be used in a Java application even when not using a database at all. Example:

```
import java.sql.*;
import org.h2.tools.Csv;
import org.h2.tools.SimpleResultSet;
public class TestCsv {
    public static void main(String[] args) throws Exception {
        SimpleResultSet rs = new SimpleResultSet();
        rs.addColumn("NAME", Types.VARCHAR, 255, 0);
        rs.addColumn("EMAIL", Types.VARCHAR, 255, 0);
        rs.addRow("Bob Meier", "bob.meier@abcde.abc");
        rs.addRow("John Jones", "john.jones@abcde.abc");
        Csv.getInstance().write("data/test.csv", rs, null);
    }
}
```

Reading a CSV File from a Java Application

It is possible to read a CSV file without opening a database. Example:

```
import java.sql.*;
import org.h2.tools.Csv;
public class TestCsv {
    public static void main(String[] args) throws Exception {
        ResultSet rs = Csv.getInstance().
```

```
            read("data/test.csv", null, null);
        ResultSetMetaData meta = rs.getMetaData();
        while (rs.next()) {
            for (int i = 0; i < meta.getColumnCount(); i++) {
                System.out.println(
                    meta.getColumnLabel(i + 1) + ": " +
                    rs.getString(i + 1));
            }
            System.out.println();
        }
        rs.close();
    }
}
```

Upgrade, Backup, and Restore

Database Upgrade

The recommended way to upgrade from one version of the database engine to the next version is to create a backup of the database (in the form of a SQL script) using the old engine, and then execute the SQL script using the new engine.

Backup using the Script Tool

There are different ways to backup a database. For example, it is possible to copy the database files. However, this is not recommended while the database is in use. Also, the database files are not human readable and quite large. The recommended way to backup a database is to create a compressed SQL script file. This can be done using the Script tool:

```
java org.h2.tools.Script -url jdbc:h2:~/test -user sa -script test.zip -options compression zip
```

It is also possible to use the SQL command SCRIPT to create the backup of the database. For more information about the options, see the SQL command SCRIPT. The backup can be done remotely, however the file will be created on the server side. The built in FTP server could be used to retrieve the file from the server.

Restore from a Script

To restore a database from a SQL script file, you can use the RunScript tool:

```
java org.h2.tools.RunScript -url jdbc:h2:~/test -user sa -script test.zip -options compression zip
```

For more information about the options, see the SQL command RUNSCRIPT. The restore can be done remotely, however the file needs to be on the server side. The built in FTP server could be used to copy the file to the server. It is also possible to use the SQL command RUNSCRIPT to execute a SQL script. SQL script files may contain references to other script files, in the form of RUNSCRIPT commands. However, when using the server mode, the references script files need to be available on the server side.

Online Backup

The BACKUP SQL statement and the Backup tool both create a zip file with all database files. However, the contents of this file are not human readable. Other than the SCRIPT statement, the BACKUP statement does not lock the database objects, and therefore does not block other users. The resulting backup is transactionally consistent:

```
BACKUP TO 'backup.zip'
```

The Backup tool (org.h2.tools.Backup) can not be used to create a online backup; the database must not be in use while running this program.

Creating a backup while the database is running is not supported, except if the file systems support creating snapshots. The problem is that it can't be guaranteed that the data is copied in the right order.

Command Line Tools

This database comes with a number of command line tools. To get more information about a tool, start it with the parameter '-?', for example:

```
java -cp h2*.jar org.h2.tools.Backup -?
```

The command line tools are:

- **Backup** creates a backup of a database.
- **ChangeFileEncryption** allows changing the file encryption password or algorithm of a database.
- **Console** starts the browser based H2 Console.
- **ConvertTraceFile** converts a .trace.db file to a Java application and SQL script.
- **CreateCluster** creates a cluster from a standalone database.
- **DeleteDbFiles** deletes all files belonging to a database.
- **Recover** helps recovering a corrupted database.
- **Restore** restores a backup of a database.
- **RunScript** runs a SQL script against a database.
- **Script** allows converting a database to a SQL script for backup or migration.
- **Server** is used in the server mode to start a H2 server.
- **Shell** is a command line database tool.

The tools can also be called from an application by calling the main or another public method. For details, see the Javadoc documentation.

Using OpenOffice Base

OpenOffice.org Base supports database access over the JDBC API. To connect to a H2 database using OpenOffice Base, you first need to add the JDBC driver to OpenOffice. The steps to connect to a H2 database are:

- Start OpenOffice Writer, go to [Tools], [Options]
- Make sure you have selected a Java runtime environment in OpenOffice.org / Java
- Click [Class Path...], [Add Archive...]
- Select your h2 jar file (location is up to you, could be wherever you choose)
- Click [OK] (as much as needed), stop OpenOffice (including the Quickstarter)
- Start OpenOffice Base
- Connect to an existing database; select [JDBC]; [Next]
- Example datasource URL: jdbc:h2:~/test
- JDBC driver class: org.h2.Driver

Now you can access the database stored in the current users home directory.

To use H2 in NeoOffice (OpenOffice without X11):

- In NeoOffice, go to [NeoOffice], [Preferences]
- Look for the page under [NeoOffice], [Java]
- Click [Class Path], [Add Archive...]
- Select your h2 jar file (location is up to you, could be wherever you choose)
- Click [OK] (as much as needed), restart NeoOffice.

Now, when creating a new database using the "Database Wizard" :

- Click [File], [New], [Database].
- Select [Connect to existing database] and the select [JDBC]. Click next.

- Example datasource URL: jdbc:h2:~/test
- JDBC driver class: org.h2.Driver

Another solution to use H2 in NeoOffice is:

- Package the h2 jar within an extension package
- Install it as a Java extension in NeoOffice

This can be done by create it using the NetBeans OpenOffice plugin. See also Extensions Development .

Java Web Start / JNLP

When using Java Web Start / JNLP (Java Network Launch Protocol), permissions tags must be set in the .jnlp file, and the application .jar file must be signed. Otherwise, when trying to write to the file system, the following exception will occur: java.security.AccessControlException: access denied (java.io.FilePermission ... read). Example permission tags:

```
<security>
    <all-permissions/>
</security>
```

Using a Connection Pool

For H2, opening a connection is fast if the database is already open. Still, using a connection pool improves performance if you open and close connections a lot. A simple connection pool is included in H2. It is based on the Mini Connection Pool Manager from Christian d'Heureuse. There are other, more complex, open source connection pools available, for example the Apache Commons DBCP . For H2, it is about twice as faster to get a connection from the built-in connection pool than to get one using DriverManager.getConnection(). The build-in connection pool is used as follows:

```
import java.sql.*;
import org.h2.jdbcx.JdbcConnectionPool;
public class Test {
    public static void main(String[] args) throws Exception {
        JdbcConnectionPool cp = JdbcConnectionPool.create(
            "jdbc:h2:~/test", "sa", "sa");
        for (int i = 0; i < args.length; i++) {
            Connection conn = cp.getConnection();
            conn.createStatement().execute(args[i]);
            conn.close();
        }
        cp.dispose();
    }
}
```

Fulltext Search

H2 includes two fulltext search implementations. One is using Apache Lucene, and the other (the native implementation) stores the index data in special tables in the database.

Using the Native Fulltext Search

To initialize, call:

```
CREATE ALIAS IF NOT EXISTS FT_INIT FOR "org.h2.fulltext.FullText.init";
CALL FT_INIT();
```

You need to initialize it in each database where you want to use it. Afterwards, you can create a fulltext index for a table using:

```
CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR);
INSERT INTO TEST VALUES(1, 'Hello World');
CALL FT_CREATE_INDEX('PUBLIC', 'TEST', NULL);
```

PUBLIC is the schema, TEST is the table name. The list of column names (column separated) is optional, in this case all columns are indexed. The index is updated in realtime. To search the index, use the following query:

```
SELECT * FROM FT_SEARCH('Hello', 0, 0);
```

This will produce a result set that contains the query needed to retrieve the data:

QUERY: "PUBLIC"."TEST" WHERE "ID"=1

To get the raw data, use FT_SEARCH_DATA('Hello', 0, 0); . The result contains the columns SCHEMA (the schema name), TABLE (the table name), COLUMNS (an array of column names), and KEYS (an array of objects). To join a table, use a join as in: SELECT T.* FROM FT_SEARCH_DATA('Hello', 0, 0) FT, TEST T WHERE FT.TABLE='TEST' AND T.ID= FT.KEYS[0];

You can also call the index from within a Java application:

```
org.h2.fulltext.FullText.search(conn, text, limit, offset);
org.h2.fulltext.FullText.searchData(conn, text, limit, offset);
```

Using the Lucene Fulltext Search

To use the Lucene full text search, you need the Lucene library in the classpath. How to do that depends on the application; if you use the H2 Console, you can add the Lucene jar file to the environment variables H2DRIVERS or CLASSPATH. To initialize the Lucene fulltext search in a database, call:

```
CREATE ALIAS IF NOT EXISTS FTL_INIT FOR "org.h2.fulltext.FullTextLucene.init";
CALL FTL_INIT();
```

You need to initialize it in each database where you want to use it. Afterwards, you can create a full text index for a table using:

```
CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR);
INSERT INTO TEST VALUES(1, 'Hello World');
CALL FTL_CREATE_INDEX('PUBLIC', 'TEST', NULL);
```

PUBLIC is the schema, TEST is the table name. The list of column names (column separated) is optional, in this case all columns are indexed. The index is updated in realtime. To search the index, use the following query:

```
SELECT * FROM FTL_SEARCH('Hello', 0, 0);
```

This will produce a result set that contains the query needed to retrieve the data:

QUERY: "PUBLIC"."TEST" WHERE "ID"=1

To get the raw data, use FTL_SEARCH_DATA('Hello', 0, 0); . The result contains the columns SCHEMA (the schema name), TABLE (the table name), COLUMNS (an array of column names), and KEYS (an array of objects). To join a table, use a join as in: SELECT T.* FROM FTL_SEARCH_DATA('Hello', 0, 0) FT, TEST T WHERE FT.TABLE='TEST' AND T.ID= FT.KEYS[0];

You can also call the index from within a Java application:

```
org.h2.fulltext.FullTextLucene.search(conn, text, limit, offset);
org.h2.fulltext.FullTextLucene.searchData(conn, text, limit, offset);
```

User-Defined Variables

This database supports user-defined variables. Variables start with @ and can be used wherever expressions or parameters are allowed. Variables are not persisted and session scoped, that means only visible from within the session in which they are defined. A value is usually assigned using the SET command:

```
SET @USER = 'Joe';
```

The value can also be changed using the SET() method. This is useful in queries:

```
SET @TOTAL = NULL;
SELECT X, SET(@TOTAL, IFNULL(@TOTAL, 1.) * X) F FROM SYSTEM_RANGE(1, 50);
```

Variables that are not set evaluate to NULL. The data type of a user-defined variable is the data type of the value assigned to it, that means it is not necessary (or possible) to declare variable names before using them. There are no restrictions on the assigned values; large objects (LOBs) are supported as well.

Date and Time

Date, time and timestamp values support ISO 8601 formatting, including time zone:

```
CALL TIMESTAMP '2008-01-01 12:00:00+01:00';
```

If the time zone is not set, the value is parsed using the current time zone setting of the system. Date and time information is stored in H2 database files in GMT (Greenwich Mean Time). If the database is opened using another system time zone, the date and time will change accordingly. If you want to move a database from one time zone to the other and don't want this to happen, you need to create a SQL script file using the SCRIPT command or Script tool, and then load the database using the RUNSCRIPT command or the RunScript tool in the new time zone.

Using Spring

Use the following configuration to start and stop the H2 TCP server using the Spring Framework:

```
<bean id="org.h2.tools.Server"
          class="org.h2.tools.Server"
          factory-method="createTcpServer"
          init-method="start"
          destroy-method="stop">
   <constructor-arg value="-tcp,-tcpAllowOthers,true,-tcpPort,8043" />
</bean>
```

The "destroy-method" will help prevent exceptions on hot-redeployment or when restarting the server.

Features

Feature List

Main Features

- Very fast database engine
- Open source
- Written in Java
- Supports standard SQL, JDBC API
- Embedded and Server mode, Clustering support
- Strong security features
- The PostgreSQL ODBC driver can be used
- Multi version concurrency

Additional Features

- Disk based or in-memory databases and tables, read-only database support, temporary tables
- Transaction support (read committed and serializable transaction isolation), 2-phase-commit
- Multiple connections, table level locking
- Cost based optimizer, using a genetic algorithm for complex queries, zero-administration
- Scrollable and updatable result set support, large result set, external result sorting, functions can return a result set
- Encrypted database (AES or XTEA), SHA-256 password encryption, encryption functions, SSL

SQL Support

- Support for multiple schemas, information schema

- Referential integrity / foreign key constraints with cascade, check constraints
- Inner and outer joins, subqueries, read only views and inline views
- Triggers and Java functions / stored procedures
- Many built-in functions, including XML and lossless data compression
- Wide range of data types including large objects (BLOB/CLOB) and arrays
- Sequence and autoincrement columns, computed columns (can be used for function based indexes)
- ORDER BY, GROUP BY, HAVING, UNION, LIMIT, TOP
- Collation support, users, roles
- Compatibility modes for IBM DB2, Apache Derby, HSQLDB, MS SQL Server, MySQL, Oracle, and PostgreSQL.

Security Features

- Includes a solution for the SQL injection problem
- User password authentication uses SHA-256 and salt
- For server mode connections, user passwords are never transmitted in plain text over the network (even when using insecure connections; this only applies to the TCP server and not to the H2 Console however; it also doesn't apply if you set the password in the database URL)
- All database files (including script files that can be used to backup data) can be encrypted using AES-256 and XTEA encryption algorithms
- The remote JDBC driver supports TCP/IP connections over SSL/TLS
- The built-in web server supports connections over SSL/TLS
- Passwords can be sent to the database using char arrays instead of Strings

Other Features and Tools

- Small footprint (smaller than 1 MB), low memory requirements
- Multiple index types (b-tree, tree, hash)
- Support for multi-dimensional indexes
- CSV (comma separated values) file support
- Support for linked tables, and a built-in virtual 'range' table
- EXPLAIN PLAN support, sophisticated trace options
- Database closing can be delayed or disabled to improve the performance
- Web-based Console application (translated to many languages) with autocomplete
- The database can generate SQL script files
- Contains a recovery tool that can dump the contents of the database
- Support for variables (for example to calculate running totals)
- Automatic re-compilation of prepared statements
- Uses a small number of database files
- Uses a checksum for each record and log entry for data integrity
- Well tested (high code coverage, randomized stress tests)

Comparison to Other Database Engines

Feature	H2	Derby	HSQLDB	MySQL	PostgreSQL
Pure Java	Yes	Yes	Yes	No	No
Embedded Mode (Java)	Yes	Yes	Yes	No	No
Performance (Embedded)	Fast	Slow	Fast	N/A	N/A
In-Memory Mode	Yes	No	Yes	No	No
Transaction Isolation	Yes	Yes	No	Yes	Yes
Cost Based Optimizer	Yes	Yes	No	Yes	Yes
Explain Plan	Yes	No	Yes	Yes	Yes
Clustering	Yes	No	No	Yes	Yes
Encrypted Database	Yes	Yes	No	No	No
Linked Tables	Yes	No	Partially *1	Partially *2	No
ODBC Driver	Yes	No	No	Yes	Yes
Fulltext Search	Yes	No	No	Yes	Yes
User-Defined Datatypes	Yes	No	No	Yes	Yes
Files per Database	Few	Many	Few	Many	Many

Table Level Locking	Yes	Yes	No	Yes	Yes
Row Level Locking	Yes *9	Yes	No	Yes	Yes
Multi Version Concurrency	Yes	No	No	No	Yes
Role Based Security	Yes	Yes *3	Yes	Yes	Yes
Updatable Result Sets	Yes	Yes *7	No	Yes	Yes
Sequences	Yes	No	Yes	No	Yes
Limit and Offset	Yes	No	Yes	Yes	Yes
Temporary Tables	Yes	Yes *4	Yes	Yes	Yes
Information Schema	Yes	No *8	No *8	Yes	Yes
Computed Columns	Yes	No	No	No	Yes *6
Case Insensitive Columns	Yes	No	Yes	Yes	Yes *6
Custom Aggregate Functions	Yes	No	No	Yes	Yes
Footprint (jar/dll size)	~1 MB *5	~2 MB	~700 KB	~4 MB	~6 MB

*1 HSQLDB supports text tables.
*2 MySQL supports linked MySQL tables under the name 'federated tables'.
*3 Derby support for roles based security and password checking as an option.
*4 Derby only supports global temporary tables.
*5 The default H2 jar file contains debug information, jar files for other databases do not.
*6 PostgreSQL supports functional indexes.
*7 Derby only supports updatable result sets if the query is not sorted.
*8 Derby and HSQLDB don't support standard compliant information schema tables.
*9 H2 supports row level locks when using multi version concurrency.

Derby and HSQLDB

After an unexpected process termination (for example power failure), H2 can recover safely and automatically without any user interaction. For Derby and HSQLDB, some manual steps are required ('Another instance of Derby may have already booted the database' / 'The database is already in use by another process').

DaffodilDb and One$Db

It looks like the development of this database has stopped. The last release was February 2006.

McKoi

It looks like the development of this database has stopped. The last release was August 2004

H2 in Use

For a list of applications that work with or use H2, see: Links .

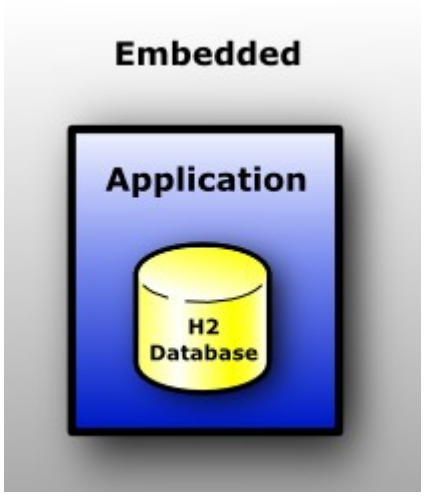
Connection Modes

The following connection modes are supported:

- Embedded mode (local connections using JDBC)
- Server mode (remote connections using JDBC or ODBC over TCP/IP)
- Mixed mode (local and remote connections at the same time)

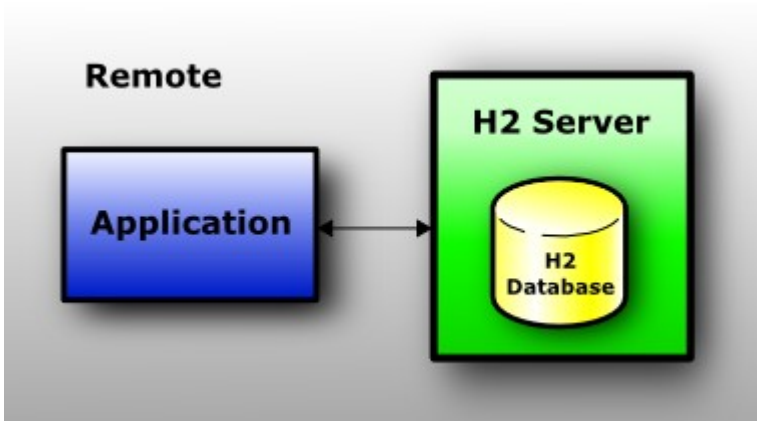
Embedded Mode

In embedded mode, an application opens a database from within the same JVM using JDBC. This is the fastest and easiest connection mode. The disadvantage is that a database may only be open in one virtual machine (and class loader) at any time. As in all modes, both persistent and in-memory databases are supported. There is no limit on the number of database open concurrently, or on the number of open connections.



Server Mode

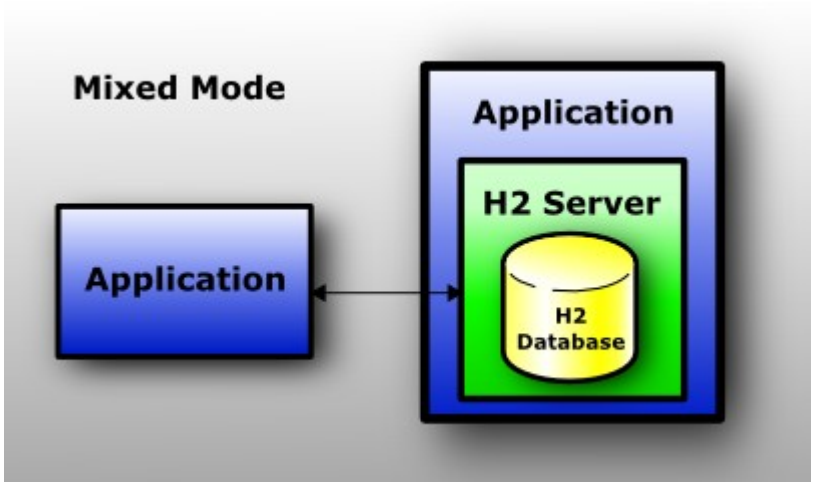
When using the server mode (sometimes called remote mode or client/server mode), an application opens a database remotely using the JDBC or ODBC API. A server needs to be started within the same or another virtual machine, or on another computer. Many applications can connect to the same database at the same time. The server mode is slower than the embedded mode, because all data is transferred over TCP/IP. As in all modes, both persistent and in-memory databases are supported. There is no limit on the number of database open concurrently, or on the number of open connections.



Mixed Mode

The mixed mode is a combination of the embedded and the server mode. The first application that connects to a database does that in embedded mode, but also starts a server so that other applications (running in different processes or virtual machines) can concurrently access the same data. The local connections are as fast as if the database is used in just the embedded mode, while the remote connections are a bit slower.

The server can be started and stopped from within the application (using the server API), or automatically (automatic mixed mode). When using the automatic mixed mode , all clients that want to connect to the database (no matter if it's an local or remote connection) can do so using the exact same database URL.



Database URL Overview

This database supports multiple connection modes and connection settings. This is achieved using different database URLs. Settings in the URLs are not case sensitive.

Topic	URL Format and Examples
Embedded (local) connection	jdbc:h2:[file:][<path>]<databaseName> jdbc:h2:~/test jdbc:h2:file:/data/sample jdbc:h2:file:C:/data/sample (Windows only)
In-memory (private)	jdbc:h2:mem:
In-memory (named)	jdbc:h2:mem:<databaseName> jdbc:h2:mem:test_mem
Server mode (remote connections) using TCP/IP	jdbc:h2:tcp://<server>[:<port>]/[<path>]<databaseName> jdbc:h2:tcp://localhost/~/test jdbc:h2:tcp://dbserv:8084/~/sample
Server mode (remote connections) using SSL/TLS	jdbc:h2:ssl://<server>[:<port>]/<databaseName> jdbc:h2:ssl://secureserv:8085/~/sample;
Using encrypted files	jdbc:h2:<url>;CIPHER=[AES|XTEA] jdbc:h2:ssl://secureserv/~/testdb;CIPHER=AES jdbc:h2:file:~/secure;CIPHER=XTEA
File locking methods	jdbc:h2:<url>;FILE_LOCK={NO|FILE|SOCKET} jdbc:h2:file:~/quickAndDirty;FILE_LOCK=NO jdbc:h2:file:~/private;CIPHER=XTEA;FILE_LOCK=SOCKET
Only open if it already exists	jdbc:h2:<url>;IFEXISTS=TRUE jdbc:h2:file:~/sample;IFEXISTS=TRUE
Don't close the database when the VM exits	jdbc:h2:<url>;DB_CLOSE_ON_EXIT=FALSE
User name and/or password	jdbc:h2:<url>[;USER=<username>][;PASSWORD=<value>] jdbc:h2:file:~/sample;USER=sa;PASSWORD=123
Log index changes	jdbc:h2:<url>;LOG=2 jdbc:h2:file:~/sample;LOG=2
Debug trace settings	jdbc:h2:<url>;TRACE_LEVEL_FILE=<level 0..3> jdbc:h2:file:~/sample;TRACE_LEVEL_FILE=3
Ignore unknown settings	jdbc:h2:<url>;IGNORE_UNKNOWN_SETTINGS=TRUE
Custom file access mode	jdbc:h2:<url>;ACCESS_MODE_LOG=rws;ACCESS_MODE_DATA=rws
Database in a zip file	jdbc:h2:zip:<zipFileName>!/<databaseName> jdbc:h2:zip:~/db.zip!/test
Compatibility mode	jdbc:h2:<url>;MODE=<databaseType> jdbc:h2:~/test;MODE=MYSQL
Auto-reconnect	jdbc:h2:<url>;AUTO_RECONNECT=TRUE jdbc:h2:tcp://localhost/~/test;AUTO_RECONNECT=TRUE
Automatic mixed mode	jdbc:h2:<url>;AUTO_SERVER=TRUE jdbc:h2:~/test;AUTO_SERVER=TRUE
Changing other settings	jdbc:h2:<url>;<setting>=<value>[;<setting>=<value>...] jdbc:h2:file:~/sample;TRACE_LEVEL_SYSTEM_OUT=3

Connecting to an Embedded (Local) Database

The database URL for connecting to a local database is jdbc:h2:[file:][<path>]<databaseName> . The prefix file: is optional. If no or only a relative path is used, then the current working directory is used as a starting point. The case sensitivity of the path and database name depend on the operating system, however it is recommended to use lowercase letters only. The database name must be at least three characters long (a limitation of File.createTempFile). To point to the user home directory, use ~/, as in: jdbc:h2:~/test.

Memory-Only Databases

For certain use cases (for example: rapid prototyping, testing, high performance operations, read-only databases), it may not be required to persist data, or persist changes to the data. This database supports the memory-only mode, where the data is not persisted.

In some cases, only one connection to a memory-only database is required. This means the database to be opened is private. In this case, the database URL is jdbc:h2:mem: Opening two connections within the same virtual machine means opening two different (private) databases.

Sometimes multiple connections to the same memory-only database are required. In this case, the database URL must include a name. Example: jdbc:h2:mem:db1 . Accessing the same database in this way only works within the same virtual machine and class loader environment.

It is also possible to access a memory-only database remotely (or from multiple processes in the same machine) using TCP/IP or SSL/TLS. An example database URL is: jdbc:h2:tcp://localhost/mem:db1 .

By default, closing the last connection to a database closes the database. For an in-memory database, this means the content is lost. To keep the database open, add ;DB_CLOSE_DELAY=-1 to the database URL. To keep the content of an in-memory database as long as the virtual machine is alive, use jdbc:h2:mem:test;DB_CLOSE_DELAY=-1 .

Database Files Encryption

The database files can be encrypted. Two encryption algorithms are supported: AES and XTEA. To use file encryption, you need to specify the encryption algorithm (the 'cipher') and the file password (in addition to the user password) when connecting to the database.

Creating a New Database with File Encryption

By default, a new database is automatically created if it does not exist yet. To create an encrypted database, connect to it as it would already exist.

Connecting to an Encrypted Database

The encryption algorithm is set in the database URL, and the file password is specified in the password field, before the user password. A single space separates the file password and the user password; the file password itself may not contain spaces. File passwords and user passwords are case sensitive. Here is an example to connect to a password-encrypted database:

```
Class.forName("org.h2.Driver");
String url = "jdbc:h2:~/test;CIPHER=AES";
String user = "sa";
String pwds = "filepwd userpwd";
conn = DriverManager.
    getConnection(url, user, pwds);
```

Encrypting or Decrypting a Database

To encrypt an existing database, use the ChangeFileEncryption tool. This tool can also decrypt an encrypted database, or change the file encryption key. The tool is available from within the H2 Console in the Tools section, or you can run it from the command line. The following command line will encrypt the database 'test' in the user home directory with the file password 'filepwd' and the encryption algorithm AES:

```
java -cp h2*.jar org.h2.tools.ChangeFileEncryption -dir ~ -db test -cipher AES -encrypt filepwd
```

Database File Locking

Whenever a database is opened, a lock file is created to signal other processes that the database is in use. If database is closed, or if the process that opened the database terminates, this lock file is deleted.

The following file locking methods are implemented:

- The default method is 'file' and uses a watchdog thread to protect the database file. The watchdog reads the lock file each second.
- The second method is 'socket' and opens a server socket. The socket method does not require reading the lock file every second. The socket method should only be used if the database files are only accessed by one (and always the same) computer.
- It is also possible to open the database without file locking; in this case it is up to the application to protect the database files.

To open the database with a different file locking method, use the parameter 'FILE_LOCK'. The following code opens the database with the 'socket' locking method:

```
String url = "jdbc:h2:~/test;FILE_LOCK=SOCKET";
```

The following code forces the database to not create a lock file at all. Please note that this is unsafe as another process is able to open the same database, possibly leading to data corruption:

```
String url = "jdbc:h2:~/test;FILE_LOCK=NO";
```

For more information about the algorithms, see Advanced / File Locking Protocols .

Opening a Database Only if it Already Exists

By default, when an application calls DriverManager.getConnection(url, ...) and the database specified in the URL does not yet exist, a new (empty) database is created. In some situations, it is better to restrict creating new databases, and only allow to open existing databases. To do this, add ;ifexists=true to the database URL. In this case, if the database does not already exist, an exception is thrown when trying to connect. The connection only succeeds when the database already exists. The complete URL may look like this:

```
String url = "jdbc:h2:/data/sample;IFEXISTS=TRUE";
```

Closing a Database

Delayed Database Closing

Usually, a database is closed when the last connection to it is closed. In some situations this slows down the application, for example when it is not possible to keep at least one connection open. The automatic closing of a database can be delayed or disabled with the SQL statement SET DB_CLOSE_DELAY <seconds>. The parameter <seconds> specifies the number of seconds to keep a database open after the last connection to it was closed. The following statement will keep a database open for 10 seconds after the last connection was closed:

```
SET DB_CLOSE_DELAY 10
```

The value -1 means the database is not closed automatically. The value 0 is the default and means the database is closed when the last connection is closed. This setting is persistent and can be set by an administrator only. It is possible to set the value in the database URL: jdbc:h2:~/test;DB_CLOSE_DELAY=10 .

Don't Close a Database when the VM Exits

By default, a database is closed when the last connection is closed. However, if it is never closed, the database is closed when the virtual machine exits normally, using a shutdown hook. In some situations, the database should not be closed in this case, for example because the database is still used at virtual machine shutdown (to store the shutdown process in the database for example). For those cases, the automatic closing of the database can be disabled in the database URL. The first connection (the one that is opening the database) needs to set the option in the database URL (it is not possible to change the setting afterwards). The database URL to disable database closing on exit is:

```
String url = "jdbc:h2:~/test;DB_CLOSE_ON_EXIT=FALSE";
```

Log Index Changes

Usually, changes to the index file are not logged for performance. If the index file is corrupt or missing when opening a database, it is re-created from the data. The index file can get corrupt when the database is not shut down correctly, because of power failure or abnormal program termination. In some situations, for example when using very large databases (over a few hundred MB), re-creating the index file takes very long. In these situations it may be better to log changes to the index file, so that recovery from a corrupted index file is fast. To enable log index changes, add LOG=2 to the URL, as in jdbc:h2:~/test;LOG=2 . This setting should be specified when connecting. The update performance of the database will be reduced when using this option.

Ignore Unknown Settings

Some applications (for example OpenOffice.org Base) pass some additional parameters when connecting to the database. Why those parameters are passed is unknown. The parameters PREFERDOSLIKELINEENDS and IGNOREDRIVERPRIVILEGES are such examples; they are simply ignored to improve the compatibility with OpenOffice.org. If an application passes other parameters when connecting to the database, usually the database throws an exception saying the parameter is not supported. It is possible to ignored such parameters by adding ;IGNORE_UNKNOWN_SETTINGS=TRUE to the database URL.

Changing Other Settings when Opening a Connection

In addition to the settings already described, other database settings can be passed in the database URL. Adding ;setting=value at the end of a database URL is the same as executing the statement SET setting value just after connecting. For a list of supported settings, see SQL Grammar .

Custom File Access Mode

Usually, the database opens log, data and index files with the access mode 'rw', meaning read-write (except for read only databases, where the mode 'r' is used). To open a database in read-only mode if the files are not read-only, use ACCESS_MODE_DATA=r. Also supported are 'rws' and 'rwd'. The access mode used for log files is set via ACCESS_MODE_LOG; for data and index files use ACCESS_MODE_DATA. These settings must be specified in the database URL:

```
String url = "jdbc:h2:~/test;ACCESS_MODE_LOG=rws;ACCESS_MODE_DATA=rws";
```

For more information see Durability Problems . On many operating systems the access mode 'rws' does not guarantee that the data is written to the disk.

Multiple Connections

Opening Multiple Databases at the Same Time

An application can open multiple databases at the same time, including multiple connections to the same database. The number of open database is only limited by the memory available.

Multiple Connections to the Same Database: Client/Server

If you want to access the same database at the same time from different processes or computers, you need to use the client / server mode. In this case, one process acts as the server, and the other processes (that could reside on other computers as well) connect to the server via TCP/IP (or SSL/TLS over TCP/IP for improved security).

Multithreading Support

This database is multithreading-safe. That means, if an application is multi-threaded, it does not need to worry about synchronizing access to the database. Internally, most requests to the same database are synchronized. That means an application can use multiple threads that access the same database at the same time, however if one thread executes a long running query, the other threads need to wait.

Locking, Lock-Timeout, Deadlocks

The database uses table level locks to give each connection a consistent state of the data. There are two kinds of locks: read locks (shared locks) and write locks (exclusive locks). All locks are released when the transaction commits or rolls back. When using the default transaction isolation level 'read committed', read locks are already released after each statement.

If a connection wants to reads from a table, and there is no write lock on the table, then a read lock is added to the table. If there is a write lock, then this connection waits for the other connection to release the lock. If a connection cannot get a lock for a specified time, then a lock timeout exception is thrown.

Usually, SELECT statements will generate read locks. This includes subqueries. Statements that modify data use write locks. It is also possible to lock a table exclusively without modifying data, using the statement SELECT ... FOR UPDATE. The statements COMMIT and ROLLBACK releases all open locks. The commands SAVEPOINT and ROLLBACK TO SAVEPOINT don't affect locks. The locks are also released when the autocommit mode changes, and for connections with autocommit set to true (this is the default), locks are released after each statement. The following statements generate locks:

Type of Lock	SQL Statement
Read	SELECT * FROM TEST; CALL SELECT MAX(ID) FROM TEST; SCRIPT;
Write	SELECT * FROM TEST WHERE 1=0 FOR UPDATE;
Write	INSERT INTO TEST VALUES(1, 'Hello'); INSERT INTO TEST SELECT * FROM TEST; UPDATE TEST SET NAME='Hi'; DELETE FROM TEST;
Write	ALTER TABLE TEST ...; CREATE INDEX ... ON TEST ...; DROP INDEX ...;

The number of seconds until a lock timeout exception is thrown can be set separately for each connection using the SQL command SET LOCK_TIMEOUT <milliseconds>. The initial lock timeout (that is the timeout used for new connections) can be set using the SQL command SET DEFAULT_LOCK_TIMEOUT <milliseconds>. The default lock timeout is persistent.

Database File Layout

There are a number of files created for persistent databases. Unlike some other databases, not every table and/or index is stored in its own file. Instead, usually only the following files are created: a data file, an index file, a log file, and a database lock file (exists only while the database is in use). In addition to that, a file is created for each large object (CLOB/BLOB) larger than a certain size, and temporary files for large result sets. If the database trace option is enabled, trace files are created. The following files can be created by the database:

File Name	Description	Number of Files
test.data.db	Data file. Contains the data for all tables. Format: <database>.data.db	1 per database
test.index.db	Index file. Contains the data for all (b tree) indexes. Format: <database>.index.db	1 per database
test.0.log.db	Transaction log file. The transaction log is used for recovery. Format: <database>.<id>.log.db	0 or more per database
test.lock.db	Database lock file. Exists only while the database is open. Format: <database>.lock.db	1 per database
test.trace.db	Trace file. Contains trace information. Format: <database>.trace.db If the file is too big, it is renamed to <database>.trace.db.old	1 per database
test.lobs.db/1.t15.lob.db	Large object. Contains the data for BLOB or CLOB values. Format: <id>.t<tableId>.lob.db	1 per value
test.123.temp.db	Temporary file. Contains a temporary blob or a large result set. Format: <database>.<id>.temp.db	1 per object

Moving and Renaming Database Files

Database name and location are not stored inside the database files.

While a database is closed, the files can be moved to another directory, and they can be renamed as well (as long as all files start with the same name).

As there is no platform specific data in the files, they can be moved to other operating systems without problems.

Backup

When the database is closed, it is possible to backup the database files. Please note that index files do not need to be backed up, because they contain redundant data, and will be recreated automatically if they don't exist.

To backup data while the database is running, the SQL command SCRIPT can be used.

Logging and Recovery

Whenever data is modified in the database and those changes are committed, the changes are logged to disk (except for in-memory objects). The changes to the data file itself are usually written later on, to optimize disk access. If there is a power failure, the data and index files are not up-to-date. But because the changes are in the log file, the next time the database is opened, the changes that are in the log file are re-applied automatically.

Please note that index file updates are not logged by default. If the database is opened and recovery is required, the index file is rebuilt from scratch.

There is usually only one log file per database. This file grows until the database is closed successfully, and is then deleted. Or, if the file gets too big, the database switches to another log file (with a higher id). It is possible to force the log switching by using the CHECKPOINT command.

If the database file is corrupted, because the checksum of a record does not match (for example, if the file was edited with another application), the database can be opened in recovery mode. In this case, errors in the database are logged but not thrown. The database should be backed up to a script and re-built as soon as possible. To open the database in the recovery mode, use a database URL must contain RECOVER=1, as in jdbc:h2:~/test;RECOVER=1. Indexes are rebuilt in this case, and the summary (object allocation table) is not read in this case, so opening the database takes longer.

Compatibility

All database engines behave a little bit different. Where possible, H2 supports the ANSI SQL standard, and tries to be compatible to other databases. There are still a few differences however:

In MySQL text columns are case insensitive by default, while in H2 they are case sensitive. However H2 supports case insensitive columns as well. To create the tables with case insensitive texts, append IGNORECASE=TRUE to the database URL (example: jdbc:h2:~/test;IGNORECASE=TRUE).

Compatibility Modes

For certain features, this database can emulate the behavior of specific databases. Not all features or differences of those databases are implemented. Here is the list of currently supported modes and the differences to the regular mode:

DB2 Compatibility Mode

To use the IBM DB2 mode, use the database URL jdbc:h2:~/test;MODE=DB2 or the SQL statement SET MODE DB2 .

- For aliased columns, ResultSetMetaData.getColumnName() returns the alias name and getTableName() returns null.
- Support for the syntax [OFFSET .. ROW] [FETCH ... ONLY] as an alternative for LIMIT .. OFFSET.

Derby Compatibility Mode

To use the Apache Derby mode, use the database URL jdbc:h2:~/test;MODE=Derby or the SQL statement SET MODE Derby .

- For aliased columns, ResultSetMetaData.getColumnName() returns the alias name and getTableName() returns null.
- For unique indexes, NULL is distinct. That means only one row with NULL in one of the columns is allowed.

HSQLDB Compatibility Mode

To use the HSQLDB mode, use the database URL jdbc:h2:~/test;MODE=HSQLDB or the SQL statement SET MODE HSQLDB .

- For aliased columns, ResultSetMetaData.getColumnName() returns the alias name and getTableName() returns null.
- When converting the scale of decimal data, the number is only converted if the new scale is smaller than the current scale. Usually, the scale is converted and 0s are added if required.
- Concatenation with NULL results in NULL. Usually, NULL is treated as an empty string if only one of the operands is NULL, and NULL is only returned if both operands are NULL.
- For unique indexes, NULL is distinct. That means only one row with NULL in one of the columns is allowed.

MS SQL Server Compatibility Mode

To use the MS SQL Server mode, use the database URL jdbc:h2:~/test;MODE=MSSQLServer or the SQL statement SET MODE MSSQLServer .

- For aliased columns, ResultSetMetaData.getColumnName() returns the alias name and getTableName() returns null.
- Identifiers may be quoted using square brackets as in [Test].
- For unique indexes, NULL is distinct. That means only one row with NULL in one of the columns is allowed.

MySQL Compatibility Mode

To use the MySQL mode, use the database URL jdbc:h2:~/test;MODE=MySQL or the SQL statement SET MODE MySQL .

- When inserting data, if a column is defined to be NOT NULL and NULL is inserted, then a 0 (or empty string, or the current timestamp for timestamp columns) value is used. Usually, this operation is not allowed and an exception is thrown.
- Creating indexes in the CREATE TABLE statement is allowed.
- Meta data calls return identifiers in lower case.
- When converting a floating point number to an integer, the fractional digits are not truncated, but the value is rounded.

Oracle Compatibility Mode

To use the Oracle mode, use the database URL jdbc:h2:~/test;MODE=Oracle or the SQL statement SET MODE Oracle .

- For aliased columns, ResultSetMetaData.getColumnName() returns the alias name and getTableName() returns null.
- When using unique indexes, multiple rows with NULL in all columns are allowed, however it is not allowed to have multiple rows with the same values otherwise.

PostgreSQL Compatibility Mode

To use the PostgreSQL mode, use the database URL jdbc:h2:~/test;MODE=PostgreSQL or the SQL statement SET MODE PostgreSQL .

- For aliased columns, ResultSetMetaData.getColumnName() returns the alias name and getTableName() returns null.
- Concatenation with NULL results in NULL. Usually, NULL is treated as an empty string if only one of the operands is NULL, and NULL is only returned if both operands are NULL.
- When converting a floating point number to an integer, the fractional digits are not be truncated, but the value is rounded.
- The system columns 'CTID' and 'OID' are supported.

Auto-Reconnect

The auto-reconnect feature causes the JDBC driver to reconnect to the database if the connection is lost. The automatic re-connect only occurs when auto-commit is enabled; if auto-commit is disabled, an exception is thrown.

Re-connecting will open a new session. After an automatic re-connect, variables and local temporary tables definitions (excluding data) are re-created. The contents of the system table INFORMATION_SCHEMA.SESSION_STATE contains all client side state that is re-created.

Automatic Mixed Mode

Multiple processes can access the same database without having to start the server manually. To do that, append ;AUTO_SERVER=TRUE to the database URL. You can use the same database URL no matter if the database is already open or not.

When using this mode, the first connection to the database is made in embedded mode, and additionally a server is started internally. If the database is already open in another process, the server mode is used automatically.

The application that opens the first connection to the database uses the embedded mode, which is faster than the server mode. Therefore the main application should open the database first if possible. The first connection automatically starts a server on a random port. This server allows remote connections, however only to this database (to ensure that, the client reads .lock.db file and sends the the random key that is stored there to the server). When the first connection is closed, the server stops. If other (remote) connections are still open, one of them will then start a server (auto-reconnect is enabled automatically).

All processes need to have access to the database files. If the first connection is closed (the connection that started the server), open transactions of other connections will be rolled back. Explicit client/server connections (using jdbc:h2:tcp:// or ssl://) are not supported. This mode is not supported for in-memory databases.

Here is an example how to use this mode. Application 1 and 2 are not necessarily started on the same computer, but they need to have access to the database files. Application 1 and 2 are typically two different processes (however they could run within the same process).

```
// Application 1:
DriverManager.getConnection("jdbc:h2:/data/test;AUTO_SERVER=TRUE");

// Application 2:
DriverManager.getConnection("jdbc:h2:/data/test;AUTO_SERVER=TRUE");
```

Using the Trace Options

To find problems in an application, it is sometimes good to see what database operations where executed. This database offers the following trace features:

- Trace to System.out and/or a file
- Support for trace levels OFF, ERROR, INFO, and DEBUG
- The maximum size of the trace file can be set
- It is possible to generate Java source code from the trace file
- Trace can be enabled at runtime by manually creating a file

Trace Options

The simplest way to enable the trace option is setting it in the database URL. There are two settings, one for System.out (TRACE_LEVEL_SYSTEM_OUT) tracing, and one for file tracing (TRACE_LEVEL_FILE). The trace levels are 0 for OFF, 1 for ERROR (the default), 2 for INFO and 3 for DEBUG. A database URL with both levels set to DEBUG is:

```
jdbc:h2:~/test;TRACE_LEVEL_FILE=3;TRACE_LEVEL_SYSTEM_OUT=3
```

The trace level can be changed at runtime by executing the SQL command SET TRACE_LEVEL_SYSTEM_OUT level (for System.out tracing) or SET TRACE_LEVEL_FILE level (for file tracing). Example:

```
SET TRACE_LEVEL_SYSTEM_OUT 3
```

Setting the Maximum Size of the Trace File

When using a high trace level, the trace file can get very big quickly. The default size limit is 16 MB, if the trace file exceeds this limit, it is renamed to .old and a new file is created. If another .old file exists, it is deleted. The size limit can be changed using the SQL statement SET TRACE_MAX_FILE_SIZE maximumFileSizeInMB . Example:

```
SET TRACE_MAX_FILE_SIZE 1
```

Java Code Generation

When setting the trace level to INFO or DEBUG, Java source code is generated as well. This allows to reproduce problems more easily. The trace file looks like this:

```
...
12-20 20:58:09 jdbc[0]:
/**/dbMeta3.getURL();
12-20 20:58:09 jdbc[0]:
/**/dbMeta3.getTables(null, "", null, new String[]{"TABLE", "VIEW"});
...
```

To filter the Java source code, use the ConvertTraceFile tool as follows:

```
java -cp h2*.jar org.h2.tools.ConvertTraceFile
    -traceFile "~/test.trace.db" -javaClass "Test"
```

The generated file Test.java will contain the Java source code. The generated source code may be too large to compile (the size of a Java method is limited). If this is the case, the source code needs to be split in multiple methods. The password is not listed in the trace file and therefore not included in the source code.

Using Other Logging APIs

By default, this database uses its own native 'trace' facility. This facility is called 'trace' and not 'log' within this database to avoid confusion with the transaction log. Trace messages can be written to both file and System.out. In most cases, this is sufficient, however sometimes it is better to use the same facility as the application, for example Log4j. To do that, this database support SLF4J.

SLF4J is a simple facade for various logging APIs and allows to plug in the desired implementation at deployment time. SLF4J supports implementations such as Logback, Log4j, Jakarta Commons Logging (JCL), Java logging, x4juli, and Simple Log.

To enable SLF4J, set the file trace level to 4 in the database URL:

```
jdbc:h2:~/test;TRACE_LEVEL_FILE=4
```

Changing the log mechanism is not possible after the database is open, that means executing the SQL statement SET TRACE_LEVEL_FILE 4 when the database is already open will not have the desired effect. To use SLF4J, all required jar files need to be in the classpath. If it does not work, check the file <database>.trace.db for error messages.

Read Only Databases

If the database files are read-only, then the database is read-only as well. It is not possible to create new tables, add or modify data in this database. Only SELECT and CALL statements are allowed. To create a read-only database, close the database so

that the log file gets smaller. Do not delete the log file. Then, make the database files read-only using the operating system. When you open the database now, it is read-only. There are two ways an application can find out whether database is read-only: by calling Connection.isReadOnly() or by executing the SQL statement CALL READONLY().

Read Only Databases in Zip or Jar File

To create a read-only database in a zip file, first create a regular persistent database, and then create a backup. If you are using a database named 'test', an easy way to do that is using the Backup tool or the BACKUP SQL statement:

```
BACKUP TO 'data.zip'
```

The database must not have pending changes, that means you need to close all connections to the database, open one single connection, and then execute the statement. Afterwards, you can log out, and directly open the database in the zip file using the following database URL:

```
jdbc:h2:zip:~/data.zip!/test
```

Databases in zip files are read-only. The performance for some queries will be slower than when using a regular database, because random access in zip files is not supported (only streaming). How much this affects the performance depends on the queries and the data. The database is not read in memory; therefore large databases are supported as well. The same indexes are used as when using a regular database.

Graceful Handling of Low Disk Space Situations

If the database needs more disk space, it calls the database event listener if one is installed. The application may then delete temporary files, or display a message and wait until the user has resolved the problem. To install a listener, run the SQL statement SET DATABASE_EVENT_LISTENER or use a database URL of the form jdbc:h2:~/test;DATABASE_EVENT_LISTENER='com.acme.DbListener' (the quotes around the class name are required). See also the DatabaseEventListener API.

Opening a Corrupted Database

If a database cannot be opened because the boot info (the SQL script that is run at startup) is corrupted, then the database can be opened by specifying a database event listener. The exceptions are logged, but opening the database will continue.

Computed Columns / Function Based Index

Function indexes are not directly supported by this database, but they can be emulated by using computed columns. For example, if an index on the upper-case version of a column is required, create a computed column with the upper-case version of the original column, and create an index for this column:

```
CREATE TABLE ADDRESS(
    ID INT PRIMARY KEY,
    NAME VARCHAR,
    UPPER_NAME VARCHAR AS UPPER(NAME)
);
CREATE INDEX IDX_U_NAME ON ADDRESS(UPPER_NAME);
```

When inserting data, it is not required (and not allowed) to specify a value for the upper-case version of the column, because the value is generated. But you can use the column when querying the table:

```
INSERT INTO ADDRESS(ID, NAME) VALUES(1, 'Miller');
SELECT * FROM ADDRESS WHERE UPPER_NAME='MILLER';
```

Multi-Dimensional Indexes

A tool is provided to execute efficient multi-dimension (spatial) range queries. This database does not support a specialized spatial index (R-Tree or similar). Instead, the B-Tree index is used. For each record, the multi-dimensional key is converted (mapped) to a single dimensional (scalar) value. This value specifies the location on a space-filling curve.

Currently, Z-order (also called N-order or Morton-order) is used; Hilbert curve could also be used, but the implementation is more complex. The algorithm to convert the multi-dimensional value is called bit-interleaving. The scalar value is indexed using a B-Tree index (usually using a computed column).

The method can result in a drastic performance improvement over just using an index on the first column. Depending on the data and number of dimensions, the improvement is usually higher than factor 5. The tool generates a SQL query from a specified multi-dimensional range. The method used is not database dependent, and the tool can easily be ported to other databases. For an example how to use the tool, please have a look at the sample code provided in TestMultiDimension.java.

Using Passwords

Using Secure Passwords

Remember that weak passwords can be broken no matter of the encryption and security protocol. Don't use passwords that can be found in a dictionary. Also appending numbers does not make them secure. A way to create good passwords that can be remembered is, take the first letters of a sentence, use upper and lower case characters, and creatively include special characters. Example:

i'sE2rtPiUKtT (it's easy to remember this password if you know the trick)

Passwords: Using Char Arrays instead of Strings

Java Strings are immutable objects and cannot be safely 'destroyed' by the application. After creating a String, it will remain in the main memory of the computer at least until it is garbage collected. The garbage collection cannot be controlled by the application, and even if it is garbage collected the data may still remain in memory. It might also be possible that the part of memory containing the password is swapped to disk (because not enough main memory is available).

An attacker might have access to the swap file of the operating system. It is therefore a good idea to use char arrays instead of Strings to store passwords. Char arrays can be cleared (filled with zeros) after use, and therefore the password will not be stored in the swap file.

This database supports using char arrays instead of String to pass user and file passwords. The following code can be used to do that:

```
import java.sql.*;
import java.util.*;
public class Test {
    public static void main(String[] args) throws Exception {
        Class.forName("org.h2.Driver");
        String url = "jdbc:h2:~/test";
        Properties prop = new Properties();
        prop.setProperty("user", "sa");
        System.out.print("Password?");
        char[] password = System.console().readPassword();
        prop.put("password", password);
        Connection conn = null;
        try {
```

```
            conn = DriverManager.getConnection(url, prop);
        } finally {
            Arrays.fill(password, (char) 0);
        }
        conn.close();
    }
}
```

This example requires Java 1.6. When using Swing, use javax.swing.JPasswordField.

Passing the User Name and/or Password in the URL

Instead of passing the user name as a separate parameter as in Connection conn = DriverManager. getConnection("jdbc:h2:~/test", "sa", "123"); the user name (and/or password) can be supplied in the URL itself: Connection conn = DriverManager. getConnection("jdbc:h2:~/test;USER=sa;PASSWORD=123"); The settings in the URL override the settings passed as a separate parameter.

User-Defined Functions and Stored Procedures

In addition to the built-in functions, this database supports user-defined Java functions. In this database, Java functions can be used as stored procedures as well. A function must be declared (registered) before it can be used. Only static Java methods are supported; both the class and the method must be public. Example Java method:

```
package acme;
import java.math.*;
public class Function {
    public static boolean isPrime(int value) {
        return new BigInteger(String.valueOf(value)).isProbablePrime(100);
    }
}
```

The Java function must be registered in the database by calling CREATE ALIAS:

```
CREATE ALIAS IS_PRIME FOR "acme.Function.isPrime"
```

For a complete sample application, see src/test/org/h2/samples/Function.java.

Function Data Type Mapping

Functions that accept non-nullable parameters such as 'int' will not be called if one of those parameters is NULL. Instead, the result of the function is NULL. If the function should be called if a parameter is NULL, you need to use 'java.lang.Integer' instead of 'int'.

SQL types are mapped to Java classes and vice-versa as in the JDBC API. For details, see Data Types . There are two special cases: java.lang.Object is mapped to OTHER (a serialized object). Therefore, java.lang.Object can not be used to match all SQL types (matching all SQL types is not supported). The second special case is Object[]: arrays of any class are mapped to ARRAY.

Functions that require a Connection

If the first parameter of a Java function is a java.sql.Connection, then the connection to database is provided. This connection does not need to be closed before returning. When calling the method from within the SQL statement, this connection parameter does not need to be (can not be) specified.

Functions throwing an Exception

If a function throws an Exception, then the current statement is rolled back and the exception is thrown to the application.

Functions returning a Result Set

Functions may returns a result set. Such a function can be called with the CALL statement:

```
public static ResultSet query(Connection conn, String sql) throws SQLException {
    return conn.createStatement().executeQuery(sql);
}

CREATE ALIAS QUERY FOR "org.h2.samples.Function.query";
CALL QUERY('SELECT * FROM TEST');
```

Using SimpleResultSet

A function can create a result set using the SimpleResultSet tool:

```
import org.h2.tools.SimpleResultSet;
...
public static ResultSet simpleResultSet() throws SQLException {
    SimpleResultSet rs = new SimpleResultSet();
    rs.addColumn("ID", Types.INTEGER, 10, 0);
    rs.addColumn("NAME", Types.VARCHAR, 255, 0);
    rs.addRow(0, "Hello");
    rs.addRow(1, "World");
    return rs;
}

CREATE ALIAS SIMPLE FOR "org.h2.samples.Function.simpleResultSet";
CALL SIMPLE();
```

Using a Function as a Table

A function that returns a result set can be used like a table. However, in this case the function is called at least twice: first while parsing the statement to collect the column names (with parameters set to null where not known at compile time). And then, while executing the statement to get the data (maybe multiple times if this is a join). If the function is called just to get the column list, the URL of the connection passed to the function is jdbc:columnlist:connection . Otherwise, the URL of the connection is jdbc:default:connection .

```
public static ResultSet getMatrix(Connection conn, Integer size)
        throws SQLException {
    SimpleResultSet rs = new SimpleResultSet();
    rs.addColumn("X", Types.INTEGER, 10, 0);
    rs.addColumn("Y", Types.INTEGER, 10, 0);
    String url = conn.getMetaData().getURL();
    if (url.equals("jdbc:columnlist:connection")) {
        return rs;
    }
    for (int s = size.intValue(), x = 0; x < s; x++) {
        for (int y = 0; y < s; y++) {
            rs.addRow(x, y);
        }
    }
    return rs;
}

CREATE ALIAS MATRIX FOR "org.h2.samples.Function.getMatrix";
SELECT * FROM MATRIX(4) ORDER BY X, Y;
```

Triggers

This database supports Java triggers that are called before or after a row is updated, inserted or deleted. Triggers can be used for complex consistency checks, or to update related data in the database. It is also possible to use triggers to simulate materialized views. For a complete sample application, see src/test/org/h2/samples/TriggerSample.java. A Java trigger must implement the interface org.h2.api.Trigger:

```
import org.h2.api.Trigger;
...
public class TriggerSample implements Trigger {
    public void init(Connection conn, String schemaName, String triggerName,
            String tableName, boolean before, int type) {
    public void fire(Connection conn,
            Object[] oldRow, Object[] newRow)
            throws SQLException {
    }
}
```

The connection can be used to query or update data in other tables. The trigger then needs to be defined in the database:

```
CREATE TRIGGER INV_INS AFTER INSERT ON INVOICE
    FOR EACH ROW CALL "org.h2.samples.TriggerSample"
```

The trigger can be used to veto a change, by throwing a SQLException.

Compacting a Database

Empty space in the database file is re-used automatically. To re-build the indexes, the simplest way is to delete the .index.db file while the database is closed. However in some situations (for example after deleting a lot of data in a database), one sometimes wants to shrink the size of the database (compact a database). Here is a sample function to do this:

```
public static void compact(String dir, String dbName,
        String user, String password) throws Exception {
    String url = "jdbc:h2:" + dir + "/" + dbName;
    String file = "data/test.sql";
    Script.execute(url, user, password, file);
    DeleteDbFiles.execute(dir, dbName, true);
    RunScript.execute(url, user, password, file, null, false);
}
```

See also the sample application org.h2.samples.Compact. The commands SCRIPT / RUNSCRIPT can be used as well to create a backup of a database and re-build the database from the script.

Cache Settings

The database keeps most frequently used data and index pages in the main memory. The amount of memory used for caching can be changed using the setting CACHE_SIZE. This setting can be set in the database connection URL (jdbc:h2:~/test;CACHE_SIZE=131072), or it can be changed at runtime using SET CACHE_SIZE size.

This database supports two cache page replacement algorithms: LRU (the default) and TQ. For LRU, the pages that were least frequently used are removed from the cache if it becomes full. The TQ (Two Queue, also called 2Q) algorithm is a bit more complicated: basically two queues are used. It is more resistant to table scans, however the overhead is a bit higher compared to the LRU. To use the cache algorithm TQ, use a database URL of the form jdbc:h2:~/test;CACHE_TYPE=TQ. The cache algorithm cannot be changed once the database is open.

Also supported is a second level soft reference cache. Rows in this cache are only garbage collected on low memory. By default the second level cache is disabled. To enable it, use the prefix SOFT_. Example: jdbc:h2:~/test;CACHE_TYPE=SOFT_LRU .

To get information about page reads and writes, and the current caching algorithm in use, call SELECT * FROM INFORMATION_SCHEMA.SETTINGS. The number of pages read / written is listed for the data and index file.

Performance

Performance Comparison

In many cases H2 is faster than other (open source and not open source) database engines. Please note this is mostly a single connection benchmark run on one computer.

Embedded

Test Case	Unit	H2	HSQLDB	Derby
Simple: Init	ms	547	532	2594
Simple: Query (random)	ms	250	391	1515
Simple: Query (sequential)	ms	188	313	1406
Simple: Update (random)	ms	812	1750	17704
Simple: Delete (sequential)	ms	203	250	8843
Simple: Memory Usage	MB	7	11	11
BenchA: Init	ms	578	719	3328
BenchA: Transactions	ms	3047	2406	12907
BenchA: Memory Usage	MB	10	15	10
BenchB: Init	ms	2141	2406	11562
BenchB: Transactions	ms	1125	1375	3625
BenchB: Memory Usage	MB	9	11	8
BenchC: Init	ms	688	594	4500
BenchC: Transactions	ms	1906	64062	6047
BenchC: Memory Usage	MB	11	17	11
Executed statements	#	322929	322929	322929
Total time	ms	11485	74798	74031
Statements per second	#	28117	4317	4362

Client-Server

Test Case	Unit	H2	HSQLDB	Derby	PostgreSQL	MySQL
Simple: Init	ms	2782	2656	5625	4563	3484
Simple: Query (random)	ms	3093	2703	6688	4812	3860
Simple: Query (sequential)	ms	2969	2594	6437	4719	3625
Simple: Update (random)	ms	2969	3531	18250	5953	5125
Simple: Delete (sequential)	ms	1047	1250	6875	2485	2390
Simple: Memory Usage	MB	7	11	14	0	0
BenchA: Init	ms	2250	2453	6031	4328	3625
BenchA: Transactions	ms	10250	9016	21484	15609	11172
BenchA: Memory Usage	MB	10	15	10	0	1
BenchB: Init	ms	9500	10672	22609	19609	13406

BenchB: Transactions	ms	2734	2656	3875	4688	2531
BenchB: Memory Usage	MB	10	11	11	1	1
BenchC: Init	ms	1860	1484	6890	2219	3438
BenchC: Transactions	ms	9046	63266	18641	11703	7421
BenchC: Memory Usage	MB	12	17	13	0	1
Executed statements	#	322929	322929	322929	322929	322929
Total time	ms	48500	102281	123405	80688	60077
Statements per second	#	6658	3157	2616	4002	5375

Benchmark Results and Comments

H2

Version 1.1.114 (2009-06-01) was used for the test. For simpler operations, the performance of H2 is about the same as for HSQLDB. For more complex queries, the query optimizer is very important. However H2 is not very fast in every case, certain kind of queries may still be slow. One situation where is H2 is slow is large result sets, because they are buffered to disk if more than a certain number of records are returned. The advantage of buffering is, there is no limit on the result set size. The open/close time is almost fixed, because of the file locking protocol: the engine waits some time after opening a database to ensure the database files are not opened by another process.

HSQLDB

Version 1.8.0.10 was used for the test. Cached tables are used in this test (hsqldb.default_table_type=cached), and the write delay is 1 second (SET WRITE_DELAY 1). HSQLDB is fast when using simple operations. HSQLDB is very slow in the last test (BenchC: Transactions), probably because is has a bad query optimizer. One query where HSQLDB is slow is a two-table join:

```
SELECT COUNT(DISTINCT S_I_ID) FROM ORDER_LINE, STOCK
WHERE OL_W_ID=? AND OL_D_ID=? AND OL_O_ID<? AND OL_O_ID>=?
AND S_W_ID=? AND S_I_ID=OL_I_ID AND S_QUANTITY<?
```

The PolePosition benchmark also shows that the query optimizer does not do a very good job for some queries. Another disadvantage of HSQLDB is the slow startup / shutdown time (currently not listed) when using bigger databases. The reason is, a backup of the whole data is made whenever the database is opened or closed.

Derby

Version 10.4.2.0 was used for the test. Derby is clearly the slowest embedded database in this test. This seems to be a structural problem, because all operations are really slow. It will be hard for the developers of Derby to improve the performance to a reasonable level. A few problems have been identified: leaving autocommit on is a problem for Derby. If it is switched off during the whole test, the results are about 20% better for Derby. Derby supports a testing mode (system property derby.system.durability=test) where durability is disabled. According to the documentation, this setting should be used for testing only, as the database may not recover after a crash. Enabling this setting improves performance by a factor of 2.6 (embedded mode) or 1.4 (server mode). Even if enabled, Derby is still less than half as fast as H2 in default mode.

PostgreSQL

Version 8.3.7 was used for the test. The following options where changed in postgresql.conf: fsync = off, commit_delay = 1000. PostgreSQL is run in server mode. It looks like the base performance is slower than MySQL, the reason could be the network layer. The memory usage number is incorrect, because only the memory usage of the JDBC driver is measured.

MySQL

Version 5.1.34-community was used for the test. MySQL was run with the InnoDB backend. The setting innodb_flush_log_at_trx_commit (found in the my.ini file) was set to 0. Otherwise (and by default), MySQL is really slow (around 140 statements per second in this test) because it tries to flush the data to disk for each commit. For small transactions (when autocommit is on) this is really slow. But many use cases use small or relatively small transactions. Too bad this setting is not listed in the configuration wizard, and it always overwritten when using the wizard. You need to change this setting

manually in the file my.ini, and then restart the service. The memory usage number is incorrect, because only the memory usage of the JDBC driver is measured.

Firebird

Firebird 1.5 (default installation) was tested, but the results are not published currently. It is possible to run the performance test with the Firebird database, and any information on how to configure Firebird for higher performance are welcome.

Why Oracle / MS SQL Server / DB2 are Not Listed

The license of these databases does not allow to publish benchmark results. This doesn't mean that they are fast. They are in fact quite slow, and need a lot of memory. But you will need to test this yourself. SQLite was not tested because the JDBC driver doesn't support transactions.

About this Benchmark

How to Run

This test was executed as follows:

```
build benchmark
```

Separate Process per Database

For each database, a new process is started, to ensure the previous test does not impact the current test.

Number of Connections

This is mostly a single-connection benchmark. BenchB uses multiple connections; the other tests use one connection.

Real-World Tests

Good benchmarks emulate real-world use cases. This benchmark includes 4 test cases: BenchSimple uses one table and many small updates / deletes. BenchA is similar to the TPC-A test, but single connection / single threaded (see also: www.tpc.org). BenchB is similar to the TPC-B test, using multiple connections (one thread per connection). BenchC is similar to the TPC-C test, but single connection / single threaded.

Comparing Embedded with Server Databases

This is mainly a benchmark for embedded databases (where the application runs in the same virtual machine as the database engine). However MySQL and PostgreSQL are not Java databases and cannot be embedded into a Java application. For the Java databases, both embedded and server modes are tested.

Test Platform

This test is run on Windows XP with the virus scanner switched off. The VM used is Sun JDK 1.5.

Multiple Runs

When a Java benchmark is run first, the code is not fully compiled and therefore runs slower than when running multiple times. A benchmark should always run the same test multiple times and ignore the first run(s). This benchmark runs three times, but only the last run is measured.

Memory Usage

It is not enough to measure the time taken, the memory usage is important as well. Performance can be improved by using a bigger cache, but the amount of memory is limited. HSQLDB tables are kept fully in memory by default; this benchmark uses 'disk based' tables for all databases. Unfortunately, it is not so easy to calculate the memory usage of PostgreSQL and MySQL, because they run in a different process than the test. This benchmark currently does not print memory usage of those databases.

Delayed Operations

Some databases delay some operations (for example flushing the buffers) until after the benchmark is run. This benchmark waits between each database tested, and each database runs in a different process (sequentially).

Transaction Commit / Durability

Durability means transaction committed to the database will not be lost. Some databases (for example MySQL) try to enforce this by default by calling fsync() to flush the buffers, but most hard drives don't actually flush all data. Calling fsync() slows down transaction commit a lot, but doesn't always make data durable. When comparing the results, it is important to think about the effect. Many database suggest to 'batch' operations when possible. This benchmark switches off autocommit when loading the data, and calls commit after each 1000 inserts. However many applications need 'short' transactions at runtime (a commit after each update). This benchmark commits after each update / delete in the simple benchmark, and after each business transaction in the other benchmarks. For databases that support delayed commits, a delay of one second is used.

Using Prepared Statements

Wherever possible, the test cases use prepared statements.

Currently Not Tested: Startup Time

The startup time of a database engine is important as well for embedded use. This time is not measured currently. Also, not tested is the time used to create a database and open an existing database. Here, one (wrapper) connection is opened at the start, and for each step a new connection is opened and then closed.

PolePosition Benchmark

The PolePosition is an open source benchmark. The algorithms are all quite simple. It was developed / sponsored by db4o.

Test Case	Unit	H2	HSQLDB	MySQL
Melbourne write	ms	369	249	2022
Melbourne read	ms	47	49	93
Melbourne read_hot	ms	24	43	95
Melbourne delete	ms	147	133	176
Sepang write	ms	965	1201	3213
Sepang read	ms	765	948	3455
Sepang read_hot	ms	789	859	3563
Sepang delete	ms	1384	1596	6214
Bahrain write	ms	1186	1387	6904
Bahrain query_indexed_string	ms	336	170	693
Bahrain query_string	ms	18064	39703	41243
Bahrain query_indexed_int	ms	104	134	678
Bahrain update	ms	191	87	159

Bahrain delete	ms	1215	729	6812
Imola retrieve	ms	198	194	4036
Barcelona write	ms	413	832	3191
Barcelona read	ms	119	160	1177
Barcelona query	ms	20	5169	101
Barcelona delete	ms	388	319	3287
Total	ms	26724	53962	87112

There are a few problems with the PolePosition test:

- HSQLDB uses in-memory tables by default while H2 uses persistent tables. The HSQLDB version included in PolePosition does not support changing this, so you need to replace poleposition-0.20/lib/hsqldb.jar with a newer version (for example hsqldb-1.8.0.7.jar), and then use the setting hsqldb.connecturl=jdbc:hsqldb:file:data/hsqldb/dbbench2;hsqldb.default_table_type=cached;sql.enforce_size=true in Jdbc.properties.
- HSQLDB keeps the database open between tests, while H2 closes the database (losing all the cache). To change that, use the database URL jdbc:h2:file:data/h2/dbbench;DB_CLOSE_DELAY=-1
- The amount of cache memory is quite important, specially for the PolePosition test. Unfortunately, the PolePosition test does not take this into account.

Application Profiling

Analyze First

Before trying to optimize performance, it is important to understand where the problem is (what part of the application is slow). Blind optimization or optimization based on guesses should be avoided, because usually it is not an efficient strategy. There are various ways to analyze an application. Sometimes two implementations can be compared using System.currentTimeMillis(). But this does not work for complex applications with many modules, and for memory problems.

A good tool to measure both memory usage and performance is the YourKit Java Profiler .

A simple way to profile an application is to use the built-in profiling tool of java. Example:

```
java -Xrunhprof:cpu=samples,depth=16 com.acme.Test
```

Unfortunately, it is only possible to profile the application from start to end. Another solution is to create a number of full thread dumps. To do that, first run jps -l to get the process id, and then run jstack <pid> or kill -QUIT <pid> (Linux) or press Ctrl+C (Windows).

Database Profiling

The ConvertTraceFile tool generates SQL statement statistics at the end of the SQL script file. The format used is similar to the profiling data generated when using java -Xrunhprof. As an example, execute the the following script using the H2 Console:

```
SET TRACE_LEVEL_FILE 3;
DROP TABLE IF EXISTS TEST;
CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
@LOOP 1000 INSERT INTO TEST VALUES(?, ?);
SET TRACE_LEVEL_FILE 0;
```

Now convert the .trace.db file using the ConvertTraceFile tool:

```
java -cp h2*.jar org.h2.tools.ConvertTraceFile
```

```
    -traceFile "~/test.trace.db" -script "~/test.sql"
```

The generated file test.sql will contain the SQL statements as well as the following profiling data (results vary):

```
----------------------------------------
-- SQL Statement Statistics
-- time: total time in milliseconds (accumulated)
-- count: how many times the statement ran
-- result: total update count or row count
----------------------------------------
-- self accu    time   count  result sql
--  62%  62%     158    1000    1000 INSERT INTO TEST VALUES(?, ?);
--  37% 100%      93       1       0 CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
--   0% 100%       0       1       0 DROP TABLE IF EXISTS TEST;
--   0% 100%       0       1       0 SET TRACE_LEVEL_FILE 3;
```

Database Performance Tuning

Use a Modern JVM

Newer JVMs are faster. Upgrading to the latest version of your JVM can provide a "free" boost to performance. On the Windows platform, switching from the default Client JVM to the Server JVM using the -server command-line option improves performance at the cost of a slight increase in start-up time.

Virus Scanners

Some virus scanners scan files every time they are accessed. It is very important for performance that database files are not scanned for viruses. The database engine does never interprets the data stored in the files as programs, that means even if somebody would store a virus in a database file, this would be harmless (when the virus does not run, it cannot spread). Some virus scanners allow to exclude files by suffix. Make sure files ending with .db are not scanned.

Using the Trace Options

If the main performance hot spots are in the database engine, in many cases the performance can be optimized by creating additional indexes, or changing the schema. Sometimes the application does not directly generate the SQL statements, for example if an O/R mapping tool is used. To view the SQL statements and JDBC API calls, you can use the trace options. For more information, see Using the Trace Options .

Index Usage

This database uses indexes to improve the performance of SELECT, UPDATE and DELETE statements. If a column is used in the WHERE clause of a query, and if an index exists on this column, then the index can be used. Multi-column indexes are used if all or the first columns of the index are used. Both equality lookup and range scans are supported. Indexes are used to order result sets, but only if the condition uses the same index or no index at all. The results are sorted in memory if required. Indexes are created automatically for primary key and unique constraints. Indexes are also created for foreign key constraints, if required. For other columns, indexes need to be created manually using the CREATE INDEX statement.

Optimizer

This database uses a cost based optimizer. For simple and queries and queries with medium complexity (less than 7 tables in the join), the expected cost (running time) of all possible plans is calculated, and the plan with the lowest cost is used. For more complex queries, the algorithm first tries all possible combinations for the first few tables, and the remaining tables added using a greedy algorithm (this works well for most joins). Afterwards a genetic algorithm is used to test at most 2000 distinct plans. Only left-deep plans are evaluated.

Expression Optimization

After the statement is parsed, all expressions are simplified automatically if possible. Operations are evaluated only once if all parameters are constant. Functions are also optimized, but only if the function is constant (always returns the same result for the same parameter values). If the WHERE clause is always false, then the table is not accessed at all.

COUNT(*) Optimization

If the query only counts all rows of a table, then the data is not accessed. However, this is only possible if no WHERE clause is used, that means it only works for queries of the form SELECT COUNT(*) FROM table.

Updating Optimizer Statistics / Column Selectivity

When executing a query, at most one index per joined table can be used. If the same table is joined multiple times, for each join only one index is used. Example: for the query SELECT * FROM TEST T1, TEST T2 WHERE T1.NAME='A' AND T2.ID=T1.ID, two index can be used, in this case the index on NAME for T1 and the index on ID for T2.

If a table has multiple indexes, sometimes more than one index could be used. Example: if there is a table TEST(ID, NAME, FIRSTNAME) and an index on each column, then two indexes could be used for the query SELECT * FROM TEST WHERE NAME='A' AND FIRSTNAME='B', the index on NAME or the index on FIRSTNAME. It is not possible to use both indexes at the same time. Which index is used depends on the selectivity of the column. The selectivity describes the 'uniqueness' of values in a column. A selectivity of 100 means each value appears only once, and a selectivity of 1 means the same value appears in many or most rows. For the query above, the index on NAME should be used if the table contains more distinct names than first names.

The SQL statement ANALYZE can be used to automatically estimate the selectivity of the columns in the tables. This command should be run from time to time to improve the query plans generated by the optimizer.

In-Memory (Hash) Indexes

Using in-memory indexes, specially in-memory hash indexes, can speed up queries and data manipulation.

In-memory indexes are automatically used for in-memory databases, but can also be created for persistent databases using CREATE MEMORY TABLE . In many cases, the rows itself will also be kept in-memory. Please note this may cause memory problems for large tables.

In-memory hash indexes are backed by a hash table and are usually faster than regular indexes. However, hash indexes only supports direct lookup (WHERE ID = ?) but not range scan (WHERE ID < ?). To use hash indexes, use HASH as in: CREATE UNIQUE HASH INDEX and CREATE TABLE ...(ID INT PRIMARY KEY HASH,...) .

Optimization Examples

See src/test/org/h2/samples/optimizations.sql for a few examples of queries that benefit from special optimizations built into the database.

Cache Size and Type

By default the cache size of H2 is quite small. Consider using a larger cache size, or enable the second level soft reference cache. See also Cache Settings .

Data Types

Each data type has different storage and performance characteristics:

- The DECIMAL/NUMERIC type is slower and requires more storage than the REAL and DOUBLE types.
- Text types are slower to read, write, and compare than numeric types and generally require more storage.
- See Large Objects for information on BINARY vs. BLOB and VARCHAR vs. CLOB performance.
- Parsing and formatting takes longer for the TIME, DATE, and TIMESTAMP types than the numeric types.

- SMALLINT/TINYINT/BOOLEAN are not significantly smaller or faster to work with than INTEGER in most modes.

Fast Database Import

To speed up large imports, consider using the following options temporarily:

- SET CACHE_SIZE (a large cache is faster)
- SET LOCK_MODE 0 (disable locking)
- SET LOG 0 (disable the transaction log)
- SET UNDO_LOG 0 (disable the session undo log)

These options can be set in the database URL: jdbc:h2:~/test;CACHE_SIZE=65536;LOCK_MODE=0;LOG=0;UNDO_LOG=0 . Most of those options are not recommended for regular use, that means you need to reset them after use.

Advanced Topics

Result Sets

Limiting the Number of Rows

Before the result is returned to the application, all rows are read by the database. Server side cursors are not supported currently. If only the first few rows are interesting for the application, then the result set size should be limited to improve the performance. This can be done using LIMIT in a query (example: SELECT * FROM TEST LIMIT 100), or by using Statement.setMaxRows(max).

Large Result Sets and External Sorting

For large result set, the result is buffered to disk. The threshold can be defined using the statement SET MAX_MEMORY_ROWS. If ORDER BY is used, the sorting is done using an external sort algorithm. In this case, each block of rows is sorted using quick sort, then written to disk; when reading the data, the blocks are merged together.

Large Objects

Storing and Reading Large Objects

If it is possible that the objects don't fit into memory, then the data type CLOB (for textual data) or BLOB (for binary data) should be used. For these data types, the objects are not fully read into memory, by using streams. To store a BLOB, use PreparedStatement.setBinaryStream. To store a CLOB, use PreparedStatement.setCharacterStream. To read a BLOB, use ResultSet.getBinaryStream, and to read a CLOB, use ResultSet.getCharacterStream. When using the client/server mode, large BLOB and CLOB data is stored in a temporary file on the client side.

When to use CLOB/BLOB

This database stores large LOB (CLOB and BLOB) objects as separate files. Small LOB objects are stored in-place, the threshold can be set using MAX_LENGTH_INPLACE_LOB , but there is still an overhead to use CLOB/BLOB. Because of this, BLOB and CLOB should never be used for columns with a maximum size below about 200 bytes. The best threshold depends on the use case; reading in-place objects is faster than reading from separate files, but slows down the performance of operations that don't involve this column.

Large Object Compression

CLOB and BLOB values can be compressed by using SET COMPRESS_LOB . The LZF algorithm is faster but needs more disk space. By default compression is disabled, which usually speeds up write operations. If you store many large compressible values such as XML, HTML, text, and uncompressed binary files, then compressing can save a lot of disk space (sometimes more than 50%), and read operations may even be faster.

Linked Tables

This database supports linked tables, which means tables that don't exist in the current database but are just links to another database. To create such a link, use the CREATE LINKED TABLE statement:

```
CREATE LINKED TABLE LINK('org.postgresql.Driver', 'jdbc:postgresql:test', 'sa', 'sa', 'TEST');
```

You can then access the table in the usual way. Whenever the linked table is accessed, the database issues specific queries over JDBC. Using the example above, if you issue the query SELECT * FROM LINK WHERE ID=1 , then the following query is run against the PostgreSQL database: SELECT * FROM TEST WHERE ID=? . The same happens for insert and update statements. Only simple statements are executed against the target database, that means no joins. Prepared statements are used where possible.

To view the statements that are executed against the target table, set the trace level to 3.

There is a restriction: when inserting into a linked table, and when updating a linked table, NULL and values that are not set are both inserted as NULL. This may not have the desired effect if the default value for this column in the target table is not NULL.

If multiple linked tables point to the same database (using the same database URL), the connection is shared. To disable this, set the system property h2.shareLinkedConnections to false.

The CREATE LINKED TABLE statement supports an optional schema name parameter. See the grammar for details.

Transaction Isolation

Transaction isolation is provided for all data manipulation language (DML) statements. Most data definition language (DDL) statements commit the current transaction. See the grammar for details.

This database supports the following transaction isolation levels:

- **Read Committed**
  This is the default level. Read locks are released immediately. Higher concurrency is possible when using this level.
  To enable, execute the SQL statement 'SET LOCK_MODE 3'
  or append ;LOCK_MODE=3 to the database URL: jdbc:h2:~/test;LOCK_MODE=3
- **Serializable**
  To enable, execute the SQL statement 'SET LOCK_MODE 1'
  or append ;LOCK_MODE=1 to the database URL: jdbc:h2:~/test;LOCK_MODE=1
- **Read Uncommitted**
  This level means that transaction isolation is disabled.
  To enable, execute the SQL statement 'SET LOCK_MODE 0'
  or append ;LOCK_MODE=0 to the database URL: jdbc:h2:~/test;LOCK_MODE=0

When using the isolation level 'serializable', dirty reads, non-repeatable reads, and phantom reads are prohibited.

- **Dirty Reads**
  Means a connection can read uncommitted changes made by another connection.
  Possible with: read uncommitted
- **Non-Repeatable Reads**
  A connection reads a row, another connection changes a row and commits, and the first connection re-reads the same row and gets the new result.
  Possible with: read uncommitted, read committed
- **Phantom Reads**
  A connection reads a set of rows using a condition, another connection inserts a row that falls in this condition and commits, then the first connection re-reads using the same condition and gets the new row.
  Possible with: read uncommitted, read committed

Table Level Locking

The database allows multiple concurrent connections to the same database. To make sure all connections only see consistent data, table level locking is used by default. This mechanism does not allow high concurrency, but is very fast. Shared locks and exclusive locks are supported. Before reading from a table, the database tries to add a shared lock to the table (this is only possible if there is no exclusive lock on the object by another connection). If the shared lock is added successfully, the table can be read. It is allowed that other connections also have a shared lock on the same object. If a connection wants to write to a table (update or delete a row), an exclusive lock is required. To get the exclusive lock, other connection must not have any locks on the object. After the connection commits, all locks are released. This database keeps all locks in memory.

Lock Timeout

If a connection cannot get a lock on an object, the connection waits for some amount of time (the lock timeout). During this time, hopefully the connection holding the lock commits and it is then possible to get the lock. If this is not possible because the other connection does not release the lock for some time, the unsuccessful connection will get a lock timeout exception. The lock timeout can be set individually for each connection.

Multi-Version Concurrency Control (MVCC)

The MVCC feature allows higher concurrency than using (table level or row level) locks. When using MVCC in this database, delete, insert and update operations will only issue a shared lock on the table. An exclusive lock is still used when adding or removing columns, when dropping the table, and when using SELECT ... FOR UPDATE. Connections only 'see' committed data, and own changes. That means, if connection A updates a row but doesn't commit this change yet, connection B will see the old value. Only when the change is committed, the new value is visible by other connections (read committed). If multiple connections concurrently try to update the same row, the database waits until it can apply the change, but at most until the lock timeout expires.

To use the MVCC feature, append MVCC=TRUE to the database URL:

```
jdbc:h2:~/test;MVCC=TRUE
```

The MVCC feature is not fully tested yet. The limitations of the MVCC mode are: it can not be used at the same time as MULTI_THREADED; the complete undo log must fit in memory when using multi-version concurrency (the setting MAX_MEMORY_UNDO has no effect).

Clustering / High Availability

This database supports a simple clustering / high availability mechanism. The architecture is: two database servers run on two different computers, and on both computers is a copy of the same database. If both servers run, each database operation is executed on both computers. If one server fails (power, hardware or network failure), the other server can still continue to work. From this point on, the operations will be executed only on one server until the other server is back up.

Clustering can only be used in the server mode (the embedded mode does not support clustering). It is possible to restore the cluster without stopping the server, however it is critical that no other application is changing the data in the first database while the second database is restored, so restoring the cluster is currently a manual process.

To initialize the cluster, use the following steps:

- Create a database
- Use the CreateCluster tool to copy the database to another location and initialize the clustering. Afterwards, you have two databases containing the same data.
- Start two servers (one for each copy of the database)
- You are now ready to connect to the databases with the client application(s)

Using the CreateCluster Tool

To understand how clustering works, please try out the following example. In this example, the two databases reside on the same computer, but usually, the databases will be on different servers.

- Create two directories: server1 and server2. Each directory will simulate a directory on a computer.
- Start a TCP server pointing to the first directory. You can do this using the command line:

```
java org.h2.tools.Server
    -tcp -tcpPort 9101
    -baseDir server1
```

- Start a second TCP server pointing to the second directory. This will simulate a server running on a second (redundant) computer. You can do this using the command line:

```
java org.h2.tools.Server
    -tcp -tcpPort 9102
    -baseDir server2
```

- Use the CreateCluster tool to initialize clustering. This will automatically create a new, empty database if it does not exist. Run the tool on the command line:

```
java org.h2.tools.CreateCluster
    -urlSource jdbc:h2:tcp://localhost:9101/~/test
    -urlTarget jdbc:h2:tcp://localhost:9102/~/test
    -user sa
    -serverList localhost:9101,localhost:9102
```

- You can now connect to the databases using an application or the H2 Console using the JDBC URL jdbc:h2:tcp://localhost:9101,localhost:9102/~/test
- If you stop a server (by killing the process), you will notice that the other machine continues to work, and therefore the database is still accessible.
- To restore the cluster, you first need to delete the database that failed, then restart the server that was stopped, and re-run the CreateCluster tool.

Clustering Algorithm and Limitations

Read-only queries are only executed against the first cluster node, but all other statements are executed against all nodes. There is currently no load balancing made to avoid problems with transactions. The following functions may yield different results on different cluster nodes and must be executed with care: RANDOM_UUID(), SECURE_RAND(), SESSION_ID(), MEMORY_FREE(), MEMORY_USED(), CSVREAD(), CSVWRITE(), RAND() [when not using a seed]. Those functions should not be used directly in modifying statements (for example INSERT, UPDATE, or MERGE). However, they can be used in read-only statements and the result can then be used for modifying statements.

Two Phase Commit

The two phase commit protocol is supported. 2-phase-commit works as follows:

- Autocommit needs to be switched off
- A transaction is started, for example by inserting a row
- The transaction is marked 'prepared' by executing the SQL statement PREPARE COMMIT transactionName
- The transaction can now be committed or rolled back
- If a problem occurs before the transaction was successfully committed or rolled back (for example because a network problem occurred), the transaction is in the state 'in-doubt'
- When re-connecting to the database, the in-doubt transactions can be listed with SELECT * FROM INFORMATION_SCHEMA.IN_DOUBT
- Each transaction in this list must now be committed or rolled back by executing COMMIT TRANSACTION transactionName or ROLLBACK TRANSACTION transactionName
- The database needs to be closed and re-opened to apply the changes

Compatibility

This database is (up to a certain point) compatible to other databases such as HSQLDB, MySQL and PostgreSQL. There are certain areas where H2 is incompatible.

Transaction Commit when Autocommit is On

At this time, this database engine commits a transaction (if autocommit is switched on) just before returning the result. For a query, this means the transaction is committed even before the application scans through the result set, and before the result set is closed. Other database engines may commit the transaction in this case when the result set is closed.

Keywords / Reserved Words

There is a list of keywords that can't be used as identifiers (table names, column names and so on), unless they are quoted (surrounded with double quotes). The list is currently:

CROSS, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, DISTINCT, EXCEPT, EXISTS, FALSE, FOR, FROM, FULL, GROUP, HAVING, INNER, INTERSECT, IS, JOIN, LIKE, LIMIT, MINUS, NATURAL, NOT, NULL, ON, ORDER, PRIMARY, ROWNUM, SELECT, SYSDATE, SYSTIME, SYSTIMESTAMP, TODAY, TRUE, UNION, WHERE

Certain words of this list are keywords because they are functions that can be used without '()' for compatibility, for example CURRENT_TIMESTAMP.

Standards Compliance

This database tries to be as much standard compliant as possible. For the SQL language, ANSI/ISO is the main standard. There are several versions that refer to the release date: SQL-92, SQL:1999, and SQL:2003. Unfortunately, the standard documentation is not freely available. Another problem is that important features are not standardized. Whenever this is the case, this database tries to be compatible to other databases.

Run as Windows Service

Using a native wrapper / adapter, Java applications can be run as a Windows Service. There are various tools available to do that. The Java Service Wrapper from Tanuki Software, Inc. (http://wrapper.tanukisoftware.org) is included in the installation. Batch files are provided to install, start, stop and uninstall the H2 Database Engine Service. This service contains the TCP Server and the H2 Console web application. The batch files are located in the directory H2/service.

Install the Service

The service needs to be registered as a Windows Service first. To do that, double click on 1_install_service.bat. If successful, a command prompt window will pop up and disappear immediately. If not, a message will appear.

Start the Service

You can start the H2 Database Engine Service using the service manager of Windows, or by double clicking on 2_start_service.bat. Please note that the batch file does not print an error message if the service is not installed.

Connect to the H2 Console

After installing and starting the service, you can connect to the H2 Console application using a browser. Double clicking on 3_start_browser.bat to do that. The default port (8082) is hard coded in the batch file.

Stop the Service

To stop the service, double click on 4_stop_service.bat. Please note that the batch file does not print an error message if the service is not installed or started.

Uninstall the Service

To uninstall the service, double click on 5_uninstall_service.bat. If successful, a command prompt window will pop up and disappear immediately. If not, a message will appear.

ODBC Driver

This database does not come with its own ODBC driver at this time, but it supports the PostgreSQL network protocol. Therefore, the PostgreSQL ODBC driver can be used. Support for the PostgreSQL network protocol is quite new and should be viewed as experimental. It should not be used for production applications.

To use the PostgreSQL ODBC driver on 64 bit versions of Windows, first run c:/windows/syswow64/odbcad32.exe . At this point you set up your DSN just like you would on any other system. See also: Re: ODBC Driver on Windows 64 bit

ODBC Installation

First, the ODBC driver must be installed. Any recent PostgreSQL ODBC driver should work, however version 8.2 (psqlodbc-08_02*) or newer is recommended. The Windows version of the PostgreSQL ODBC driver is available at http://www.postgresql.org/ftp/odbc/versions/msi .

Starting the Server

After installing the ODBC driver, start the H2 Server using the command line:

```
java -cp h2*.jar org.h2.tools.Server
```

The PG Server (PG for PostgreSQL protocol) is started as well. By default, databases are stored in the current working directory where the server is started. Use -baseDir to save databases in another directory, for example the user home directory:

```
java -cp h2*.jar org.h2.tools.Server -baseDir ~
```

The PG server can be started and stopped from within a Java application as follows:

```
Server server = Server.createPgServer("-baseDir", "~");
server.start();
...
server.stop();
```

By default, only connections from localhost are allowed. To allow remote connections, use -pgAllowOthers when starting the server.

ODBC Configuration

After installing the driver, a new Data Source must be added. In Windows, run odbcad32.exe to open the Data Source Administrator. Then click on 'Add...' and select the PostgreSQL Unicode driver. Then click 'Finish'. You will be able to change the connection properties:

Property	Example	Remarks
Data Source	H2 Test	The name of the ODBC Data Source
Database	test	The database name. Only simple names are supported at this time; relative or absolute path are not supported in the database name. By default, the database is stored in the current working directory where the Server is started except when the -baseDir setting is used. The name must be at least 3 characters.
Server	localhost	The server name or IP address. By default, only remote connections are allowed
User Name	sa	The database user name.
SSL Mode	disabled	At this time, SSL is not supported.
Port	5435	The port where the PG Server is listening.
Password	sa	The database password.

To improve performance, please enable 'server side prepare' under Options / Datasource / Page 2 / Server side prepare.

Afterwards, you may use this data source.

PG Protocol Support Limitations

At this time, only a subset of the PostgreSQL network protocol is implemented. Also, there may be compatibility problems on the SQL level, with the catalog, or with text encoding. Problems are fixed as they are found. Currently, statements can not be canceled when using the PG protocol.

PostgreSQL ODBC Driver Setup requires a database password; that means it is not possible to connect to H2 databases without password. This is a limitation of the ODBC driver.

Security Considerations

Currently, the PG Server does not support challenge response or encrypt passwords. This may be a problem if an attacker can listen to the data transferred between the ODBC driver and the server, because the password is readable to the attacker. Also, it is currently not possible to use encrypted SSL connections. Therefore the ODBC driver should not be used where security is important.

Using H2 in Microsoft .NET

The database can be used from Microsoft .NET even without using Java, by using IKVM.NET. You can access a H2 database on .NET using the JDBC API, or using the ADO.NET interface.

Using the ADO.NET API on .NET

An implementation of the ADO.NET interface is available in the open source project H2Sharp .

Using the JDBC API on .NET

- Install the .NET Framework from Microsoft . Mono has not yet been tested.
- Install IKVM.NET .
- Copy the h2*.jar file to ikvm/bin
- Run the H2 Console using: ikvm -jar h2*.jar
- Convert the H2 Console to an .exe file using: ikvmc -target:winexe h2*.jar . You may ignore the warnings.
- Create a .dll file using (change the version accordingly): ikvmc.exe -target:library -version:1.0.69.0 h2*.jar

If you want your C# application use H2, you need to add the h2.dll and the IKVM.OpenJDK.ClassLibrary.dll to your C# solution. Here some sample code:

```
using System;
using java.sql;

class Test
{
    static public void Main()
    {
        org.h2.Driver.load();
        Connection conn = DriverManager.getConnection("jdbc:h2:~/test", "sa", "sa");
        Statement stat = conn.createStatement();
        ResultSet rs = stat.executeQuery("SELECT 'Hello World'");
        while (rs.next())
        {
            Console.WriteLine(rs.getString(1));
        }
    }
}
```

ACID

In the database world, ACID stands for:

- Atomicity: transactions must be atomic, meaning either all tasks are performed or none.
- Consistency: all operations must comply with the defined constraints.
- Isolation: transactions must be isolated from each other.
- Durability: committed transaction will not be lost.

Atomicity

Transactions in this database are always atomic.

Consistency

By default, this database is always in a consistent state. Referential integrity rules are enforced except when explicitly disabled.

Isolation

For H2, as with most other database systems, the default isolation level is 'read committed'. This provides better performance, but also means that transactions are not completely isolated. H2 supports the transaction isolation levels 'serializable', 'read committed', and 'read uncommitted'.

Durability

This database does not guarantee that all committed transactions survive a power failure. Tests show that all databases sometimes lose transactions on power failure (for details, see below). Where losing transactions is not acceptable, a laptop or

UPS (uninterruptible power supply) should be used. If durability is required for all possible cases of hardware failure, clustering should be used, such as the H2 clustering mode.

Durability Problems

Complete durability means all committed transaction survive a power failure. Some databases claim they can guarantee durability, but such claims are wrong. A durability test was run against H2, HSQLDB, PostgreSQL, and Derby. All of those databases sometimes lose committed transactions. The test is included in the H2 download, see org.h2.test.poweroff.Test.

Ways to (Not) Achieve Durability

Making sure that committed transactions are not lost is more complicated than it seems first. To guarantee complete durability, a database must ensure that the log record is on the hard drive before the commit call returns. To do that, databases use different methods. One is to use the 'synchronous write' file access mode. In Java, RandomAccessFile supports the modes "rws" and "rwd":

- rwd: every update to the file's content is written synchronously to the underlying storage device.
- rws: in addition to rwd, every update to the metadata is written synchronously.

A test (org.h2.test.poweroff.TestWrite) with one of those modes achieves around 50 thousand write operations per second. Even when the operating system write buffer is disabled, the write rate is around 50 thousand operations per second. This feature does not force changes to disk because it does not flush all buffers. The test updates the same byte in the file again and again. If the hard drive was able to write at this rate, then the disk would need to make at least 50 thousand revolutions per second, or 3 million RPM (revolutions per minute). There are no such hard drives. The hard drive used for the test is about 7200 RPM, or about 120 revolutions per second. There is an overhead, so the maximum write rate must be lower than that.

Calling fsync flushes the buffers. There are two ways to do that in Java:

- FileDescriptor.sync(). The documentation says that this forces all system buffers to synchronize with the underlying device. Sync is supposed to return after all in-memory modified copies of buffers associated with this FileDescriptor have been written to the physical medium.
- FileChannel.force() (since JDK 1.4). This method is supposed to force any updates to this channel's file to be written to the storage device that contains it.

By default, MySQL calls fsync for each commit. When using one of those methods, only around 60 write operations per second can be achieved, which is consistent with the RPM rate of the hard drive used. Unfortunately, even when calling FileDescriptor.sync() or FileChannel.force(), data is not always persisted to the hard drive, because most hard drives do not obey fsync(): see Your Hard Drive Lies to You . In Mac OS X, fsync does not flush hard drive buffers. See Bad fsync? . So the situation is confusing, and tests prove there is a problem.

Trying to flush hard drive buffers is hard, and if you do the performance is very bad. First you need to make sure that the hard drive actually flushes all buffers. Tests show that this can not be done in a reliable way. Then the maximum number of transactions is around 60 per second. Because of those reasons, the default behavior of H2 is to delay writing committed transactions.

In H2, after a power failure, a bit more than one second of committed transactions may be lost. To change the behavior, use SET WRITE_DELAY and CHECKPOINT SYNC. Most other databases support commit delay as well. In the performance comparison, commit delay was used for all databases that support it.

Running the Durability Test

To test the durability / non-durability of this and other databases, you can use the test application in the package org.h2.test.poweroff. Two computers with network connection are required to run this test. One computer just listens, while the test application is run (and power is cut) on the other computer. The computer with the listener application opens a TCP/IP port and listens for an incoming connection. The second computer first connects to the listener, and then created the databases and starts inserting records. The connection is set to 'autocommit', which means after each inserted record a commit is performed automatically. Afterwards, the test computer notifies the listener that this record was inserted successfully. The listener computer displays the last inserted record number every 10 seconds. Now, switch off the power manually, then restart the computer, and run the application again. You will find out that in most cases, none of the databases contains all the records that the listener computer knows about. For details, please consult the source code of the listener and test application.

Using the Recover Tool

The recover tool can be used to extract the contents of a data file, even if the database is corrupted. It also extracts the content of the log file or large objects (CLOB or BLOB). To run the tool, type on the command line:

```
java -cp h2*.jar org.h2.tools.Recover
```

For each database in the current directory, a text file will be created. This file contains raw insert statements (for the data) and data definition (DDL) statements to recreate the schema of the database. This file can be executed using the RunScript tool or a RUNSCRIPT FROM SQL statement. The script includes at least one CREATE USER statement. If you run the script against a database that was created with the same user, or if there are conflicting users, running the script will fail. Consider running the script against a database that was created with a user name that is not in the script.

File Locking Protocols

Whenever a database is opened, a lock file is created to signal other processes that the database is in use. If the database is closed, or if the process that opened the database terminates, this lock file is deleted.

In special cases (if the process did not terminate normally, for example because there was a power failure), the lock file is not deleted by the process that created it. That means the existence of the lock file is not a safe protocol for file locking. However, this software uses a challenge-response protocol to protect the database files. There are two methods (algorithms) implemented to provide both security (that is, the same database files cannot be opened by two processes at the same time) and simplicity (that is, the lock file does not need to be deleted manually by the user). The two methods are 'file method' and 'socket methods'.

File Locking Method 'File'

The default method for database file locking is the 'File Method'. The algorithm is:

- When the lock file does not exist, it is created (using the atomic operation File.createNewFile). Then, the process waits a little bit (20ms) and checks the file again. If the file was changed during this time, the operation is aborted. This protects against a race condition when a process deletes the lock file just after one create it, and a third process creates the file again. It does not occur if there are only two writers.
- If the file can be created, a random number is inserted together with the locking method ('file'). Afterwards, a watchdog thread is started that checks regularly (every second once by default) if the file was deleted or modified by another (challenger) thread / process. Whenever that occurs, the file is overwritten with the old data. The watchdog thread runs with high priority so that a change to the lock file does not get through undetected even if the system is very busy. However, the watchdog thread does use very little resources (CPU time), because it waits most of the time. Also, the watchdog only reads from the hard disk and does not write to it.
- If the lock file exists, and it was modified in the 20 ms, the process waits for some time (up to 10 times). If it was still changed, an exception is thrown (database is locked). This is done to eliminate race conditions with many concurrent writers. Afterwards, the file is overwritten with a new version (challenge). After that, the thread waits for 2 seconds. If there is a watchdog thread protecting the file, he will overwrite the change and this process will fail to lock the database. However, if there is no watchdog thread, the lock file will still be as written by this thread. In this case, the file is deleted and atomically created again. The watchdog thread is started in this case and the file is locked.

This algorithm is tested with over 100 concurrent threads. In some cases, when there are many concurrent threads trying to lock the database, they block each other (meaning the file cannot be locked by any of them) for some time. However, the file never gets locked by two threads at the same time. However using that many concurrent threads / processes is not the common use case. Generally, an application should throw an error to the user if it cannot open a database, and not try again in a (fast) loop.

File Locking Method 'Socket'

There is a second locking mechanism implemented, but disabled by default. To use it, append ;FILE_LOCK=SOCKET to the database URL. The algorithm is:

- If the lock file does not exist, it is created. Then a server socket is opened on a defined port, and kept open. The port and IP address of the process that opened the database is written into the lock file.
- If the lock file exists, and the lock method is 'file', then the software switches to the 'file' method.
- If the lock file exists, and the lock method is 'socket', then the process checks if the port is in use. If the original process is still running, the port is in use and this process throws an exception (database is in use). If the original process died (for example due to a power failure, or abnormal termination of the virtual machine), then the port was released. The new process deletes the lock file and starts again.

This method does not require a watchdog thread actively polling (reading) the same file every second. The problem with this method is, if the file is stored on a network share, two processes (running on different computers) could still open the same database files, if they do not have a direct TCP/IP connection.

Protection against SQL Injection

What is SQL Injection

This database engine provides a solution for the security vulnerability known as 'SQL Injection'. Here is a short description of what SQL injection means. Some applications build SQL statements with embedded user input such as:

```
String sql = "SELECT * FROM USERS WHERE PASSWORD='"+pwd+"'";
ResultSet rs = conn.createStatement().executeQuery(sql);
```

If this mechanism is used anywhere in the application, and user input is not correctly filtered or encoded, it is possible for a user to inject SQL functionality or statements by using specially built input such as (in this example) this password: ' OR ''='. In this case the statement becomes:

```
SELECT * FROM USERS WHERE PASSWORD='' OR ''='';
```

Which is always true no matter what the password stored in the database is. For more information about SQL Injection, see Glossary and Links.

Disabling Literals

SQL Injection is not possible if user input is not directly embedded in SQL statements. A simple solution for the problem above is to use a PreparedStatement:

```
String sql = "SELECT * FROM USERS WHERE PASSWORD=?";
PreparedStatement prep = conn.prepareStatement(sql);
prep.setString(1, pwd);
ResultSet rs = prep.executeQuery();
```

This database provides a way to enforce usage of parameters when passing user input to the database. This is done by disabling embedded literals in SQL statements. To do this, execute the statement:

```
SET ALLOW_LITERALS NONE;
```

Afterwards, SQL statements with text and number literals are not allowed any more. That means, SQL statement of the form WHERE NAME='abc' or WHERE CustomerId=10 will fail. It is still possible to use PreparedStatements and parameters as described above. Also, it is still possible to generate SQL statements dynamically, and use the Statement API, as long as the SQL statements do not include literals. There is also a second mode where number literals are allowed: SET ALLOW_LITERALS NUMBERS. To allow all literals, execute SET ALLOW_LITERALS ALL (this is the default setting). Literals can only be enabled or disabled by an administrator.

Using Constants

Disabling literals also means disabling hard-coded 'constant' literals. This database supports defining constants using the CREATE CONSTANT command. Constants can be defined only when literals are enabled, but used even when literals are disabled. To avoid name clashes with column names, constants can be defined in other schemas:

```
CREATE SCHEMA CONST AUTHORIZATION SA;
CREATE CONSTANT CONST.ACTIVE VALUE 'Active';
CREATE CONSTANT CONST.INACTIVE VALUE 'Inactive';
SELECT * FROM USERS WHERE TYPE=CONST.ACTIVE;
```

Even when literals are enabled, it is better to use constants instead of hard-coded number or text literals in queries or views. With constants, typos are found at compile time, the source code is easier to understand and change.

Using the ZERO() Function

It is not required to create a constant for the number 0 as there is already a built-in function ZERO():

```
SELECT * FROM USERS WHERE LENGTH(PASSWORD)=ZERO();
```

Protection against Remote Access

By default this database does not allow others to connect when starting the H2 Console, the TCP server, or the PG server. Remote access can be enabled using the command line options -webAllowOthers, -tcpAllowOthers, and -pgAllowOthers. If you enable remote access, please also consider using the options -baseDir and -ifExists, so that remote users can not create new databases or access existing databases with weak passwords. Also, ensure the existing accessible databases are protected using a strong password.

Restricting Class Loading and Usage

By default there is no restriction on loading classes and executing Java code for admins. That means an admin may call system functions such as System.setProperty by executing:

```
CREATE ALIAS SET_PROPERTY FOR "java.lang.System.setProperty";
CALL SET_PROPERTY('abc', '1');
CREATE ALIAS GET_PROPERTY FOR "java.lang.System.getProperty";
CALL GET_PROPERTY('abc');
```

To restrict users (including admins) from loading classes and executing code, the list of allowed classes can be set in the system property h2.allowedClasses in the form of a comma separated list of classes or patterns (items ending with '*'). By default all classes are allowed. Example:

```
java -Dh2.allowedClasses=java.lang.Math,com.acme.*
```

This mechanism is used for all user classes, including database event listeners, trigger classes, user-defined functions, user-defined aggregate functions, and JDBC driver classes (with the exception of the H2 driver) when using the H2 Console.

Security Protocols

The following paragraphs document the security protocols used in this database. These descriptions are very technical and only intended for security experts that already know the underlying security primitives.

User Password Encryption

When a user tries to connect to a database, the combination of user name, @, and password are hashed using SHA-256, and this hash value is transmitted to the database. This step does not protect against an attacker that re-uses the value if he is able to listen to the (unencrypted) transmission between the client and the server. But, the passwords are never transmitted as plain text, even when using an unencrypted connection between client and server. That means if a user reuses the same password for different things, this password is still protected up to some point. See also 'RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication' for more information.

When a new database or user is created, a new cryptographically secure random salt value is generated. The size of the salt is 64 bits. Using the random salt reduces the risk of an attacker pre-calculating hash values for many different (commonly used) passwords.

The combination of user-password hash value (see above) and salt is hashed using SHA-256. The resulting value is stored in the database. When a user tries to connect to the database, the database combines user-password hash value with the stored salt value and calculates the hash value. Other products use multiple iterations (hash the hash value again and again), but this is not done in this product to reduce the risk of denial of service attacks (where the attacker tries to connect with bogus passwords, and the server spends a lot of time calculating the hash value for each password). The reasoning is: if the attacker has access to the hashed passwords, he also has access to the data in plain text, and therefore does not need the password any more. If the data is protected by storing it on another computer and only accessible remotely, then the iteration count is not required at all.

File Encryption

The database files can be encrypted using two different algorithms: AES-128 and XTEA (using 32 rounds). The reasons for supporting XTEA is performance (XTEA is about twice as fast as AES) and to have an alternative algorithm if AES is suddenly broken.

When a user tries to connect to an encrypted database, the combination of the word 'file', @, and the file password is hashed using SHA-256. This hash value is transmitted to the server.

When a new database file is created, a new cryptographically secure random salt value is generated. The size of the salt is 64 bits. The combination of the file password hash and the salt value is hashed 1024 times using SHA-256. The reason for the iteration is to make it harder for an attacker to calculate hash values for common passwords.

The resulting hash value is used as the key for the block cipher algorithm (AES-128 or XTEA with 32 rounds). Then, an initialization vector (IV) key is calculated by hashing the key again using SHA-256. This is to make sure the IV is unknown to the attacker. The reason for using a secret IV is to protect against watermark attacks.

Before saving a block of data (each block is 8 bytes long), the following operations are executed: first, the IV is calculated by encrypting the block number with the IV key (using the same block cipher algorithm). This IV is combined with the plain text using XOR. The resulting data is encrypted using the AES-128 or XTEA algorithm.

When decrypting, the operation is done in reverse. First, the block is decrypted using the key, and then the IV is calculated combined with the decrypted text using XOR.

Therefore, the block cipher mode of operation is CBC (cipher-block chaining), but each chain is only one block long. The advantage over the ECB (electronic codebook) mode is that patterns in the data are not revealed, and the advantage over multi block CBC is that flipped cipher text bits are not propagated to flipped plaintext bits in the next block.

Database encryption is meant for securing the database while it is not in use (stolen laptop and so on). It is not meant for cases where the attacker has access to files while the database is in use. When he has write access, he can for example replace pieces of files with pieces of older versions and manipulate data like this.

File encryption slows down the performance of the database engine. Compared to unencrypted mode, database operations take about 2.2 times longer when using XTEA, and 2.5 times longer using AES (embedded mode).

Wrong Password / User Name Delay

To protect against remote brute force password attacks, the delay after each unsuccessful login gets double as long. Use the system properties h2.delayWrongPasswordMin and h2.delayWrongPasswordMax to change the minimum (the default is 250 milliseconds) or maximum delay (the default is 4000 milliseconds, or 4 seconds). The delay only applies for those using the wrong password. Normally there is no delay for a user that knows the correct password, with one exception: after using the wrong password, there is a delay of up to (randomly distributed) the same delay as for a wrong password. This is to protect

against parallel brute force attacks, so that an attacker needs to wait for the whole delay. Delays are synchronized. This is also required to protect against parallel attacks.

There is only one exception message for both wrong user and for wrong password, to make it harder to get the list of user names. It is not possible from the stack trace to see if the user name was wrong or the password.

HTTPS Connections

The web server supports HTTP and HTTPS connections using SSLServerSocket. There is a default self-certified certificate to support an easy starting point, but custom certificates are supported as well.

SSL/TLS Connections

Remote SSL/TLS connections are supported using the Java Secure Socket Extension (SSLServerSocket / SSLSocket). By default, anonymous SSL is enabled. The default cipher suite is SSL_DH_anon_WITH_RC4_128_MD5 .

To use your own keystore, set the system properties javax.net.ssl.keyStore and javax.net.ssl.keyStorePassword before starting the H2 server and client. See also Customizing the Default Key and Trust Stores, Store Types, and Store Passwords for more information.

To disable anonymous SSL, set the system property h2.enableAnonymousSSL to false.

Universally Unique Identifiers (UUID)

This database supports the UUIDs. Also supported is a function to create new UUIDs using a cryptographically strong pseudo random number generator. With random UUIDs, the chance of two having the same value can be calculated using the probability theory. See also 'Birthday Paradox'. Standardized randomly generated UUIDs have 122 random bits. 4 bits are used for the version (Randomly generated UUID), and 2 bits for the variant (Leach-Salz). This database supports generating such UUIDs using the built-in function RANDOM_UUID(). Here is a small program to estimate the probability of having two identical UUIDs after generating a number of values:

```
public class Test {
    public static void main(String[] args) throws Exception {
        double x = Math.pow(2, 122);
        for (int i = 35; i < 62; i++) {
            double n = Math.pow(2, i);
            double p = 1 - Math.exp(-(n * n) / 2 / x);
            System.out.println("2^" + i + "=" + (1L << i) +
                    " probability: 0" +
                    String.valueOf(1 + p).substring(1));
        }
    }
}
```

Some values are:

```
2^36=68'719'476'736 probability: 0.000'000'000'000'000'4
2^41=2'199'023'255'552 probability: 0.000'000'000'000'4
2^46=70'368'744'177'664 probability: 0.000'000'000'4
```

To help non-mathematicians understand what those numbers mean, here a comparison: one's annual risk of being hit by a meteorite is estimated to be one chance in 17 billion, that means the probability is about 0.000'000'000'06.

Settings Read from System Properties

Some settings of the database can be set on the command line using -DpropertyName=value. It is usually not required to change those settings manually. The settings are case sensitive. Example:

```
java -Dh2.serverCachedObjects=256 org.h2.tools.Server
```

The current value of the settings can be read in the table INFORMATION_SCHEMA.SETTINGS.

For a complete list of settings, see SysProperties .

Setting the Server Bind Address

Usually server sockets accept connections on any/all local addresses. This may be a problem on multi-homed hosts. To bind only to one address, use the system property h2.bindAddress. This setting is used for both regular server sockets and for SSL server sockets. IPv4 and IPv6 address formats are supported.

Pluggable File System

This database supports a pluggable file system API. The file system implementation is selected using a file name prefix. The following file systems are included:

- **zip:** read-only zip-file based file system. Format: zip:/zipFileName!/fileName.
- **nio:** file system that uses FileChannel instead of RandomAccessFile (faster in some operating systems).
- **nioMapped:** file system that uses memory mapped files (faster in some operating systems).
- **split:** file system that splits files in 1 GB files (stackable with other file systems).
- **memFS:** in-memory file system (experimental; used for testing).
- **memLZF:** compressing in-memory file system (experimental; used for testing).

As an example, to use the the **nio** file system, use the following database URL: jdbc:h2:nio:~/test .

To register a new file system, extend the classes org.h2.store.fs.FileSystem and FileObject, and call the method FileSystem.register before using it.

Limits and Limitations

This database has the following known limitations:

- Database file size limits (excluding BLOB and CLOB data): With the default storage mechanism, the maximum file size is currently 256 GB for the data, and 256 GB for the index. With the page store (experimental): 4 TB or higher.
- BLOB and CLOB size limit: every CLOB or BLOB can be up to 256 GB.
- The maximum file size for FAT or FAT32 file systems is 4 GB. That means when using FAT or FAT32, the limit is 4 GB for the data. This is the limitation of the file system. The database does provide a workaround for this problem, it is to use the file name prefix 'split:'. In that case files are split into files of 1 GB by default. An example database URL is: jdbc:h2:split:~/test .
- The maximum number of rows per table is 2'147'483'648.
- Main memory requirements: The larger the database, the more main memory is required. With the default storage mechanism, the minimum main memory required for a 12 GB database is around 240 MB. With the page store (experimental), the minimum main memory required is much lower, around 1 MB for each 8 GB database file size.
- Limit on the complexity of SQL statements. Statements of the following form will result in a stack overflow exception:

```
SELECT * FROM DUAL WHERE X = 1
OR X = 2 OR X = 2 OR X = 2 OR X = 2 OR X = 2
```

```
-- repeat previous line 500 times --
```

- There is no limit for the following entities, except the memory and storage capacity: maximum identifier length (table name, column name, and so on); maximum number of tables, columns, indexes, triggers, and other database objects; maximum statement length, number of parameters per statement, tables per statement, expressions in order by, group by, having, and so on; maximum rows per query; maximum columns per table, columns per index, indexes per table, lob columns per table, and so on; maximum row length, index row length, select row length; maximum length of a varchar column, decimal column, literal in a statement.
- For limitations on data types, see the documentation of the respective Java data type or the data type documentation of this database.

Glossary and Links

Term	Description
AES-128	A block encryption algorithm. See also: Wikipedia: AES
Birthday Paradox	Describes the higher than expected probability that two persons in a room have the same birthday. Also valid for randomly generated UUIDs. See also: Wikipedia: Birthday Paradox
Digest	Protocol to protect a password (but not to protect data). See also: RFC 2617: HTTP Digest Access Authentication
GCJ	Compiler for Java. GNU Compiler for the Java and NativeJ (commercial)
HTTPS	A protocol to provide security to HTTP connections. See also: RFC 2818: HTTP Over TLS
Modes of Operation	Wikipedia: Block cipher modes of operation
Salt	Random number to increase the security of passwords. See also: Wikipedia: Key derivation function
SHA-256	A cryptographic one-way hash function. See also: Wikipedia: SHA hash functions
SQL Injection	A security vulnerability where an application embeds SQL statements or expressions in user input. See also: Wikipedia: SQL Injection
Watermark Attack	Security problem of certain encryption programs where the existence of certain data can be proven without decrypting. For more information, search in the internet for 'watermark attack cryptoloop'
SSL/TLS	Secure Sockets Layer / Transport Layer Security. See also: Java Secure Socket Extension (JSSE)
XTEA	A block encryption algorithm. See also: Wikipedia: XTEA

SQL Grammar

Commands (Data Manipulation)

SELECT
INSERT
UPDATE
DELETE
BACKUP
CALL
EXPLAIN
MERGE
RUNSCRIPT
SCRIPT
SHOW

Commands (Data Definition)

ALTER INDEX RENAME
ALTER SEQUENCE
ALTER TABLE ADD
ALTER TABLE ADD CONSTRAINT
ALTER TABLE ALTER COLUMN
ALTER TABLE ALTER COLUMN RENAME
ALTER TABLE ALTER COLUMN RESTART
ALTER TABLE ALTER COLUMN SELECTIVITY
ALTER TABLE ALTER COLUMN SET DEFAULT
ALTER TABLE ALTER COLUMN SET NOT NULL
ALTER TABLE ALTER COLUMN SET NULL
ALTER TABLE DROP COLUMN
ALTER TABLE DROP CONSTRAINT
ALTER TABLE SET
ALTER TABLE RENAME
ALTER USER ADMIN
ALTER USER RENAME
ALTER USER SET PASSWORD
ALTER VIEW
ANALYZE
COMMENT
CREATE AGGREGATE
CREATE ALIAS
CREATE CONSTANT
CREATE DOMAIN
CREATE INDEX
CREATE LINKED TABLE
CREATE ROLE
CREATE SCHEMA
CREATE SEQUENCE
CREATE TABLE
CREATE TRIGGER
CREATE USER
CREATE VIEW
DROP AGGREGATE
DROP ALIAS
DROP ALL OBJECTS
DROP CONSTANT
DROP DOMAIN
DROP INDEX
DROP ROLE
DROP SCHEMA
DROP SEQUENCE
DROP TABLE
DROP TRIGGER
DROP USER

DROP VIEW
TRUNCATE TABLE

Commands (Other)

CHECKPOINT
CHECKPOINT SYNC
COMMIT
COMMIT TRANSACTION
GRANT RIGHT
GRANT ROLE
HELP
PREPARE COMMIT
REVOKE RIGHT
REVOKE ROLE
ROLLBACK
ROLLBACK TRANSACTION
SAVEPOINT
SET @
SET ALLOW_LITERALS
SET AUTOCOMMIT
SET CACHE_SIZE
SET CLUSTER
SET COLLATION
SET COMPRESS_LOB
SET DATABASE_EVENT_LISTENER
SET DB_CLOSE_DELAY
SET DEFAULT_LOCK_TIMEOUT
SET DEFAULT_TABLE_TYPE
SET EXCLUSIVE
SET IGNORECASE
SET LOCK_MODE
SET LOCK_TIMEOUT
SET LOG
SET MAX_LENGTH_INPLACE_LOB
SET MAX_LOG_SIZE
SET MAX_MEMORY_ROWS
SET MAX_MEMORY_UNDO
SET MAX_OPERATION_MEMORY
SET MODE
SET MULTI_THREADED
SET OPTIMIZE_REUSE_RESULTS
SET PASSWORD
SET QUERY_TIMEOUT
SET REFERENTIAL_INTEGRITY
SET SALT HASH
SET SCHEMA
SET SCHEMA_SEARCH_PATH
SET THROTTLE
SET TRACE_LEVEL
SET TRACE_MAX_FILE_SIZE
SET UNDO_LOG
SET WRITE_DELAY
SHUTDOWN

Other Grammar

Comments
Select Part
From Part
Constraint
Referential Constraint
Table Expression
Order
Index Column
Expression

And Condition
Condition
Condition Right Hand Side
Compare
Operand
Summand
Factor
Term
Value
Case
Case When
Cipher
Select Expression
Data Type
Name
Alias
Quoted Name
String
Dollar Quoted String
Int
Long
Hex Number
Decimal
Double
Date
Time
Timestamp
Boolean
Bytes
Array
Null
Hex
Digit

System Tables

Information Schema
Range Table

SELECT

```
{SELECT selectPart FROM fromPart | FROM fromPart SELECT selectPart}
[WHERE expression] [GROUP BY expression [,...]] [HAVING expression]
[{UNION [ALL] | MINUS | EXCEPT | INTERSECT} select]
[ORDER BY order [,...]] [LIMIT expression [OFFSET expression]
[SAMPLE_SIZE rowCountInt]] [FOR UPDATE]
```

Selects data from a table or multiple tables.

LIMIT limits the number of rows returned by the query, OFFSET specified how many rows to skip. SAMPLE_SIZE limits the number of rows read for aggregate queries. If FOR UPDATE is specified, the tables are locked for writing.

Example:

```
SELECT * FROM TEST;
SELECT * FROM (SELECT ID, COUNT(*) FROM TEST
    GROUP BY ID UNION SELECT NULL, COUNT(*) FROM TEST)
    ORDER BY 1 NULLS LAST;
```

INSERT

```
INSERT INTO tableName [(columnName [,...])]
{VALUES {( [{DEFAULT | expression} [,...]] )} [,...] | select}
```

Inserts a new row / new rows into a table.

Example:

INSERT INTO TEST VALUES(1, 'Hello')

UPDATE

```
UPDATE tableName SET {columnName= {DEFAULT | expression} } [,...]
[WHERE expression]
```

Updates data in a table.

Example:

UPDATE TEST SET NAME='Hi' WHERE ID=1

DELETE

```
DELETE FROM tableName [WHERE expression]
```

Deletes rows form a table.

Example:

DELETE FROM TEST WHERE ID=2

BACKUP

```
BACKUP TO fileNameString
```

Backs up the database files to a .zip file. Objects are not locked. Admin rights are required to execute this command.

Example:

BACKUP TO 'backup.zip'

CALL

```
CALL expression
```

Calculates a simple expression.

Example:

CALL 15*25

EXPLAIN

```
EXPLAIN [PLAN FOR] {select | insert | update | delete}
```

Shows the execution plan for a statement.

Example:

EXPLAIN SELECT * FROM TEST WHERE ID=1

MERGE

```
MERGE INTO tableName [(columnName [,...])] [KEY(columnName [,...])]
{VALUES {( [{DEFAULT | expression} [,...]] )} [,...] | select}
```

Updates existing rows, and insert rows that don't exist. If no key column is specified, the primary key columns are used to find the row. If more than one row per new row is affected, an exception is thrown. If the table contains an auto-incremented key or identity column, and the row was updated, the generated key is set to 0; otherwise it is set to the new key.

Example:

MERGE INTO TEST KEY(ID) VALUES(2, 'World')

RUNSCRIPT

```
RUNSCRIPT FROM fileNameString
[COMPRESSION {DEFLATE|LZF|ZIP|GZIP}]
[CIPHER cipher PASSWORD string]
[CHARSET charsetString]
```

Runs a SQL script from a file. The script is a text file containing SQL statements; each statement must end with ';'. This command can be used to restore a database from a backup. The password must be in single quotes; it is case sensitive and can contain spaces.

The compression algorithm must match the one used when creating the script. When using encryption, only DEFLATE and LZF are supported (LZF is faster but uses more space). Instead of a file, an URL may be used.

Admin rights are required to execute this command.

Example:

RUNSCRIPT FROM 'backup'

SCRIPT

```
SCRIPT [SIMPLE] [NODATA] [NOPASSWORDS] [NOSETTINGS] [DROP]
[BLOCKSIZE blockSizeInt] [TO fileNameString
[COMPRESSION {DEFLATE|LZF|ZIP|GZIP}]
```

```
[CIPHER cipher PASSWORD string]]
```

Creates a SQL script with or without the insert statements. The simple format does not use multi-row insert statements. If no file name is specified, the script is returned as a result set. This command can be used to create a backup of the database. For long term storage, it is more portable than copying the database files.

If the DROP option is specified, drop statements are created for tables, views, and sequences. If the block size is set, CLOB and BLOB values larger than this size are split into separate blocks. If a file name is specified, then the whole script (including insert statements) is written to this file, and a result set without the insert statements is returned. When using encryption, only DEFLATE and LZF are supported (LZF is faster but uses more space).

This command locks objects while it is running. The password must be in single quotes; it is case sensitive and can contain spaces.

Example:

SCRIPT NODATA

SHOW

```
SHOW { SCHEMAS | TABLES [FROM schemaName] |
COLUMNS FROM tableName [FROM schemaName] }
```

Lists the schemas, tables, or the columns of a table.

Example:

SHOW TABLES

ALTER INDEX RENAME

```
ALTER INDEX indexName RENAME TO newIndexName
```

Renames an index. This command commits an open transaction.

Example:

ALTER INDEX IDXNAME RENAME TO IDX_TEST_NAME

ALTER SEQUENCE

```
ALTER SEQUENCE sequenceName [RESTART WITH long] [INCREMENT BY long]
```

Changes the next value and the increment of a sequence. This command does not commit the current transaction; however the new value is used by other transactions immediately, and rolling back this command has no effect.

Example:

ALTER SEQUENCE SEQ_ID RESTART WITH 1000

ALTER TABLE ADD

```
ALTER TABLE tableName ADD name dataType [DEFAULT expression]
[[NOT] NULL] [AUTO_INCREMENT | IDENTITY] [BEFORE columnName]
```

Adds a new column to a table. This command commits an open transaction.

Example:

ALTER TABLE TEST ADD CREATEDATE TIMESTAMP

ALTER TABLE ADD CONSTRAINT

```
ALTER TABLE tableName ADD constraint [CHECK|NOCHECK]
```

Adds a constraint to a table. If NOCHECK is specified, existing rows are not checked for consistency (the default is to check consistency for existing rows). It is not possible to disable checking for unique constraints. This command commits an open transaction.

Example:

ALTER TABLE TEST ADD CONSTRAINT NAME_UNIQUE UNIQUE(NAME)

ALTER TABLE ALTER COLUMN

```
ALTER TABLE tableName ALTER COLUMN columnName dataType
[DEFAULT expression] [NOT [NULL]] [AUTO_INCREMENT | IDENTITY]
```

Changes the data type of a column. The operation fails if the data can not be converted. This command commits an open transaction.

Example:

ALTER TABLE TEST ALTER COLUMN NAME CLOB

ALTER TABLE ALTER COLUMN RENAME

```
ALTER TABLE tableName ALTER COLUMN columnName RENAME TO name
```

Renames a column. This command commits an open transaction.

Example:

ALTER TABLE TEST ALTER COLUMN NAME RENAME TO TEXT

ALTER TABLE ALTER COLUMN RESTART

```
ALTER TABLE tableName ALTER COLUMN columnName RESTART WITH long
```

Changes the next value of an auto increment column. The column must already be an auto increment column. The same transactional rules as for ALTER SEQUENCE apply.

Example:

ALTER TABLE TEST ALTER COLUMN ID RESTART WITH 10000

ALTER TABLE ALTER COLUMN SELECTIVITY

```
ALTER TABLE tableName ALTER COLUMN columnName SELECTIVITY int
```

Sets the selectivity (1-100) for a column. Setting the selectivity to 0 means the default value. Selectivity is used by the cost based optimizer to calculate the estimated cost of an index. Selectivity 100 means values are unique, 10 means every distinct value appears 10 times on average. This command commits an open transaction.

Example:

ALTER TABLE TEST ALTER COLUMN NAME SELECTIVITY 100

ALTER TABLE ALTER COLUMN SET DEFAULT

```
ALTER TABLE tableName ALTER COLUMN columnName SET DEFAULT expression
```

Changes the default value of a column. This command commits an open transaction.

Example:

ALTER TABLE TEST ALTER COLUMN NAME SET DEFAULT ''

ALTER TABLE ALTER COLUMN SET NOT NULL

```
ALTER TABLE tableName ALTER COLUMN columnName SET NOT NULL
```

Sets a column to not allow NULL. Rows may not contains NULL in this column. This command commits an open transaction.

Example:

ALTER TABLE TEST ALTER COLUMN NAME SET NOT NULL

ALTER TABLE ALTER COLUMN SET NULL

```
ALTER TABLE tableName ALTER COLUMN columnName SET NULL
```

Sets a column to allow NULL. The row may not be part of a primary key or multi-column hash index. Single column indexes on this column are dropped. This command commits an open transaction.

Example:

ALTER TABLE TEST ALTER COLUMN NAME SET NULL

ALTER TABLE DROP COLUMN

```
ALTER TABLE tableName DROP COLUMN columnName
```

Removes a column from a table. This command commits an open transaction.

Example:

ALTER TABLE TEST DROP COLUMN NAME

ALTER TABLE DROP CONSTRAINT

```
ALTER TABLE tableName DROP
{CONSTRAINT [IF EXISTS] constraintName | PRIMARY KEY}
```

Removes a constraint or a primary key from a table. This command commits an open transaction.

Example:

ALTER TABLE TEST DROP CONSTRAINT UNIQUE_NAME

ALTER TABLE SET

```
ALTER TABLE tableName SET REFERENTIAL_INTEGRITY
{FALSE | TRUE [CHECK|NOCHECK]}
```

Disables or enables referential integrity checking for a table. This command can be used inside a transaction. Enabling referential integrity does not check existing data, except if CHECK is specified. Use SET REFERENTIAL_INTEGRITY to disable it for all tables; the global flag and the flag for each table are independent.

This command commits an open transaction.

Example:

ALTER TABLE TEST SET REFERENTIAL_INTEGRITY FALSE

ALTER TABLE RENAME

```
ALTER TABLE tableName RENAME TO newName
```

Renames a table. This command commits an open transaction.

Example:

ALTER TABLE TEST RENAME TO MY_DATA

ALTER USER ADMIN

```
ALTER USER userName ADMIN {TRUE | FALSE}
```

Switches the admin flag of a user on or off.

Only unquoted or uppercase user names are allowed. Admin rights are required to execute this command. This command commits an open transaction.

Example:

ALTER USER TOM ADMIN TRUE

ALTER USER RENAME

```
ALTER USER userName RENAME TO newUserName
```

Renames a user. After renaming a user, the password becomes invalid and needs to be changed as well.

Only unquoted or uppercase user names are allowed. Admin rights are required to execute this command. This command commits an open transaction.

Example:

ALTER USER TOM RENAME TO THOMAS

ALTER USER SET PASSWORD

```
ALTER USER userName SET {PASSWORD string | SALT bytes HASH bytes}
```

Changes the password of a user. Only unquoted or uppercase user names are allowed. The password must be enclosed in single quotes. It is case sensitive and can contain spaces. The salt and hash values are hex strings.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

ALTER USER SA SET PASSWORD 'rioyxlgt'

ALTER VIEW

```
ALTER VIEW viewName RECOMPILE
```

Recompiles a view after the underlying tables have been changed or created. This command commits an open transaction.

Example:

ALTER VIEW ADDRESS_VIEW RECOMPILE

ANALYZE

```
ANALYZE [SAMPLE_SIZE rowCountInt]
```

Updates the selectivity statistics of all tables. The selectivity is used by the cost based optimizer to select the best index for a given query. If no sample size is set, up to 10000 rows per table are read. The value 0 means all rows are read. The selectivity can be set manually using ALTER TABLE ALTER COLUMN SELECTIVITY. Manual values are overwritten by this statement. The selectivity is available in the INFORMATION_SCHEMA.COLUMNS table.

This command commits an open transaction.

Example:

ANALYZE SAMPLE_SIZE 1000

COMMENT

```
COMMENT ON { { TABLE | VIEW | CONSTANT | CONSTRAINT | ALIAS | INDEX | ROLE
| SCHEMA | SEQUENCE | TRIGGER | USER | DOMAIN } [schemaName.]objectName }
| { COLUMN [schemaName.]tableName.columnName } IS expression
```

Sets the comment of a database object. Use NULL to remove the comment.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

COMMENT ON TABLE TEST IS 'Table used for testing'

CREATE AGGREGATE

```
CREATE AGGREGATE [IF NOT EXISTS] newAggregateName FOR className
```

Creates a new user-defined aggregate function. The method name must be the full qualified class name. The class must implement the interface org.h2.api.AggregateFunction.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

CREATE AGGREGATE MEDIAN FOR "com.acme.db.Median"

CREATE ALIAS

```
CREATE ALIAS [IF NOT EXISTS] newFunctionAliasName [DETERMINISTIC]
FOR classAndMethodName
```

Creates a new function alias. The method name must be the full qualified class and method name, and may optionally include the parameter classes as in "java.lang.Integer.parseInt(java.lang.String, int)"). The class and the method must both be public, and the method must be static.

Deterministic functions always return the same value for the same parameters. If the first parameter of the Java function is a java.sql.Connection, then a connection to the database is provided. This connection must not be closed. If the class contains multiple methods with the given name but different parameter count, all methods are mapped.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

```
CREATE ALIAS MY_SQRT FOR "java.lang.Math.sqrt";
CREATE ALIAS GET_SYSTEM_PROPERTY FOR "java.lang.System.getProperty";
CALL GET_SYSTEM_PROPERTY('java.class.path');
CALL GET_SYSTEM_PROPERTY('com.acme.test', 'true');
```

CREATE CONSTANT

```
CREATE CONSTANT [IF NOT EXISTS] newConstantName VALUE expression
```

Creates a new constant. This command commits an open transaction.

Example:

```
CREATE CONSTANT ONE VALUE 1
```

CREATE DOMAIN

```
CREATE DOMAIN [IF NOT EXISTS] newDomainName AS dataType [DEFAULT expression]
[[NOT] NULL] [SELECTIVITY selectivity] [CHECK condition]
```

Creates a new data type (domain). The check condition must evaluate to true or to NULL (to prevent NULL, use NOT NULL). In the condition, the term VALUE refers to the value being tested.

This command commits an open transaction.

Example:

```
CREATE DOMAIN EMAIL AS VARCHAR(255) CHECK (POSITION('@', VALUE) > 1)
```

CREATE INDEX

```
CREATE {[UNIQUE [HASH]] INDEX [IF NOT EXISTS] newIndexName
| PRIMARY KEY [HASH]} ON tableName(indexColumn [,...])
```

Creates a new index. This command commits an open transaction.

Example:

```
CREATE INDEX IDXNAME ON TEST(NAME)
```

CREATE LINKED TABLE

```
CREATE [[GLOBAL | LOCAL] TEMPORARY] LINKED TABLE [IF NOT EXISTS]
name(driverString, urlString, userString, passwordString,
[originalSchemaString,] originalTableString) [EMIT UPDATES | READONLY]
```

Creates a table link to an external table. The driver name may be empty if the driver is already loaded. If the schema name is not set, only one table with that name may exist in the target database.

Usually, for update statements, the old rows are deleted first and then the new rows are inserted. It is possible to emit update statements (except on rollback), however in this case multi-row unique key updates may not always work. Linked tables to the same database share one connection. If a query is used instead of the original table name, the table is read only.

To use JNDI to get the connection, the driver class must be a javax.naming.Context (for example javax.naming.InitialContext), and the URL must be the resource name (for example java:comp/env/jdbc/Test).

Admin rights are required to execute this command. This command commits an open transaction.

Example:

```
CREATE LINKED TABLE LINK('org.h2.Driver', 'jdbc:h2:test2', 'sa', 'sa', 'TEST');
CREATE LINKED TABLE LINK('', 'jdbc:h2:test2', 'sa', 'sa',
    '(SELECT * FROM TEST WHERE ID>0)');
CREATE LINKED TABLE LINK('javax.naming.InitialContext',
    'java:comp/env/jdbc/Test', NULL, NULL, '(SELECT * FROM TEST WHERE ID>0)');
```

CREATE ROLE

```
CREATE ROLE [IF NOT EXISTS] newRoleName
```

Creates a new role. This command commits an open transaction.

Example:

```
CREATE ROLE READONLY
```

CREATE SCHEMA

```
CREATE SCHEMA [IF NOT EXISTS] name [AUTHORIZATION ownerUserName]
```

Creates a new schema. If no owner is specified, the current user is used. The user that executes the command must have admin rights, as well as the owner.

This command commits an open transaction.

Example:

```
CREATE SCHEMA TEST_SCHEMA AUTHORIZATION SA
```

CREATE SEQUENCE

```
CREATE SEQUENCE [IF NOT EXISTS] newSequenceName [START WITH long]
[INCREMENT BY long] [CACHE long]
```

Creates a new sequence. The data type of a sequence is BIGINT. Used values are never re-used, even when the transaction is rolled back. The cache is the number of pre-allocated numbers. If the system crashes without closing the database, at most this many numbers are lost. The default cache size is 32.

This command commits an open transaction.

Example:

CREATE SEQUENCE SEQ_ID

CREATE TABLE

```
CREATE [CACHED | MEMORY | TEMP | [GLOBAL | LOCAL] TEMPORARY]
TABLE [IF NOT EXISTS] name
{ ( {name dataType [{AS computedColumnExpression | DEFAULT expression}]
[[NOT] NULL] [{AUTO_INCREMENT | IDENTITY}[(startInt [, incrementInt])]]
[SELECTIVITY selectivity] [PRIMARY KEY [HASH] | UNIQUE] | constraint} [,...] )
[AS select] [NOT PERSISTENT] } | { AS select }
```

Creates a new table.

Cached tables (the default) are persistent, and the number of rows is not limited by the main memory. Memory tables are persistent, but the index data is kept in main memory, that means memory tables should not get too large. Tables with the NOT PERSISTENT modifier are kept fully in memory, and all rows are lost when the database is closed. Temporary tables are not persistent. Temporary tables can be global (accessible by all connections) or local (only accessible by the current connection). The default is for temporary tables is global.

Identity and auto-increment columns are columns with a sequence as the default. The column declared as the identity columns is implicitly the primary key column of this table (unlike auto-increment columns).

This command commits an open transaction.

Example:

CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255))

CREATE TRIGGER

```
CREATE TRIGGER [IF NOT EXISTS] newTriggerName
{BEFORE | AFTER} {INSERT | UPDATE | DELETE} [,...]
ON tableName
[FOR EACH ROW] [QUEUE int] [NOWAIT]
CALL triggeredClassName
```

Creates a new trigger. The trigger class must be public. Nested and inner classes are not supported. Before triggers are called after data conversion is made, default values are set, null and length constraint checks have been made; but before other constraints have been checked.

This command commits an open transaction.

Example:

CREATE TRIGGER TRIG_INS BEFORE INSERT ON TEST FOR EACH ROW CALL "MyTrigger"

CREATE USER

```
CREATE USER [IF NOT EXISTS] newUserName
{PASSWORD string | SALT bytes HASH bytes}
[ADMIN]
```

Creates a new user. For compatibility, only unquoted or uppercase user names are allowed. The password must be in single quotes. It is case sensitive and can contain spaces. The salt and hash values are hex strings.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

CREATE USER GUEST PASSWORD 'abc'

CREATE VIEW

```
CREATE [FORCE] VIEW [IF NOT EXISTS] newViewName [(columnName [,..])]
AS select
```

Creates a new view. If the force option is used, then the view is created even if the underlying table(s) don't exist.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

CREATE VIEW TEST_VIEW AS SELECT * FROM TEST WHERE ID < 100

DROP AGGREGATE

```
DROP AGGREGATE [IF EXISTS] aggregateName
```

Drops an existing user-defined aggregate function.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

CREATE AGGREGATE MEDIAN

DROP ALIAS

```
DROP ALIAS [IF EXISTS] functionAliasName
```

Drops an existing function alias.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

CREATE ALIAS MY_SQRT

DROP ALL OBJECTS

```
DROP ALL OBJECTS [DELETE FILES]
```

Drops all existing views, tables, sequences, schemas, function aliases, roles, user-defined aggregate functions, domains, and users (except the current user). If DELETE FILES is specified, the database files will be removed when the last user disconnects from the database. Warning: this command can not be rolled back.

Admin rights are required to execute this command.

Example:

DROP ALL OBJECTS

DROP CONSTANT

```
DROP CONSTANT [IF EXISTS] constantName
```

Drops a constant. This command commits an open transaction.

Example:

DROP CONSTANT ONE

DROP DOMAIN

```
DROP DOMAIN [IF EXISTS] domainName
```

Drops a data type (domain). This command commits an open transaction.

Example:

DROP DOMAIN EMAIL

DROP INDEX

```
DROP INDEX [IF EXISTS] indexName
```

Drops an index. This command commits an open transaction.

Example:

DROP INDEX IF EXISTS IDXNAME

DROP ROLE

```
DROP ROLE [IF EXISTS] roleName
```

Drops a role. This command commits an open transaction.

Example:

DROP ROLE READONLY

DROP SCHEMA

```
DROP SCHEMA [IF EXISTS] schemaName
```

Drops a schema. This command commits an open transaction.

Example:

DROP SCHEMA TEST_SCHEMA

DROP SEQUENCE

```
DROP SEQUENCE [IF EXISTS] sequenceName
```

Drops a sequence. This command commits an open transaction.

Example:

DROP SEQUENCE SEQ_ID

DROP TABLE

```
DROP TABLE [IF EXISTS] tableName [,...]
```

Drops an existing table, or a list of existing tables. This command commits an open transaction.

Example:

DROP TABLE TEST

DROP TRIGGER

```
DROP TRIGGER [IF EXISTS] triggerName
```

Drops an existing trigger. This command commits an open transaction.

Example:

DROP TRIGGER TRIG_INS

DROP USER

```
DROP USER [IF EXISTS] userName
```

Drops a user. The current user cannot be dropped. For compatibility, only unquoted or uppercase user names are allowed.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

DROP USER TOM

DROP VIEW

```
DROP VIEW [IF EXISTS] viewName
```

Drops a view. This command commits an open transaction.

Example:

DROP VIEW TEST_VIEW

TRUNCATE TABLE

```
TRUNCATE TABLE tableName
```

Removes all rows from a table. Other than DELETE FROM without where clause, this command can not be rolled back. This command is faster than DELETE without where clause. Only regular data tables without foreign key constraints can be truncated.

This command commits an open transaction.

Example:

TRUNCATE TABLE TEST

CHECKPOINT

```
CHECKPOINT
```

Flushes the log and data files and switches to a new log file if possible.

Admin rights are required to execute this command.

Example:

CHECKPOINT

CHECKPOINT SYNC

```
CHECKPOINT SYNC
```

Flushes the log, data and index files and forces all system buffers be written to the underlying device.

Admin rights are required to execute this command.

Example:

CHECKPOINT SYNC

COMMIT

```
COMMIT [WORK]
```

Commits a transaction.

Example:

COMMIT

COMMIT TRANSACTION

```
COMMIT TRANSACTION transactionName
```

Sets the resolution of an in-doubt transaction to 'commit'.

Admin rights are required to execute this command. This command is part of the 2-phase-commit protocol.

Example:

COMMIT TRANSACTION XID_TEST

GRANT RIGHT

```
GRANT {SELECT | INSERT | UPDATE | DELETE | ALL} [,...]
ON tableName [,...] TO {PUBLIC | userName | roleName}
```

Grants rights for a table to a user or role.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

GRANT SELECT ON TEST TO READONLY

GRANT ROLE

```
GRANT roleName TO {PUBLIC | userName | roleName}
```

Grants a role to a user or role.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

GRANT READONLY TO PUBLIC

HELP

```
HELP [anything [...]]
```

Displays the help pages of SQL commands or keywords.

Example:

HELP SELECT

PREPARE COMMIT

```
PREPARE COMMIT newTransactionName
```

Prepares committing a transaction. This command is part of the 2-phase-commit protocol.

Example:

PREPARE COMMIT XID_TEST

REVOKE RIGHT

```
REVOKE {SELECT | INSERT | UPDATE | DELETE | ALL} [,...]
ON tableName [,...] FROM {PUBLIC | userName | roleName}
```

Removes rights for a table from a user or role.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

REVOKE SELECT ON TEST FROM READONLY

REVOKE ROLE

```
REVOKE roleName
FROM {PUBLIC | userName | roleName}
```

Removes a role from a user or role.

Admin rights are required to execute this command. This command commits an open transaction.

Example:

REVOKE READONLY FROM TOM

ROLLBACK

```
ROLLBACK [TO SAVEPOINT savepointName]
```

Rolls back a transaction. If a savepoint name is used, the transaction is only rolled back to the specified savepoint.

Example:

ROLLBACK

ROLLBACK TRANSACTION

```
ROLLBACK TRANSACTION transactionName
```

Sets the resolution of an in-doubt transaction to 'rollback'.

Admin rights are required to execute this command. This command is part of the 2-phase-commit protocol.

Example:

ROLLBACK TRANSACTION XID_TEST

SAVEPOINT

```
SAVEPOINT savepointName
```

Create a new savepoint. See also ROLLBACK. Savepoints are only valid until the transaction is committed or rolled back.

Example:

SAVEPOINT HALF_DONE

SET @

```
SET @variableName [=] expression
```

Updates a user-defined variable. This command does not commit a transaction, and rollback does not affect it.

Example:

SET @TOTAL=0

SET ALLOW_LITERALS

```
SET ALLOW_LITERALS {NONE|ALL|NUMBERS}
```

This setting can help solve the SQL injection problem. By default, text and number literals are allowed in SQL statements. However, this enables SQL injection if the application dynamically builds SQL statements. SQL injection is not possible if user data is set using parameters ('?').

NONE means literals of any kind are not allowed, only parameters and constants are allowed. NUMBERS mean only numerical and boolean literals are allowed. ALL means all literals are allowed (default).

See also CREATE CONSTANT.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent. This setting can be appended to the database URL: jdbc:h2:test;ALLOW_LITERALS=NONE

Example:

SET ALLOW_LITERALS NONE

SET AUTOCOMMIT

```
SET AUTOCOMMIT {TRUE | ON | FALSE | OFF}
```

Switches auto commit on or off. This setting can be appended to the database URL: jdbc:h2:test;AUTOCOMMIT=OFF

Example:

SET AUTOCOMMIT OFF

SET CACHE_SIZE

```
SET CACHE_SIZE int
```

Sets the size of the cache in KB (each KB being 1024 bytes). The default value is 16384 (16 MB). The value is rounded to the next higher power of two. Depending on the virtual machine, the actual memory required may be higher.

This setting is persistent and affects all connections as there is only one cache per database. Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent. This setting can be appended to the database URL: jdbc:h2:test;CACHE_SIZE=8192

Example:

SET CACHE_SIZE 8192

SET CLUSTER

```
SET CLUSTER serverListString
```

This command should not be used directly by an application, the statement is executed automatically by the system. The behavior may change in future releases. Sets the cluster server list. An empty string switches off the cluster mode. Switching on the cluster mode requires admin rights, but any user can switch it off (this is automatically done when the client detects the other server is not responding).

Admin rights are required to execute this command. This command commits an open transaction.

Example:

SET CLUSTER ''

SET COLLATION

```
SET [DATABASE] COLLATION
{OFF | collationName
[STRENGTH {PRIMARY | SECONDARY | TERTIARY | IDENTICAL}]}
```

Sets the collation used for comparing strings. This command can only be executed if there are no tables defined. See java.text.Collator for details about STRENGTH.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent.

Example:

SET COLLATION ENGLISH

SET COMPRESS_LOB

```
SET COMPRESS_LOB {NO|LZF|DEFLATE}
```

Sets the compression algorithm for BLOB and CLOB data. Compression is usually slower, but needs less disk space. LZF is faster but uses more space.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent.

Example:

SET COMPRESS_LOB LZF

SET DATABASE_EVENT_LISTENER

```
SET DATABASE_EVENT_LISTENER classNameString
```

Sets the event listener class. An empty string ('') means no listener should be used. This setting is not persistent.

Admin rights are required to execute this command, except if it is set when opening the database (in this case it is reset just after opening the database). This setting can be appended to the database URL: jdbc:h2:test;DATABASE_EVENT_LISTENER='sample.MyListener'

Example:

SET DATABASE_EVENT_LISTENER 'sample.MyListener'

SET DB_CLOSE_DELAY

```
SET DB_CLOSE_DELAY int
```

Sets the delay for closing a database if all connections are closed. The value -1 means the database is never closed until the close delay is set to some other value or SHUTDOWN is called. The value 0 means no delay (default; the database is closed if the last connection to it is closed). Values 1 and larger mean the number of seconds the database is left open after closing the last connection.

If the application exits normally or System.exit is called, the database is closed immediately, even if a delay is set.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent. This setting can be appended to the database URL: jdbc:h2:test;DB_CLOSE_DELAY=-1

Example:

SET DB_CLOSE_DELAY -1

SET DEFAULT_LOCK_TIMEOUT

```
SET DEFAULT LOCK_TIMEOUT int
```

Sets the default lock timeout (in milliseconds) in this database that is used for the new sessions. The default value for this setting is 1000 (one second).

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent.

Example:

SET DEFAULT_LOCK_TIMEOUT 5000

SET DEFAULT_TABLE_TYPE

```
SET DEFAULT_TABLE_TYPE {MEMORY | CACHED}
```

Sets the default table storage type that is used when creating new tables. Memory tables are kept fully in the main memory (including indexes), however changes to the data are stored in the log file. The size of memory tables is limited by the memory. The default is CACHED.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent.

Example:

SET DEFAULT_TABLE_TYPE MEMORY

SET EXCLUSIVE

```
SET EXCLUSIVE {TRUE | FALSE}
```

Switched the database to exclusive mode and back. In exclusive mode, new connections are rejected, and operations by other connections are paused until the exclusive mode is disabled. Only the connection that set the exclusive mode can disable it. When the connection is closed, it is automatically disabled.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent.

Example:

SET EXCLUSIVE TRUE

SET IGNORECASE

```
SET IGNORECASE {TRUE|FALSE}
```

If IGNORECASE is enabled, text columns in newly created tables will be case-insensitive. Already existing tables are not affected. The effect of case-insensitive columns is similar to using a collation with strength PRIMARY. Case-insensitive columns are compared faster than when using a collation.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent.

Example:

SET IGNORECASE TRUE

SET LOCK_MODE

SET LOCK_MODE int

Sets the lock mode. The values 0, 1, 2, and 3 are supported. The default is 3 (READ_COMMITTED).

The value 0 means no locking (should only be used for testing; also known as READ_UNCOMMITTED). Please note that using SET LOCK_MODE 0 while at the same time using multiple connections may result in inconsistent transactions.

The value 1 means table level locking (also known as SERIALIZABLE).

The value 2 means table level locking with garbage collection (if the application does not close all connections).

The value 3 means table level locking, but read locks are released immediately (default; also known as READ_COMMITTED).

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent. This setting can be appended to the database URL: jdbc:h2:test;LOCK_MODE=3

Example:

SET LOCK_MODE 1

SET LOCK_TIMEOUT

SET LOCK_TIMEOUT int

Sets the lock timeout (in milliseconds) for the current session. The default value for this setting is 1000 (one second).

This command does not commit a transaction, and rollback does not affect it. This setting can be appended to the database URL: jdbc:h2:test;LOCK_TIMEOUT=10000

Example:

SET LOCK_TIMEOUT 1000

SET LOG

SET LOG int

Enabled or disables writing to the transaction log file. The values 0, 1, and 2 are supported.

0 means logging is disabled (faster). 1 means logging of the data is enabled, but logging of the index changes is disabled (default). 2 means logging of both data and index changes are enabled.

Transaction logging can be disabled to improve the performance when durability is not important, for example while running tests or when loading the database. Warning: it may not be possible to recover the database if logging is disabled and the application terminates abnormally. If logging of index changes is enabled, opening a database that was crashed becomes faster because the indexes don't need to be rebuilt.

Admin rights are required to execute this command. This command commits an open transaction. This setting can be appended to the database URL: jdbc:h2:test;LOG=2

Example:

SET LOG 0

SET MAX_LENGTH_INPLACE_LOB

```
SET MAX_LENGTH_INPLACE_LOB int
```

Sets the maximum size of an in-place LOB object. LOB objects larger that this size are stored in a separate file, otherwise stored directly in the database (in-place). The default max size is 1024.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent.

Example:

SET MAX_LENGTH_INPLACE_LOB 128

SET MAX_LOG_SIZE

```
SET MAX_LOG_SIZE int
```

Sets the maximum file size of a log file, in megabytes. If the file exceeds the limit, a new file is created. Old files (that are not used for recovery) are deleted automatically, but multiple log files may exist for some time. The default max size is 32 MB.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent.

Example:

SET MAX_LOG_SIZE 2

SET MAX_MEMORY_ROWS

```
SET MAX_MEMORY_ROWS int
```

The maximum number of rows in a result set that are kept in-memory. If more rows are read, then the rows are buffered to disk. The default value is 10000.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent.

Example:

SET MAX_MEMORY_ROWS 1000

SET MAX_MEMORY_UNDO

```
SET MAX_MEMORY_UNDO int
```

The maximum number of undo records per a session that are kept in-memory. If a transaction is larger, the records are buffered to disk. The default value is 50000. Changes to tables without a primary key can not be buffered to disk. This setting is not supported when using multi-version concurrency.

Admin rights are required to execute this command. This command commits an open transaction. This setting is persistent.

Example:

SET MAX_MEMORY_UNDO 1000

SET MAX_OPERATION_MEMORY

```
SET MAX_OPERATION_MEMORY int
```

Sets the maximum memory used for large operations (delete and insert), in bytes. Operations that use more memory are buffered to disk, slowing down the operation. The default max size is 100000. 0 means no limit.

This setting is not persistent. Admin rights are required to execute this command. This setting can be appended to the database URL: jdbc:h2:test;MAX_OPERATION_MEMORY=10000

Example:

SET MAX_OPERATION_MEMORY 0

SET MODE

```
SET MODE {REGULAR | DB2 | DERBY | HSQLDB |
MSSQLSERVER | MYSQL | ORACLE | POSTGRESQL}
```

Changes to another database compatibility mode. For details, see Compatibility Modes in the feature section.

This setting is not persistent. Admin rights are required to execute this command. This command commits an open transaction. This setting can be appended to the database URL: jdbc:h2:test;MODE=MYSQL

Example:

SET MODE HSQLDB

SET MULTI_THREADED

```
SET MULTI_THREADED {0|1}
```

Enabled (1) or disabled (0) multi-threading inside the database engine. By default, this setting is disabled. Currently, enabling this is experimental only.

This is a global setting, which means it is not possible to open multiple databases with different modes at the same time in the same virtual machine. This setting is not persistent, however the value is kept until the virtual machine exits or it is changed.

Admin rights are required to execute this command. This command commits an open transaction. This setting can be appended to the database URL: jdbc:h2:test;MULTI_THREADED=1

Example:

SET MULTI_THREADED 1

SET OPTIMIZE_REUSE_RESULTS

```
SET OPTIMIZE_REUSE_RESULTS {0|1}
```

Enabled (1) or disabled (0) the result reuse optimization. If enabled, subqueries and views used as subqueries are only re-run if the data in one of the tables was changed. This option is enabled by default.

Admin rights are required to execute this command. This command commits an open transaction. This setting can be appended to the database URL: jdbc:h2:test;OPTIMIZE_REUSE_RESULTS=0

Example:

SET OPTIMIZE_REUSE_RESULTS 0

SET PASSWORD

```
SET PASSWORD string
```

Changes the password of the current user. The password must be in single quotes. It is case sensitive and can contain spaces.

This command commits an open transaction.

Example:

SET PASSWORD 'abcstzri!.5'

SET QUERY_TIMEOUT

```
SET QUERY_TIMEOUT int
```

Set the query timeout of the current session to the given value. The timeout is in milliseconds. All kinds of statements will throw an exception if they take longer than the given value. The default timeout is 0, meaning no timeout.

This command does not commit a transaction, and rollback does not affect it.

Example:

SET QUERY_TIMEOUT 10000

SET REFERENTIAL_INTEGRITY

```
SET REFERENTIAL_INTEGRITY [TRUE|FALSE]
```

Disabled or enables referential integrity checking for the whole database. Enabling it does not check existing data. Use ALTER TABLE SET to disable it only for one table.

This setting is not persistent. This command commits an open transaction. Admin rights are required to execute this command.

Example:

SET REFERENTIAL_INTEGRITY FALSE

SET SALT HASH

```
SET SALT bytes HASH bytes
```

Sets the password salt and hash for the current user. The password must be in single quotes. It is case sensitive and can contain spaces.

This command commits an open transaction.

Example:

SET SALT '00' HASH '1122'

SET SCHEMA

```
SET SCHEMA schemaName
```

Changes the default schema of the current connection. The default schema is used in statements where no schema is set explicitly. The default schema for new connections is PUBLIC.

This command does not commit a transaction, and rollback does not affect it. This setting can be appended to the database URL: jdbc:h2:test;SCHEMA=ABC

Example:

SET SCHEMA INFORMATION_SCHEMA

SET SCHEMA_SEARCH_PATH

```
SET SCHEMA_SEARCH_PATH schemaName [,...]
```

Changes the schema search path of the current connection. The default schema is used in statements where no schema is set explicitly. The default schema for new connections is PUBLIC.

This command does not commit a transaction, and rollback does not affect it. This setting can be appended to the database URL: jdbc:h2:test;SCHEMA_SEARCH_PATH=ABC,DEF

Example:

SET SCHEMA_SEARCH_PATH INFORMATION_SCHEMA, PUBLIC

SET THROTTLE

```
SET THROTTLE int
```

Sets the throttle for the current connection. The value is the number of milliseconds delay after each 50 ms. The default value is 0 (throttling disabled).

This command does not commit a transaction, and rollback does not affect it. This setting can be appended to the database URL: jdbc:h2:test;THROTTLE=50

Example:

SET THROTTLE 200

SET TRACE_LEVEL

```
SET {TRACE_LEVEL_FILE | TRACE_LEVEL_SYSTEM_OUT} int
```

Sets the trace level for file the file or system out stream. Levels are: 0=off, 1=error, 2=info, 3=debug. The default level is 1 for file and 0 for system out. To use SLF4J, append ;TRACE_LEVEL_FILE=4 to the database URL when opening the database.

This setting is not persistent. Admin rights are required to execute this command. This command does not commit a transaction, and rollback does not affect it. This setting can be appended to the database URL: jdbc:h2:test;TRACE_LEVEL_SYSTEM_OUT=3

Example:

SET TRACE_LEVEL_SYSTEM_OUT 3

SET TRACE_MAX_FILE_SIZE

```
SET TRACE_MAX_FILE_SIZE int
```

Sets the maximum trace file size. If the file exceeds the limit, the file is renamed to .old and a new file is created. If another .old file exists, it is deleted. The default max size is 16 MB.

This setting is persistent. Admin rights are required to execute this command. This command commits an open transaction. This setting can be appended to the database URL: jdbc:h2:test;TRACE_MAX_FILE_SIZE=3

Example:

SET TRACE_MAX_FILE_SIZE 10

SET UNDO_LOG

```
SET UNDO_LOG int
```

Enables (1) or disables (0) the per session undo log. The undo log is enabled by default. When disabled, transactions can not be rolled back. This setting should only be used for bulk operations that don't need to be atomic.

This command commits an open transaction.

Example:

SET UNDO_LOG 0

SET WRITE_DELAY

```
SET WRITE_DELAY int
```

Set the maximum delay between a commit and flushing the log, in milliseconds. This setting is persistent. The default is 500 ms.

Admin rights are required to execute this command. This command commits an open transaction. This setting can be appended to the database URL: jdbc:h2:test;WRITE_DELAY=0

Example:

SET WRITE_DELAY 2000

SHUTDOWN

```
SHUTDOWN [IMMEDIATELY|COMPACT|SCRIPT]
```

This statement is closes all open connections to the database and closes the database. This command usually does not need to be used, as the database is closed automatically when the last connection to it is closed.

If no option is used, then all connections are closed. If the IMMEDIATELY option is used, the database files are closed as if the hard drive stops working, without rollback of the open transactions; otherwise any open transaction are rolled back before closing the connection. COMPACT and SCRIPT are only supported for compatibility and have no effect.

Admin rights are required to execute this command.

Example:

SHUTDOWN

Comments

```
-- anythingUntilEndOfLine
| // anythingUntilEndOfLine
| /* anythingUntilEndComment */
```

Comments can be used anywhere in a command and are ignored by the database. Line comments end with a newline. Block comments cannot be nested, but can be multiple lines long.

Example:

// This is a comment

Select Part

```
[TOP term] [DISTINCT | ALL] selectExpression [,...]
```

The SELECT part of a query.

Example:

DISTINCT *

From Part

```
tableExpression [,...]
```

The FROM part of a query.

Example:

FROM TEST

Constraint

```
PRIMARY KEY [HASH] (columnName [,...])
| [CONSTRAINT [IF NOT EXISTS] newConstraintName] {
CHECK expression
| UNIQUE (columnName [,...])
| referentialConstraint}
```

Defines a constraint. The check condition must evaluate to true or to NULL (to prevent NULL, use NOT NULL).

Example:

PRIMARY KEY(ID, NAME)

Referential Constraint

```
FOREIGN KEY (columnName [,...])
REFERENCES [refTableName] [(refColumnName[,...])]
[ON DELETE {CASCADE | RESTRICT | NO ACTION | SET {DEFAULT|NULL}}]
[ON UPDATE {CASCADE | SET {DEFAULT|NULL}}]
```

Defines a referential constraint. If the table name is not specified, then the same table is referenced. As this database does not support deferred checking, RESTRICT and NO ACTION will both throw an exception if the constraint is violated. If the referenced columns are not specified, then the primary key columns are used. The required indexes are automatically created if required.

Example:

FOREIGN KEY(ID) REFERENCES TEST(ID)

Table Expression

```
{[schemaName.] tableName | (select)} [[AS] newTableAlias]
```

```
[{{LEFT | RIGHT} [OUTER] | [INNER] | CROSS | NATURAL}
JOIN tableExpression [[AS] newTableAlias] [ON expression] ]
```

Joins a table. The join expression is not supported for cross and natural joins. A natural join is an inner join, where the condition is automatically on the columns with the same name.

Example:

TEST AS T LEFT JOIN TEST AS T1 ON T.ID = T1.ID

Order

```
{int | expression} [ASC | DESC] [NULLS {FIRST | LAST}]
```

Sorts the result by the given column number, or by an expression. If the expression is a single parameter, then the value is interpreted as a column number. Negative column numbers reverse the sort order.

Example:

NAME DESC NULLS LAST

Index Column

```
columnName [ASC | DESC] [NULLS {FIRST | LAST}]
```

Indexes this column in ascending or descending order. Usually it is not required to specify the order; however doing so will speed up large queries that order the column in the same way.

Example:

NAME

Expression

```
andCondition [OR andCondition]
```

Value or condition.

Example:

ID=1 OR NAME='Hi'

And Condition

```
condition [AND condition]
```

Value or condition.

Example:

ID=1 AND NAME='Hi'

Condition

```
operand [conditionRightHandSide] | NOT condition | EXISTS (select)
```

Boolean value or condition.

Example:

ID<>2

Condition Right Hand Side

```
compare { {{ALL|ANY|SOME}(select)} | operand }
| IS [NOT] NULL
| BETWEEN operand AND operand
| IN ({select | expression[,...]})
| [NOT] LIKE operand [ESCAPE string]
| [NOT] REGEXP operand
```

The right hand side of a condition.

When comparing with LIKE, the wildcards characters are _ (any one character) and % (any characters). The database uses an index when comparing with LIKE except if the operand starts with a wildcard. To search for the characters % and _, the characters need to be escaped. The default escape character is \ (backslash). To select no escape character, use ESCAPE '' (empty string).

When comparing with REGEXP, regular expression matching is used. See Java Matcher.find for details.

Example:

LIKE 'Jo%'

Compare

```
<> | <= | >= | = | < | > | !=
```

Comparison operator. The operator != is the same as <>.

Example:

<>

Operand

```
summand [ || summand]
```

A value or a concatenation of values.

Example:

'Hi' || ' Eva'

Summand

```
factor [{+ | -} factor]
```

A value or a numeric sum.

Example:

ID + 20

Factor

```
term [{* | /} term]
```

A value or a numeric factor.

Example:

ID * 10

Term

```
value
| columnName
| ?[int]
| NEXT VALUE FOR sequenceName
| function
| {- | +} term
| (expression)
| select
| case
| caseWhen
| tableAlias.columnName
```

A value. Parameters can be indexed, for example ?1 meaning the first parameter.

Example:

'Hello'

Value

```
string | dollarQuotedString | hexNumber | int | long | decimal | double |
date | time | timestamp | boolean | bytes | array | null
```

A value of any data type, or null.

Example:

10

Case

```
CASE expression {WHEN expression THEN expression}
[...] [ELSE expression] END
```

Returns the first expression where the value is equal to the test expression. If no else part is specified, return NULL.

Example:

CASE CNT WHEN 0 THEN 'No' WHEN 1 THEN 'One' ELSE 'Some' END

Case When

```
CASE {WHEN expression THEN expression}
[...] [ELSE expression] END
```

Returns the first expression where the condition is true. If no else part is specified, return NULL.

Example:

CASE WHEN CNT<10 THEN 'Low' ELSE 'High' END

Cipher

```
[AES | XTEA]
```

Two algorithms are supported, AES (AES-256) and XTEA (using 32 rounds). The AES algorithm is about half as fast as XTEA.

Example:

AES

Select Expression

```
* | expression [[AS] columnAlias] | tableAlias.*
```

An expression in a SELECT statement.

Example:

ID AS VALUE

Data Type

```
intType | booleanType | tinyintType | smallintType | bigintType | identityType |
decimalType | doubleType | realType | dateType | timeType | timestampType |
binaryType | otherType | varcharType | varcharIgnorecaseType | charType
blobType | clobType | uuidType | arrayType
```

A data type definition.

Example:

INT

Name

```
{ { A-Z|_ } [ { A-Z|_|0-9} [...] ] } | quotedName
```

Names are not case sensitive. There is no maximum name length.

Example:

TEST

Alias

```
name
```

An alias is a name that is only valid in the context of the statement.

Example:

A

Quoted Name

```
"anythingExceptDoubleQuote"
```

Quoted names are case sensitive, and can contain spaces. There is no maximum name length. Two double quotes can be used to create a single double quote inside an identifier.

Example:

"FirstName"

String

```
'anythingExceptSingleQuote'
```

A string starts and ends with a single quote. Two single quotes can be used to create a single quote inside a string.

Example:

'John''s car'

Dollar Quoted String

```
$$anythingExceptTwoDollarSigns$$
```

A string starts and ends with two dollar signs. Two dollar signs are not allowed within the text. A whitespace is required before the first set of dollar signs. No escaping is required within the text.

Example:

$$John's car$$

Int

```
[- | +] digit [...]
```

The maximum integer number is 2147483647, the minimum is -2147483648.

Example:

10

Long

```
[- | +] digit [...]
```

Long numbers are between -9223372036854775808 and 9223372036854775807.

Example:

100000

Hex Number

```
[+ | -] 0x hex
```

A number written in hexadecimal notation.

Example:

0xff

Decimal

```
[- | +] digit [...] [. digit [...] ]
```

Number with fixed precision and scale.

Example:

-1600.05

Double

```
[- | +] digit [...]
[. digit [...] [E [- | +] exponentDigit [...] ]]
```

The limitations are the same as for the Java data type Double.

Example:

-1.4e-10

Date

```
DATE 'yyyy-MM-dd'
```

A date literal. The limitations are the same as for the Java data type java.sql.Date, but for compatibility with other databases the suggested minimum and maximum years are 0001 and 9999.

Example:

DATE '2004-12-31'

Time

```
TIME 'hh:mm:ss'
```

A time literal.

Example:

TIME '23:59:59'

Timestamp

```
TIMESTAMP 'yyyy-MM-dd hh:mm:ss[.nnnnnnnnn]'
```

A timestamp literal. The limitations are the same as for the Java data type java.sql.Timestamp, but for compatibility with other databases the suggested minimum and maximum years are 0001 and 9999.

Example:

TIMESTAMP '2005-12-31 23:59:59'

Boolean

```
TRUE | FALSE
```

A boolean value.

Example:

TRUE

Bytes

```
X'hex'
```

A binary value. The hex value is not case sensitive.

Example:

X'01FF'

Array

```
( expression [,..] )
```

An array of values.

Example:

(1, 2)

Null

```
NULL
```

NULL is a value without data type and means 'unknown value'.

Example:

NULL

Hex

```
{{ digit | a-f | A-F } {digit | a-f | A-F }} [...]
```

The hexadecimal representation of a number or of bytes. Two characters are one byte.

Example:

cafe

Digit

0-9

A digit.

Example:

0

Information Schema

The system tables in the schema 'INFORMATION_SCHEMA' contain the meta data of all tables in the database as well as the current settings.

Table	Columns
CATALOGS	CATALOG_NAME
COLLATIONS	NAME, KEY
COLUMNS	TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, ORDINAL_POSITION, COLUMN_DEFAULT, IS_NULLABLE, DATA_TYPE, CHARACTER_MAXIMUM_LENGTH, CHARACTER_OCTET_LENGTH, NUMERIC_PRECISION, NUMERIC_PRECISION_RADIX, NUMERIC_SCALE, CHARACTER_SET_NAME, COLLATION_NAME, TYPE_NAME, NULLABLE, IS_COMPUTED, SELECTIVITY, CHECK_CONSTRAINT, SEQUENCE_NAME, REMARKS, SOURCE_DATA_TYPE
COLUMN_PRIVILEGES	GRANTOR, GRANTEE, TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, PRIVILEGE_TYPE, IS_GRANTABLE
CONSTANTS	CONSTANT_CATALOG, CONSTANT_SCHEMA, CONSTANT_NAME, DATA_TYPE, REMARKS, SQL, ID
CONSTRAINTS	CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, UNIQUE_INDEX_NAME, CHECK_EXPRESSION, COLUMN_LIST, REMARKS, SQL, ID
CROSS_REFERENCES	PKTABLE_CATALOG, PKTABLE_SCHEMA, PKTABLE_NAME, PKCOLUMN_NAME, FKTABLE_CATALOG, FKTABLE_SCHEMA, FKTABLE_NAME, FKCOLUMN_NAME, ORDINAL_POSITION, UPDATE_RULE, DELETE_RULE, FK_NAME, PK_NAME, DEFERRABILITY
DOMAINS	DOMAIN_CATALOG, DOMAIN_SCHEMA, DOMAIN_NAME, COLUMN_DEFAULT, IS_NULLABLE, DATA_TYPE, PRECISION, SCALE, TYPE_NAME, SELECTIVITY, CHECK_CONSTRAINT, REMARKS, SQL, ID
FUNCTION_ALIASES	ALIAS_CATALOG, ALIAS_SCHEMA, ALIAS_NAME, JAVA_CLASS, JAVA_METHOD, DATA_TYPE, COLUMN_COUNT, RETURNS_RESULT, REMARKS, ID
FUNCTION_COLUMNS	ALIAS_CATALOG, ALIAS_SCHEMA, ALIAS_NAME, JAVA_CLASS, JAVA_METHOD, COLUMN_COUNT, POS, COLUMN_NAME, DATA_TYPE, TYPE_NAME, PRECISION, SCALE, RADIX, NULLABLE, COLUMN_TYPE, REMARKS, COLUMN_DEFAULT
HELP	ID, SECTION, TOPIC, SYNTAX, TEXT
INDEXES	TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, NON_UNIQUE, INDEX_NAME, ORDINAL_POSITION, COLUMN_NAME, CARDINALITY, PRIMARY_KEY, INDEX_TYPE_NAME, IS_GENERATED, INDEX_TYPE, ASC_OR_DESC, PAGES, FILTER_CONDITION, REMARKS, SQL, ID, SORT_TYPE, CONSTRAINT_NAME
IN_DOUBT	TRANSACTION, STATE
LOCKS	TABLE_SCHEMA, TABLE_NAME, SESSION_ID, LOCK_TYPE
RIGHTS	GRANTEE, GRANTEETYPE, GRANTEDROLE, RIGHTS, TABLE_SCHEMA, TABLE_NAME, ID
ROLES	NAME, REMARKS, ID
SCHEMATA	CATALOG_NAME, SCHEMA_NAME, SCHEMA_OWNER, DEFAULT_CHARACTER_SET_NAME, DEFAULT_COLLATION_NAME, IS_DEFAULT, REMARKS, ID

SEQUENCES	SEQUENCE_CATALOG, SEQUENCE_SCHEMA, SEQUENCE_NAME, CURRENT_VALUE, INCREMENT, IS_GENERATED, REMARKS, CACHE, ID
SESSIONS	ID, USER_NAME, SESSION_START, STATEMENT, STATEMENT_START
SESSION_STATE	KEY, SQL
SETTINGS	NAME, VALUE
TABLES	TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE, STORAGE_TYPE, SQL, REMARKS, LAST_MODIFICATION, ID, TYPE_NAME
TABLE_PRIVILEGES	GRANTOR, GRANTEE, TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, PRIVILEGE_TYPE, IS_GRANTABLE
TABLE_TYPES	TYPE
TRIGGERS	TRIGGER_CATALOG, TRIGGER_SCHEMA, TRIGGER_NAME, TRIGGER_TYPE, TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, BEFORE, JAVA_CLASS, QUEUE_SIZE, NO_WAIT, REMARKS, SQL, ID
TYPE_INFO	TYPE_NAME, DATA_TYPE, PRECISION, PREFIX, SUFFIX, PARAMS, AUTO_INCREMENT, MINIMUM_SCALE, MAXIMUM_SCALE, RADIX, POS, CASE_SENSITIVE, NULLABLE, SEARCHABLE
USERS	NAME, ADMIN, REMARKS, ID
VIEWS	TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, VIEW_DEFINITION, CHECK_OPTION, IS_UPDATABLE, STATUS, REMARKS, ID

Range Table

The range table is a dynamic system table that contains all values from a start to an end value. The table contains one column called X. Both the start and end values are included in the result. The table is used as follows:

```
SELECT X FROM SYSTEM_RANGE(1, 10);
```

Functions

Aggregate Functions

AVG
BOOL_AND
BOOL_OR
COUNT
GROUP_CONCAT
MAX
MIN
SUM
SELECTIVITY
STDDEV_POP
STDDEV_SAMP
VAR_POP
VAR_SAMP

Numeric Functions

ABS
ACOS
ASIN
ATAN
COS
COT
SIN
TAN
ATAN2
BITAND
BITOR
BITXOR
MOD
CEILING
DEGREES
EXP
FLOOR
LOG
LOG10
RADIANS
SQRT
PI
POWER
RAND
RANDOM_UUID
ROUND
ROUNDMAGIC
SECURE_RAND
SIGN
ENCRYPT
DECRYPT
HASH
TRUNCATE
COMPRESS
EXPAND
ZERO

String Functions

ASCII
BIT_LENGTH
LENGTH
OCTET_LENGTH

CHAR
CONCAT
DIFFERENCE
HEXTORAW
RAWTOHEX
INSTR
INSERT Function
LOWER
UPPER
LEFT
RIGHT
LOCATE
POSITION
LPAD
RPAD
LTRIM
RTRIM
TRIM
REGEXP_REPLACE
REPEAT
REPLACE
SOUNDEX
SPACE
STRINGDECODE
STRINGENCODE
STRINGTOUTF8
SUBSTRING
UTF8TOSTRING
XMLATTR
XMLNODE
XMLCOMMENT
XMLCDATA
XMLSTARTDOC
XMLTEXT

Time and Date Functions

CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
DATEADD
DATEDIFF
DAYNAME
DAY_OF_MONTH
DAY_OF_WEEK
DAY_OF_YEAR
EXTRACT
FORMATDATETIME
HOUR
MINUTE
MONTH
MONTHNAME
PARSEDATETIME
QUARTER
SECOND
WEEK
YEAR

System Functions

ARRAY_GET
ARRAY_LENGTH
AUTOCOMMIT
CANCEL_SESSION
CASEWHEN Function
CAST

COALESCE
CONVERT
CURRVAL
CSVREAD
CSVWRITE
DATABASE
DATABASE_PATH
FILE_READ
GREATEST
IDENTITY
IFNULL
LEAST
LOCK_MODE
LOCK_TIMEOUT
LINK_SCHEMA
MEMORY_FREE
MEMORY_USED
NEXTVAL
NULLIF
READONLY
ROWNUM
SCHEMA
SESSION_ID
SET
TABLE
TRANSACTION_ID
USER

AVG

```
AVG([DISTINCT] {int | long | decimal | double}): value
```

The average (mean) value. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

AVG(X)

BOOL_AND

```
BOOL_AND(boolean): boolean
```

Returns true if all expressions are true. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

BOOL_AND(ID>10)

BOOL_OR

```
BOOL_OR(boolean): boolean
```

Returns true if any expression is true. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

BOOL_OR(NAME LIKE 'W%')

COUNT

```
COUNT(*) | COUNT([DISTINCT] expression): int
```

The count of all row, or of the non-null values. If no rows are selected, the result is 0. Aggregates are only allowed in select statements.

Example:

COUNT(*)

GROUP_CONCAT

```
GROUP_CONCAT([DISTINCT] string [ORDER BY {expression [ASC|DESC]}[,...]]
[SEPARATOR expression]): string
```

Concatenates strings with a separator. The default separator is a ',' (without space). If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

GROUP_CONCAT(NAME ORDER BY ID SEPARATOR ', ')

MAX

```
MAX(value): value
```

The highest value. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

MAX(NAME)

MIN

```
MIN(value): value
```

The lowest value. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

MIN(NAME)

SUM

```
SUM([DISTINCT] {int | long | decimal | double}): value
```

The sum of all values. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

SUM(X)

SELECTIVITY

```
SELECTIVITY(value): int
```

Estimates the selectivity (0-100) of a value. The value is defined as (100 * distinctCount / rowCount). The selectivity of 0 rows is 0 (unknown). Up to 10000 values are kept in memory. Aggregates are only allowed in select statements.

Example:

SELECT SELECTIVITY(FIRSTNAME), SELECTIVITY(NAME) FROM TEST WHERE ROWNUM()<20000

STDDEV_POP

```
STDDEV_POP([DISTINCT] double): double
```

The population standard deviation. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

STDDEV_POP(X)

STDDEV_SAMP

```
STDDEV_SAMP([DISTINCT] double): double
```

The sample standard deviation. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

STDDEV(X)

VAR_POP

```
VAR_POP([DISTINCT] double): double
```

The population variance (square of the population standard deviation). If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

VAR_POP(X)

VAR_SAMP

```
VAR_SAMP([DISTINCT] double): double
```

The sample variance (square of the sample standard deviation). If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

VAR_SAMP(X)

ABS

```
ABS({int | long | decimal | double}): value
```

See also Java Math.abs. Please note that Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE and Math.abs(Long.MIN_VALUE) == Long.MIN_VALUE.

Example:

ABS(ID)

ACOS

```
ACOS(double): double
```

See also Java Math.* functions.

Example:

ACOS(D)

ASIN

```
ASIN(double): double
```

See also Java Math.* functions.

Example:

ASIN(D)

ATAN

```
ATAN(double): double
```

See also Java Math.* functions.

Example:

ATAN(D)

COS

```
COS(double): double
```

See also Java Math.* functions.

Example:

COS(ANGLE)

COT

```
COT(double): double
```

See also Java Math.* functions.

Example:

COT(ANGLE)

SIN

```
SIN(double): double
```

See also Java Math.* functions.

Example:

SIN(ANGLE)

TAN

```
TAN(double): double
```

See also Java Math.* functions.

Example:

TAN(ANGLE)

ATAN2

```
ATAN2(double, double): double
```

See also Java Math.atan2.

Example:

ATAN2(X, Y)

BITAND

```
BITAND(long, long): long
```

See also Java operator &.

Example:

BITAND(A, B)

BITOR

```
BITOR(long, long): long
```

See also Java operator |.

Example:

BITOR(A, B)

BITXOR

```
BITXOR(long, long): long
```

See also Java operator ^.

Example:

BITXOR(A, B)

MOD

```
MOD(long, long): long
```

See also Java operator %.

Example:

MOD(A, B)

CEILING

```
CEILING(double): double
```

See also Java Math.ceil.

Example:

LOG(A)

DEGREES

```
DEGREES(double): double
```

See also Java Math.toDegrees.

Example:

DEGREES(A)

EXP

```
EXP(double): double
```

See also Java Math.exp.

Example:

EXP(A)

FLOOR

```
FLOOR(double): double
```

See also Java Math.floor.

Example:

FLOOR(A)

LOG

```
LOG(double): double
```

See also Java Math.log.

Example:

LOG(A)

LOG10

```
LOG10(double): double
```

See also Java Math.log10 (in Java 5).

Example:

LOG10(A)

RADIANS

```
RADIANS(double): double
```

See also Java Math.toRadians.

Example:

RADIANS(A)

SQRT

```
SQRT(double): double
```

See also Java Math.sqrt.

Example:

SQRT(A)

PI

```
PI(): double
```

See also Java Math.PI.

Example:

PI()

POWER

```
POWER(double, double): double
```

See also Java Math.pow.

Example:

POWER(A, B)

RAND

```
RAND([int]): double
```

Calling the function without parameter returns the next a pseudo random number. Calling it with an parameter seeds the session's random number generator.

Example:

RAND()

RANDOM_UUID

```
RANDOM_UUID(): UUID
```

Returns a new UUID with 122 pseudo random bits.

Example:

RANDOM_UUID()

ROUND

```
ROUND(double, digitsInt): double
```

Rounds to a number of digits.

Example:

ROUND(VALUE, 2)

ROUNDMAGIC

```
ROUNDMAGIC(double): double
```

This function rounds numbers in a good way, but it is slow. It has a special handling for numbers around 0. Only numbers smaller or equal +/-1000000000000 are supported. The value is converted to a String internally, and then the last last 4 characters are checked. '000x' becomes '0000' and '999x' becomes '999999', which is rounded automatically.

Example:

ROUNDMAGIC(VALUE/3*3)

SECURE_RAND

SECURE_RAND(int): bytes

Generates a number of cryptographically secure random numbers.

Example:

CALL SECURE_RAND(16)

SIGN

SIGN({int | long | decimal | double}): int

Returns -1 if the value is smaller 0, 0 if zero, and otherwise 1.

Example:

SIGN(VALUE)

ENCRYPT

ENCRYPT(algorithmString, keyBytes, dataBytes): bytes

Encrypts data using a key. Supported algorithms are XTEA and AES. The block size is 16 bytes.

Example:

CALL ENCRYPT('AES', '00', STRINGTOUTF8('Test'))

DECRYPT

DECRYPT(algorithmString, keyBytes, dataBytes): bytes

Decrypts data using a key. Supported algorithms are XTEA and AES. The block size is 16 bytes.

Example:

```
CALL TRIM(CHAR(0) FROM UTF8TOSTRING(
    DECRYPT('AES', '00', '3fabb4de8f1ee2e97d7793bab2db1116')))
```

HASH

```
HASH(algorithmString, dataBytes, iterationInt): bytes
```

Calculate the hash value using an algorithm, and repeat this process for a number of iterations. Currently, the only algorithm supported is SHA256.

Example:

CALL HASH('SHA256', STRINGTOUTF8('Password'), 1000)

TRUNCATE

```
TRUNCATE(double, digitsInt): double
```

Truncates to a number of digits (to the next value closer to 0).

Example:

TRUNCATE(VALUE, 2)

COMPRESS

```
COMPRESS(dataBytes [, algorithmString]): bytes
```

Compresses the data using the specified compression algorithm. Supported algorithms are: LZF (faster but lower compression; default), and DEFLATE (higher compression). Compression does not always reduce size. Very small objects and objects with little redundancy may get larger.

Example:

COMPRESS(STRINGTOUTF8('Test'))

EXPAND

```
EXPAND(bytes): bytes
```

Expands data that was compressed using the COMPRESS function.

Example:

UTF8TOSTRING(EXPAND(COMPRESS(STRINGTOUTF8('Test'))))

ZERO

```
ZERO(): int
```

Returns the value 0. This function can be used even if numeric literals are disabled.

Example:

ZERO()

ASCII

```
ASCII(string): int
```

Returns the ASCII value of the first character in the string.

Example:

ASCII('Hi')

BIT_LENGTH

```
BIT_LENGTH(string): long
```

Returns the number of bits in a string. For BLOB, CLOB, BYTES and JAVA_OBJECT, the precision is used. Each character needs 16 bits.

Example:

BIT_LENGTH(NAME)

LENGTH

```
{LENGTH | CHAR_LENGTH | CHARACTER_LENGTH}(string): long
```

Returns the number of characters in a string. For BLOB, CLOB, BYTES and JAVA_OBJECT, the precision is used.

Example:

LENGTH(NAME)

OCTET_LENGTH

```
OCTET_LENGTH(string): long
```

Returns the number of bytes in a string. For BLOB, CLOB, BYTES and JAVA_OBJECT, the precision is used. Each character needs 2 bytes.

Example:

OCTET_LENGTH(NAME)

CHAR

```
{CHAR | CHR}(int): string
```

Returns the character that represents the ASCII value.

Example:

CHAR(65)

CONCAT

```
CONCAT(string, string [,...]): string
```

Combines strings.

Example:

CONCAT(NAME, '!')

DIFFERENCE

```
DIFFERENCE(string, string): int
```

Returns the difference between the sounds of two strings.

Example:

DIFFERENCE(T1.NAME, T2.NAME)

HEXTORAW

```
HEXTORAW(string): string
```

Converts a hex representation of a string to a string. 4 hex characters per string character are used.

Example:

HEXTORAW(DATA)

RAWTOHEX

```
RAWTOHEX(string): string
```

Converts a string to the hex representation. 4 hex characters per string character are used.

Example:

RAWTOHEX(DATA)

INSTR

```
INSTR(string, searchString, [, startInt]): int
```

Returns the location of a search string in a string (s). If a start position is used, the characters before it are ignored. If position is negative, the rightmost location is returned. 0 is returned if the search string is not found.

Example:

INSTR(EMAIL,'@')

INSERT Function

```
INSERT(originalString, startInt, lengthInt, addString): string
```

Inserts a additional string into the original string at a specified start position. The length specifies the number of characters that are removed at the start position in the original string.

Example:

INSERT(NAME, 1, 1, ' ')

LOWER

```
{LOWER | LCASE}(string): string
```

Converts a string to lowercase.

Example:

LOWER(NAME)

UPPER

```
{UPPER | UCASE}(string): string
```

Converts a string to uppercase.

Example:

UPPER(NAME)

LEFT

```
LEFT(string, int): string
```

Returns the leftmost number of characters.

Example:

LEFT(NAME, 3)

RIGHT

RIGHT(string, int): string

Returns the rightmost number of characters.

Example:

RIGHT(NAME, 3)

LOCATE

LOCATE(searchString, string [, startInt]): int

Returns the location of a search string in a string. If a start position is used, the characters before it are ignored. If position is negative, the rightmost location is returned. 0 is returned if the search string is not found.

Example:

LOCATE('.', NAME)

POSITION

POSITION(searchString, string): int

Returns the location of a search string in a string. See also LOCATE.

Example:

POSITION('.', NAME)

LPAD

LPAD(string, int[, paddingString]): string

Left pad the string to the specified length. If the length is shorter than the string, it will be truncated at the end. If the padding string is not set, spaces will be used.

Example:

LPAD(AMOUNT, 10, '*')

RPAD

```
RPAD(string, int[, paddingString]): string
```

Right pad the string to the specified length. If the length is shorter than the string, it will be truncated. If the padding string is not set, spaces will be used.

Example:

RPAD(TEXT, 10, '-')

LTRIM

```
LTRIM(string): string
```

Removes all leading spaces from a string.

Example:

LTRIM(NAME)

RTRIM

```
RTRIM(string): string
```

Removes all trailing spaces from a string.

Example:

RTRIM(NAME)

TRIM

```
TRIM([{LEADING | TRAILING | BOTH} [string] FROM]
string): string
```

Removes all leading spaces, trailing spaces, or spaces at both ends, from a string. Other characters can be removed as well.

Example:

TRIM(BOTH '_' FROM NAME)

REGEXP_REPLACE

```
REGEXP_REPLACE(inputString, regexString, replacementString): string
```

Replaces each substring that matches a regular expression. For details, see the Java String.replaceAll() method.

Example:

REGEXP_REPLACE('Hello World', ' +', ' ')

REPEAT

REPEAT(string, int): string

Returns a string repeated some number of times.

Example:

REPEAT(NAME || ' ', 10)

REPLACE

REPLACE(string, searchString [, replacementString]): string

Replaces all occurrences of a search string in a text with another string. If no replacement is specified, the search string is just removed from the original string.

Example:

REPLACE(NAME, ' ')

SOUNDEX

SOUNDEX(string): string

Returns a four character code representing the sound of a string. See also http://www.archives.gov/genealogy/census/soundex.html .

Example:

SOUNDEX(NAME)

SPACE

SPACE(int): string

Returns a string consisting of a number of spaces.

Example:

SPACE(80)

STRINGDECODE

```
STRINGDECODE(string): string
```

Converts a encoded string using the Java string literal encoding format. Special characters are \b, \t, \n, \f, \r, \", \\, \<octal>, \u<unicode>.

Example:

CALL STRINGENCODE(STRINGDECODE('Lines 1\nLine 2'))

STRINGENCODE

```
STRINGENCODE(string): string
```

Encodes special characters in a string using the Java string literal encoding format. Special characters are \b, \t, \n, \f, \r, \", \\, \<octal>, \u<unicode>.

Example:

CALL STRINGENCODE(STRINGDECODE('Lines 1\nLine 2'))

STRINGTOUTF8

```
STRINGTOUTF8(string): bytes
```

Encodes a string to a byte array using the UTF8 encoding format.

Example:

CALL UTF8TOSTRING(STRINGTOUTF8('This is a test'))

SUBSTRING

```
{SUBSTRING | SUBSTR}(string, startInt [, lengthInt]): string
```

Returns a substring of a string starting at a position. The length is optional. Also supported is: SUBSTRING(string FROM start [FOR length]).

Example:

SUBSTR(NAME, 1)

UTF8TOSTRING

```
UTF8TOSTRING(bytes): string
```

Decodes a byte array in the UTF8 format to a string.

Example:

CALL UTF8TOSTRING(STRINGTOUTF8('This is a test'))

XMLATTR

```
XMLATTR(nameString, valueString): string
```

Creates an XML attribute element of the form name="value". The value is encoded as XML text.

Example:

CALL XMLNODE('a', XMLATTR('href', 'http://h2database.com'))

XMLNODE

```
XMLNODE(elementString [, attributesString [, contentString]]): string
```

Create an XML node element.

Example:

CALL XMLNODE('a', XMLATTR('href', 'http://h2database.com'), 'H2')

XMLCOMMENT

```
XMLCOMMENT(commentString): string
```

Creates an XML comment. Two dashes (--) are converted to - -.

Example:

CALL XMLCOMMENT('Test')

XMLCDATA

```
XMLCDATA(valueString): string
```

Creates an XML CDATA element. If the value contains ']]>', an XML text element is created instead.

Example:

CALL XMLCDATA('data')

XMLSTARTDOC

```
XMLSTARTDOC(): string
```

The string '<?xml version="1.0"?>' is returned.

Example:

CALL XMLSTARTDOC()

XMLTEXT

XMLTEXT(valueString): string

Creates an XML text element.

Example:

CALL XMLTEXT('test')

ARRAY_GET

ARRAY_GET(arrayExpression, indexExpression): varchar

Returns one element of an array.

Example:

CALL ARRAY_GET(('Hello', 'World'), 2)

ARRAY_LENGTH

ARRAY_GET(arrayExpression): int

Returns the length of an array.

Example:

CALL ARRAY_LENGTH(('Hello', 'World'))

AUTOCOMMIT

AUTOCOMMIT(): boolean

Returns true if auto commit is switched on for this session.

Example:

AUTOCOMMIT()

CANCEL_SESSION

```
CANCEL_SESSION(sessionInt): boolean
```

Cancels the currently executing statement of another session. The method only works if the multithreaded kernel is enabled (see SET MULTI_THREADED). Returns true if the statement was canceled, false if the session is closed or no statement is currently executing.

Admin rights are required to execute this command.

Example:

CANCEL_SESSION(3)

CASEWHEN Function

```
CASEWHEN(boolean, aValue, bValue): value
```

Returns 'a' if the boolean expression is true, otherwise 'b'.

Example:

CASEWHEN(ID=1, 'A', 'B')

CAST

```
CAST(value AS dataType): value
```

Converts a value to another data type. When converting a text to a number, the default Java conversion rules are used (prefixes 0x or # for hexadecimal numbers, prefix 0 for octal numbers).

Example:

CAST(NAME AS INT)

COALESCE

```
COALESCE(aValue, bValue [,...]): value
```

Returns the first value that is not null.

Example:

COALESCE(A, B, C)

CONVERT

```
CONVERT(value, dataType): value
```

Converts a value to another data type.

Example:

CONVERT(NAME, INT)

CURRVAL

```
CURRVAL([schemaName, ] sequenceString): long
```

Returns the current (last) value of the sequence, independent of the session. If the sequence was just created, the method returns (start - interval). If the schema name is not set, the current schema is used. If the schema name is not set, the sequence name is converted to uppercase (for compatibility).

Example:

CURRVAL('TEST_SEQ')

CSVREAD

```
CSVREAD(fileNameString [, columnNamesString [, charsetString
[, fieldSeparatorString [, fieldDelimiterString [, escapeCharacterString
[, nullString]]]]]]): resultSet
```

Returns the result set of reading the CSV (comma separated values) file. For each parameter, NULL means the default value should be used.

If the column names are specified (a list of column names separated with the fieldSeparator), those are used they are read from the file, otherwise (or if they are set to NULL) the first line of the file is interpreted as the column names. Column names are case sensitive, that means you need to use quoted identifiers unless the column names are capitalized (see below).

The default charset is the default value for this system, and the default field separator is a comma. Missing unquoted values as well as data that matches nullString is parsed as NULL. All columns of type VARCHAR.

This function can be used like a table: SELECT * FROM CSVREAD(...). Instead of a file, an URL may be used, for example jar:file:///c:/temp/example.zip!/org/example/nested.zip. Admin rights are required to execute this command.

Example:

```
CALL CSVREAD('test.csv');
-- Read a file containing the columns ID, NAME with
-- UTF-8 encoding and the pipe (|) as field separator
CALL CSVREAD('test2.csv', 'ID|NAME', 'UTF-8', '|');
-- Read a semicolon-separated file
SELECT * FROM CSVREAD('data/test.csv', NULL, NULL, ';');
SELECT "Last Name" FROM CSVREAD('address.csv');
```

CSVWRITE

```
CSVWRITE(fileNameString, queryString [, charsetString [, fieldSeparatorString
[, fieldDelimiterString [, escapeCharacterString [, nullString
[, lineSeparatorString]]]]]]): int
```

Writes a CSV (comma separated values). The file is overwritten if it exists. For each parameter, NULL means the default value should be used. The default charset is the default value for this system, and the default field separator is a comma.

The values are converted to text using the default string representation; if another conversion is required you need to change the select statement accordingly. The parameter nullString is used when writing NULL (by default nothing is written when NULL appears). The default line separator is the default value for this system ('line.separator' system property).

The returned value is the number or rows written. Admin rights are required to execute this command.

Example:

```
CALL CSVWRITE('test.csv', 'SELECT * FROM TEST');
-- Write a file with UTF-8 encoding and the pipe (|) as field separator
CALL CSVWRITE('test2.csv', 'SELECT * FROM TEST', 'UTF-8', '|');
```

DATABASE

```
DATABASE(): string
```

Returns the name of the database.

Example:

```
CALL DATABASE();
```

DATABASE_PATH

```
DATABASE_PATH(): string
```

Returns the directory of the database files and the database name, if it is file based. Returns NULL otherwise.

Example:

```
CALL DATABASE_PATH();
```

FILE_READ

```
FILE_READ(fileNameString [,encodingString]): value
```

Returns the contents of a file. If only one parameter is supplied, the data are returned as a BLOB. If two parameters are used, the data is returned as a CLOB (text). The second parameter is the character set to use, NULL meaning the default character set for this system. File names and URLs are supported. Admin rights are required to execute this command.

Example:

```
SELECT LENGTH(FILE_READ('~/.h2.server.properties')) LEN;
SELECT FILE_READ('http://localhost:8182/stylesheet.css', NULL) CSS;
```

GREATEST

```
GREATEST(aValue, bValue [,...]): value
```

Returns the largest value that is not NULL, or NULL if all values are NULL.

Example:

CALL GREATEST(1, 2, 3);

IDENTITY

IDENTITY(): long

Returns the last inserted identity value for this session.

Example:

CALL IDENTITY();

IFNULL

IFNULL(aValue, bValue): value

Returns the value of 'a' if it is not null, otherwise 'b'.

Example:

CALL IFNULL(NULL, '');

LEAST

LEAST(aValue, bValue [,...]): value

Returns the smallest value that is not NULL, or NULL if all values are NULL.

Example:

CALL LEAST(1, 2, 3);

LOCK_MODE

LOCK_MODE(): int

Returns the current lock mode. See SET LOCK_MODE.

Example:

CALL LOCK_MODE();

LOCK_TIMEOUT

LOCK_TIMEOUT(): int

Returns the lock timeout of the current session (in milliseconds).

Example:

LOCK_TIMEOUT()

LINK_SCHEMA

```
LINK_SCHEMA(targetSchemaString, driverString, urlString,
userString, passwordString, sourceSchemaString): resultSet
```

Creates table links for all tables in a schema. If tables with the same name already exist, they are dropped first. The target schema is created automatically if it does not yet exist. The driver name may be empty if the driver is already loaded. The list of tables linked is returned. Admin rights are required to execute this command.

Example:

CALL LINK_SCHEMA('TEST2', '', 'jdbc:h2:test2', 'sa', 'sa', 'PUBLIC');

MEMORY_FREE

```
MEMORY_FREE(): int
```

Returns the free memory in KB (where 1024 bytes is a KB). The garbage is run before returning the value. Admin rights are required to execute this command.

Example:

MEMORY_FREE()

MEMORY_USED

```
MEMORY_USED(): int
```

Returns the used memory in KB (where 1024 bytes is a KB). The garbage is run before returning the value. Admin rights are required to execute this command.

Example:

MEMORY_USED()

NEXTVAL

```
NEXTVAL([schemaName, ] sequenceString): long
```

Returns the next value of the sequence. Used values are never re-used, even when the transaction is rolled back. If the schema name is not set, the current schema is used, and the sequence name is converted to uppercase (for compatibility).

Example:

NEXTVAL('TEST_SEQ')

NULLIF

```
NULLIF(aValue, bValue): value
```

Returns NULL if 'a' is equals to 'b', otherwise 'a'.

Example:

NULLIF(A, B)

READONLY

```
READONLY(): boolean
```

Returns true if the database is read-only.

Example:

READONLY()

ROWNUM

```
ROWNUM(): int
```

Returns the number of the current row. This function is supported for SELECT statements, as well as for DELETE and UPDATE. The first row has the row number 1, and is calculated before ordering and grouping the result set. To get the row number after ordering and grouping, use a subquery.

Example:

SELECT ROWNUM(), * FROM TEST;
SELECT ROWNUM(), * FROM (SELECT * FROM TEST ORDER BY NAME);

SCHEMA

```
SCHEMA(): string
```

Returns the name of the default schema for this session.

Example:

CALL SCHEMA()

SESSION_ID

```
SESSION_ID(): int
```

Returns the unique session id number for the current database connection. This id stays the same while the connection is open. The database engine may re-use a session id after the connection is closed.

Example:

```
CALL SESSION_ID()
```

SET

```
SET(@variableName, value): value
```

Updates a variable with the given value. The new value is returned. When used in a query, the value is updated in the order the rows are read.

Example:

```
SELECT X, SET(@I, IFNULL(@I, 0)+X) RUNNING_TOTAL FROM SYSTEM_RANGE(1, 10)
```

TABLE

```
TABLE|TABLE_DISTINCT( { name dataType = expression } [,..]): result set
```

Returns the result set. TABLE_DISTINCT removes duplicate rows.

Example:

```
SELECT * FROM TABLE(ID INT=(1, 2), NAME VARCHAR=('Hello', 'World'))
```

TRANSACTION_ID

```
TRANSACTION_ID(): string
```

Returns the current transaction id for this session. This method returns NULL if there is no uncommitted change, or if the the database is not persisted. Otherwise a value of the following form is returned: logFileId-position-sessionId. The value is unique across database restarts (values are not re-used).

Example:

```
CALL TRANSACTION_ID()
```

USER

```
{USER | CURRENT_USER}(): string
```

Returns the name of the current user of this session.

Example:

CURRENT_USER()

CURRENT_DATE

{CURRENT_DATE[()] | CURDATE() | SYSDATE | TODAY}: date

Returns the current date.

Example:

CURRENT_DATE()

CURRENT_TIME

{CURRENT_TIME[()] | CURTIME()}: time

Returns the current time.

Example:

CURRENT_TIME()

CURRENT_TIMESTAMP

{CURRENT_TIMESTAMP[([int])] | NOW([int])}: timestamp

Returns the current timestamp. The precision parameter for nanoseconds precision is optional.

Example:

CURRENT_TIMESTAMP()

DATEADD

DATEADD(unitString, addInt, timestamp): timestamp

Adds units to a timestamp. The string indicates the unit. Use negative values to subtract units. The same units as in the EXTRACT function are supported.

Example:

DATEADD('MONTH', 1, DATE '2001-01-31')

DATEDIFF

```
DATEDIFF(unitString, aTimestamp, bTimestamp): long
```

Returns the difference between two timestamps. The string indicates the unit. The same units as in the EXTRACT function are supported.

Example:

DATEDIFF('YEAR', T1.CREATED, T2.CREATED)

DAYNAME

```
DAYNAME(date): string
```

Returns the name of the day (in English).

Example:

DAYNAME(CREATED)

DAY_OF_MONTH

```
DAY_OF_MONTH(date): int
```

Returns the day of the month (1-31).

Example:

DAY_OF_MONTH(CREATED)

DAY_OF_WEEK

```
DAY_OF_WEEK(date): int
```

Returns the day of the week (1 means Sunday).

Example:

DAY_OF_WEEK(CREATED)

DAY_OF_YEAR

```
DAY_OF_YEAR(date): int
```

Returns the day of the year (1-366).

Example:

DAY_OF_YEAR(CREATED)

EXTRACT

```
EXTRACT(
{YEAR | YY | MONTH | MM | DAY | DD | DAY_OF_YEAR | DOY |
HOUR | HH | MINUTE | MI | SECOND | SS | MILLISECOND | MS}
FROM timestamp): int
```

Returns a specific value from a timestamps.

Example:

EXTRACT(SECOND FROM CURRENT_TIMESTAMP)

FORMATDATETIME

```
FORMATDATETIME(timestamp, formatString [, localeString
[, timeZoneString]]): string
```

Formats a date, time or timestamp as a string. The most important format characters are: y year, M month, d day, H hour, m minute, s second For details of the format, see java.text.SimpleDateFormat.

Example:

```
CALL FORMATDATETIME(TIMESTAMP '2001-02-03 04:05:06',
    'EEE, d MMM yyyy HH:mm:ss z', 'en', 'GMT')
```

HOUR

```
HOUR(timestamp): int
```

Returns the hour (0-23) from a timestamp.

Example:

HOUR(CREATED)

MINUTE

```
MINUTE(timestamp): int
```

Returns the minute (0-59) from a timestamp.

Example:

MINUTE(CREATED)

MONTH

```
MONTH(timestamp): int
```

Returns the month (1-12) from a timestamp.

Example:

MONTH(CREATED)

MONTHNAME

```
MONTHNAME(date): string
```

Returns the name of the month (in English).

Example:

MONTHNAME(CREATED)

PARSEDATETIME

```
PARSEDATETIME(string, formatString [, localeString [, timeZoneString]]): string
```

Parses a string and returns a timestamp. The most important format characters are: y year, M month, d day, H hour, m minute, s second For details of the format, see java.text.SimpleDateFormat.

Example:

```
CALL PARSEDATETIME('Sat, 3 Feb 2001 03:05:06 GMT',
    'EEE, d MMM yyyy HH:mm:ss z', 'en', 'GMT')
```

QUARTER

```
QUARTER(timestamp): int
```

Returns the quarter (1-4) from a timestamp.

Example:

QUARTER(CREATED)

SECOND

```
SECOND(timestamp): int
```

Returns the second (0-59) from a timestamp.

Example:

SECOND(CREATED)

WEEK

```
WEEK(timestamp): int
```

Returns the week (1-53) from a timestamp. This method uses the current system locale.

Example:

WEEK(CREATED)

YEAR

```
YEAR(timestamp): int
```

Returns the year from a timestamp.

Example:

YEAR(CREATED)

Data Types

INT Type

```
INT | INTEGER | MEDIUMINT | INT4 | SIGNED
```

Possible values: -2147483648 to 2147483647.

Mapped to java.lang.Integer.

Example:

INT

BOOLEAN Type

```
BOOLEAN | BIT | BOOL
```

Possible values: TRUE and FALSE.

Mapped to java.lang.Boolean.

Example:

BOOLEAN

TINYINT Type

```
TINYINT
```

Possible values are: -128 to 127.

Mapped to java.lang.Byte.

Example:

TINYINT

SMALLINT Type

```
SMALLINT | INT2 | YEAR
```

Possible values: -32768 to 32767.

Mapped to java.lang.Short.

Example:

SMALLINT

BIGINT Type

```
BIGINT | INT8
```

Possible values: -9223372036854775808 to 9223372036854775807.

Mapped to java.lang.Long.

Example:

BIGINT

IDENTITY Type

```
IDENTITY
```

Auto-Increment value. Possible values: -9223372036854775808 to 9223372036854775807. Used values are never re-used, even when the transaction is rolled back.

Mapped to java.lang.Long.

Example:

IDENTITY

DECIMAL Type

```
{DECIMAL | NUMBER | DEC | NUMERIC} ( precisionInt [, scaleInt] )
```

Data type with fixed precision and scale. This data type is recommended for storing currency values.

Mapped to java.math.BigDecimal.

Example:

DECIMAL(20, 2)

DOUBLE Type

```
{DOUBLE [PRECISION] | FLOAT | FLOAT4 | FLOAT8}
```

Floating point number. Should not be used to represent currency values, because of rounding problems.

Mapped to java.lang.Double.

Example:

DOUBLE

REAL Type

```
REAL
```

Single precision floating point number. Should not be used to represent currency values, because of rounding problems.

Mapped to java.lang.Float.

Example:

REAL

TIME Type

```
TIME
```

The format is hh:mm:ss.

Mapped to java.sql.Time.

Example:

TIME

DATE Type

```
DATE
```

The format is yyyy-MM-dd.

Mapped to java.sql.Date

Example:

DATE

TIMESTAMP Type

```
{TIMESTAMP | DATETIME | SMALLDATETIME}
```

The format is yyyy-MM-dd hh:mm:ss[.nnnnnnnnn].

Mapped to java.sql.Timestamp (java.util.Date is also supported).

Example:

TIMESTAMP

BINARY Type

```
{BINARY | VARBINARY | LONGVARBINARY | RAW | BYTEA}
[( precisionInt )]
```

Represents a byte array. For very long arrays, use BLOB. The maximum size is 2 GB, but the whole object is kept in memory when using this data type. The precision is a size constraint; only the actual data is persisted. For large text data BLOB or CLOB should be used.

Mapped to byte[].

Example:

BINARY(1000)

OTHER Type

```
OTHER
```

This type allows storing serialized Java objects. Internally, a byte array is used. Serialization and deserialization is done on the client side only. Deserialization is only done get getObject is called. Java operations cannot be executed inside the database engine for security reasons. Use PreparedStatement.setObject to store values.

Mapped to java.lang.Object (or any subclass).

Example:

OTHER

VARCHAR Type

```
{VARCHAR | LONGVARCHAR |
VARCHAR2 | NVARCHAR | NVARCHAR2 | VARCHAR_CASESENSITIVE}
[( precisionInt )]
```

Unicode String. Use two single quotes ('') to create a quote. The maximum precision is Integer.MAX_VALUE. The precision is a size constraint; only the actual data is persisted. The whole text is kept in memory when using this data type. For large text data CLOB should be used; see there for details.

Mapped to java.lang.String.

Example:

VARCHAR(255)

VARCHAR_IGNORECASE Type

```
VARCHAR_IGNORECASE [( precisionInt )]
```

Same as VARCHAR, but not case sensitive when comparing. Stored in mixed case. The maximum precision is Integer.MAX_VALUE characters, but the whole text is kept in memory when using this data type. For large text data CLOB should be used; see there for details.

Mapped to java.lang.String.

Example:

VARCHAR_IGNORECASE

CHAR Type

```
{CHAR | CHARACTER | NCHAR}
[( precisionInt )]
```

This type is supported for compatibility with other databases and older applications. The difference to VARCHAR is that trailing spaces are ignored and not persisted. Unicode String. Use two single quotes ('') to create a quote. The maximum precision is Integer.MAX_VALUE. The precision is a size constraint; only the actual data is persisted. The whole text is kept in memory when using this data type. For large text data CLOB should be used; see there for details.

Mapped to java.lang.String.

Example:

CHAR(10)

BLOB Type

```
{BLOB | TINYBLOB | MEDIUMBLOB | LONGBLOB | IMAGE | OID}
[( precisionInt )]
```

Like BINARY, but intended for very large values such as files or images. Unlike when using BINARY, large objects are not kept fully in-memory. Use PreparedStatement.setBinaryStream to store values. See also CLOB and Advanced / Large Objects.

Mapped to java.sql.Blob (java.io.InputStream is also supported).

Example:

BLOB

CLOB Type

```
{CLOB | TINYTEXT | TEXT | MEDIUMTEXT | LONGTEXT | NTEXT | NCLOB}
[( precisionInt )]
```

CLOB is like VARCHAR, but intended for very large values. Unlike when using VARCHAR, large CLOB objects are not kept fully in-memory; instead, they are streamed. CLOB should be used for documents and texts with arbitrary size such as XML or HTML documents, text files, or memo fields of unlimited size. Use PreparedStatement.setCharacterStream to store values. See also Advanced / Large Objects.

VARCHAR should be used for text with relatively short average size (for example shorter than 200 characters). Short CLOB values are stored inline, but there is an overhead compared to VARCHAR.

Mapped to java.sql.Clob (java.io.Reader is also supported).

Example:

CLOB

UUID Type

```
UUID
```

Universally unique identifier. This is a 128 bit value. Use PreparedStatement.setBytes or setString to store values.

Mapped to java.util.UUID.

Example:

UUID

ARRAY Type

```
ARRAY
```

An array of values. Use a value list (1, 2) or PreparedStatement.setObject(.., new Object[]{..}) to store values.

Mapped to java.lang.Object[] (arrays of any non-primitive type are also supported).

Example:

ARRAY

Build

Portability

This database is written in Java and therefore works on many platforms. It can also be compiled to a native executable using GCJ.

Environment

A Java Runtime Environment (JRE) version 1.5 or higher is required to run this database.

To build the database executables, the following software stack was used. Newer version or compatible software works too.

- Mac OS X and Windows XP
- Sun JDK Version 1.5 and 1.6
- Eclipse Version 3.4
- Eclipse Plugins: Subclipse 1.4.6 , Eclipse Checkstyle Plug-in 4.4.2 , EclEmma Java Code Coverage 1.3.0
- Emma Java Code Coverage
- Mozilla Firefox 3.0
- OpenOffice 3.0
- NSIS 2.38 (Nullsoft Scriptable Install System)
- Maven 2.0.9

Building the Software

You need to install a JDK, for example the Sun JDK version 1.5 or 1.6. Ensure that Java binary directory is included in the PATH environment variable, and that the environment variable JAVA_HOME points to your Java installation. On the command line, go to the directory h2 and execute the following command:

```
build -?
```

For Linux and OS X, use ./build.sh instead of build .

You will get a list of targets. If you want to build the jar file, execute (Windows):

```
build jar
```

Switching the Source Code

By default the source code uses Java 1.5 features, however Java 1.6 is supported as well. To switch the source code to the install version of Java, run:

```
build switchSource
```

Build Targets

The build system can generate smaller jar files as well. The following targets are currently supported:

- jarClient creates the h2client.jar. This only contains the JDBC client.
- jarSmall creates the file h2small.jar. This only contains the embedded database. Debug information is disabled.
- jarJaqu creates the file h2jaqu.jar. This only contains the JaQu (Java Query) implementation. All other jar files do not include JaQu.
- javadocImpl creates the Javadocs of the implementation.

To create the h2client.jar file, go to the directory h2 and execute the following command:

```
build jarClient
```

Using Maven 2

Using a Central Repository

You can include the database in your Maven 2 project as a dependency. Example:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.1.118</version>
</dependency>
```

New versions of this database are first uploaded to http://hsql.sourceforge.net/m2-repo/ and then automatically synchronized with the main maven repository; however after a new release it may take a few hours before they are available there.

Using Snapshot Version

To build a 'snapshot' H2 .jar file and upload it the to the local Maven 2 repository, execute the following command:

```
build mavenInstallLocal
```

Afterwards, you can include the database in your Maven 2 project as a dependency:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

Translating

The translation of this software is split into the following parts:

- H2 Console: src/main/org/h2/server/web/res/_text_*.properties
- Error messages: src/main/org/h2/res/_messages_*.properties

- Web site: src/docsrc/text/_docs_*.utf8.txt

To translate the H2 Console, start it and select Preferences / Translate. The conversion between UTF-8 and Java encoding (using the \u syntax), as well as the HTML entities (&#..;) is automated by running the tool PropertiesToUTF8. The web site translation is automated as well, using build docs .

Providing Patches

If you like to provide patches, please consider the following guidelines to simplify merging them:

- Only use Java 1.5 features (do not use Java 1.6) (see Environment).
- Follow the coding style used in the project, and use Checkstyle (see above) to verify. For example, do not use tabs (use spaces instead). The checkstyle configuration is in src/installer/checkstyle.xml .
- Please provide test cases and integrate them into the test suite. For Java level tests, see src/test/org/h2/test/TestAll.java . For SQL level tests, see src/test/org/h2/test/test.in.txt or testSimple.in.txt .
- The test cases should cover at least 90% of the changed and new code; use a code coverage tool to verify that (see above). or use the build target 'coverage'.
- Verify that you did not break other features: run the test cases by executing build test .
- Provide end user documentation if required (src/docsrc/html/*).
- Document grammar changes in src/main/org/h2/res/help.csv
- Provide a change log entry (src/docsrc/html/changelog.html).
- Verify the spelling using build spellcheck . If required add the new words to src/tools/org/h2/build/doc/dictionary.txt .
- Run the src/installer/buildRelease to find and fix formatting errors.
- Verify the formatting using build docs and build javadoc .
- Submit patches as .patch files (compressed if big). To create a patch using Eclipse, use Team / Create Patch.

For legal reasons, patches need to be public in the form of an email to the group , or in the form of an issue report or attachment . Significant contributions need to include the following statement:

```
"I wrote the code, it's mine, and I'm contributing it to H2 for distribution
multiple-licensed under the H2 License, version 1.0, and under the
Eclipse Public License, version 1.0 (http://h2database.com/html/license.html)."
```

Automated Build

This build process is automated and runs regularly. The build process includes running the tests and code coverage, using the command line ./build.sh clean jar coverage -Dh2.ftpPassword=... uploadBuild . The last results are available here:

- Test Output
- Code Coverage Summary
- Code Coverage Details (download, 1.3 MB)
- Build Newsfeed
- Latest Jar File (download, 1 MB)

History and Roadmap

Change Log

The up-to-date change log is available at http://www.h2database.com/html/changelog.html

Roadmap

The current roadmap is available at http://www.h2database.com/html/roadmap.html

History of this Database Engine

The development of H2 was started in May 2004, but it was first published on December 14th 2005. The main author of H2, Thomas Mueller, is also the original developer of Hypersonic SQL. In 2001, he joined PointBase Inc. where he created PointBase Micro. At that point, he had to discontinue Hypersonic SQL, but then the HSQLDB Group was formed to continued to work on the Hypersonic SQL codebase. The name H2 stands for Hypersonic 2; however H2 does not share any code with Hypersonic SQL or HSQLDB. H2 is built from scratch.

Why Java

A few reasons using a Java database are:

- Very simple to integrate in Java applications
- Support for many different platforms
- More secure than native applications (no buffer overflows)
- User defined functions (or triggers) run very fast
- Unicode support

Some people think that Java is still too slow for low level operations, but this is not the case (not any more). In general, the code can be written a lot faster than using C or C++. Like that, it is possible to concentrate on improving the algorithms (that make the application faster) rather than porting the code and dealing with low level stuff (such as memory management or dealing with threads). Garbage collection is now probably faster than manual memory management.

A lot of features are already built in (for example Unicode, network libraries). It is very easy to write secure code because buffer overflows can not occur. Some features such as the reflection mechanism can be used for randomized testing.

Java is also future proof: a lot of companies support Java, and it is now open source.

This software does not rely on many Java libraries or other software, to increase the portability and ease of use, and for performance reasons. For example, the encryption algorithms and many library functions are implemented in the database instead of using the existing libraries. Libraries that are not available in open source Java implementations (such as Swing) are not used or only used for specific features.

Supporters

Many thanks for those who helped by finding and reporting bugs, gave valuable feedback, spread the word and have translated this project. Also many thanks to the donors who contributed via PayPal:

- SkyCash, Poland
- Donald Bleyl, USA
- lumber-mill.co.jp, Japan
- Frank Berger, Germany
- Ashwin Jayaprakash, USA
- Florent Ramiere, France
- Jun Iyama, Japan
- Antonio Casqueiro, Portugal
- Oliver Computing LLC, USA
- Harpal Grover Consulting Inc., USA
- Elisabetta Berlini, Italy
- William Gilbert, USA
- Antonio Dieguez, Chile
- Ontology Works, USA
- Pete Haidinyak, USA
- William Osmond, USA
- Joachim Ansorg, Germany
- Oliver Soerensen, Germany
- Christos Vasilakis, Greece
- Fyodor Kupolov, Denmark

Frequently Asked Questions

Are there Known Bugs? When is the Next Release?

Usually, bugs get fixes as they are found. There is a release every few weeks. Here is the list of known and confirmed issues:

- Tomcat and Glassfish 3 set most static fields (final or non-final) to null when unloading a web application. This can cause a NullPointerException in H2 versions 1.1.107 and older, and may still not work in newer versions. Please report it if you run into this issue. In Tomcat >= 6.0 this behavior can be disabled by setting the system property org.apache.catalina.loader.WebappClassLoader.ENABLE_CLEAR_REFERENCES to false, however Tomcat may then run out of memory. A known workaround is to put the h2.jar file in a shared lib directory (common/lib).
- Some problems have been found with right outer join. Internally, it is converted to left outer join, which does not always produce the same results as other databases when used in combination with other joins.
- When using Install4j before 4.1.4 on Linux and enabling 'pack200', the h2*.jar becomes corrupted by the install process, causing application failure. A workaround is to add an empty file h2*.jar.nopack next to the h2*.jar file. This problem is solved in Install4j 4.1.4.

Is this Database Engine Open Source?

Yes. It is free to use and distribute, and the source code is included. See also under license.

My Query is Slow

Slow SELECT (or DELETE, UPDATE, MERGE) statement can have multiple reasons. Follow this checklist:

- Run ANALYZE (see documentation for details).
- Run the query with EXPLAIN and check if indexes are used (see documentation for details).
- If required, create additional indexes and try again using ANALYZE and EXPLAIN.
- If it doesn't help please report the problem.

How to Create a New Database?

By default, a new database is automatically created if it does not yet exist.

How to Connect to a Database?

The database driver is org.h2.Driver , and the database URL starts with jdbc:h2: . To connect to a database using JDBC, use the following code:

```
Class.forName("org.h2.Driver");
Connection conn = DriverManager.getConnection("jdbc:h2:~/test", "sa", "");
```

Where are the Database Files Stored?

When using database URLs like jdbc:h2:~/test, the database is stored in the user directory. For Windows, this is usually "C:\Documents and Settings\<userName>". If the base directory is not set (as in jdbc:h2:test), the database files are stored in the directory where the application is started (the current working directory). When using the H2 Console application from the start menu, this is "<Installation Directory>/bin". The base directory can be set in the database URL. A fixed or relative path can be used. When using the URL jdbc:h2:file:data/sample, the database is stored in the directory "data" (relative to the current working directory). The directory is created automatically if it does not yet exist. It is also possible to use the fully qualified directory name (and for Windows, drive name). Example: jdbc:h2:file:C:/data/test

What is the Size Limit (Maximum Size) of a Database?

See Limits and Limitations .

Is it Reliable?

Some users have reported that after a power failure, the database can sometimes not be opened because the index file is corrupt. In that case, the index file can be deleted (it is automatically re-created). To avoid this, append ;LOG=2 to the database URL. See also: SET LOG . This problem will be solved using the new 'page store' mechanism (currently experimental).

That is not easy to say. It is still a quite new product. A lot of tests have been written, and the code coverage of these tests is very high. Randomized stress tests are run regularly. But there are probably still bugs that have not yet been found (as with most software). Some features are known to be dangerous, they are only supported for situations where performance is more important than reliability. Those dangerous features are:

- Disabling the transaction log mechanism using SET LOG 0.
- Using the transaction isolation level READ_UNCOMMITTED (LOCK_MODE 0) while at the same time using multiple connections.
- Disabling database file protection using FILE_LOCK=NO in the database URL.
- Disabling referential integrity using SET REFERENTIAL_INTEGRITY FALSE.

In addition to that, running out of memory should be avoided. In older versions, OutOfMemory errors while using the database could corrupt a databases.

Areas that are not fully tested:

- Platforms other than Windows XP, Linux, Mac OS X, or JVMs other than Sun 1.5 or 1.6
- The features AUTO_SERVER and AUTO_RECONNECT
- The MVCC (multi version concurrency) mode
- Cluster mode, 2-phase commit, savepoints
- 24/7 operation
- Some operations on databases larger than 500 MB may be slower than expected
- The optimizer may not always select the best plan
- Fulltext search
- Operations on LOBs over 2 GB

Areas considered Experimental:

- The PostgreSQL server
- The new page store
- Multi-threading within the engine using SET MULTI_THREADED=1
- Compatibility modes for other databases (only some features are implemented)

Why is Opening my Database Slow?

If it takes a long time to open a database, in most cases it was not closed the last time. This is specially a problem for larger databases. To close a database, close all connections to it before the application ends, or execute the command SHUTDOWN. The database is also closed when the virtual machine exits normally by using a shutdown hook. However killing a Java process or calling Runtime.halt will prevent this. The reason why opening is slow in this situations is that indexes are re-created. If you can not guarantee the database is closed, consider using SET LOG 2 (see SQL Grammar).

To find out what the problem is, open the database in embedded mode using the H2 Console. This will print progress information. If you have many 'Creating index' lines it is an indication that the database was not closed the last time.

Other possible reasons are: the database is very big (many GB), or contains linked tables that are slow to open.

Is the GCJ Version Stable? Faster?

The GCJ version is not as stable as the Java version. When running the regression test with the GCJ version, sometimes the application just stops at what seems to be a random point without error message. Currently, the GCJ version is also slower than when using the Sun VM. However, the startup of the GCJ version is faster than when using a VM.

How to Translate this Project?

For more information, see Build/Translating .