H2 Database Engine

Version 1.3.168 (2012-07-13)

Table of Contents

H2 Database Engine....1
Quickstart....12
Embedding H2 in an Application....12
The H2 Console Application....12
Step-by-Step....12
Installation....12
Start the Console....12
Login....13
Sample....14
Execute....15
Disconnect....16
End....16
Installation....17
Requirements....17
Database Engine....17
H2 Console....17
Supported Platforms....17
Installing the Software....17
Directory Structure....17
Tutorial....18
Starting and Using the H2 Console....18
Firewall....19
Testing Java....19
Error Message 'Port may be in use'....19
Using another Port....19
Connecting to the Server using a Browser....19
Multiple Concurrent Sessions....19
Login....20
Error Messages....20
Adding Database Drivers....20
Using the H2 Console....20
Inserting Table Names or Column Names....20
Disconnecting and Stopping the Application....20
Special H2 Console Syntax....20
Settings of the H2 Console....21
Connecting to a Database using JDBC....21
Creating New Databases....22
Using the Server....22
Starting the Server Tool from Command Line....22
Connecting to the TCP Server....22
Starting the TCP Server within an Application....22
Stopping a TCP Server from Another Process....22
Using Hibernate....23
Using TopLink and Glassfish....23
Using EclipseLink....23
Using Apache ActiveMQ....23
Using H2 within NetBeans....24
Using H2 with jOOQ....24
Using Databases in Web Applications....24
Embedded Mode....25
Server Mode....25
Using a Servlet Listener to Start and Stop a Database....25
Using the H2 Console Servlet....25
Android....26
CSV (Comma Separated Values) Support....27
Reading a CSV File from Within a Database....27
Importing Data from a CSV File....27
Writing a CSV File from Within a Database....27
Writing a CSV File from a Java Application....27
Reading a CSV File from a Java Application....28
Upgrade, Backup, and Restore....28
Database Upgrade....28
Backup using the Script Tool....28
Restore from a Script....28
Online Backup....28
Command Line Tools....29

The Shell Tool....29
Using OpenOffice Base....30
Java Web Start / JNLP....30
Using a Connection Pool....30
Fulltext Search....31
Using the Native Fulltext Search....31
Using the Lucene Fulltext Search....31
User-Defined Variables....32
Date and Time....33
Using Spring....33
Using the TCP Server....33
Error Code Incompatibility....33
Java Management Extension (JMX)....34
Features....35
Feature List....35
Main Features....35
Additional Features....35
SQL Support....35
Security Features....36
Other Features and Tools....36
Comparison to Other Database Engines....36
DaffodilDb and One$Db....37
McKoi....37
H2 in Use....37
Connection Modes....37
Embedded Mode....37
Server Mode....38
Mixed Mode....38
Database URL Overview....39
Connecting to an Embedded (Local) Database....40
In-Memory Databases....40
Database Files Encryption....40
Creating a New Database with File Encryption....40
Connecting to an Encrypted Database....40
Encrypting or Decrypting a Database....41
Database File Locking....41
Opening a Database Only if it Already Exists....41
Closing a Database....41
Delayed Database Closing....41
Don't Close a Database when the VM Exits....42
Execute SQL on Connection....42
Ignore Unknown Settings....42
Changing Other Settings when Opening a Connection....42
Custom File Access Mode....42
Multiple Connections....43
Opening Multiple Databases at the Same Time....43
Multiple Connections to the Same Database: Client/Server....43
Multithreading Support....43
Locking, Lock-Timeout, Deadlocks....43
Avoiding Deadlocks....44
Database File Layout....44
Moving and Renaming Database Files....44
Backup....44
Logging and Recovery....44
Compatibility....44
Compatibility Modes....45
DB2 Compatibility Mode....45
Derby Compatibility Mode....45
HSQLDB Compatibility Mode....45
MS SQL Server Compatibility Mode....45
MySQL Compatibility Mode....45
Oracle Compatibility Mode....46
PostgreSQL Compatibility Mode....46
Auto-Reconnect....46
Automatic Mixed Mode....46
Page Size....47
Using the Trace Options....47
Trace Options....47
Setting the Maximum Size of the Trace File....47
Java Code Generation....48
Using Other Logging APIs....48

Read Only Databases....48
Read Only Databases in Zip or Jar File....48
Opening a Corrupted Database....49
Computed Columns / Function Based Index....49
Multi-Dimensional Indexes....49
User-Defined Functions and Stored Procedures....50
Referencing a Compiled Method....50
Declaring Functions as Source Code....50
Method Overloading....51
Function Data Type Mapping....51
Functions That Require a Connection....51
Functions Throwing an Exception....51
Functions Returning a Result Set....51
Using SimpleResultSet....51
Using a Function as a Table....52
Triggers....52
Compacting a Database....53
Cache Settings....53
Performance....55
Performance Comparison....55
Embedded....55
Client-Server....55
Benchmark Results and Comments....56
H2....56
HSQLDB....56
Derby....56
PostgreSQL....56
MySQL....56
Firebird....56
Why Oracle / MS SQL Server / DB2 are Not Listed....56
About this Benchmark....57
How to Run....57
Separate Process per Database....57
Number of Connections....57
Real-World Tests....57
Comparing Embedded with Server Databases....57
Test Platform....57
Multiple Runs....57
Memory Usage....57
Delayed Operations....57
Transaction Commit / Durability....58
Using Prepared Statements....58
Currently Not Tested: Startup Time....58
PolePosition Benchmark....58
Database Performance Tuning....59
Keep Connections Open or Use a Connection Pool....59
Use a Modern JVM....59
Virus Scanners....59
Using the Trace Options....59
Index Usage....59
How Data is Stored Internally....59
Optimizer....60
Expression Optimization....60
COUNT(*) Optimization....60
Updating Optimizer Statistics / Column Selectivity....60
In-Memory (Hash) Indexes....60
Use Prepared Statements....60
Prepared Statements and IN(...)....60
Optimization Examples....61
Cache Size and Type....61
Data Types....61
Sorted Insert Optimization....61
Using the Built-In Profiler....61
Application Profiling....61
Analyze First....61
Database Profiling....62
Statement Execution Plans....62
Displaying the Scan Count....63
Special Optimizations....63
How Data is Stored and How Indexes Work....64
Indexes....64

Using Multiple Indexes....65
Fast Database Import....66
Advanced....67
Result Sets....67
Statements that Return a Result Set....67
Limiting the Number of Rows....67
Large Result Sets and External Sorting....67
Large Objects....68
Storing and Reading Large Objects....68
When to use CLOB/BLOB....68
Large Object Compression....68
Linked Tables....68
Updatable Views....68
Transaction Isolation....69
Table Level Locking....69
Lock Timeout....69
Multi-Version Concurrency Control (MVCC)....70
Clustering / High Availability....70
Using the CreateCluster Tool....70
Detect Which Cluster Instances are Running....71
Clustering Algorithm and Limitations....71
Two Phase Commit....71
Compatibility....72
Transaction Commit when Autocommit is On....72
Keywords / Reserved Words....72
Standards Compliance....72
Supported Character Sets, Character Encoding, and Unicode....72
Run as Windows Service....72
Install the Service....72
Start the Service....73
Connect to the H2 Console....73
Stop the Service....73
Uninstall the Service....73
Additional JDBC drivers....73
ODBC Driver....73
ODBC Installation....73
Starting the Server....73
ODBC Configuration....74
PG Protocol Support Limitations....74
Security Considerations....74
Using Microsoft Access....74
Using H2 in Microsoft .NET....75
Using the ADO.NET API on .NET....75
Using the JDBC API on .NET....75
ACID....75
Atomicity....75
Consistency....75
Isolation....76
Durability....76
Durability Problems....76
Ways to (Not) Achieve Durability....76
Running the Durability Test....77
Using the Recover Tool....77
File Locking Protocols....77
File Locking Method 'File'....77
File Locking Method 'Socket'....78
File Locking Method 'FS'....78
File Locking Method 'Serialized'....78
Using Passwords....79
Using Secure Passwords....79
Passwords: Using Char Arrays instead of Strings....79
Passing the User Name and/or Password in the URL....79
Password Hash....79
Protection against SQL Injection....80
What is SQL Injection....80
Disabling Literals....80
Using Constants....80
Using the ZERO() Function....81
Protection against Remote Access....81
Restricting Class Loading and Usage....81
Security Protocols....81

User Password Encryption....81
File Encryption....82
Wrong Password / User Name Delay....82
HTTPS Connections....83
SSL/TLS Connections....83
Universally Unique Identifiers (UUID)....83
Recursive Queries....83
Settings Read from System Properties....84
Setting the Server Bind Address....84
Pluggable File System....84
Split File System....85
Database Upgrade....85
Limits and Limitations....86
Glossary and Links....86
SQL Grammar....87
Index....87
Commands (Data Manipulation)....87
Commands (Data Definition)....87
Commands (Other)....88
Other Grammar....88
System Tables....89
SELECT....89
INSERT....90
UPDATE....90
DELETE....90
BACKUP....90
CALL....91
EXPLAIN....91
MERGE....91
RUNSCRIPT....91
SCRIPT....92
SHOW....92
ALTER INDEX RENAME....92
ALTER SCHEMA RENAME....93
ALTER SEQUENCE....93
ALTER TABLE ADD....93
ALTER TABLE ADD CONSTRAINT....93
ALTER TABLE ALTER....93
ALTER TABLE DROP COLUMN....94
ALTER TABLE DROP CONSTRAINT....94
ALTER TABLE SET....94
ALTER TABLE RENAME....95
ALTER USER ADMIN....95
ALTER USER RENAME....95
ALTER USER SET PASSWORD....95
ALTER VIEW....96
ANALYZE....96
COMMENT....96
CREATE AGGREGATE....96
CREATE ALIAS....97
CREATE CONSTANT....97
CREATE DOMAIN....97
CREATE INDEX....98
CREATE LINKED TABLE....98
CREATE ROLE....98
CREATE SCHEMA....99
CREATE SEQUENCE....99
CREATE TABLE....99
CREATE TRIGGER....100
CREATE USER....100
CREATE VIEW....101
DROP AGGREGATE....101
DROP ALIAS....101
DROP ALL OBJECTS....101
DROP CONSTANT....102
DROP DOMAIN....102
DROP INDEX....102
DROP ROLE....102
DROP SCHEMA....102
DROP SEQUENCE....102
DROP TABLE....103

DROP TRIGGER....103
DROP USER....103
DROP VIEW....103
TRUNCATE TABLE....103
CHECKPOINT....104
CHECKPOINT SYNC....104
COMMIT....104
COMMIT TRANSACTION....104
GRANT RIGHT....105
GRANT ROLE....105
HELP....105
PREPARE COMMIT....105
REVOKE RIGHT....105
REVOKE ROLE....106
ROLLBACK....106
ROLLBACK TRANSACTION....106
SAVEPOINT....106
SET @....106
SET ALLOW_LITERALS....107
SET AUTOCOMMIT....107
SET CACHE_SIZE....107
SET CLUSTER....108
SET COLLATION....108
SET COMPRESS_LOB....108
SET DATABASE_EVENT_LISTENER....108
SET DB_CLOSE_DELAY....109
SET DEFAULT_LOCK_TIMEOUT....109
SET DEFAULT_TABLE_TYPE....109
SET EXCLUSIVE....109
SET IGNORECASE....110
SET LOG....110
SET LOCK_MODE....110
SET LOCK_TIMEOUT....111
SET MAX_LENGTH_INPLACE_LOB....111
SET MAX_LOG_SIZE....111
SET MAX_MEMORY_ROWS....112
SET MAX_MEMORY_UNDO....112
SET MAX_OPERATION_MEMORY....112
SET MODE....112
SET MULTI_THREADED....113
SET OPTIMIZE_REUSE_RESULTS....113
SET PASSWORD....113
SET QUERY_TIMEOUT....113
SET REFERENTIAL_INTEGRITY....114
SET SALT HASH....114
SET SCHEMA....114
SET SCHEMA_SEARCH_PATH....114
SET THROTTLE....115
SET TRACE_LEVEL....115
SET TRACE_MAX_FILE_SIZE....115
SET UNDO_LOG....115
SET WRITE_DELAY....116
SHUTDOWN....116
Alias....116
And Condition....116
Array....117
Boolean....117
Bytes....117
Case....117
Case When....117
Cipher....118
Column Definition....118
Comments....118
Compare....118
Condition....119
Condition Right Hand Side....119
Constraint....119
Constraint Name Definition....119
Csv Options....120
Data Type....120
Date....120

Decimal....120
Digit....121
Dollar Quoted String....121
Expression....121
Factor....121
Hex....121
Hex Number....122
Index Column....122
Int....122
Long....122
Name....122
Null....123
Number....123
Numeric....123
Operand....123
Order....123
Quoted Name....124
Referential Constraint....124
Referential Action....124
Script Compression....124
Select Expression....124
String....125
Summand....125
Table Expression....125
Values Expression....125
Term....125
Time....126
Timestamp....126
Value....126
Information Schema....126
Range Table....127
Functions....128
Index....128
Aggregate Functions....128
Numeric Functions....128
String Functions....129
Time and Date Functions....129
System Functions....130
AVG....130
BOOL_AND....130
BOOL_OR....131
COUNT....131
GROUP_CONCAT....131
MAX....131
MIN....131
SUM....132
SELECTIVITY....132
STDDEV_POP....132
STDDEV_SAMP....132
VAR_POP....132
VAR_SAMP....133
ABS....133
ACOS....133
ASIN....133
ATAN....133
COS....134
COSH....134
COT....134
SIN....134
SINH....134
TAN....134
TANH....135
ATAN2....135
BITAND....135
BITOR....135
BITXOR....135
MOD....136
CEILING....136
DEGREES....136
EXP....136
FLOOR....136

LOG....137
LOG10....137
RADIANS....137
SQRT....137
PI....137
POWER....137
RAND....138
RANDOM_UUID....138
ROUND....138
ROUNDMAGIC....138
SECURE_RAND....139
SIGN....139
ENCRYPT....139
DECRYPT....139
HASH....139
TRUNCATE....140
COMPRESS....140
EXPAND....140
ZERO....140
ASCII....140
BIT_LENGTH....141
LENGTH....141
OCTET_LENGTH....141
CHAR....141
CONCAT....141
DIFFERENCE....142
HEXTORAW....142
RAWTOHEX....142
INSTR....142
INSERT Function....142
LOWER....143
UPPER....143
LEFT....143
RIGHT....143
LOCATE....143
POSITION....144
LPAD....144
RPAD....144
LTRIM....144
RTRIM....144
TRIM....145
REGEXP_REPLACE....145
REPEAT....145
REPLACE....145
SOUNDEX....145
SPACE....146
STRINGDECODE....146
STRINGENCODE....146
STRINGTOUTF8....146
SUBSTRING....146
UTF8TOSTRING....147
XMLATTR....147
XMLNODE....147
XMLCOMMENT....147
XMLCDATA....147
XMLSTARTDOC....148
XMLTEXT....148
ARRAY_GET....148
ARRAY_LENGTH....148
ARRAY_CONTAINS....148
AUTOCOMMIT....148
CANCEL_SESSION....149
CASEWHEN Function....149
CAST....149
COALESCE....149
CONVERT....150
CURRVAL....150
CSVREAD....150
CSVWRITE....151
DATABASE....151
DATABASE_PATH....151

DECODE....151
FILE_READ....151
GREATEST....152
IDENTITY....152
IFNULL....152
LEAST....152
LOCK_MODE....153
LOCK_TIMEOUT....153
LINK_SCHEMA....153
MEMORY_FREE....153
MEMORY_USED....153
NEXTVAL....154
NULLIF....154
NVL2....154
READONLY....154
ROWNUM....154
SCHEMA....155
SCOPE_IDENTITY....155
SESSION_ID....155
SET....155
TABLE....155
TRANSACTION_ID....156
TRUNCATE_VALUE....156
USER....156
CURRENT_DATE....156
CURRENT_TIME....156
CURRENT_TIMESTAMP....157
DATEADD....157
DATEDIFF....157
DAYNAME....157
DAY_OF_MONTH....157
DAY_OF_WEEK....158
DAY_OF_YEAR....158
EXTRACT....158
FORMATDATETIME....158
HOUR....158
MINUTE....159
MONTH....159
MONTHNAME....159
PARSEDATETIME....159
QUARTER....159
SECOND....160
WEEK....160
YEAR....160
Data Types....161
Index....161
INT Type....161
BOOLEAN Type....161
TINYINT Type....161
SMALLINT Type....162
BIGINT Type....162
IDENTITY Type....162
DECIMAL Type....162
DOUBLE Type....163
REAL Type....163
TIME Type....163
DATE Type....163
TIMESTAMP Type....163
BINARY Type....164
OTHER Type....164
VARCHAR Type....164
VARCHAR_IGNORECASE Type....164
CHAR Type....165
BLOB Type....165
CLOB Type....165
UUID Type....166
ARRAY Type....166
Build....167
Portability....167
Environment....167
Building the Software....167

Switching the Source Code....167
Build Targets....168
Using Lucene 2 / 3....168
Using Maven 2....168
Using a Central Repository....168
Maven Plugin to Start and Stop the TCP Server....168
Using Snapshot Version....168
Using Eclipse....169
Translating....169
Providing Patches....169
Reporting Problems or Requests....170
Automated Build....170
Generating Railroad Diagrams....170
History and Roadmap....171
Change Log....171
Roadmap....171
History of this Database Engine....171
Why Java....171
Supporters....171
Frequently Asked Questions....173
I Have a Problem or Feature Request....173
Are there Known Bugs? When is the Next Release?....173
Is this Database Engine Open Source?....173
Is Commercial Support Available?....173
How to Create a New Database?....174
How to Connect to a Database?....174
Where are the Database Files Stored?....174
What is the Size Limit (Maximum Size) of a Database?....174
Is it Reliable?....174
Why is Opening my Database Slow?....175
My Query is Slow....175
H2 is Very Slow....175
Column Names are Incorrect?....175
Float is Double?....175
Is the GCJ Version Stable? Faster?....175
How to Translate this Project?....176
How to Contribute to this Project?....176

Quickstart

Embedding H2 in an Application
The H2 Console Application

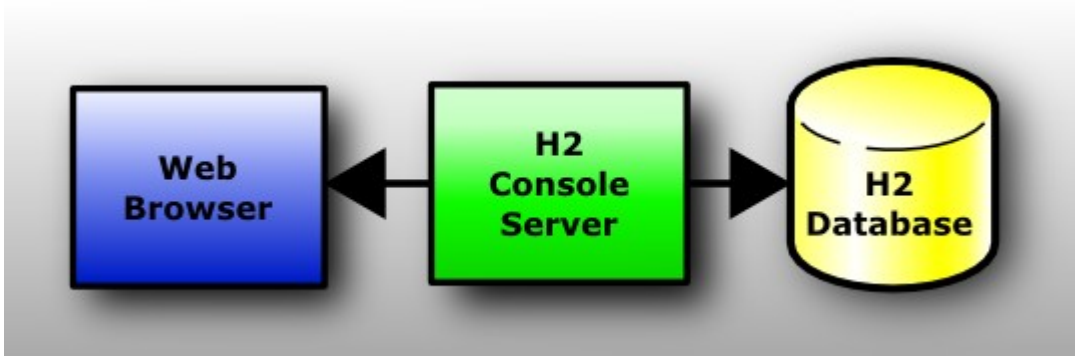
Embedding H2 in an Application

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

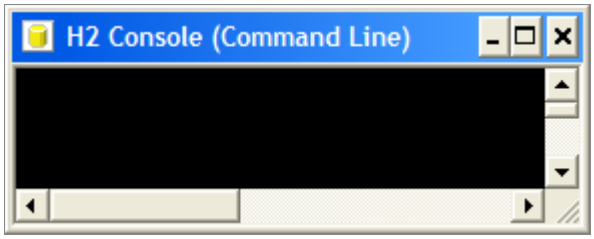
Installation

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)]:

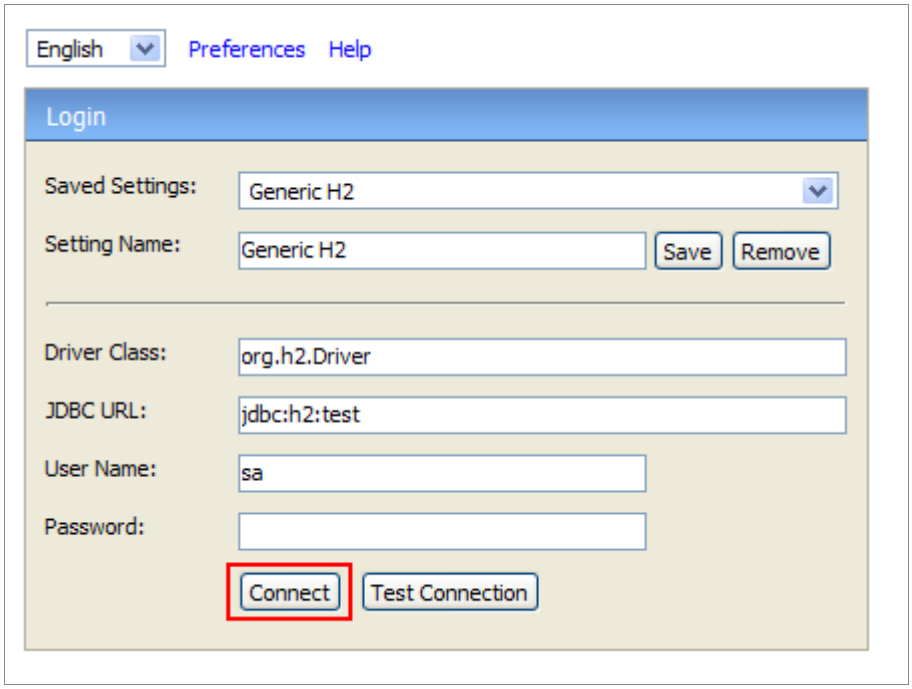
A new console window appears:



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

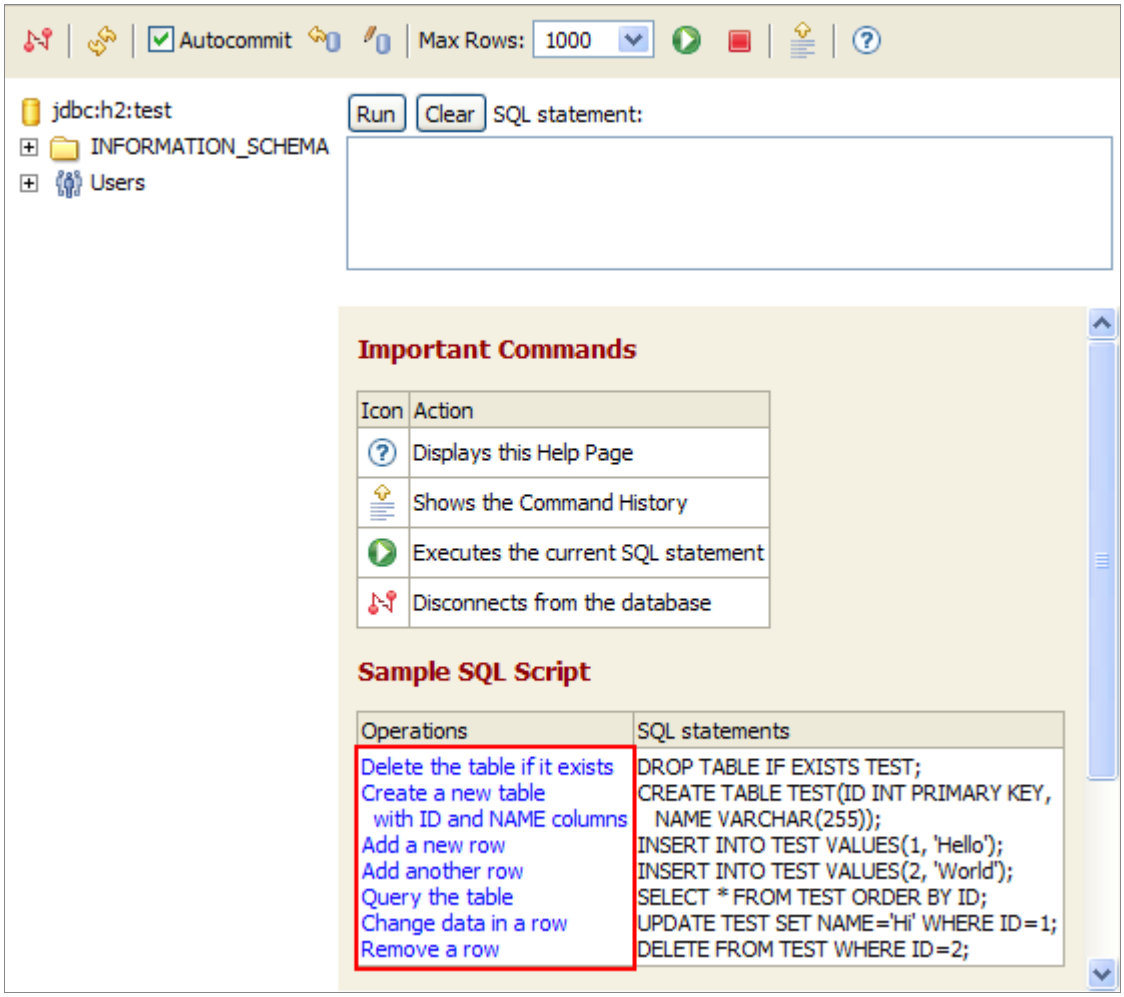
Select [Generic H2] and click [Connect]:



You are now logged in.

Sample

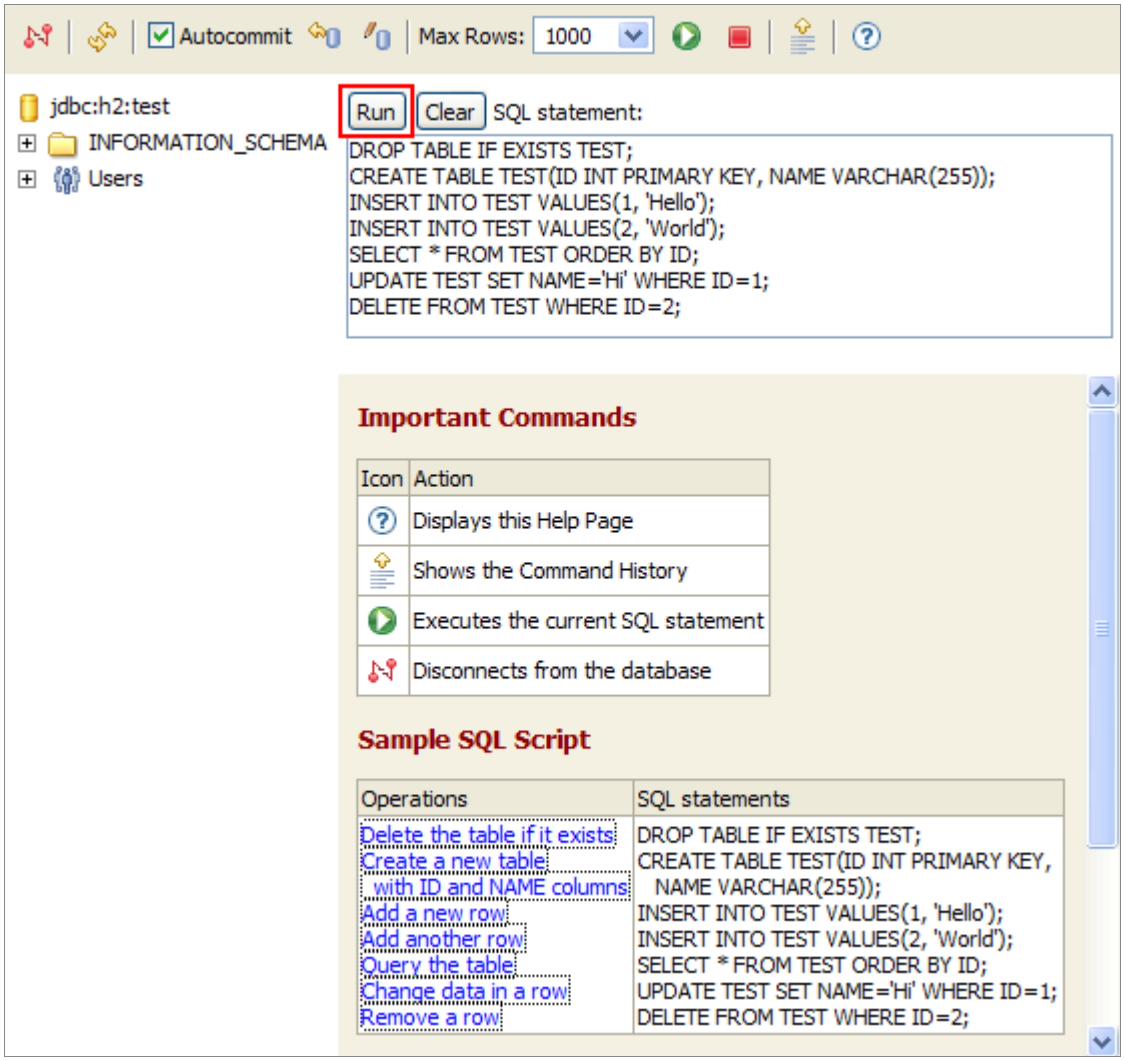
Click on the [Sample SQL Script]:



The SQL commands appear in the command area.

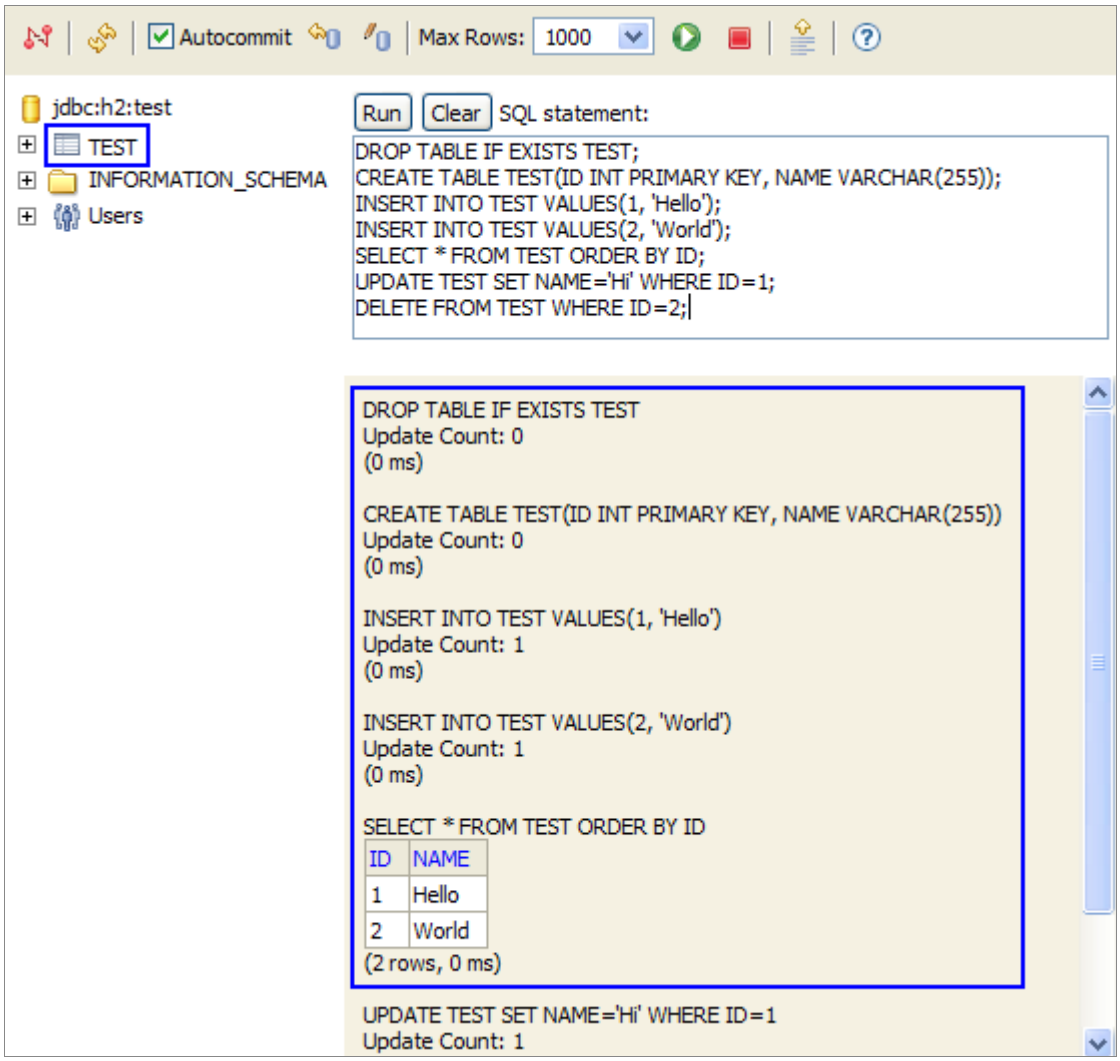
Execute

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
Supported Platforms
Installing the Software
Directory Structure

Requirements

To run this database, the following software stack is known to work. Other software most likely also works, but is not tested as much.

Database Engine

- Windows XP or Vista, Mac OS X, or Linux
- Sun JDK 1.5 or newer (using Retroweaver to convert to JDK 1.4 should work, but is not regularly tested)
- Recommended Windows file system: NTFS (FAT32 only supports files up to 4 GB)

H2 Console

- Mozilla Firefox

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 1.6, but it also works in many other operating systems and using other Java runtime environments. All major operating systems (Windows XP, Windows Vista, Windows 7, Mac OS, Ubuntu,...) are supported.

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:

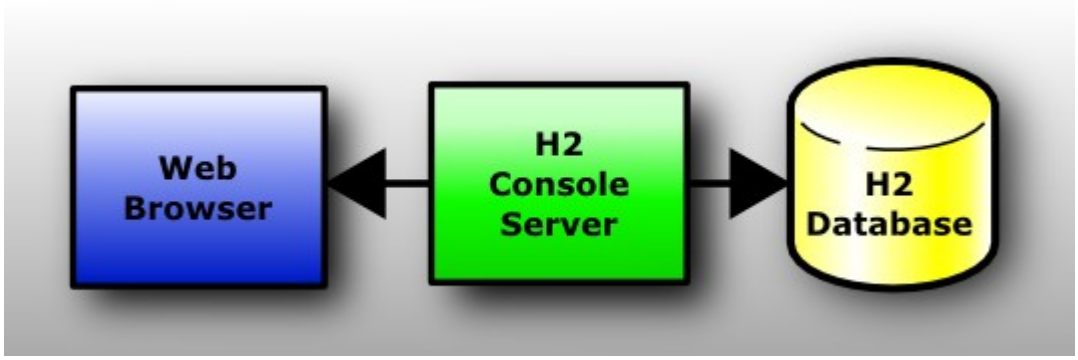
Directory	Contents
bin	JAR and batch files
docs	Documentation
docs/html	HTML pages
docs/javadoc	Javadoc files
ext	External dependencies (downloaded when building)
service	Tools to run the database as a Windows Service
src	Source files
src/docsrc	Documentation sources
src/installer	Installer, shell, and release build script
src/main	Database engine source code
src/test	Test source code
src/tools	Tools and database adapters source code

Tutorial

Starting and Using the H2 Console
Special H2 Console Syntax
Settings of the H2 Console
Connecting to a Database using JDBC
Creating New Databases
Using the Server
Using Hibernate
Using TopLink and Glassfish
Using EclipseLink
Using Apache ActiveMQ
Using H2 within NetBeans
Using H2 with jOOQ
Using Databases in Web Applications
Android
CSV (Comma Separated Values) Support
Upgrade, Backup, and Restore
Command Line Tools
The Shell Tool
Using OpenOffice Base
Java Web Start / JNLP
Using a Connection Pool
Fulltext Search
User-Defined Variables
Date and Time
Using Spring
Java Management Extension (JMX)

Starting and Using the H2 Console

The H2 Console application lets you access a database using a browser. 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 H2 Console:

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 -jar h2*.jar
```

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'.

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 default port of the H2 Console is already in use by another application, then a different port needs to be configured. The settings are stored in a properties file. For details, see Settings of the H2 Console. The relevant entry is webPort.

If no port is specified for the TCP and PG servers, each service will try to listen on its default port. If the default port is already in use, a random port is used.

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

To register additional JDBC drivers (MySQL, PostgreSQL, HSQLDB,...), add the jar file names to the environment variables H2DRIVERS or CLASSPATH. Example (Windows): to add the HSQLDB JDBC driver C:\Programs\hsqldb\lib\hsqldb.jar, set the environment variable H2DRIVERS to C:\Programs\hsqldb\lib\hsqldb.jar.

Multiple drivers can be set; entries need to be separated by ; (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. Type a SQL command in the query panel and click [Run]. The result appears just below the command.

Inserting Table Names or Column Names

To insert table and column names into the script, click on the item in the tree. If you click on a table while the query is empty, then SELECT * FROM ... is added. While typing a query, the table that was used is expanded in the tree. For example if you type SELECT * FROM TEST T WHERE T. then the table TEST is expanded.

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.

Special H2 Console Syntax

The H2 Console supports a few built-in commands. Those are interpreted within the H2 Console, so they work with any database. Built-in commands need to be at the beginning of a statement (before any remarks), otherwise they are not parsed correctly. If in doubt, add ; before the command.

Command(s)	Description
@autocommit_true; @autocommit_false;	Enable or disable autocommit.
@cancel;	Cancel the currently running statement.
@columns null null TEST; @index_info null null TEST; @tables; @tables null null TEST;	Call the corresponding DatabaseMetaData.get method. Patterns are case sensitive (usually identifiers are uppercase). For information about the parameters, see the Javadoc documentation. Missing parameters at the end of the line are set to null. The complete list of metadata commands is: @attributes, @best_row_identifier, @catalogs, @columns, @column_privileges, @cross_references, @exported_keys, @imported_keys, @index_info,

	@primary_keys, @procedures, @procedure_columns, @schemas, @super_tables, @super_types, @tables, @table_privileges, @table_types, @type_info, @udts, @version_columns
@edit select * from test;	Use an updatable result set.
@generated insert into test() values();	Show the result of Statement.getGeneratedKeys().
@history;	List the command history.
@info;	Display the result of various Connection and DatabaseMetaData methods.
@list select * from test;	Show the result set in list format (each column on its own line, with row numbers).
@loop 1000 select ?, ?/*rnd*/; @loop 1000 @statement select ?;	Run the statement this many times. Parameters (?) are set using a loop from 0 up to x - 1. Random values are used for each ?/*rnd*/. A Statement object is used instead of a PreparedStatement if @statement is used. Result sets are read until ResultSet.next() returns false. Timing information is printed.
@maxrows 20;	Set the maximum number of rows to display.
@memory;	Show the used and free memory. This will call System.gc().
@meta select 1;	List the ResultSetMetaData after running the query.
@parameter_meta select ?;	Show the result of the PreparedStatement.getParameterMetaData() calls. The statement is not executed.
@prof_start; call hash('SHA256', '', 1000000); @prof_stop;	Start/stop the built-in profiling tool. The top 3 stack traces of the statement(s) between start and stop are listed (if there are 3).
@prof_stop; @sleep 10; @prof_stop;	Sleep for a number of seconds. Used to profile a long running query or operation that is running in another session (but in the same process).
@transaction_isolation; @transaction_isolation 2;	Display (without parameters) or change (with parameters 1, 2, 4, 8) the transaction isolation level.

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. Supported settings are:

- webAllowOthers: allow other computers to connect.
- webPort: the port of the H2 Console
- webSSL: use encrypted (HTTPS) connections.

In addition to those settings, the properties of the last recently used connection are listed in the form <number>=<name>|<driver>|<url>|<user> using the escape character \. Example: 1=Generic H2 (Embedded)|org.h2.Driver|jdbc\:h2\:~/test|sa

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.

Auto-creating new database can be disabled, see Opening a Database Only if it Already Exists.

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). Please note that only the web server supports browser connections. The servers can be started in different ways, one is using the Server tool. Starting the server doesn't open a database - databases are opened as soon as a client connects.

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 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.

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. Please note that you can't connection with a web browser to this URL. You can only connect using a H2 client (over JDBC).

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 some old versions of Hibernate was buggy. A patch for Hibernate has been submitted and is now applied. You can rename it to H2Dialect.java and include this as a patch in your application, or upgrade to a version of Hibernate where this is fixed.

When using Hibernate, try to use the H2Dialect if possible. When using the H2Dialect, compatibility modes such as MODE=MySQL are not supported. When using such a compatibility mode, use the Hibernate dialect for the corresponding database instead of the H2Dialect; but please note H2 does not support all features of all databases.

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.

To use H2 within Glassfish, copy the h2*.jar to the directory glassfish/glassfish/lib.

Using EclipseLink

To use H2 in EclipseLink, use the platform class org.eclipse.persistence.platform.database.H2Platform. If this platform is not available in your version of EclipseLink, you can use the OraclePlatform instead in many case. See also H2Platform.

Using Apache ActiveMQ

When using H2 as the backend database for Apache ActiveMQ, please use the TransactDatabaseLocker instead of the default locking mechanism. Otherwise the database file will grow without bounds. The problem is that the default locking mechanism uses an uncommitted UPDATE transaction, which keeps the transaction log from shrinking (causes the database file to grow). Instead of using an UPDATE statement, the TransactDatabaseLocker uses SELECT ... FOR UPDATE which is not problematic. To use it, change the ApacheMQ configuration element <jdbcPersistenceAdapter> element, property databaseLocker="org.apache.activemq.store.jdbc.adapter.TransactDatabaseLocker". However, using the MVCC mode will again result in the same problem. Therefore, please do not use the MVCC mode in this case. Another (more dangerous) solution is to set useDatabaseLock to false.

Using H2 within NetBeans

The project H2 Database Engine Support For NetBeans allows you to start and stop the H2 server from within the IDE.

There is a known issue when using the Netbeans SQL Execution Window: before executing a query, another query in the form SELECT COUNT(*) FROM <query> is run. This is a problem for queries that modify state, such as SELECT SEQ.NEXTVAL. In this case, two sequence values are allocated instead of just one.

Using H2 with jOOQ

jOOQ adds a thin layer on top of JDBC, allowing for type-safe SQL construction, including advanced SQL, stored procedures and advanced data types. jOOQ takes your database schema as a base for code generation. If this is your example schema:

```
CREATE TABLE USER (ID INT, NAME VARCHAR(50));
```

then run the jOOQ code generator on the command line using this command:

```
java -cp jooq.jar;jooq-meta.jar;jooq-codegen.jar;h2-1.3.158.jar;.
org.jooq.util.GenerationTool /codegen.xml
```

...where codegen.xml is on the classpath and contains this information

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<configuration xmlns="http://www.jooq.org/xsd/jooq-codegen-2.3.0.xsd">
    <jdbc>
        <driver>org.h2.Driver</driver>
        <url>jdbc:h2:~/test</url>
        <user>sa</user>
        <password></password>
    </jdbc>
    <generator>
        <name>org.jooq.util.DefaultGenerator</name>
        <database>
            <name>org.jooq.util.h2.H2Database</name>
            <includes>.*</includes>
            <excludes></excludes>
            <inputSchema>PUBLIC</inputSchema>
        </database>
        <generate></generate>
        <target>
            <packageName>org.jooq.h2.generated</packageName>
            <directory>./src</directory>
        </target>
    </generator>
</configuration>
```

Using the generated source, you can query the database as follows:

```
Factory create = new H2Factory(connection);
Result<UserRecord> result =
create.selectFrom(USER)
    .where(NAME.like("Johnny%"))
    .orderBy(ID)
    .fetch();
```

See more details on jOOQ Homepage and in the jOOQ Tutorial

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 this tool 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");
```

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>
    <!--
    <init-param>
        <param-name>webAllowOthers</param-name>
        <param-value></param-value>
    </init-param>
    <init-param>
        <param-name>trace</param-name>
        <param-value></param-value>
    </init-param>
    -->
    <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
```

Android

You can use this database on an Android device (using the Dalvik VM) instead of or in addition to SQLite. So far, only very few tests and benchmarks were run, but it seems that performance is similar to SQLite, except for opening and closing a database, which is not yet optimized in H2 (H2 takes about 0.2 seconds, and SQLite about 0.02 seconds). Read operations seem to be a bit faster than SQLite, and write operations seem to be slower. So far, only very few tests have been run, and everything seems to work as expected. Fulltext search was not yet tested, however the native fulltext search should work.

Reasons to use H2 instead of SQLite are:

- Full Unicode support including UPPER() and LOWER().
- Streaming API for BLOB and CLOB data.
- Fulltext search.
- Multiple connections.
- User defined functions and triggers.
- Database file encryption.
- Reading and writing CSV files (this feature can be used out side the database as well).
- Referential integrity and check constraints.
- Better data type and SQL support.
- In-memory databases, read-only databases, linked tables.
- Better compatibility with other databases which simplifies porting applications.
- Possibly better performance (so far for read operations).
- Server mode (accessing a database on a different machine over TCP/IP).

Currently only the JDBC API is supported (it is planned to support the Android database API in future releases). Both the regular H2 jar file and the smaller h2small-*.jar can be used. To create the smaller jar file, run the command ./build.sh jarSmall (Linux / Mac OS) or build.bat jarSmall (Windows).

The database files needs to be stored in a place that is accessible for the application. Example:

```
String url = "jdbc:h2:/data/data/" +
    "com.example.hello" +
    "/data/hello" +
    ";FILE_LOCK=FS" +
    ";PAGE_SIZE=1024" +
    ";CACHE_SIZE=8192";
Class.forName("org.h2.Driver");
conn = DriverManager.getConnection(url);
```

```
...
```

Limitations: Using a connection pool is currently not supported, because the required javax.sql. classes are not available on Android.

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.

Reading a CSV File from Within a Database

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

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

Please note for performance reason, CSVREAD should not be used inside a join. Instead, import the data first (possibly into a temporary table), create the required indexes if necessary, and then query this table.

Importing Data from a CSV File

A fast way to load or import data (sometimes called 'bulk load') from a CSV file is to combine table creation with import. Optionally, the column names and data types can be set when creating the table. Another option is to use INSERT INTO ... SELECT.

```
CREATE TABLE TEST AS SELECT * FROM CSVREAD('test.csv');
CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255))
    AS SELECT * FROM CSVREAD('test.csv');
```

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');
```

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");
        new Csv().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 = new Csv().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

The recommended way to backup a database is to create a compressed SQL script file. This will result in a small, human readable, and database version independent backup. Creating the script will also verify the checksums of the database file. The Script tool is ran as follows:

```
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 the database file. However, the contents of this file are not human readable.

The resulting backup is transactionally consistent, meaning the consistency and atomicity rules apply.

```
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 by copying the database files while the database is running is not supported, except if the file systems support creating snapshots. With other file systems, 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.

The Shell Tool

The Shell tool is a simple interactive command line tool. To start it, type:

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

You will be asked for a database URL, JDBC driver, user name, and password. The connection setting can also be set as command line parameters. After connecting, you will get the list of options. The built-in commands don't need to end with a semicolon, but SQL statements are only executed if the line ends with a semicolon ;. This allows to enter multi-line statements:

```
sql> select * from test
...> where id = 0;
```

By default, results are printed as a table. For results with many column, consider using the list mode:

```
sql> list
Result list mode is now on
sql> select * from test;
ID  : 1
NAME: Hello

ID  : 2
NAME: World
(2 rows, 0 ms)
```

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 name, TEST is the table name. The list of column names (comma 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. Currently Apache Lucene version 2.x is used by default for H2 version 1.2.x, and Lucene version 3.x is used by default for H2 version 1.3.x. 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 name, TEST is the table name. The list of column names (comma 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);
```

The Lucene fulltext search supports searching in specific column only. Column names must be uppercase (except if the original columns are double quoted). For column names starting with an underscore (_), another underscore needs to be added. Example:

```
CREATE ALIAS IF NOT EXISTS FTL_INIT FOR "org.h2.fulltext.FullTextLucene.init";
CALL FTL_INIT();
DROP TABLE IF EXISTS TEST;
CREATE TABLE TEST(ID INT PRIMARY KEY, FIRST_NAME VARCHAR, LAST_NAME VARCHAR);
CALL FTL_CREATE_INDEX('PUBLIC', 'TEST', NULL);
INSERT INTO TEST VALUES(1, 'John', 'Wayne');
INSERT INTO TEST VALUES(2, 'Elton', 'John');
SELECT * FROM FTL_SEARCH_DATA('John', 0, 0);
SELECT * FROM FTL_SEARCH_DATA('LAST_NAME:John', 0, 0);
CALL FTL_DROP_ALL();
```

The Lucene fulltext search implementation is not synchronized internally. If you update the database and query the fulltext search concurrently (directly using the Java API of H2 or Lucene itself), you need to ensure operations are properly synchronized. If this is not the case, you may get exceptions such as org.apache.lucene.store.AlreadyClosedException: this IndexReader is closed.

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. Rolling back a transaction does not affect the value of a user-defined variable.

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 without time zone information. If the database is opened using another system time zone, the date and time will be the same. That means if you store the value '2000-01-01 12:00:00' in one time zone, then close the database and open the database again in a different time zone, you will also get '2000-01-01 12:00:00'. Please note that changing the time zone after the H2 driver is loaded is not supported.

Using Spring

Using the TCP Server

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.

Error Code Incompatibility

There is an incompatibility with the Spring JdbcTemplate and H2 version 1.3.154 and newer, because of a change in the error code. This will cause the JdbcTemplate to not detect a duplicate key condition, and so a DataIntegrityViolationException is thrown instead of DuplicateKeyException. See also the issue SPR-8235. The workaround is to add the following XML file to the root of the classpath:

```
<beans
   xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
      "http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd"
   >
   <import resource="classpath:org/springframework/jdbc/support/sql-error-codes.xml"/>
   <bean id = "H2" class="org.springframework.jdbc.support.SQLErrorCodes">
      <property name="badSqlGrammarCodes">
         <value>
            42000,42001,42101,42102,42111,42112,42121,42122,42132
         </value>
      </property>
      <property name="duplicateKeyCodes">
         <value>23001,23505</value>
      </property>
      <property name="dataIntegrityViolationCodes">
         <value>22003,22012,22025,23000</value>
      </property>
      <property name="dataAccessResourceFailureCodes">
         <value>90046,90100,90117,90121,90126</value>
```

```
      </property>
      <property name="cannotAcquireLockCodes">
         <value>50200</value>
      </property>
   </bean>
</beans>
```

Java Management Extension (JMX)

Management over JMX is supported, but not enabled by default. To enable JMX, append ;JMX=TRUE to the database URL when opening the database. Various tools support JMX, one such tool is the jconsole. When opening the jconsole, connect to the process where the database is open (when using the server mode, you need to connect to the server process). Then go to the MBeans section. Under org.h2 you will find one entry per database. The object name of the entry is the database short name, plus the path (each colon is replaced with an underscore character).

The following attributes and operations are supported:

- CacheSize: the cache size currently in use in KB.
- CacheSizeMax (read/write): the maximum cache size in KB.
- Exclusive: whether this database is open in exclusive mode or not.
- FileReadCount: the number of file read operations since the database was opened.
- FileSize: the file size in KB.
- FileWriteCount: the number of file write operations since the database was opened.
- FileWriteCountTotal: the number of file write operations since the database was created.
- LogMode (read/write): the current transaction log mode. See SET LOG for details.
- Mode: the compatibility mode (REGULAR if no compatibility mode is used).
- MultiThreaded: true if multi-threaded is enabled.
- Mvcc: true if MVCC is enabled.
- ReadOnly: true if the database is read-only.
- TraceLevel (read/write): the file trace level.
- Version: the database version in use.
- listSettings: list the database settings.
- listSessions: list the open sessions, including currently executing statement (if any) and locked tables (if any).

To enable JMX, you may need to set the system properties com.sun.management.jmxremote and com.sun.management.jmxremote.port as required by the JVM.

Features

Feature List
Comparison to Other Database Engines
H2 in Use
Connection Modes
Database URL Overview
Connecting to an Embedded (Local) Database
In-Memory Databases
Database Files Encryption
Database File Locking
Opening a Database Only if it Already Exists
Closing a Database
Ignore Unknown Settings
Changing Other Settings when Opening a Connection
Custom File Access Mode
Multiple Connections
Database File Layout
Logging and Recovery
Compatibility
Auto-Reconnect
Automatic Mixed Mode
Page Size
Using the Trace Options
Using Other Logging APIs
Read Only Databases
Read Only Databases in Zip or Jar File
Computed Columns / Function Based Index
Multi-Dimensional Indexes
User-Defined Functions and Stored Procedures
Triggers
Compacting a Database
Cache Settings

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, including support for the ICU4J library
- Support for users and 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-128 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
- Supports the EXPLAIN PLAN statement; 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

This comparison is based on H2 1.3, Apache Derby version 10.8, HSQLDB 2.2, MySQL 5.5, PostgreSQL 9.0.

Feature	H2	Derby	HSQLDB	MySQL	PostgreSQL
Pure Java	Yes	Yes	Yes	No	No
Embedded Mode (Java)	Yes	Yes	Yes	No	No
In-Memory Mode	Yes	Yes	Yes	No	No
Explain Plan	Yes	Yes *12	Yes	Yes	Yes
Built-in Clustering / Replication	Yes	Yes	No	Yes	Yes
Encrypted Database	Yes	Yes *10	Yes *10	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
Domains (User-Defined Types)	Yes	No	Yes	Yes	Yes
Files per Database	Few	Many	Few	Many	Many
Row Level Locking	Yes *9	Yes	Yes *9	Yes	Yes
Multi Version Concurrency	Yes	No	Yes	Yes	Yes
Multi-Threaded Statement Processing	No *11	Yes	Yes	Yes	Yes
Role Based Security	Yes	Yes *3	Yes	Yes	Yes

Updatable Result Sets	Yes	Yes *7	Yes	Yes	Yes
Sequences	Yes	Yes	Yes	No	Yes
Limit and Offset	Yes	Yes *13	Yes	Yes	Yes
Window Functions	No *15	No *15	No	No	Yes
Temporary Tables	Yes	Yes *4	Yes	Yes	Yes
Information Schema	Yes	No *8	Yes	Yes	Yes
Computed Columns	Yes	Yes	Yes	No	Yes *6
Case Insensitive Columns	Yes	Yes *14	Yes	Yes	Yes *6
Custom Aggregate Functions	Yes	No	Yes	Yes	Yes
CLOB/BLOB Compression	Yes	No	No	No	Yes
Footprint (jar/dll size)	~1 MB *5	~2 MB	~1 MB	~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 doesn't support standard compliant information schema tables.
*9 When using MVCC (multi version concurrency).
*10 Derby and HSQLDB don't hide data patterns well.
*11 The MULTI_THREADED option is not enabled by default, and not yet supported when using MVCC.
*12 Derby doesn't support the EXPLAIN statement, but it supports runtime statistics and retrieving statement execution plans.
*13 Derby doesn't support the syntax LIMIT .. [OFFSET ..], however it supports FETCH FIRST .. ROW[S] ONLY.
*14 Using collations. *15 Derby and H2 support ROW_NUMBER() OVER().

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.

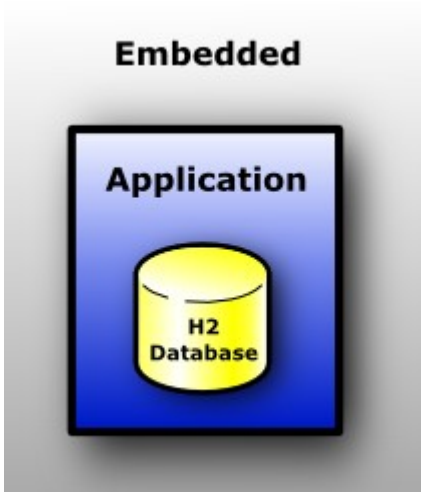
Connection Modes

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

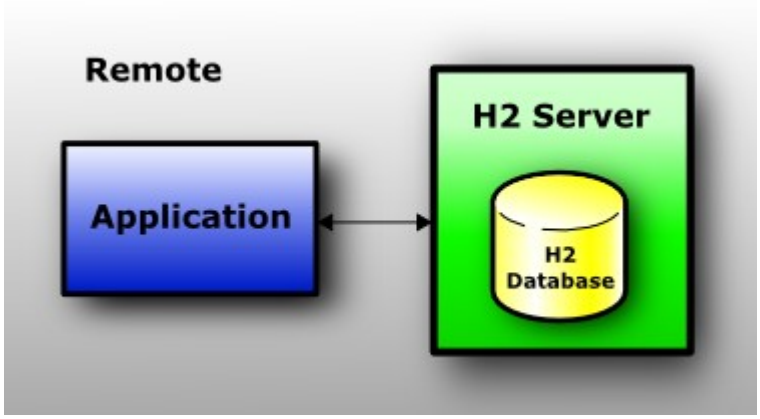
In embedded mode, an application opens a database from within the same JVM using JDBC. This is the fastest and easiest connection mode. The disadvantage is that a database may only be open in one virtual machine (and class loader) at any time. As in all modes, both persistent and in-memory databases 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, by connecting to this server. Internally, the server process opens the database(s) in embedded mode.

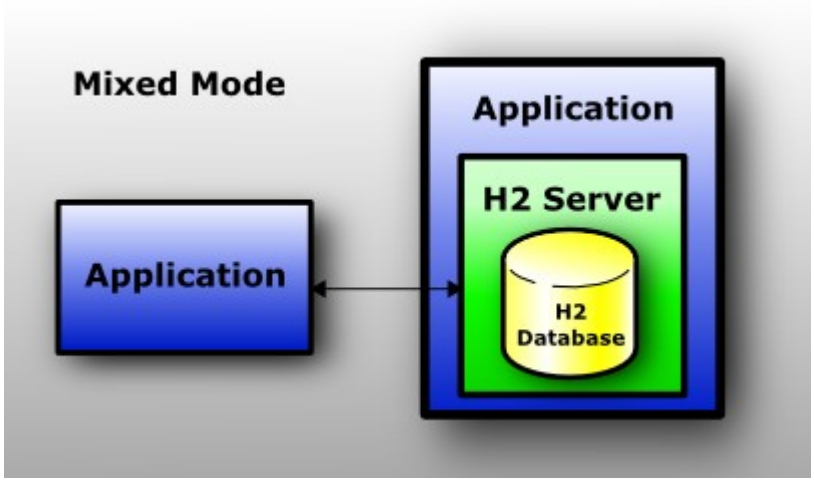
The server mode is slower than the embedded mode, because all data is transferred over TCP/IP. As in all modes, both persistent and in-memory databases are supported. There is no limit on the number of database open concurrently per server, or on the number of open connections.



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 jdbc:h2:tcp://localhost/mem:test
Server mode (remote connections) using SSL/TLS	jdbc:h2:ssl://<server>[:<port>]/<databaseName> jdbc:h2:ssl://localhost:8085/~/sample;
Using encrypted files	jdbc:h2:<url>;CIPHER=[AES|XTEA] jdbc:h2:ssl://localhost/~/test;CIPHER=AES jdbc:h2:file:~/secure;CIPHER=XTEA
File locking methods	jdbc:h2:<url>;FILE_LOCK={FILE|SOCKET|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
Execute SQL on connection	jdbc:h2:<url>;INIT=RUNSCRIPT FROM '~/create.sql' jdbc:h2:file:~/sample;INIT=RUNSCRIPT FROM '~/create.sql'\;RUNSCRIPT FROM '~/populate.sql'
User name and/or password	jdbc:h2:<url>[;USER=<username>][;PASSWORD=<value>] jdbc:h2:file:~/sample;USER=sa;PASSWORD=123
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_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

Page size	jdbc:h2:<url>;PAGE_SIZE=512
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.

In-Memory 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 in-memory mode, where the data is not persisted.

In some cases, only one connection to a in-memory 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 in-memory database are required. In this case, the database URL must include a name. Example: jdbc:h2:mem:db1. Accessing the same database using this URL only works within the same virtual machine and class loader environment.

To access an in-memory database from another process or from another computer, you need to start a TCP server in the same process as the in-memory database was created. The other processes then need to access the database over TCP/IP or SSL/TLS, using a database URL such as: 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. Failing to do so will result in a corrupted database. Using the method NO 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.

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";
```

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";
```

Execute SQL on Connection

Sometimes, particularly for in-memory databases, it is useful to be able to execute DDL or DML commands automatically when a client connects to a database. This functionality is enabled via the INIT property. Note that multiple commands may be passed to INIT, but the semicolon delimiter must be escaped, as in the example below.

```
String url = "jdbc:h2:mem;INIT=runscript from '~/create.sql'\\;runscript from '~/populate.sql'";
```

Please note the double backslash is only required in a Java or properties file. In a GUI, or in an XML file, only one backslash is required:

```
<property name="url" value=
"jdbc:h2:mem:test;INIT=create schema if not exists test\;runscript from '~/sql/populate.sql';DB_CLOSE_DELAY=-1"
/>
```

Backslashes within the init script (for example within a runscript statement, to specify the folder names in Windows) need to be escaped as well (using a second backslash). It might be simpler to avoid backslashes in folder names for this reason; use forward slashes instead.

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 the database file 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 database file is not read-only, use ACCESS_MODE_DATA=r. Also supported are rws and rwd. This setting must be specified in the database URL:

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

An application should normally use one connection per thread. This database synchronizes access to the same connection, but other databases may not do this.

Locking, Lock-Timeout, Deadlocks

Unless multi-version concurrency is used, 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.

Avoiding Deadlocks

To avoid deadlocks, ensure that all transactions lock the tables in the same order (for example in alphabetical order), and avoid upgrading read locks to write locks. Both can be achieved using explicitly locking tables using SELECT ... FOR UPDATE.

Database File Layout

The following files are created for persistent databases:

File Name	Description	Number of Files
test.h2.db	Database file. Contains the transaction log, indexes, and data for all tables. Format: <database>.h2.db	1 per database
test.lock.db	Database lock file. Automatically (re-)created while the database is in use. Format: <database>.lock.db	1 per database (only if in use)
test.trace.db	Trace file (if the trace option is enabled). Contains trace information. Format: <database>.trace.db Renamed to <database>.trace.db.old is too big.	0 or 1 per database
test.lobs.db/*	Directory containing one file for each BLOB or CLOB value larger than a certain size. Format: <id>.t<tableId>.lob.db	1 per large object
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 of the same database start with the same name and the respective extensions are unchanged).

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.

To backup data while the database is running, the SQL commands SCRIPT and BACKUP can be used.

Logging and Recovery

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

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. However, only a small subset of the differences between databases are implemented in this way. 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.
- Concatenating NULL with another value results in the other value.
- Support the pseudo-table SYSIBM.SYSDUMMY1.

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.
- Concatenating NULL with another value results in the other value.
- Support the pseudo-table SYSIBM.SYSDUMMY1.

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.
- For unique indexes, NULL is distinct. That means only one row with NULL in one of the columns is allowed.
- Text can be concatenated using '+'.

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.
- Concatenating NULL with another value results in the other value.
- Text can be concatenated using '+'.

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 using INDEX(..) or KEY(..). Example: create table test(id int primary key, name varchar(255), key idx_name(name));
- 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.
- Concatenating NULL with another value results in the other value.

Text comparison in MySQL is case insensitive by default, while in H2 it is case sensitive (as in most other databases). H2 does support case insensitive text comparison, but it needs to be set separately, using SET IGNORECASE TRUE. This affects comparison using =, LIKE, REGEXP.

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.
- Concatenating NULL with another value results in the other value.

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.
- 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.
- LOG(x) is base 10 in this mode.

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. To enable this mode, append ;AUTO_RECONNECT=TRUE to the database URL.

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.

If another connection uses the database in exclusive mode (enabled using SET EXCLUSIVE 1 or SET EXCLUSIVE 2), then this connection will try to re-connect until the exclusive mode ends.

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 independent of whether the database is already open or not. This feature doesn't work with in-memory databases. Example database URL:

```
jdbc:h2:/data/test;AUTO_SERVER=TRUE
```

Use the same URL for all connections to this database. Internally, when using this mode, the first connection to the database is made in embedded mode, and additionally a server is started internally (as a daemon thread). If the database is already open in another process, the server mode is used automatically. The IP address and port of the server are stored in the file .lock.db, that's why in-memory databases can't be supported.

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 (this may not be a problem if you don't disable autocommit). 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");
```

When using this feature, by default the server uses any free TCP port. The port can be set manually using AUTO_SERVER_PORT=9090.

Page Size

The page size for new databases is 2 KB (2048), unless the page size is set explicitly in the database URL using PAGE_SIZE= when the database is created. The page size of existing databases can not be changed, so this property needs to be set when the database is created.

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 to a file
- Support for trace levels OFF, ERROR, INFO, 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 such file exists, it is deleted. To limit the size to a certain number of megabytes, use SET TRACE_MAX_FILE_SIZE mb. 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 simplifies reproducing problems. 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. The logger name is h2database. 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. Then, make the database file read-only. 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().

Using the Custom Access Mode r the database can also be opened in read-only mode, even if the database file is not read only.

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. The database must not have pending changes, that means you need to close all connections to the database first. To speed up opening the read-only database and running queries, the database should be closed using SHUTDOWN DEFRAG. If you are using a database named test, an easy way to create a zip file is using the Backup tool. You can start the tool from the command line, or from within the H2 Console (Tools - Backup). Please note that the database must be closed when the backup is created. Therefore, the SQL statement BACKUP TO can not be used.

When the zip file is created, you can 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.

If the database is larger than a few megabytes, performance is much better if the database file is split into multiple smaller files, because random access in compressed files is not possible. See also the sample application ReadOnlyDatabaseInZip.

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

A computed column is a column whose value is calculated before storing. The formula is evaluated when the row is inserted, and re-evaluated every time the row is updated. One use case is to automatically update the last-modification time:

```
CREATE TABLE TEST(ID INT, NAME VARCHAR, LAST_MOD TIMESTAMP AS NOW());
```

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.

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. A function can be defined using source code, or as a reference to a compiled class that is available in the classpath. By default, the function aliases are stored in the current schema.

Referencing a Compiled Method

When referencing a method, the class must already be compiled and included in the classpath where the database is running. Only static Java methods are supported; both the class and the method must be public. Example Java class:

```
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 ... FOR:

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

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

Declaring Functions as Source Code

When defining a function alias with source code, the database tries to compile the source code using the Sun Java compiler (the class com.sun.tools.javac.Main) if the tools.jar is in the classpath. If not, javac is run as a separate process. Only the source code is stored in the database; the class is compiled each time the database is re-opened. Source code is usually passed as dollar quoted text to avoid escaping problems, however single quotes can be used as well. Example:

```
CREATE ALIAS NEXT_PRIME AS $$
String nextPrime(String value) {
    return new BigInteger(value).nextProbablePrime().toString();
}
$$;
```

By default, the three packages java.util, java.math, java.sql are imported. The method name (nextPrime in the example above) is ignored. Method overloading is not supported when declaring functions as source code, that means only one method may be declared for an alias. If different import statements are required, they must be declared at the beginning and separated with the tag @CODE:

```
CREATE ALIAS IP_ADDRESS AS $$
import java.net.*;
@CODE
String ipAddress(String host) throws Exception {
    return InetAddress.getByName(host).getHostAddress();
}
$$;
```

The following template is used to create a complete Java class:

```
package org.h2.dynamic;
< import statements before the tag @CODE; if not set:
import java.util.*;
import java.math.*;
import java.sql.*;
>
public class <aliasName> {
```

```
    public static <sourceCode>
}
```

Method Overloading

Multiple methods may be bound to a SQL function if the class is already compiled and included in the classpath. Each Java method must have a different number of arguments. Method overloading is not supported when declaring functions as source code.

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.

SQL types are mapped to Java classes and vice-versa as in the JDBC API. For details, see Data Types. There are a few 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. Objects of type org.h2.value.Value (the internal value class) are passed through without conversion.

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. SQLException are directly re-thrown to the calling application; all other exceptions are first converted to a SQLException.

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. The trigger class must be available in the classpath of the database engine (when using the server mode, it must be in the classpath of the server).

```
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) {
        // initialize the trigger object is necessary
    }

    public void fire(Connection conn,
            Object[] oldRow, Object[] newRow)
            throws SQLException {
        // the trigger is fired
    }

    public void close() {
        // the database is closed
    }

    public void remove() {
        // the trigger was dropped
    }

}
```

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.

As an alternative to implementing the Trigger interface, an application can extend the abstract class org.h2.tools.TriggerAdapter. This will allows to use the ResultSet interface within trigger implementations. In this case, only the fire method needs to be implemented:

```
import org.h2.tools.TriggerAdapter;
...
public class TriggerSample implements TriggerAdapter {

    public void fire(Connection conn, ResultSet oldRow, ResultSet newRow)
            throws SQLException {
        // the trigger is fired
    }

}
```

Compacting a Database

Empty space in the database file re-used automatically. When closing the database, the database is automatically compacted for up to 200 milliseconds by default. To compact more, use the SQL statement SHUTDOWN COMPACT. However re-creating the database may further reduce the database size because this will re-build the indexes. 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 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. The size of the cache, as represented by CACHE_SIZE is measured in KB, with each KB being 1024 bytes. This setting has no effect for in-memory databases. For persistent databases, the setting is stored in the database and re-used when the database is opened the next time. However, when opening an existing database, the cache size is set to at most half the amount of memory available for the virtual machine (Runtime.getRuntime().maxMemory()), even if the cache size setting stored in the database is larger; however the setting stored in the database is kept. Setting the cache size in the database URL or explicitly using SET CACHE_SIZE overrides this value (even if larger than the physical memory). To get the current used maximum cache size, use the query SELECT * FROM INFORMATION_SCHEMA.SETTINGS WHERE NAME = 'info.CACHE_MAX_SIZE'

An experimental scan-resistant cache algorithm "Two Queue" (2Q) is available. To enable it, append ;CACHE_TYPE=TQ to the database URL. The cache might not actually improve performance. If you plan to use it, please run your own test cases first.

Also included is an experimental 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. The cache might not actually improve performance. If you plan to use it, please run your own test cases first.

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.

Performance

Performance Comparison
PolePosition Benchmark
Database Performance Tuning
Using the Built-In Profiler
Application Profiling
Database Profiling
Statement Execution Plans
How Data is Stored and How Indexes Work
Fast Database Import

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	241	431	1027
Simple: Query (random)	ms	193	267	748
Simple: Query (sequential)	ms	89	179	658
Simple: Update (random)	ms	406	772	12175
Simple: Delete (sequential)	ms	155	266	6281
Simple: Memory Usage	MB	7	13	16
BenchA: Init	ms	200	251	1075
BenchA: Transactions	ms	1071	1458	8142
BenchA: Memory Usage	MB	8	14	12
BenchB: Init	ms	787	1584	4163
BenchB: Transactions	ms	465	875	2744
BenchB: Memory Usage	MB	17	13	10
BenchC: Init	ms	348	225	922
BenchC: Transactions	ms	1382	865	3527
BenchC: Memory Usage	MB	12	20	11
Executed statements	#	322929	322929	322929
Total time	ms	5337	7173	41462
Statements per second	#	60507	45020	7788

Client-Server

Test Case	Unit	H2	HSQLDB	Derby	PostgreSQL	MySQL
Simple: Init	ms	1715	2096	3008	3093	3084
Simple: Query (random)	ms	2615	2119	4450	3201	3313
Simple: Query (sequential)	ms	2531	1944	4019	3163	3295
Simple: Update (random)	ms	1862	2486	13929	4404	4391
Simple: Delete (sequential)	ms	778	1118	7032	1682	1882
Simple: Memory Usage	MB	8	14	18	1	2
BenchA: Init	ms	1264	1686	2734	2867	3225
BenchA: Transactions	ms	5998	6829	14323	11491	10571
BenchA: Memory Usage	MB	9	18	14	1	2
BenchB: Init	ms	5571	7553	11636	12226	12553
BenchB: Transactions	ms	1931	3417	3435	2407	2149
BenchB: Memory Usage	MB	18	16	13	2	2
BenchC: Init	ms	1333	968	1769	1693	2645

BenchC: Transactions	ms	6508	4330	7910	7564	6108
BenchC: Memory Usage	MB	12	21	14	2	2
Executed statements	#	322929	322929	322929	322929	322929
Total time	ms	32106	34546	74245	53791	53216
Statements per second	#	10058	9347	4349	6003	6068

Benchmark Results and Comments

H2

Version 1.2.137 (2010-06-06) was used for the test. For most operations, the performance of H2 is about the same as for HSQLDB. One situation where 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.

HSQLDB

Version 2.0.0 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).

Derby

Version 10.6.1.0 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 calls FileChannel.force(false), but only twice per log file (not on each commit). Disabling this call improves performance for Derby by about 2%. Unlike H2, Derby does not call FileDescriptor.sync() on each checkpoint. 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 8.4.4 was used for the test. The following options where changed in postgresql.conf: fsync = off, commit_delay = 1000. PostgreSQL is run in server mode. The memory usage number is incorrect, because only the memory usage of the JDBC driver is measured.

MySQL

Version 5.1.47-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 / my.cnf file) was set to 0. Otherwise (and by default), MySQL is 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 / my.cnf, 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 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 Mac OS X 10.6. No virus scanner was used, and disk indexing was disabled. The VM used is Sun JDK 1.6.

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 the method 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. This test was not run for a longer time, so please be aware that the results below are for older database versions (H2 version 1.1, HSQLDB 1.8, Java 1.4).

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 the file 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.

Database Performance Tuning

Keep Connections Open or Use a Connection Pool

If your application opens and closes connections a lot (for example, for each request), you should consider using a connection pool. Opening a connection using DriverManager.getConnection is specially slow if the database is closed. By default the database is closed if the last connection is closed.

If you open and close connections a lot but don't want to use a connection pool, consider keeping a 'sentinel' connection open for as long as the application runs, or use delayed database closing. See also Closing a database.

Use a Modern JVM

Newer JVMs are faster. Upgrading to the latest version of your JVM can provide a "free" boost to performance. 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 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. Ensure files ending with .db are not scanned.

Using the Trace Options

If the 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, DELETE. 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.

How Data is Stored Internally

For persistent databases, if a table is created with a single column primary key of type BIGINT, INT, SMALLINT, TINYINT, then the data of the table is organized in this way. This is sometimes also called a "clustered index" or "index organized table".

H2 internally stores table data and indexes in the form of b-trees. Each b-tree stores entries as a list of unique keys (one or more columns) and data (zero or more columns). The table data is always organized in the form of a "data b-tree" with a single column key of type long. If a single column primary key of type BIGINT, INT, SMALLINT, TINYINT is specified when creating the table (or just after creating the table, but before inserting any rows), then this column is used as the key of the data b-tree. If no primary key has been specified, if the primary key column is of another data type, or if the primary key contains more than one column, then a hidden auto-increment column of type BIGINT is added to the table, which is used as the key for the data b-tree. All other columns of the table are stored within the data area of this data b-tree (except for large BLOB, CLOB columns, which are stored externally).

For each additional index, one new "index b-tree" is created. The key of this b-tree consists of the indexed columns, plus the key of the data b-tree. If a primary key is created after the table has been created, or if the primary key contains multiple column, or if the primary key is not of the data types listed above, then the primary key is stored in a new index b-tree.

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 join can be used. If the same table is joined multiple times, for each join only one index is used (the same index could be used for both joins, or each join could use a different index). 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,...).

Use Prepared Statements

If possible, use prepared statements with parameters.

Prepared Statements and IN(...)

Avoid generating SQL statements with a variable size IN(...) list. Instead, use a prepared statement with arrays as in the following example:

```
PreparedStatement prep = conn.prepareStatement(
    "SELECT * FROM TABLE(X INT=?) T INNER JOIN TEST ON T.X=TEST.ID");
prep.setObject(1, new Object[] { "1", "2" });
ResultSet rs = prep.executeQuery();
```

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.

Sorted Insert Optimization

To reduce disk space usage and speed up table creation, an optimization for sorted inserts is available. When used, b-tree pages are split at the insertion point. To use this optimization, add SORTED before the SELECT statement:

```
CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR) AS
    SORTED SELECT X, SPACE(100) FROM SYSTEM_RANGE(1, 100);
INSERT INTO TEST
    SORTED SELECT X, SPACE(100) FROM SYSTEM_RANGE(101, 200);
```

Using the Built-In Profiler

A very simple Java profiler is built-in. To use it, use the following template:

```
import org.h2.util.Profiler;
Profiler prof = new Profiler();
prof.startCollecting();
// .... some long running process, at least a few seconds
prof.stopCollecting();
System.out.println(prof.getTop(3));
```

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 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).

A simple profiling tool is included in H2. To use it, the application needs to be changed slightly. Example:

```
import org.h2.util;
...
Profiler profiler = new Profiler();
profiler.startCollecting();
// application code
System.out.println(profiler.getTop(3));
```

The profiler is built into the H2 Console tool, to analyze databases that open slowly. To use it, run the H2 Console, and then click on 'Test Connection'. Afterwards, click on "Test successful" and you get the most common stack traces, which helps to find out why it took so long to connect. You will only get the stack traces if opening the database took more than a few seconds.

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. For this to work, the trace level needs to be 2 or higher (TRACE_LEVEL_FILE=2). The easiest way to set the trace level is to append the setting to the database URL, for example: jdbc:h2:~/test;TRACE_LEVEL_FILE=2 or jdbc:h2:tcp://localhost/~/test;TRACE_LEVEL_FILE=2. As an example, execute the the following script using the H2 Console:

```
SET TRACE_LEVEL_FILE 2;
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;
```

After running the test case, convert the .trace.db file using the ConvertTraceFile tool. The trace file is located in the same directory as the database file.

```
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;
```

Statement Execution Plans

The SQL statement EXPLAIN displays the indexes and optimizations the database uses for a statement. The following statements support EXPLAIN: SELECT, UPDATE, DELETE, MERGE, INSERT. The following query shows that the database uses the primary key index to search for rows:

```
EXPLAIN SELECT * FROM TEST WHERE ID=1;
SELECT
    TEST.ID,
    TEST.NAME
FROM PUBLIC.TEST
    /* PUBLIC.PRIMARY_KEY_2: ID = 1 */
WHERE ID = 1
```

For joins, the tables in the execution plan are sorted in the order they are processed. The following query shows the database first processes the table INVOICE (using the primary key). For each row, it will additionally check that the value of the column AMOUNT is larger than zero, and for those rows the database will search in the table CUSTOMER (using the primary key). The query plan contains some redundancy so it is a valid statement.

```
CREATE TABLE CUSTOMER(ID IDENTITY, NAME VARCHAR);
CREATE TABLE INVOICE(ID IDENTITY,
    CUSTOMER_ID INT REFERENCES CUSTOMER(ID),
    AMOUNT NUMBER);

EXPLAIN SELECT I.ID, C.NAME FROM CUSTOMER C, INVOICE I
WHERE I.ID=10 AND AMOUNT>0 AND C.ID=I.CUSTOMER_ID;

SELECT
    I.ID,
    C.NAME
FROM PUBLIC.INVOICE I
    /* PUBLIC.PRIMARY_KEY_9: ID = 10 */
    /* WHERE (I.ID = 10)
        AND (AMOUNT > 0)
    */
INNER JOIN PUBLIC.CUSTOMER C
    /* PUBLIC.PRIMARY_KEY_5: ID = I.CUSTOMER_ID */
    ON 1=1
WHERE (C.ID = I.CUSTOMER_ID)
    AND ((I.ID = 10)
    AND (AMOUNT > 0))
```

Displaying the Scan Count

EXPLAIN ANALYZE additionally shows the scanned rows per table and pages read from disk per table or index. This will actually execute the query, unlike EXPLAIN which only prepares it. The following query scanned 1000 rows, and to do that had to read 85 pages from the data area of the table. Running the query twice will not list the pages read from disk, because they are now in the cache. The tableScan means this query doesn't use an index.

```
EXPLAIN ANALYZE SELECT * FROM TEST;
SELECT
    TEST.ID,
    TEST.NAME
FROM PUBLIC.TEST
    /* PUBLIC.TEST.tableScan */
    /* scanCount: 1000 */
/*
total: 85
TEST.TEST_DATA read: 85 (100%)
*/
```

The cache will prevent the pages are read twice. H2 reads all columns of the row unless only the columns in the index are read. Except for large CLOB and BLOB, which are not store in the table.

Special Optimizations

For certain queries, the database doesn't need to read all rows, or doesn't need to sort the result even if ORDER BY is used.

For queries of the form SELECT COUNT(*), MIN(ID), MAX(ID) FROM TEST, the query plan includes the line /* direct lookup */ if the data can be read from an index.

For queries of the form SELECT DISTINCT CUSTOMER_ID FROM INVOICE, the query plan includes the line /* distinct */ if there is an non-unique or multi-column index on this column, and if this column has a low selectivity.

For queries of the form SELECT * FROM TEST ORDER BY ID, the query plan includes the line /* index sorted */ to indicate there is no separate sorting required.

For queries of the form SELECT * FROM TEST GROUP BY ID ORDER BY ID, the query plan includes the line /* group sorted */ to indicate there is no separate sorting required.

How Data is Stored and How Indexes Work

Internally, each row in a table is identified by a unique number, the row id. The rows of a table are stored with the row id as the key. The row id is a number of type long. If a table has a single column primary key of type INT or BIGINT, then the value of this column is the row id, otherwise the database generates the row id automatically. There is a (non-standard) way to access the row id: using the _ROWID_ pseudo-column:

```
CREATE TABLE ADDRESS(FIRST_NAME VARCHAR, NAME VARCHAR, CITY VARCHAR, PHONE VARCHAR);
INSERT INTO ADDRESS VALUES('John', 'Miller', 'Berne', '123 456 789');
INSERT INTO ADDRESS VALUES('Philip', 'Jones', 'Berne', '123 012 345');
SELECT _ROWID_, * FROM ADDRESS;
```

The data is stored in the database as follows:

ROWID	FIRST_NAME	NAME	CITY	PHONE
1	John	Miller	Berne	123 456 789
2	Philip	Jones	Berne	123 012 345

Access by row id is fast because the data is sorted by this key. If the query condition does not contain the row id (and if no other index can be used), then all rows of the table are scanned. A table scan iterates over all rows in the table, in the order of the row id. To find out what strategy the database uses to retrieve the data, use EXPLAIN SELECT:

```
SELECT * FROM ADDRESS WHERE NAME = 'Miller';

EXPLAIN SELECT PHONE FROM ADDRESS WHERE NAME = 'Miller';
SELECT
    PHONE
FROM PUBLIC.ADDRESS
    /* PUBLIC.ADDRESS.tableScan */
WHERE NAME = 'Miller';
```

Indexes

An index internally is basically just a table that contains the indexed column(s), plus the row id:

```
CREATE INDEX INDEX_PLACE ON ADDRESS(CITY, NAME, FIRST_NAME);
```

In the index, the data is sorted by the indexed columns. So this index contains the following data:

CITY	NAME	FIRST_NAME	_ROWID_
Berne	Jones	Philip	2
Berne	Miller	John	1

When the database uses an index to query the data, it searches the index for the given data, and (if required) reads the remaining columns in the main data table (retrieved using the row id). An index on city, name, and first name (multi-column index) allows to quickly search for rows when the city, name, and first name are known. If only the city and name, or only the city is known, then this index is also used (so creating an additional index on just the city is not needed). This index is also used when reading all rows, sorted by the indexed columns. However, if only the first name is known, then this index is not used:

```
EXPLAIN SELECT PHONE FROM ADDRESS WHERE CITY = 'Berne' AND NAME = 'Miller' AND FIRST_NAME = 'John';
SELECT
    PHONE
FROM PUBLIC.ADDRESS
```

```
    /* PUBLIC.INDEX_PLACE: FIRST_NAME = 'John'
       AND CITY = 'Berne'
       AND NAME = 'Miller'
     */
WHERE (FIRST_NAME = 'John')
    AND ((CITY = 'Berne')
    AND (NAME = 'Miller'));

EXPLAIN SELECT PHONE FROM ADDRESS WHERE CITY = 'Berne';
SELECT
    PHONE
FROM PUBLIC.ADDRESS
    /* PUBLIC.INDEX_PLACE: CITY = 'Berne' */
WHERE CITY = 'Berne';

EXPLAIN SELECT * FROM ADDRESS ORDER BY CITY, NAME, FIRST_NAME;
SELECT
    ADDRESS.FIRST_NAME,
    ADDRESS.NAME,
    ADDRESS.CITY,
    ADDRESS.PHONE
FROM PUBLIC.ADDRESS
    /* PUBLIC.INDEX_PLACE */
ORDER BY 3, 2, 1
/* index sorted */;

EXPLAIN SELECT PHONE FROM ADDRESS WHERE FIRST_NAME = 'John';
SELECT
    PHONE
FROM PUBLIC.ADDRESS
    /* PUBLIC.ADDRESS.tableScan */
WHERE FIRST_NAME = 'John';
```

If your application often queries the table for a phone number, then it makes sense to create an additional index on it:

```
CREATE INDEX IDX_PHONE ON ADDRESS(PHONE);
```

This index contains the phone number, and the row id:

PHONE	_ROWID_
123 012 345	2
123 456 789	1

Using Multiple Indexes

Within a query, only one index per logical table is used. Using the condition PHONE = '123 567 789' OR CITY = 'Berne' would use a table scan instead of first using the index on the phone number and then the index on the city. It makes sense to write two queries and combine then using UNION. In this case, each individual query uses a different index:

```
EXPLAIN SELECT NAME FROM ADDRESS WHERE PHONE = '123 567 789'
UNION SELECT NAME FROM ADDRESS WHERE CITY = 'Berne';

(SELECT
    NAME
FROM PUBLIC.ADDRESS
    /* PUBLIC.IDX_PHONE: PHONE = '123 567 789' */
WHERE PHONE = '123 567 789')
UNION
(SELECT
    NAME
FROM PUBLIC.ADDRESS
    /* PUBLIC.INDEX_PLACE: CITY = 'Berne' */
WHERE CITY = 'Berne')
```

Fast Database Import

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

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

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

If you have to import a lot of rows, use a PreparedStatement or use CSV import. Please note that CREATE TABLE(...) ... AS SELECT ... is faster than CREATE TABLE(...); INSERT INTO ... SELECT ....

Advanced

Result Sets
Large Objects
Linked Tables
Recursive Queries
Updatable Views
Transaction Isolation
Multi-Version Concurrency Control (MVCC)
Clustering / High Availability
Two Phase Commit
Compatibility
Standards Compliance
Run as Windows Service
ODBC Driver
Using H2 in Microsoft .NET
ACID
Durability Problems
Using the Recover Tool
File Locking Protocols
File Locking Method 'Serialized'
Using Passwords
Password Hash
Protection against SQL Injection
Protection against Remote Access
Restricting Class Loading and Usage
Security Protocols
SSL/TLS Connections
Universally Unique Identifiers (UUID)
Settings Read from System Properties
Setting the Server Bind Address
Pluggable File System
Split File System
Database Upgrade
Limits and Limitations
Glossary and Links

Result Sets

Statements that Return a Result Set

The following statements return a result set: SELECT, EXPLAIN, CALL, SCRIPT, SHOW, HELP. All other statements return an update count.

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.

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=false.

The statement CREATE LINKED TABLE supports an optional schema name parameter.

The following are not supported because they may result in a deadlock: creating a linked table to the same database, and creating a linked table to another database using the server mode if the other database is open in the same server (use the embedded mode instead).

Data types that are not supported in H2 are also not supported for linked tables, for example unsigned data types if the value is outside the range of the signed type. In such cases, the columns needs to be cast to a supported type.

Updatable Views

By default, views are not updatable. To make a view updatable, use an "instead of" trigger as follows:

```
CREATE TRIGGER TRIGGER_NAME
```

```
INSTEAD OF INSERT, UPDATE, DELETE
ON VIEW_NAME
FOR EACH ROW CALL "com.acme.TriggerClassName";
```

Update the base table(s) within the trigger as required. For details, see the sample application org.h2.samples.UpdatableView.

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 after executing the statement, but write locks are kept until the transaction commits. 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**
  Both read locks and write locks are kept until the transaction commits. 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. When a lock is released, and multiple connections are waiting for it, one of them is picked at random.

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
```

MVCC is disabled by default. 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=TRUE; the complete undo log (the list of uncommitted changes) must fit in memory when using multi-version concurrency. The setting MAX_MEMORY_UNDO has no effect. It is not possible to enable or disable this setting while the database is already open. The setting must be specified in the first connection (the one that opens the database).

If MVCC is enabled, changing the lock mode (LOCK_MODE) 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). The cluster can be re-created using the CreateCluster tool without stopping the remaining server. Applications that are still connected are automatically disconnected, however when appending ;AUTO_RECONNECT=TRUE, they will recover from that.

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, 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.

Detect Which Cluster Instances are Running

To find out which cluster nodes are currently running, execute the following SQL statement:

```
SELECT VALUE FROM INFORMATION_SCHEMA.SETTINGS WHERE NAME='CLUSTER'
```

If the result is '' (two single quotes), then the cluster mode is disabled. Otherwise, the list of servers is returned, enclosed in single quote. Example: 'server1:9191,server2:9191'.

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, MERGE). However, they can be used in read-only statements and the result can then be used for modifying statements. Using auto-increment and identity columns is currently not supported. Instead, sequence values need to be manually requested and then used to insert data (using two statements).

When using the cluster modes, result sets are read fully in memory by the client, so that there is no problem if the server dies that executed the query. Result sets must fit in memory on the client side.

The SQL statement SET AUTOCOMMIT FALSE is not supported in the cluster mode. To disable autocommit, the method Connection.setAutoCommit(false) needs to be called.

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, UNIQUE, 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.

Supported Character Sets, Character Encoding, and Unicode

H2 internally uses Unicode, and supports all character encoding systems and character sets supported by the virtual machine you use.

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. 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.

The service wrapper bundled with H2 is a 32-bit version. To use a 64-bit version of Windows (x64), you need to use a 64-bit version of the wrapper, for example the one from Simon Krenger.

When running the database as a service, absolute path should be used. Using ~ in the database URL is problematic in this case, because it means to use the home directory of the current user. The service might run without or with the wrong user, so that the database files might end up in an unexpected place.

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.

Additional JDBC drivers

To use other databases (for example MySQL), the location of the JDBC drivers of those databases need to be added to the environment variables H2DRIVERS or CLASSPATH before installing the service. 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.

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. The property column represents the property key in the odbc.ini file (which may be different from the GUI).

Property	Example	Remarks
Data Source	H2 Test	The name of the ODBC Data Source
Database	~/test;ifexists=true	The database name. This can include connections settings. 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.
Servername	localhost	The server name or IP address. By default, only remote connections are allowed
Username	sa	The database user name.
SSL	false (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. Also, H2 does not provide index meta over ODBC.

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.

The first connection that opens a database using the PostgreSQL server needs to be an administrator user. Subsequent connections don't need to be opened by an administrator.

Using Microsoft Access

When using Microsoft Access to edit data in a linked H2 table, you may need to enable the following option: Tools - Options - Edit/Find - ODBC fields.

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. This method is supposed to return after all in-memory modified copies of buffers associated with this file descriptor 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 database file, even if the database is corrupted. It also extracts the content of the transaction log and 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.

The Recover tool creates a SQL script from database file. It also processes the transaction log.

To verify the database can recover at any time, append ;RECOVER_TEST=64 to the database URL in your test environment. This will simulate an application crash after each 64 writes to the database file. A log file named databaseName.h2.db.log is created that lists the operations. The recovery is tested using an in-memory file system, that means it may require a larger heap setting.

File Locking Protocols

Multiple concurrent connections to the same database are supported, however a database file can only be open for reading and writing (in embedded mode) by one process at the same time. Otherwise, the processes would overwrite each others data and corrupt the database file. To protect against this problem, 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 stops normally, 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'.

The file locking protocols (except the file locking method 'FS') have the following limitation: if a shared file system is used, and the machine with the lock owner is sent to sleep (standby or hibernate), another machine may take over. If the machine that originally held the lock wakes up, the database may become corrupt. If this situation can occur, the application must ensure the database is closed when the application is put to sleep.

File Locking Method 'File'

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

- If the lock file does not exist, it is created (using the atomic operation File.createNewFile). Then, the process waits a little bit (20 ms) and checks the file again. If the file was changed during this time, the operation is aborted. This protects against a race condition when one process deletes the lock file just after another 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 was recently modified, the process waits for some time (up to two seconds). 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.

File Locking Method 'FS'

This database file locking mechanism uses native file system lock on the database file. No *.lock.db file is created in this case, and no background thread is started. This mechanism may not work on all systems as expected. Some systems allow to lock the same file multiple times within the same virtual machine, and on some system native file locking is not supported or files are not unlocked after a power failure.

To enable this feature, append ;FILE_LOCK=FS to the database URL.

This feature is relatively new. When using it for production, please ensure your system does in fact lock files as expected.

File Locking Method 'Serialized'

This locking mode allows to open multiple connections to the same database. The connections may be opened from multiple processes and from different computers. When writing to the database, access is automatically synchronized internally. Write operations are slower than when using the server mode, and concurrency is relatively poor. The advantage of this mode is that there is no need to start a server.

To enable this feature, append ;FILE_LOCK=SERIALIZED to the database URL.

This feature is relatively new. When using it for production, please ensure your use case is well tested (if possible with automated test cases).

One known limitation when using this mode is: queries that write to the database will fail with the exception "The database is read only", if the queries are run using Statement.executeQuery(). As a workaround, use Statement.execute()

Using Passwords

Using Secure Passwords

Remember that weak passwords can be broken regardless of the encryption and security protocols. Don't use passwords that can be found in a dictionary. Appending numbers does not make passwords 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 (but it's more important to use a long password than to use special characters). Example:

i'sE2rtPiUKtT from the sentence 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 (if not enough main memory is available), which is a problem if the attacker has access to the swap file of the operating system.

It is a good idea to use char arrays instead of strings for 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.

Password Hash

Sometimes the database password needs to be stored in a configuration file (for example in the web.xml file). In addition to connecting with the plain text password, this database supports connecting with the password hash. This means that only the hash of the password (and not the plain text password) needs to be stored in the configuration file. This will only protect others

from reading or re-constructing the plain text password (even if they have access to the configuration file); it does not protect others from accessing the database using the password hash.

To connect using the password hash instead of plain text password, append ;PASSWORD_HASH=TRUE to the database URL, and replace the password with the password hash. To calculate the password hash from a plain text password, run the following command within the H2 Console tool: @password_hash <upperCaseUserName> <password>. As an example, if the user name is sa and the password is test, run the command @password_hash SA test. Then use the resulting password hash as you would use the plain text password. When using an encrypted database, then the user password and file password need to be hashed separately. To calculate the hash of the file password, run: @password_hash file <filePassword>.

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 prepared statement:

```
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 prepared statements 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 connections from other machines when starting the H2 Console, the TCP server, or the PG server. Remote access can be enabled using the command line options -webAllowOthers, -tcpAllowOthers, -pgAllowOthers. If you enable remote access, please also consider using the options -baseDir, -ifExists, so that remote users can not create new databases or access existing databases with weak passwords. When using the option -baseDir, only databases within that directory may be accessed. Ensure the existing accessible databases are protected using strong passwords.

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 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 a bit faster as AES in some environments) and to have an alternative algorithm if AES is suddenly broken. Please note that the XTEA implementation used in this database only uses 32 rounds and not 64 rounds as recommended by its inventor (as of 2010, the best known attack is on 27 rounds).

When a user tries to connect to an encrypted database, the combination of 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 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:

Number of UUIs	Probability of Duplicates
2^36=68'719'476'736	0.000'000'000'000'000'4
2^41=2'199'023'255'552	0.000'000'000'000'4
2^46=70'368'744'177'664	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.

Recursive Queries

H2 has experimental support for recursive queries using so called "common table expressions" (CTE). Examples:

```
WITH RECURSIVE T(N) AS (
    SELECT 1
    UNION ALL
    SELECT N+1 FROM T WHERE N<10
)
SELECT * FROM T;
-- returns the values 1 .. 10

WITH RECURSIVE T(N) AS (
```

```
    SELECT 1
    UNION ALL
    SELECT N*2 FROM T WHERE N<10
)
SELECT * FROM T;
-- returns the values 1, 2, 4, 8, 16

CREATE TABLE FOLDER(ID INT PRIMARY KEY, NAME VARCHAR(255), PARENT INT);

INSERT INTO FOLDER VALUES(1, null, null), (2, 'src', 1),
(3, 'main', 2), (4, 'org', 3), (5, 'test', 2);

WITH LINK(ID, NAME, LEVEL) AS (
    SELECT ID, NAME, 0 FROM FOLDER WHERE PARENT IS NULL
    UNION ALL
    SELECT FOLDER.ID, IFNULL(LINK.NAME || '/', '') || FOLDER.NAME, LEVEL + 1
    FROM LINK INNER JOIN FOLDER ON LINK.ID = FOLDER.PARENT
)
SELECT NAME FROM LINK WHERE NAME IS NOT NULL ORDER BY ID;
-- src
-- src/main
-- src/main/org
-- src/test
```

Limitations: Recursive queries need to be of the type UNION ALL, and the recursion needs to be on the second part of the query. No tables or views with the name of the table expression may exist. Different table expression names need to be used when using multiple distinct table expressions within the same transaction and for the same session. All columns of the table expression are of type VARCHAR, and may need to be cast to the required data type. Views with recursive queries are not supported. Subqueries and INSERT INTO ... FROM with recursive queries are not supported. Parameters are only supported within the last SELECT statement (a workaround is to use session variables like @start within the table expression). The syntax is:

```
WITH RECURSIVE recursiveQueryName(columnName, ...) AS (
    nonRecursiveSelect
    UNION ALL
    recursiveSelect
)
select
```

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.
- split: file system that splits files in 1 GB files (stackable with other file systems).
- 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). Please note that there currently is a file size limitation of 2 GB when using this file system when using a 32-bit JVM. To work around this limitation, combine it with the split file system: split:nioMapped:test.
- memFS: in-memory file system (slower than mem; experimental; mainly used for testing the database engine itself).
- memLZF: compressing in-memory file system (slower than memFS but uses less memory; experimental; mainly used for testing the database engine itself).

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.FilePath, FileBase, and call the method FilePath.register before using it.

For input streams (but not for random access files), URLs may be used in addition to the registered file systems. Example: jar:file:///c:/temp/example.zip!/org/example/nested.csv. To read a stream from the classpath, use the prefix classpath:, as in classpath:/org/h2/samples/newsfeed.sql.

Split File System

The file system prefix split: is used to split logical files into multiple physical files, for example so that a database can get larger than the maximum file system size of the operating system. If the logical file is larger than the maximum file size, then the file is split as follows:

- <fileName> (first block, is always created)
- <fileName>.1.part (second block)

More physical files (*.2.part, *.3.part) are automatically created / deleted if needed. The maximum physical file size of a block is 2^30 bytes, which is also called 1 GiB or 1 GB. However this can be changed if required, by specifying the block size in the file name. The file name format is: split:<x>:<fileName> where the file size per block is 2^x. For 1 MiB block sizes, use x = 20 (because 2^20 is 1 MiB). The following file name means the logical file is split into 1 MiB blocks: split:20:test.h2.db. An example database URL for this case is jdbc:h2:split:20:~/test.

Database Upgrade

In version 1.2, H2 introduced a new file store implementation which is incompatible to the one used in versions < 1.2. To automatically convert databases to the new file store, it is necessary to include an additional jar file. The file can be found at http://h2database.com/h2mig_pagestore_addon.jar . If this file is in the classpath, every connect to an older database will result in a conversion process.

The conversion itself is done internally via 'script to' and 'runscript from'. After the conversion process, the files will be renamed from

- dbName.data.db to dbName.data.db.backup
- dbName.index.db to dbName.index.db.backup

by default. Also, the temporary script will be written to the database directory instead of a temporary directory. Both defaults can be customized via

- org.h2.upgrade.DbUpgrade.setDeleteOldDb(boolean)
- org.h2.upgrade.DbUpgrade.setScriptInTmpDir(boolean)

prior opening a database connection.

Since version 1.2.140 it is possible to let the old h2 classes (v 1.2.128) connect to the database. The automatic upgrade .jar file must be present, and the URL must start with jdbc:h2v1_1: (the JDBC driver class is org.h2.upgrade.v1_1.Driver). If the database should automatically connect using the old version if a database with the old format exists (without upgrade), and use the new version otherwise, then append ;NO_UPGRADE=TRUE to the database URL. Please note the old driver did not process the system property "h2.baseDir" correctly, so that using this setting is not supported when upgrading.

Limits and Limitations

This database has the following known limitations:

- Database file size limit: 4 TB (using the default page size of 2 KB) or higher (when using a larger page size). This limit is including CLOB and BLOB data.
- 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 2^64.
- Main memory requirements: The larger the database, the more main memory is required. With the version 1.1 storage mechanism, the minimum main memory required for a 12 GB database was around 240 MB. With the current page store, 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

Index

Commands (Data Manipulation)

SELECT
INSERT
UPDATE
DELETE
BACKUP
CALL
EXPLAIN
MERGE
RUNSCRIPT
SCRIPT
SHOW

Commands (Data Definition)

ALTER INDEX RENAME
ALTER SCHEMA RENAME
ALTER SEQUENCE
ALTER TABLE ADD
ALTER TABLE ADD CONSTRAINT
ALTER TABLE ALTER
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 LOG
SET LOCK_MODE
SET LOCK_TIMEOUT
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

Alias
And Condition
Array
Boolean
Bytes
Case
Case When
Cipher
Column Definition
Comments
Compare
Condition
Condition Right Hand Side
Constraint

Constraint Name Definition
Csv Options
Data Type
Date
Decimal
Digit
Dollar Quoted String
Expression
Factor
Hex
Hex Number
Index Column
Int
Long
Name
Null
Number
Numeric
Operand
Order
Quoted Name
Referential Constraint
Referential Action
Script Compression
Select Expression
String
Summand
Table Expression
Values Expression
Term
Time
Timestamp
Value

System Tables

Information Schema
Range Table

SELECT

```
SELECT [ TOP term ] [ DISTINCT | ALL ] selectExpression [,...]
FROM tableExpression [,...] [ 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. GROUP BY groups the the result by the given expression(s). HAVING filter rows after grouping. ORDER BY sorts the result by the given column(s) or expression(s). UNION combines the result of this query with the results of another query.

LIMIT limits the number of rows returned by the query (no limit if null or smaller than zero). OFFSET specified how many rows to skip. SAMPLE_SIZE limits the number of rows read for aggregate queries.

Multiple set operators (UNION, INTERSECT, MINUS, EXPECT) are evaluated from left to right. For compatibility with other databases and future versions of H2 please use parentheses.

If FOR UPDATE is specified, the tables are locked for writing. When using MVCC, only the selected rows are locked as in an UPDATE statement. In this case, aggregate, GROUP BY, DISTINCT queries or joins are not allowed in this case.

Example:

```
SELECT * FROM TEST;
SELECT * FROM TEST ORDER BY NAME;
SELECT ID, COUNT(*) FROM TEST GROUP BY ID;
SELECT NAME, COUNT(*) FROM TEST GROUP BY NAME HAVING COUNT(*) > 2;
SELECT 'ID' COL, MAX(ID) AS MAX FROM TEST UNION SELECT 'NAME', MAX(NAME) FROM TEST;
SELECT * FROM TEST LIMIT 1000;
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 } [,...] ) } [,...] | [ DIRECT ] [ SORTED ] select } } |
{ SET { columnName = { DEFAULT | expression } } [,...] }
```

Inserts a new row / new rows into a table.

When using DIRECT, then the results from the query are directly applied in the target table without any intermediate step.

When using SORTED, b-tree pages are split at the insertion point. This can improve performance and reduce disk usage.

Example:

```
INSERT INTO TEST VALUES(1, 'Hello')
```

UPDATE

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

Updates data in a table.

Example:

```
UPDATE TEST SET NAME='Hi' WHERE ID=1;
UPDATE PERSON P SET NAME=(SELECT A.NAME FROM ADDRESS A WHERE A.ID=P.ID);
```

DELETE

```
DELETE [ TOP term ] FROM tableName [ WHERE expression ] [ LIMIT term ]
```

Deletes rows form a table. If TOP or LIMIT is specified, at most the specified number of rows are deleted (no limit if null or smaller than zero).

Example:

```
DELETE FROM TEST WHERE ID=2
```

BACKUP

```
BACKUP TO fileNameString
```

Backs up the database files to a .zip file. Objects are not locked, but the backup is transactionally consistent because the transaction log is also copied. Admin rights are required to execute this command.

Example:

BACKUP TO 'backup.zip'

CALL

```
CALL expression
```

Calculates a simple expression. This statement returns a result set with one row, except if the called function returns a result set itself. If the called function returns an array, then each element in this array is returned as a column.

Example:

CALL 15*25

EXPLAIN

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

Shows the execution plan for a statement. When using EXPLAIN ANALYZE, the statement is actually executed, and the query plan will include the actual row scan count for each table.

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 [ scriptCompression ]
[ 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.

Instead of a file name, an URL may be used. To read a stream from the classpath, use the prefix 'classpath:'.

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.sql'

SCRIPT

```
SCRIPT [ SIMPLE ] [ NODATA ] [ NOPASSWORDS ] [ NOSETTINGS ]
[ DROP ] [ BLOCKSIZE blockSizeInt ]
[ TO fileNameString [ scriptCompression ]
[ CIPHER cipher PASSWORD string ]
[ CHARSET charsetString ] ]
[ TABLE tableName [, ...] ]
[ SCHEMA schemaName [, ...] ]
```

Creates a SQL script from the database.

SIMPLE does not use multi-row insert statements. NODATA will not emit INSERT statements. 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 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 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).

The password must be in single quotes; it is case sensitive and can contain spaces.

This command locks objects while it is running.

When using the TABLE or SCHEMA option, only the selected table(s) / schema(s) are included.

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 SCHEMA RENAME

```
ALTER SCHEMA schema RENAME TO newSchemaName
```

Renames a schema. This command commits an open transaction.

Example:

ALTER SCHEMA TEST RENAME TO PRODUCTION

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 [ COLUMN ]
{ [ IF NOT EXISTS ] columnDefinition [ BEFORE columnName ]
| ( { columnDefinition } [,...] ) }
```

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). The required indexes are automatically created if they don't exist yet. 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

```
ALTER TABLE tableName ALTER COLUMN columnName
{ { dataType [ DEFAULT expression ] [ [ NOT ] NULL ] [ AUTO_INCREMENT | IDENTITY ] }
| { RENAME TO name }
| { RESTART WITH long }
| { SELECTIVITY int }
| { SET DEFAULT expression }
| { SET NULL }
| { SET NOT NULL } }
```

Changes the data type of a column, rename a column, change the identity value, or change the selectivity.

Changing the data type fails if the data can not be converted.

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

SELECTIVITY 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.

SET DEFAULT changes the default value of a column.

SET NULL sets a column to allow NULL. The row may not be part of a primary key. Single column indexes on this column are dropped.

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 CLOB;
ALTER TABLE TEST ALTER COLUMN NAME RENAME TO TEXT;
ALTER TABLE TEST ALTER COLUMN ID RESTART WITH 10000;
ALTER TABLE TEST ALTER COLUMN NAME SELECTIVITY 100;
ALTER TABLE TEST ALTER COLUMN NAME SET DEFAULT '';
ALTER TABLE TEST ALTER COLUMN NAME SET NOT NULL;
ALTER TABLE TEST ALTER COLUMN NAME SET NULL;
```

ALTER TABLE DROP COLUMN

```
ALTER TABLE tableName DROP COLUMN [ IF EXISTS ] 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 is used for views created using CREATE FORCE VIEW. 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
{ { COLUMN [ schemaName. ] tableName.columnName }
| { { TABLE | VIEW | CONSTANT | CONSTRAINT | ALIAS | INDEX | ROLE
| SCHEMA | SEQUENCE | TRIGGER | USER | DOMAIN } [ schemaName. ] objectName } }
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 | AS sourceCodeString }
```

Creates a new function alias. Deterministic functions must always return the same value for the same parameters. The result of such functions is cached if possible.

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. The class must be available in the classpath of the database engine (when using the server mode, it must be in the classpath of the server).

When defining a function alias with source code, the Sun javac is compiler is used if the file tools.jar is in the classpath. If not, javac is run as a separate process. Only the source code is stored in the database; the class is compiled each time the database is re-opened. Source code is usually passed as dollar quoted text to avoid escaping problems. If import statements are used, then the tag @CODE must be added before the method.

If the method throws an SQLException, it is directly re-thrown to the calling application; all other exceptions are first converted to a SQLException.

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 ALIAS REVERSE AS $$ String reverse(String s) { return new StringBuilder(s).reverse().toString(); } $$;
CALL REVERSE('Test');
```

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.

Domains are usable within the whole database. They can not be created in a specific schema.

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.

Hash indexes are meant for in-memory databases and memory tables (CREATE MEMORY TABLE). For other tables, or if the index contains multiple columns, the HASH keyword is ignored. Hash indexes can only test for equality, and do not support range queries (similar to a hash table). Non-unique keys are supported.

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 the connection to the source database is lost, the connection is re-opened (this is a workaround for MySQL that disconnects after 8 hours of inactivity by default).

If a query is used instead of the original table name, the table is read only. Queries must be enclosed in parenthesis: (SELECT * FROM ORDERS).

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. Specifying the owner currently has no effect.

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. To disable caching, use the cache size 1 or lower.

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
[ ( { columnDefinition | constraint } [,...] ) ]
[ ENGINE tableEngineName ] [ NOT PERSISTENT ] [ TRANSACTIONAL ]
[ AS select ]
```

Creates a new table.

Cached tables (the default for regular tables) are persistent, and the number of rows is not limited by the main memory. Memory tables (the default for temporary tables) are persistent, but the index data is kept in main memory, that means memory tables should not get too large.

Temporary tables are deleted when closing or opening a database. Temporary tables can be global (accessible by all connections) or local (only accessible by the current connection). The default for temporary tables is global. Indexes of temporary tables are kept fully in main memory, unless the temporary table is created using CREATE CACHED TABLE.

The ENGINE option is only required when custom table implementations are used. The table engine class must implement the interface org.h2.api.TableEngine.

Tables with the NOT PERSISTENT modifier are kept fully in memory, and all rows are lost when the database is closed.

The column definition is optional if a query is specified. In that case the column list of the query is used.

This command commits an open transaction, except when using TRANSACTIONAL (only supported for temporary tables).

Example:

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

CREATE TRIGGER

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

Creates a new trigger. The trigger class must be public and implement org.h2.api.Trigger. Inner classes are not supported. The class must be available in the classpath of the database engine (when using the server mode, it must be in the classpath of the server).

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. If there are multiple triggers, the order in which they are called is undefined.

ROLLBACK can be specified in combination with INSERT, UPDATE, and DELETE. Only row based AFTER trigger can be called on ROLLBACK. Exceptions that occur within such triggers are ignored. As the operations that occur within a trigger are part of the transaction, ROLLBACK triggers are only required if an operation communicates outside of the database.

INSTEAD OF triggers are implicitly row based and behave like BEFORE triggers. Only the first such trigger is called. Such triggers on views are supported. They can be used to make views updatable.

A BEFORE SELECT trigger is fired just before the database engine tries to read from the table. The trigger can be used to update a table on demand. The trigger is called with both 'old' and 'new' set to null.

The MERGE statement will call both INSERT and UPDATE triggers. Not supported are SELECT triggers with the option FOR EACH ROW, and AFTER SELECT triggers.

Committing or rolling back a transaction within a trigger is not allowed, except for SELECT triggers.

By default a trigger is called once for each statement, without the old and new rows. FOR EACH ROW triggers are called once for each inserted, updated, or deleted row.

The trigger need to be created in the same schema as the table. The schema name does not need to be specified when creating the trigger.

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 [ OR REPLACE ] [ 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.

If the OR REPLACE clause is used an existing view will be replaced, and any dependent views will not need to be recreated. If dependent views will become invalid as a result of the change an error will be generated, but this error can be ignored if the FORCE clause is also used.

Views are not updatable except when using 'instead of' triggers.

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:

```
DROP AGGREGATE MEDIAN
```

DROP ALIAS

```
DROP ALIAS [ IF EXISTS ] existingFunctionAliasName
```

Drops an existing function alias.

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

Example:

```
DROP 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 [,...] [ RESTRICT | CASCADE ]
```

Drops an existing table, or a list of tables. The command will fail if dependent views exist and the RESTRICT clause is used (the default). All dependent views are dropped as well if the CASCADE clause is used. 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 [ RESTRICT | CASCADE ]
```

Drops an existing view. All dependent views are dropped as well if the CASCADE clause is used (the default). The command will fail if dependent views exist and the RESTRICT clause is used. This command commits an open transaction.

Example:

DROP VIEW TEST_VIEW

TRUNCATE TABLE

```
TRUNCATE TABLE tableName
```

Removes all rows from a table. Unlike 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 (except if referential integrity is disabled for this database or for this table). Linked tables can't be truncated.

This command commits an open transaction.

Example:

TRUNCATE TABLE TEST

CHECKPOINT

```
CHECKPOINT
```

Flushes the data to disk and switches to a new transaction log if possible.

Admin rights are required to execute this command.

Example:

CHECKPOINT

CHECKPOINT SYNC

```
CHECKPOINT SYNC
```

Flushes the data to disk and 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. Variables are not persisted and session scoped, that means only visible from within the session in which they are defined. 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, as it affects all connections. 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 - however this will not work as expected when using a connection pool (the connection pool manager will re-enable autocommit when returning the connection to the pool, so autocommit will only be disabled the first time the connection is used.

Example:

SET AUTOCOMMIT OFF

SET CACHE_SIZE

```
SET CACHE_SIZE int
```

Sets the size of the cache in KB (each KB being 1024 bytes) for the current database. 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. Using a very small value (specially 0) will reduce performance a lot. This setting only affects the database engine (the server in a client/server environment). It has no effect for in-memory databases.

Admin rights are required to execute this command, as it affects all connections. 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).

This command is effective immediately, but does not commit 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 the supported collations and the STRENGTH (PRIMARY is usually case- and umlaut-insensitive; SECONDARY is case-insensitive but umlaut-sensitive; TERTIARY is both case- and umlaut-sensitive; IDENTICAL is sensitive to all differences and only affects ordering).

The ICU4J collator is used if it is in the classpath. It is also used if the collation name starts with ICU4J_ (in that case, the ICU4J must be in the classpath, otherwise an exception is thrown). The default collator is used if the collation name starts with DEFAULT_ (even if ICU4J is in the classpath).

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, as it affects all connections. 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, as it affects all connections. 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, as it affects all connections. 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 the data is still stored in the database file. The size of memory tables is limited by the memory. The default is CACHED.

Admin rights are required to execute this command, as it affects all connections. This command commits an open transaction. This setting is persistent. It has no effect for in-memory databases.

Example:

SET DEFAULT_TABLE_TYPE MEMORY

SET EXCLUSIVE

```
SET EXCLUSIVE { 0 | 1 | 2 }
```

Switched the database to exclusive mode (1, 2) and back to normal mode (0).

In exclusive mode, new connections are rejected, and operations by other connections are paused until the exclusive mode is disabled. When using the value 1, existing connections stay open. When using the value 2, all existing connections are closed (and current transactions are rolled back) except the connection that executes SET EXCLUSIVE. 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.

Example:

SET EXCLUSIVE 1

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. String literals and parameters are however still considered case sensitive even if this option is set.

Admin rights are required to execute this command, as it affects all connections. This command commits an open transaction. This setting is persistent. This setting can be appended to the database URL: jdbc:h2:test;IGNORECASE=TRUE

Example:

SET IGNORECASE TRUE

SET LOG

```
SET LOG int
```

Sets the transaction log mode. The values 0, 1, and 2 are supported, the default is 2. This setting affects all connections.

LOG 0 means the transaction log is disabled completely. It is the fastest mode, but also the most dangerous: if the process is killed while the database is open in this mode, the data might be lost. It must only be used if this is not a problem, for example when initially loading a database, or when running tests.

LOG 1 means the transaction log is enabled, but FileDescriptor.sync is disabled. This setting is about half as fast as with LOG 0. This setting is useful if no protection against power failure is required, but the data must be protected against killing the process.

LOG 2 (the default) means the transaction log is enabled, and FileDescriptor.sync is called for each checkpoint. This setting is about half as fast as LOG 1. Depending on the file system, this will also protect against power failure in the majority if cases.

Admin rights are required to execute this command, as it affects all connections. This command commits an open transaction. This setting is not persistent. This setting can be appended to the database URL: jdbc:h2:test;LOG=0

Example:

SET LOG 1

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). This setting affects all connections.

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, as it affects all connections. 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 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. This setting has no effect for in-memory databases.

Admin rights are required to execute this command, as it affects all connections. 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 size of the transaction log, in megabytes. If the log is larger, and if there is no open transaction, the transaction log is truncated. If there is an option transaction, the transaction log will continue to grow however. The default max size is 16 MB. This setting has no effect for in-memory databases.

Admin rights are required to execute this command, as it affects all connections. 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, as it affects all connections. This command commits an open transaction. This setting is persistent. It has no effect for in-memory databases.

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, as it affects all connections. This command commits an open transaction. This setting is persistent. It has no effect for in-memory databases.

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, as it affects all connections. It has no effect for in-memory databases. 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, as it affects all connections. 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, as it affects all connections. 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, as it affects all connections. 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, as it affects all connections.

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, as it affects all connections. 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, as it affects all connections. 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, as it affects all connections. 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 | DEFRAG ]
```

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

If no option is used, then the database is closed normally. All connections are closed, open transactions are rolled back.

SHUTDOWN COMPACT fully compacts the database (re-creating the database may further reduce the database size). If the database is closed normally (using SHUTDOWN or by closing all connections), then the database is also compacted, but only for at most the time defined by the database setting h2.maxCompactTime in milliseconds (see there).

SHUTDOWN IMMEDIATELY closes the database files without any cleanup and without compacting.

SHUTDOWN DEFRAG re-orders the pages when closing the database so that table scans are faster.

Admin rights are required to execute this command.

Example:

SHUTDOWN COMPACT

Alias

```
name
```

An alias is a name that is only valid in the context of the statement.

Example:

A

And Condition

```
condition [ { AND condition } [...] ]
```

Value or condition.

Example:

ID=1 AND NAME='Hi'

Array

```
( [ expression, [ expression [,...] ] ] )
```

An array of values. An empty array is '()'. Trailing commas are ignored. An array with one element must contain a comma to be parsed as an array.

Example:

(1, 2)
(1,)
()

Boolean

```
TRUE | FALSE
```

A boolean value.

Example:

TRUE

Bytes

```
X&amp;apos;hex&amp;apos;
```

A binary value. The hex value is not case sensitive.

Example:

X'01FF'

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-128) and XTEA (using 32 rounds). XTEA is a bit faster than AES in some environments, but AES is more secure.

Example:

AES

Column Definition

```
columnName dataType
[ { DEFAULT expression | AS computedColumnExpression } ] [ [ NOT ] NULL ]
[ { AUTO_INCREMENT | IDENTITY } [ ( startInt [, incrementInt ] ) ] ]
[ SELECTIVITY selectivity ] [ COMMENT expression ]
[ PRIMARY KEY [ HASH ] | UNIQUE ] [ CHECK condition ]
```

Default expressions are used if no explicit value was used when adding a row. The computed column expression is evaluated and assigned whenever the row changes.

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).

Check constraints can reference columns of the table, and they can reference objects that exist while the statement is executed.

Example:

```
CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255) DEFAULT '');
CREATE TABLE TEST(ID BIGINT IDENTITY);
CREATE TABLE TEST(QUANTITY INT, PRICE DECIMAL, AMOUNT DECIMAL AS QUANTITY*PRICE);
```

Comments

```
-- anything | // anything | /* anything */
```

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
```

Compare

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

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

Example:

<>

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
| IS [ NOT ] [ DISTINCT FROM ] operand
| BETWEEN operand AND operand
| IN ( { select | expression [,...] } )
| [ NOT ] LIKE operand [ ESCAPE string ]
| [ NOT ] REGEXP operand
```

The right hand side of a condition.

The conditions IS [NOT] and IS [NOT] DISTINCT FROM are null-safe, meaning NULL is considered the same as NULL, and the condition never evaluates to NULL.

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). At most one escape character is allowed. Each character that follows the escape character in the pattern needs to match exactly. Patterns that end with an escape character are invalid and the expression returns NULL.

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

Example:

LIKE 'Jo%'

Constraint

```
[ constraintNameDefinition ] {
CHECK expression | UNIQUE ( columnName [,...] )
| referentialConstraint }
| PRIMARY KEY [ HASH ] ( columnName [,...] )
```

Defines a constraint. The check condition must evaluate to TRUE, FALSE or NULL. TRUE and NULL mean the operation is to be permitted, and FALSE means the operation is to be rejected. To prevent NULL in a column, use NOT NULL instead of a check constraint.

Example:

PRIMARY KEY(ID, NAME)

Constraint Name Definition

```
CONSTRAINT [ IF NOT EXISTS ] newConstraintName
```

Defines a constraint name.

Example:

CONSTRAINT CONST_ID

Csv Options

charsetString [, fieldSepString [, fieldDelimString [, escString [, nullString]]]]]
| optionString

Optional parameters for CSVREAD and CSVWRITE. Instead of setting the options one by one, all options can be combined into a space separated key-value pairs, as follows: STRINGDECODE('charset=UTF-8 escape=\" fieldDelimiter=\" fieldSeparator=, ' || 'lineComment=# lineSeparator=\n null= rowSeparator='). The following options are supported: caseSensitiveColumnNames (true or false; disabled by default), charset, escape, fieldDelimiter, fieldSeparator, lineComment (disabled by default), lineSeparator, null, rowSeparator (not set by default), preserveWhitespace (true or false; disabled by default), writeColumnHeader (true or false; enabled by default). For a newline or other special character, use STRINGDECODE as in the example above. A space needs to be escaped with a backslash ('\ '), and a backslash needs to be escaped with another backslash ('\\'). All other characters are not to be escaped, that means newline and tab characters are written as such.

Example:

CALL CSVWRITE('test2.csv', 'SELECT * FROM TEST', 'charset=UTF-8 fieldSeparator=|');

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

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'

Decimal

[+ | -] { { number [. number] } | { . number } } [E [+ | -] expNumber [...]]]

A decimal number with fixed precision and scale. Internally, java.lang.BigDecimal is used. To ensure the floating point representation is used, use CAST(X AS DOUBLE). There are some special decimal values: to represent positive infinity, use POWER(0, -1); for negative infinity, use (-POWER(0, -1)); for -0.0, use (-CAST(0 AS DOUBLE)); for NaN (not a number), use SQRT(-1).

Example:

SELECT -1600.05
SELECT CAST(0 AS DOUBLE)
SELECT -1.4e-10

Digit

```
0-9
```

A digit.

Example:

0

Dollar Quoted String

```
$$anything$$
```

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:

$$John's car$$

Expression

```
andCondition [ { OR andCondition } [...] ]
```

Value or condition.

Example:

ID=1 OR NAME='Hi'

Factor

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

A value or a numeric factor.

Example:

ID * 10

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

Hex Number

[+ | -] 0x hex

A number written in hexadecimal notation.

Example:

0xff

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

Int

[+ | -] number

The maximum integer number is 2147483647, the minimum is -2147483648.

Example:

10

Long

[+ | -] number

Long numbers are between -9223372036854775808 and 9223372036854775807.

Example:

100000

Name

{ { A-Z|_ } [{ A-Z|_|0-9 } [...]] } | quotedName

Names are not case sensitive. There is no maximum name length.

Example:

TEST

Null

NULL

NULL is a value without data type and means 'unknown value'.

Example:

NULL

Number

digit [...]

The maximum length of the number depends on the data type used.

Example:

100

Numeric

decimal | int | long | hexNumber

The data type of a numeric value is always the lowest possible for the given value. If the number contains a dot this is decimal; otherwise it is int, long, or decimal (depending on the value).

Example:

SELECT -1600.05
SELECT CAST(0 AS DOUBLE)
SELECT -1.4e-10

Operand

summand [{ || summand } [...]]

A value or a concatenation of values.

Example:

'Hi' || ' Eva'

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

Quoted Name

```
"anything"
```

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"

Referential Constraint

```
FOREIGN KEY ( columnName [,...] )
REFERENCES [ refTableName ] [ ( refColumnName [,...] ) ]
[ ON DELETE referentialAction ] [ ON UPDATE referentialAction ]
```

Defines a referential constraint. If the table name is not specified, then the same table is referenced. RESTRICT is the default action. If the referenced columns are not specified, then the primary key columns are used. The required indexes are automatically created if required. Some tables may not be referenced, such as metadata tables.

Example:

FOREIGN KEY(ID) REFERENCES TEST(ID)

Referential Action

```
CASCADE | RESTRICT | NO ACTION | SET { DEFAULT | NULL }
```

The action CASCADE will cause conflicting rows in the referencing (child) table to be deleted or updated. RESTRICT is the default action. As this database does not support deferred checking, RESTRICT and NO ACTION will both throw an exception if the constraint is violated. The action SET DEFAULT will set the column in the referencing (child) table to the default value, while SET NULL will set it to NULL.

Example:

FOREIGN KEY(ID) REFERENCES TEST(ID) ON UPDATE CASCADE

Script Compression

```
COMPRESSION { DEFLATE | LZF | ZIP | GZIP }
```

The compression algorithm to use for script files. LZF is faster but uses more space.

Example:

COMPRESSION LZF

Select Expression

```
* | expression [ [ AS ] columnAlias ] | tableAlias.*
```

An expression in a SELECT statement.

Example:

ID AS VALUE

String

```
&apos;anything&apos;
```

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'

Summand

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

A value or a numeric sum.

Please note the text concatenation operator is ||.

Example:

ID + 20

Table Expression

```
{ [ schemaName. ] tableName | ( select ) | valuesExpression } [ [ AS ] newTableAlias ]
[ { { LEFT | RIGHT } [ OUTER ] | [ INNER ] | CROSS | NATURAL }
JOIN tableExpression [ 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

Values Expression

```
VALUES { ( expression [,...] ) } [,...]
```

A list of rows that can be used like a table. The column list of the resulting table is C1, C2, and so on.

Example:

SELECT * FROM (VALUES(1, 'Hello'), (2, 'World')) AS V;

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. Each table has a pseudo-column named _ROWID_ that contains the unique row identifier.

Example:

'Hello'

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'

Value

string | dollarQuotedString | numeric | date | time | timestamp | boolean | bytes | array | null

A literal value of any data type, or null.

Example:

10

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, TYPE_NAME, COLUMN_COUNT, RETURNS_RESULT, REMARKS, ID, SOURCE
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, INDEX_CLASS
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_CLASS
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:

Example:

```
SELECT X FROM SYSTEM_RANGE(1, 10);
```

Functions

Index

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
COSH
COT
SIN
SINH
TAN
TANH
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
ARRAY_CONTAINS
AUTOCOMMIT
CANCEL_SESSION
CASEWHEN Function
CAST
COALESCE
CONVERT
CURRVAL
CSVREAD
CSVWRITE
DATABASE
DATABASE_PATH
DECODE
FILE_READ
GREATEST
IDENTITY
IFNULL
LEAST
LOCK_MODE
LOCK_TIMEOUT
LINK_SCHEMA
MEMORY_FREE
MEMORY_USED
NEXTVAL
NULLIF
NVL2
READONLY
ROWNUM
SCHEMA
SCOPE_IDENTITY
SESSION_ID
SET
TABLE
TRANSACTION_ID
TRUNCATE_VALUE
USER

AVG

```
AVG ( [ DISTINCT ] { numeric } )
```

The average (mean) value. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements. The returned value is of the same data type as the parameter.

Example:

AVG(X)

BOOL_AND

```
BOOL_AND(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)
```

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( { * | { [ DISTINCT ] expression } } )
```

The count of all row, or of the non-null values. This method returns a long. 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 ] )
```

Concatenates strings with a separator. The default separator is a ',' (without space). This method returns a string. 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)
```

The highest value. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements. The returned value is of the same data type as the parameter.

Example:

MAX(NAME)

MIN

```
MIN(value)
```

The lowest value. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements. The returned value is of the same data type as the parameter.

Example:

MIN(NAME)

SUM

```
SUM( [ DISTINCT ] { numeric } )
```

The sum of all values. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements. The returned value is of the same data type as the parameter.

Example:

SUM(X)

SELECTIVITY

```
SELECTIVITY(value)
```

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 ] numeric )
```

The population standard deviation. This method returns a double. 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 ] numeric )
```

The sample standard deviation. This method returns a double. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

STDDEV(X)

VAR_POP

```
VAR_POP( [ DISTINCT ] numeric )
```

The population variance (square of the population standard deviation). This method returns a double. 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 ] numeric )
```

The sample variance (square of the sample standard deviation). This method returns a double. If no rows are selected, the result is NULL. Aggregates are only allowed in select statements.

Example:

VAR_SAMP(X)

ABS

```
ABS ( { numeric } )
```

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. The returned value is of the same data type as the parameter.

Example:

ABS(ID)

ACOS

```
ACOS(numeric)
```

Calculate the arc cosine. See also Java Math.acos. This method returns a double.

Example:

ACOS(D)

ASIN

```
ASIN(numeric)
```

Calculate the arc sine. See also Java Math.asin. This method returns a double.

Example:

ASIN(D)

ATAN

```
ATAN(numeric)
```

Calculate the arc tangent. See also Java Math.atan. This method returns a double.

Example:

ATAN(D)

COS

```
COS(numeric)
```

Calculate the trigonometric cosine. See also Java Math.cos. This method returns a double.

Example:

COS(ANGLE)

COSH

```
COSH(numeric)
```

Calculate the hyperbolic cosine. See also Java Math.cosh. This method returns a double.

Example:

COSH(X)

COT

```
COT(numeric)
```

Calculate the trigonometric cotangent (1/TAN(ANGLE)). See also Java Math.* functions. This method returns a double.

Example:

COT(ANGLE)

SIN

```
SIN(numeric)
```

Calculate the trigonometric sine. See also Java Math.sin. This method returns a double.

Example:

SIN(ANGLE)

SINH

```
SINH(numeric)
```

Calculate the hyperbolic sine. See also Java Math.sinh. This method returns a double.

Example:

SINH(ANGLE)

TAN

```
TAN(numeric)
```

Calculate the trigonometric tangent. See also Java Math.tan. This method returns a double.

Example:

TAN(ANGLE)

TANH

TANH(numeric)

Calculate the hyperbolic tangent. See also Java Math.tanh. This method returns a double.

Example:

TANH(X)

ATAN2

ATAN2(numeric, numeric)

Calculate the angle when converting the rectangular coordinates to polar coordinates. See also Java Math.atan2. This method returns a double.

Example:

ATAN2(X, Y)

BITAND

BITAND(long, long)

The bitwise AND operation. This method returns a long. See also Java operator &.

Example:

BITAND(A, B)

BITOR

BITOR(long, long)

The bitwise OR operation. This method returns a long. See also Java operator |.

Example:

BITOR(A, B)

BITXOR

BITXOR(long, long)

The bitwise XOR operation. This method returns a long. See also Java operator ^.

Example:

BITXOR(A, B)

MOD

MOD(long, long)

The modulo operation. This method returns a long. See also Java operator %.

Example:

MOD(A, B)

CEILING

{ CEILING | CEIL } (numeric)

See also Java Math.ceil. This method returns a double.

Example:

CEIL(A)

DEGREES

DEGREES(numeric)

See also Java Math.toDegrees. This method returns a double.

Example:

DEGREES(A)

EXP

EXP(numeric)

See also Java Math.exp. This method returns a double.

Example:

EXP(A)

FLOOR

FLOOR(numeric)

See also Java Math.floor. This method returns a double.

Example:

FLOOR(A)

LOG

{ LOG | LN } (numeric)

See also Java Math.log. In the PostgreSQL mode, LOG(x) is base 10. This method returns a double.

Example:

LOG(A)

LOG10

LOG10(numeric)

See also Java Math.log10 (in Java 5). This method returns a double.

Example:

LOG10(A)

RADIANS

RADIANS(numeric)

See also Java Math.toRadians. This method returns a double.

Example:

RADIANS(A)

SQRT

SQRT(numeric)

See also Java Math.sqrt. This method returns a double.

Example:

SQRT(A)

PI

PI()

See also Java Math.PI. This method returns a double.

Example:

PI()

POWER

POWER(numeric, numeric)

See also Java Math.pow. This method returns a double.

Example:

POWER(A, B)

RAND

```
{ RAND | RANDOM } ( [ int ] )
```

Calling the function without parameter returns the next a pseudo random number. Calling it with an parameter seeds the session's random number generator. This method returns a double between 0 (including) and 1 (excluding).

Example:

RAND()

RANDOM_UUID

```
RANDOM_UUID()
```

Returns a new UUID with 122 pseudo random bits.

Please note that using an index on randomly generated data will result on poor performance once there are millions of rows in a table. The reason is that the cache behavior is very bad with randomly distributed data. This is a problem for any database system.

Example:

RANDOM_UUID()

ROUND

```
ROUND(numeric [, digitsInt])
```

Rounds to a number of digits, or to the nearest long if the number of digits if not set. This method returns a double.

Example:

ROUND(VALUE, 2)

ROUNDMAGIC

```
ROUNDMAGIC(numeric)
```

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. This method returns a double.

Example:

ROUNDMAGIC(VALUE/3*3)

SECURE_RAND

```
SECURE_RAND(int)
```

Generates a number of cryptographically secure random numbers. This method returns bytes.

Example:

```
CALL SECURE_RAND(16)
```

SIGN

```
SIGN ( { numeric } )
```

Returns -1 if the value is smaller 0, 0 if zero, and otherwise 1.

Example:

```
SIGN(VALUE)
```

ENCRYPT

```
ENCRYPT(algorithmString, keyBytes, dataBytes)
```

Encrypts data using a key. Supported algorithms are XTEA and AES. The block size is 16 bytes. This method returns bytes.

Example:

```
CALL ENCRYPT('AES', '00', STRINGTOUTF8('Test'))
```

DECRYPT

```
DECRYPT(algorithmString, keyBytes, dataBytes)
```

Decrypts data using a key. Supported algorithms are XTEA and AES. The block size is 16 bytes. This method returns bytes.

Example:

```
CALL TRIM(CHAR(0) FROM UTF8TOSTRING(
    DECRYPT('AES', '00', '3fabb4de8f1ee2e97d7793bab2db1116')))
```

HASH

```
HASH(algorithmString, dataBytes, iterationInt)
```

Calculate the hash value using an algorithm, and repeat this process for a number of iterations. Currently, the only algorithm supported is SHA256. This method returns bytes.

Example:

```
CALL HASH('SHA256', STRINGTOUTF8('Password'), 1000)
```

TRUNCATE

```
{ TRUNC | TRUNCATE } (numeric, digitsInt)
```

Truncates to a number of digits (to the next value closer to 0). This method returns a double.

Example:

TRUNCATE(VALUE, 2)

COMPRESS

```
COMPRESS(dataBytes [, algorithmString])
```

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. This method returns bytes.

Example:

COMPRESS(STRINGTOUTF8('Test'))

EXPAND

```
EXPAND(bytes)
```

Expands data that was compressed using the COMPRESS function. This method returns bytes.

Example:

UTF8TOSTRING(EXPAND(COMPRESS(STRINGTOUTF8('Test'))))

ZERO

```
ZERO()
```

Returns the value 0. This function can be used even if numeric literals are disabled.

Example:

ZERO()

ASCII

```
ASCII(string)
```

Returns the ASCII value of the first character in the string. This method returns an int.

Example:

ASCII('Hi')

BIT_LENGTH

```
BIT_LENGTH(string)
```

Returns the number of bits in a string. This method returns a long. 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 )
```

Returns the number of characters in a string. This method returns a long. For BLOB, CLOB, BYTES and JAVA_OBJECT, the precision is used.

Example:

LENGTH(NAME)

OCTET_LENGTH

```
OCTET_LENGTH(string)
```

Returns the number of bytes in a string. This method returns a long. For BLOB, CLOB, BYTES and JAVA_OBJECT, the precision is used. Each character needs 2 bytes.

Example:

OCTET_LENGTH(NAME)

CHAR

```
{ CHAR | CHR } ( int )
```

Returns the character that represents the ASCII value. This method returns a string.

Example:

CHAR(65)

CONCAT

```
CONCAT(string, string [,...])
```

Combines strings. This method returns a string.

Example:

CONCAT(NAME, '!')

DIFFERENCE

```
DIFFERENCE(string, string)
```

Returns the difference between the sounds of two strings. This method returns an int.

Example:

DIFFERENCE(T1.NAME, T2.NAME)

HEXTORAW

```
HEXTORAW(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)
```

Converts a string to the hex representation. 4 hex characters per string character are used. This method returns a string.

Example:

RAWTOHEX(DATA)

INSTR

```
INSTR(string, searchString, [, startInt])
```

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:

INSTR(EMAIL,'@')

INSERT Function

```
INSERT(originalString, startInt, lengthInt, addString)
```

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. This method returns a string.

Example:

INSERT(NAME, 1, 1, ' ')

LOWER

```
{ LOWER | LCASE } ( string )
```

Converts a string to lowercase.

Example:

LOWER(NAME)

UPPER

```
{ UPPER | UCASE } ( string )
```

Converts a string to uppercase.

Example:

UPPER(NAME)

LEFT

```
LEFT(string, int)
```

Returns the leftmost number of characters.

Example:

LEFT(NAME, 3)

RIGHT

```
RIGHT(string, int)
```

Returns the rightmost number of characters.

Example:

RIGHT(NAME, 3)

LOCATE

```
LOCATE(searchString, string [, startInt])
```

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)
```

Returns the location of a search string in a string. See also LOCATE.

Example:

POSITION('.', NAME)

LPAD

```
LPAD(string, int[, paddingString])
```

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])
```

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)
```

Removes all leading spaces from a string.

Example:

LTRIM(NAME)

RTRIM

```
RTRIM(string)
```

Removes all trailing spaces from a string.

Example:

RTRIM(NAME)

TRIM

```
TRIM ( [ { LEADING | TRAILING | BOTH } [ string ] FROM ] 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)
```

Replaces each substring that matches a regular expression. For details, see the Java String.replaceAll() method. If any parameter is null, the result is null.

Example:

REGEXP_REPLACE('Hello World', ' +', ' ')

REPEAT

```
REPEAT(string, int)
```

Returns a string repeated some number of times.

Example:

REPEAT(NAME || ' ', 10)

REPLACE

```
REPLACE(string, searchString [, replacementString])
```

Replaces all occurrences of a search string in a text with another string. If no replacement is specified, the search string is removed from the original string. If any parameter is null, the result is null.

Example:

REPLACE(NAME, ' ')

SOUNDEX

```
SOUNDEX(string)
```

Returns a four character code representing the sound of a string. See also http://www.archives.gov/genealogy/census/soundex.html . This method returns a string.

Example:

SOUNDEX(NAME)

SPACE

SPACE(int)

Returns a string consisting of a number of spaces.

Example:

SPACE(80)

STRINGDECODE

STRINGDECODE(string)

Converts a encoded string using the Java string literal encoding format. Special characters are \b, \t, \n, \f, \r, \", \\, \<octal>, \u<unicode>. This method returns a string.

Example:

CALL STRINGENCODE(STRINGDECODE('Lines 1\nLine 2'))

STRINGENCODE

STRINGENCODE(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>. This method returns a string.

Example:

CALL STRINGENCODE(STRINGDECODE('Lines 1\nLine 2'))

STRINGTOUTF8

STRINGTOUTF8(string)

Encodes a string to a byte array using the UTF8 encoding format. This method returns bytes.

Example:

CALL UTF8TOSTRING(STRINGTOUTF8('This is a test'))

SUBSTRING

{ SUBSTRING | SUBSTR } (string, startInt [, lengthInt])

Returns a substring of a string starting at a position. If the start index is negative, then the start index is relative to the end of the string. The length is optional. Also supported is: SUBSTRING(string FROM start [FOR length]).

Example:

```
CALL SUBSTR('[Hello]', 2, 5);
CALL SUBSTR('Hello World', -5);
```

UTF8TOSTRING

```
UTF8TOSTRING(bytes)
```

Decodes a byte array in the UTF8 format to a string.

Example:

CALL UTF8TOSTRING(STRINGTOUTF8('This is a test'))

XMLATTR

```
XMLATTR(nameString, valueString)
```

Creates an XML attribute element of the form name=value. The value is encoded as XML text. This method returns a string.

Example:

CALL XMLNODE('a', XMLATTR('href', 'http://h2database.com'))

XMLNODE

```
XMLNODE(elementString [, attributesString [, contentString [, indentBoolean]]])
```

Create an XML node element. An empty or null attribute string means no attributes are set. An empty or null content string means the node is empty. The content is indented by default if it contains a newline. This method returns a string.

Example:

CALL XMLNODE('a', XMLATTR('href', 'http://h2database.com'), 'H2')

XMLCOMMENT

```
XMLCOMMENT(commentString)
```

Creates an XML comment. Two dashes (--) are converted to - -. This method returns a string.

Example:

CALL XMLCOMMENT('Test')

XMLCDATA

```
XMLCDATA(valueString)
```

Creates an XML CDATA element. If the value contains]]>, an XML text element is created instead. This method returns a string.

Example:

CALL XMLCDATA('data')

XMLSTARTDOC

```
XMLSTARTDOC()
```

Returns the XML declaration. The result is always <?xml version=1.0?>.

Example:

CALL XMLSTARTDOC()

XMLTEXT

```
XMLTEXT(valueString [, escapeNewlineBoolean])
```

Creates an XML text element. If enabled, newline and linefeed is converted to an XML entity (&#). This method returns a string.

Example:

CALL XMLTEXT('test')

ARRAY_GET

```
ARRAY_GET(arrayExpression, indexExpression)
```

Returns one element of an array. This method returns a string.

Example:

CALL ARRAY_GET(('Hello', 'World'), 2)

ARRAY_LENGTH

```
ARRAY_LENGTH(arrayExpression)
```

Returns the length of an array.

Example:

CALL ARRAY_LENGTH(('Hello', 'World'))

ARRAY_CONTAINS

```
ARRAY_CONTAINS(arrayExpression, value)
```

Returns a boolean true if the array contains the value.

Example:

CALL ARRAY_CONTAINS(('Hello', 'World'), 'Hello')

AUTOCOMMIT

```
AUTOCOMMIT()
```

Returns true if auto commit is switched on for this session.

Example:

AUTOCOMMIT()

CANCEL_SESSION

CANCEL_SESSION(sessionInt)

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)

Returns 'a' if the boolean expression is true, otherwise 'b'. Returns the same data type as the parameter.

Example:

CASEWHEN(ID=1, 'A', 'B')

CAST

CAST(value AS dataType)

Converts a value to another data type. The following conversion rules are used: When converting a number to a boolean, 0 is false and every other value is true. When converting a boolean to a number, false is 0 and true is 1. When converting a number to a number of another type, the value is checked for overflow. When converting a number to binary, the number of bytes matches the precision. When converting a string to binary, it is hex encoded (every byte two characters); a hex string can be converted to a number by first converting it to binary. If a direct conversion is not possible, the value is first converted to a string.

Example:

```
CAST(NAME AS INT);
CAST(65535 AS BINARY);
CAST(CAST('FFFF' AS BINARY) AS INT);
```

COALESCE

{ COALESCE | NVL } (aValue, bValue [,...])

Returns the first value that is not null.

Example:

COALESCE(A, B, C)

CONVERT

```
CONVERT(value, dataType)
```

Converts a value to another data type.

Example:

CONVERT(NAME, INT)

CURRVAL

```
CURRVAL( [ schemaName, ] sequenceString )
```

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). This method returns a long.

Example:

CURRVAL('TEST_SEQ')

CSVREAD

```
CSVREAD(fileNameString [, columnsString [, csvOptions ] ] )
```

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, otherwise (or if they are set to NULL) the first line of the file is interpreted as the column names. In that case, column names that contain no special characters (only letters, '_', and digits; similar to the rule for Java identifiers) are considered case insensitive. Other column names are case sensitive, that means you need to use quoted identifiers (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.

The BOM (the byte-order-mark) character 0xfeff at the beginning of the file is ignored.

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.csv. To read a stream from the classpath, use the prefix classpath:. To read from HTTP, use the prefix http: (as in a browser).

For performance reason, CSVREAD should not be used inside a join. Instead, import the data first (possibly into a temporary table) and then use the table.

Admin rights are required to execute this command.

Example:

```
CALL CSVREAD('test.csv');
-- Read a file containing the columns ID, NAME with
CALL CSVREAD('test2.csv', 'ID|NAME', 'charset=UTF-8 fieldSeparator=|');
SELECT * FROM CSVREAD('data/test.csv', null, 'rowSeparator=;');
-- Read a tab-separated file
SELECT * FROM CSVREAD('data/test.tsv', null, 'rowSeparator=' || CHAR(9));
SELECT "Last Name" FROM CSVREAD('address.csv');
SELECT "Last Name" FROM CSVREAD('classpath:/org/acme/data/address.csv');
```

CSVWRITE

```
CSVWRITE ( fileNameString, queryString [, csvOptions [, lineSepString] ] )
```

Writes a CSV (comma separated values). The file is overwritten if it exists. If only a file name is specified, it will be written to the current working directory. 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 (system property line.separator).

The returned value is the number or rows written. Admin rights are required to execute this command.

Example:

```
CALL CSVWRITE('data/test.csv', 'SELECT * FROM TEST');
CALL CSVWRITE('data/test2.csv', 'SELECT * FROM TEST', 'charset=UTF-8 fieldSeparator=|');
-- Write a tab-separated file
CALL CSVWRITE('data/test.tsv', 'SELECT * FROM TEST', 'charset=UTF-8 fieldSeparator=' || CHAR(9));
```

DATABASE

```
DATABASE()
```

Returns the name of the database.

Example:

```
CALL DATABASE();
```

DATABASE_PATH

```
DATABASE_PATH()
```

Returns the directory of the database files and the database name, if it is file based. Returns NULL otherwise.

Example:

```
CALL DATABASE_PATH();
```

DECODE

```
DECODE(value, whenValue, thenValue [,...])
```

Returns the first matching value. NULL is considered to match NULL. If no match was found, then NULL or the last parameter (if the parameter count is even) is returned. This function is provided for Oracle compatibility (see there for details).

Example:

```
CALL DECODE(RAND()>0.5, 0, 'Red', 1, 'Black');
```

FILE_READ

```
FILE_READ(fileNameString [,encodingString])
```

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. To read a stream from the classpath, use the prefix classpath:.

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 [,...])
```

Returns the largest value that is not NULL, or NULL if all values are NULL.

Example:

```
CALL GREATEST(1, 2, 3);
```

IDENTITY

```
IDENTITY()
```

Returns the last inserted identity value for this session. This value changes whenever a new sequence number was generated, even within a trigger or Java function. See also SCOPE_IDENTITY. This method returns a long.

Example:

```
CALL IDENTITY();
```

IFNULL

```
IFNULL(aValue, bValue)
```

Returns the value of 'a' if it is not null, otherwise 'b'.

Example:

```
CALL IFNULL(NULL, '');
```

LEAST

```
LEAST(aValue, bValue [,...])
```

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()
```

Returns the current lock mode. See SET LOCK_MODE. This method returns an int.

Example:

CALL LOCK_MODE();

LOCK_TIMEOUT

```
LOCK_TIMEOUT()
```

Returns the lock timeout of the current session (in milliseconds).

Example:

LOCK_TIMEOUT()

LINK_SCHEMA

```
LINK_SCHEMA(targetSchemaString, driverString, urlString,
userString, passwordString, sourceSchemaString)
```

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 in the form of a result set. Admin rights are required to execute this command.

Example:

CALL LINK_SCHEMA('TEST2', '', 'jdbc:h2:test2', 'sa', 'sa', 'PUBLIC');

MEMORY_FREE

```
MEMORY_FREE()
```

Returns the free memory in KB (where 1024 bytes is a KB). This method returns an int. The garbage is run before returning the value. Admin rights are required to execute this command.

Example:

MEMORY_FREE()

MEMORY_USED

```
MEMORY_USED()
```

Returns the used memory in KB (where 1024 bytes is a KB). This method returns an int. The garbage is run before returning the value. Admin rights are required to execute this command.

Example:

MEMORY_USED()

NEXTVAL

```
NEXTVAL ( [ schemaName, ] sequenceString )
```

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). This method returns a long.

Example:

NEXTVAL('TEST_SEQ')

NULLIF

```
NULLIF(aValue, bValue)
```

Returns NULL if 'a' is equals to 'b', otherwise 'a'.

Example:

NULLIF(A, B)

NVL2

```
NVL2(testValue, aValue, bValue)
```

If the test value is null, then 'b' is returned. Otherwise, 'a' is returned. The data type of the returned value is the data type of 'a' if this is a text type.

Example:

NVL2(X, 'not null', 'null')

READONLY

```
READONLY()
```

Returns true if the database is read-only.

Example:

READONLY()

ROWNUM

```
{ ROWNUM() } | { ROW_NUMBER() OVER() }
```

Returns the number of the current row. This method returns a long. It 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, but after evaluating index conditions (even when the index conditions are specified in an outer query). 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);
SELECT ID FROM (SELECT T.*, ROWNUM AS R FROM TEST T) WHERE R BETWEEN 2 AND 3;
```

SCHEMA

```
SCHEMA()
```

Returns the name of the default schema for this session.

Example:

```
CALL SCHEMA()
```

SCOPE_IDENTITY

```
SCOPE_IDENTITY()
```

Returns the last inserted identity value for this session for the current scope. Changes within triggers and Java functions are ignored. See also IDENTITY(). This method returns a long.

Example:

```
CALL SCOPE_IDENTITY();
```

SESSION_ID

```
SESSION_ID()
```

Returns the unique session id number for the current database connection. This id stays the same while the connection is open. This method returns an int. The database engine may re-use a session id after the connection is closed.

Example:

```
CALL SESSION_ID()
```

SET

```
SET(@variableName, 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. When used in a subquery, not all rows might be read depending on the query plan. This can be used to implement running totals / cumulative sums.

Example:

```
SELECT X, SET(@I, IFNULL(@I, 0)+X) RUNNING_TOTAL FROM SYSTEM_RANGE(1, 10)
```

TABLE

```
{ TABLE | TABLE_DISTINCT } ( { name dataType = expression } [,...] )
```

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()
```

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. This method returns a string. The value is unique across database restarts (values are not re-used).

Example:

CALL TRANSACTION_ID()

TRUNCATE_VALUE

```
TRUNCATE_VALUE(value, precisionInt, forceBoolean)
```

Truncate a value to the required precision. The precision of the returned value may be a bit larger than requested, because fixed precision values are not truncated (unlike the numeric TRUNCATE method). Unlike CAST, the truncating a decimal value may lose precision if the force flag is set to true. The method returns a value with the same data type as the first parameter.

Example:

CALL TRUNCATE_VALUE(X, 10, TRUE);

USER

```
{ USER | CURRENT_USER } ()
```

Returns the name of the current user of this session.

Example:

CURRENT_USER()

CURRENT_DATE

```
{ CURRENT_DATE [ () ] | CURDATE() | SYSDATE | TODAY }
```

Returns the current date. This method always returns the same value within a transaction.

Example:

CURRENT_DATE()

CURRENT_TIME

```
{ CURRENT_TIME [ () ] | CURTIME() }
```

Returns the current time. This method always returns the same value within a transaction.

Example:

CURRENT_TIME()

CURRENT_TIMESTAMP

{ CURRENT_TIMESTAMP [([int])] | NOW([int]) }

Returns the current timestamp. The precision parameter for nanoseconds precision is optional. This method always returns the same value within a transaction.

Example:

CURRENT_TIMESTAMP()

DATEADD

{ DATEADD| TIMESTAMPADD } (unitString, addInt, 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. This method returns a timestamp.

Example:

DATEADD('MONTH', 1, DATE '2001-01-31')

DATEDIFF

{ DATEDIFF | TIMESTAMPDIFF } (unitString, aTimestamp, bTimestamp)

Returns the the number of crossed unit boundaries between two timestamps. This method returns a long. 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)

Returns the name of the day (in English).

Example:

DAYNAME(CREATED)

DAY_OF_MONTH

DAY_OF_MONTH(date)

Returns the day of the month (1-31).

Example:

DAY_OF_MONTH(CREATED)

DAY_OF_WEEK

```
DAY_OF_WEEK(date)
```

Returns the day of the week (1 means Sunday).

Example:

DAY_OF_WEEK(CREATED)

DAY_OF_YEAR

```
DAY_OF_YEAR(date)
```

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 )
```

Returns a specific value from a timestamps. This method returns an int.

Example:

EXTRACT(SECOND FROM CURRENT_TIMESTAMP)

FORMATDATETIME

```
FORMATDATETIME ( timestamp, formatString
[ , localeString [ , timeZoneString ] ] )
```

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. This method returns a string.

Example:

```
CALL FORMATDATETIME(TIMESTAMP '2001-02-03 04:05:06',
    'EEE, d MMM yyyy HH:mm:ss z', 'en', 'GMT')
```

HOUR

```
HOUR(timestamp)
```

Returns the hour (0-23) from a timestamp.

Example:

HOUR(CREATED)

MINUTE

```
MINUTE(timestamp)
```

Returns the minute (0-59) from a timestamp.

Example:

MINUTE(CREATED)

MONTH

```
MONTH(timestamp)
```

Returns the month (1-12) from a timestamp.

Example:

MONTH(CREATED)

MONTHNAME

```
MONTHNAME(date)
```

Returns the name of the month (in English).

Example:

MONTHNAME(CREATED)

PARSEDATETIME

```
PARSEDATETIME(string, formatString
[, localeString [, timeZoneString]])
```

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)
```

Returns the quarter (1-4) from a timestamp.

Example:

QUARTER(CREATED)

SECOND

```
SECOND(timestamp)
```

Returns the second (0-59) from a timestamp.

Example:

SECOND(CREATED)

WEEK

```
WEEK(timestamp)
```

Returns the week (1-53) from a timestamp. This method uses the current system locale.

Example:

WEEK(CREATED)

YEAR

```
YEAR(timestamp)
```

Returns the year from a timestamp.

Example:

YEAR(CREATED)

Data Types

Index

INT Type
BOOLEAN Type
TINYINT Type
SMALLINT Type
BIGINT Type
IDENTITY Type
DECIMAL Type
DOUBLE Type
REAL Type
TIME Type
DATE Type
TIMESTAMP Type
BINARY Type
OTHER Type
VARCHAR Type
VARCHAR_IGNORECASE Type
CHAR Type
BLOB Type
CLOB Type
UUID Type
ARRAY Type

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 }
```

A 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
```

A 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 time data type. The format is hh:mm:ss.

Mapped to java.sql.Time. When converted to a java.sql.Date, the date is set to 1970-01-01.

Example:

TIME

DATE Type

```
DATE
```

The date data type. The format is yyyy-MM-dd.

Mapped to java.sql.Date, with the time set to 00:00:00 (or to the next possible time if midnight doesn't exist for the given date and timezone due to a daylight saving change).

Example:

DATE

TIMESTAMP Type

```
{ TIMESTAMP | DATETIME | SMALLDATETIME }
```

The timestamp data type. 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 when 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 ) ]
```

A 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 loaded into 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. The precision is a size constraint; only the actual data is persisted.

The whole text is loaded into 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 ) ]
```

A Unicode String. 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.

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. To store values, use PreparedStatement.setBytes, setString, or setObject(uuid) (where uuid is a java.util.UUID). ResultSet.getObject will return a java.util.UUID.

Please note that using an index on randomly generated data will result on poor performance once there are millions of rows in a table. The reason is that the cache behavior is very bad with randomly distributed data. This is a problem for any database system.

For details, see the documentation of java.util.UUID.

Example:

UUID

ARRAY Type

ARRAY

An array of values. Mapped to java.lang.Object[] (arrays of any non-primitive type are also supported).

Use a value list (1, 2) or PreparedStatement.setObject(.., new Object[] {..}) to store values, and ResultSet.getObject(..) or ResultSet.getArray(..) to retrieve the values.

Example:

ARRAY

Build

Portability
Environment
Building the Software
Build Targets
Using Maven 2
Using Eclipse
Translating
Providing Patches
Reporting Problems or Requests
Automated Build
Generating Railroad Diagrams

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

To run this database, a Java Runtime Environment (JRE) version 1.5 or higher is required.

To create the database executables, the following software stack was used. To use this database, it is not required to install this software however.

- Mac OS X and Windows XP
- Sun JDK Version 1.5 and 1.6
- Eclipse Version 3.4
- Eclipse Plugins: Subclipse, Eclipse Checkstyle Plug-in, EclEmma Java Code Coverage 1.3.0
- Emma Java Code Coverage
- Mozilla Firefox
- OpenOffice
- NSIS 2.38 (Nullsoft Scriptable Install System)
- Maven

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
```

To run the build tool in shell mode, use the command line option - as in ./build.sh -.

Switching the Source Code

By default the source code uses Java 1.6 features, however Java 1.5 is supported as well. To switch the source code to the installed 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 file 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 file h2client.jar, go to the directory h2 and execute the following command:

```
build jarClient
```

Using Lucene 2 / 3

Both Apache Lucene 2 and Lucene 3 are supported. Currently Apache Lucene version 2.x is used by default for H2 version 1.2.x, and Lucene version 3.x is used by default for H2 version 1.3.x. To use a different version of Lucene when compiling, it needs to be specified as follows:

```
build -Dlucene=2 clean compile
```

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>1.3.168</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.

Maven Plugin to Start and Stop the TCP Server

A Maven plugin to start and stop the H2 TCP server is available from Laird Nelson at GitHub. To start the H2 server, use:

```
mvn com.edugility.h2-maven-plugin:1.0-SNAPSHOT:spawn
```

To stop the H2 server, use:

```
mvn com.edugility.h2-maven-plugin:1.0-SNAPSHOT:stop
```

Using Snapshot Version

To build a h2-*-SNAPSHOT.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>
```

Using Eclipse

To create an Eclipse project for H2, use the following steps:

- Install Subversion and Eclipse.
- Get the H2 source code from the Subversion repository:
  svn checkout http://h2database.googlecode.com/svn/trunk h2database-read-only
- Download all dependencies (Windows):
  build.bat download
- In Eclipse, create a new Java project from existing source code: File, New, Project, Java Project, Create project from existing source.
- Select the h2 folder, click Next and Finish.
- To resolve com.sun.javadoc import statements, you may need to manually add the file <java.home>/../lib/tools.jar to the build path.

Translating

The translation of this software is split into the following parts:

- H2 Console: src/main/org/h2/server/web/res/_text_*.prop
- Error messages: src/main/org/h2/res/_messages_*.prop

To translate the H2 Console, start it and select Preferences / Translate. After you are done, send the translated *.prop file to the Google Group. The web site is currently translated using Google.

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.
- A template of the Eclipse settings are in src/installer/eclipse.settings/*. If you want to use them, you need to copy them to the .settings directory. The formatting options (eclipseCodeStyle) are also included.
- 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/docsrc/help/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 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)."

Reporting Problems or Requests

Please consider the following checklist if you have a question, want to report a problem, or if you have a feature request:

- For bug reports, please provide a short, self contained, correct (compilable), example of the problem.
- Feature requests are always welcome, even if the feature is already on the roadmap. Your mail will help prioritize feature requests. If you urgently need a feature, consider providing a patch.
- Before posting problems, check the FAQ and do a Google search.
- When got an unexpected exception, please try the Error Analyzer tool. If this doesn't help, please report the problem, including the complete error message and stack trace, and the root cause stack trace(s).
- When sending source code, please use a public web clipboard such as Pastebin, Cl1p, or Mystic Paste to avoid formatting problems. Please keep test cases as simple and short as possible, but so that the problem can still be reproduced. As a template, use: HelloWorld.java. Method that simply call other methods should be avoided, as well as unnecessary exception handling. Please use the JDBC API and no external tools or libraries. The test should include all required initialization code, and should be started with the main method.
- For large attachments, use a public temporary storage such as Rapidshare.
- Google Group versus issue tracking: Use the Google Group for questions or if you are not sure it's a bug. If you are sure it's a bug, you can create an issue, but you don't need to (sending an email to the group is enough). Please note that only few people monitor the issue tracking system.
- For out-of-memory problems, please analyze the problem yourself first, for example using the command line option -XX:+HeapDumpOnOutOfMemoryError (to create a heap dump file on out of memory) and a memory analysis tool such as the Eclipse Memory Analyzer (MAT).
- It may take a few days to get an answers. Please do not double post.

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)

Generating Railroad Diagrams

The railroad diagrams of the SQL grammar are HTML, formatted as nested tables. The diagrams are generated as follows:

- The BNF parser (org.h2.bnf.Bnf) reads and parses the BNF from the file help.csv.
- The page parser (org.h2.server.web.PageParser) reads the template HTML file and fills in the diagrams.
- The rail images (one straight, four junctions, two turns) are generated using a simple Java application.

To generate railroad diagrams for other grammars, see the package org.h2.jcr. This package is used to generate the SQL-2 railroad diagrams for the JCR 2.0 specification.

History and Roadmap

Change Log
Roadmap
History of this Database Engine
Why Java
Supporters

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 wrote PointBase Micro, a commercial Java SQL database. At that point, he had to discontinue Hypersonic SQL. 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 code with Hypersonic SQL or HSQLDB. H2 is built from scratch.

Why Java

The main reasons to use 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 think Java is too slow for low level operations, but this is no longer true. Garbage collection for example is now faster than manual memory management.

Developing Java code is faster than developing C or C++ code. When using Java, most time can be spent on improving the algorithms instead of porting the code to different platforms or doing memory management. Features such as Unicode and network libraries are already built-in. In Java, writing secure code is easier because buffer overflows can not occur. Features such as reflection can be used for randomized testing.

Java is future proof: a lot of companies support Java. Java is now open source.

To increase the portability and ease of use, this software depends on very few libraries. Features that are not available in open source Java implementations (such as Swing) are not used, or only used for optional features.

Supporters

Many thanks for those who reported bugs, gave valuable feedback, spread the word, and translated this project. Also many thanks to the donors:

- Martin Wildam, Austria
- Code Lutin, France
- Code 42 Software, Inc., Minneapolis
- NetSuxxess GmbH, Germany

- Poker Copilot, Steve McLeod, Germany
- SkyCash, Poland
- Lumber-mill, Inc., Japan
- StockMarketEye, USA
- Eckenfelder GmbH & Co.KG, Germany
- Alessio Jacopo D'Adamo, Italy
- Ashwin Jayaprakash, USA
- Donald Bleyl, USA
- Frank Berger, Germany
- 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 Rojas, Chile
- Ontology Works, USA
- Pete Haidinyak, USA
- William Osmond, USA
- Joachim Ansorg, Germany
- Oliver Soerensen, Germany
- Christos Vasilakis, Greece
- Fyodor Kupolov, Denmark
- Jakob Jenkov, Denmark
- Stéphane Chartrand, Switzerland
- Glenn Kidd, USA
- Gustav Trede, Sweden
- Joonas Pulakka, Finland
- Bjorn Darri Sigurdsson, Iceland
- Iyama Jun, Japan
- Gray Watson, USA
- Erik Dick, Germany
- Pengxiang Shao, China
- Bilingual Marketing Group, USA
- Philippe Marschall, Switzerland
- Knut Staring, Norway
- Theis Borg, Denmark
- Joel A. Garringer, USA
- Olivier Chafik, France
- Rene Schwietzke, Germany
- Jalpesh Patadia, USA
- Takanori Kawashima, Japan
- Terrence JC Huang, China
- Jia Dong Huang, Australia
- Laurent van Roy, Belgium

Frequently Asked Questions

I Have a Problem or Feature Request
Are there Known Bugs? When is the Next Release?
Is this Database Engine Open Source?
Is Commercial Support Available?
How to Create a New Database?
How to Connect to a Database?
Where are the Database Files Stored?
What is the Size Limit (Maximum Size) of a Database?
Is it Reliable?
Why is Opening my Database Slow?
My Query is Slow
H2 is Very Slow
Column Names are Incorrect?
Float is Double?
Is the GCJ Version Stable? Faster?
How to Translate this Project?
How to Contribute to this Project?

I Have a Problem or Feature Request

Please read the support checklist.

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:

- When opening a database file in a timezone that has different daylight saving rules: the time part of dates where the daylight saving doesn't match will differ. This is not a problem within regions that use the same rules (such as, within USA, or within Europe), even if the timezone itself is different. As a workaround, export the database to a SQL script using the old timezone, and create a new database in the new timezone. This problem does not occur when using the system property "h2.storeLocalTime" (however such database files are not compatible with older versions of H2).
- Apache Harmony: there seems to be a bug in Harmony that affects H2. See HARMONY-6505.
- 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=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. This problem is fixed in H2 version 1.3.
- 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.

For a complete list, see Open Issues.

Is this Database Engine Open Source?

Yes. It is free to use and distribute, and the source code is included. See also under license.

Is Commercial Support Available?

Yes, commercial support is available, see Commercial Support.

How to Create a New Database?

By default, a new database is automatically created if it does not yet exist. See Creating New Databases.

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?

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 higher than 80% for each package. 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 or FileDescriptor.sync() using LOG=0 or LOG=1.
- Using the transaction isolation level READ_UNCOMMITTED (LOCK_MODE 0) while at the same time using multiple connections.
- Disabling database file protection using (setting FILE_LOCK to 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.

This database is well tested using automated test cases. The tests run every night and run for more than one hour. But not all areas of this database are equally well tested. When using one of the following features for production, please ensure your use case is well tested (if possible with automated test cases). The areas that are not well tested are:

- 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 file locking method 'Serialized'.
- Cluster mode, 2-phase commit, savepoints.
- 24/7 operation.
- Fulltext search.
- Operations on LOBs over 2 GB.
- The optimizer may not always select the best plan.
- Using the ICU4J collator.

Areas considered experimental are:

- The PostgreSQL server
- Multi-threading within the engine using SET MULTI_THREADED=1.

- Compatibility modes for other databases (only some features are implemented).
- The soft reference cache (CACHE_TYPE=SOFT_LRU). It might not improve performance, and out of memory issues have been reported.

Some users have reported that after a power failure, the database cannot be opened sometimes. In this case, use a backup of the database or the Recover tool. Please report such problems. The plan is that the database automatically recovers in all situations.

Why is Opening my Database Slow?

To find out what the problem is, use the H2 Console and click on "Test Connection" instead of "Login". After the "Login Successful" appears, click on it (it's a link). This will list the top stack traces. Then either analyze this yourself, or post those stack traces in the Google Group.

Other possible reasons are: the database is very big (many GB), or contains linked tables that are slow to open.

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.

H2 is Very Slow

By default, H2 closes the database when the last connection is closed. If your application closes the only connection after each operation, the database is opened and closed a lot, which is quite slow. There are multiple ways to solve this problem, see Database Performance Tuning.

Column Names are Incorrect?

For the query SELECT ID AS X FROM TEST the method ResultSetMetaData.getColumnName() returns ID, I expect it to return X. What's wrong?

This is not a bug. According the the JDBC specification, the method ResultSetMetaData.getColumnName() should return the name of the column and not the alias name. If you need the alias name, use ResultSetMetaData.getColumnLabel(). Some other database don't work like this yet (they don't follow the JDBC specification). If you need compatibility with those databases, use the Compatibility Mode, or append ;ALIAS_COLUMN_NAME=TRUE to the database URL.

This also applies to DatabaseMetaData calls that return a result set. The columns in the JDBC API are column labels, not column names.

Float is Double?

For a table defined as CREATE TABLE TEST(X FLOAT) the method ResultSet.getObject() returns a java.lang.Double, I expect it to return a java.lang.Float. What's wrong?

This is not a bug. According the the JDBC specification, the JDBC data type FLOAT is equivalent to DOUBLE, and both are mapped to java.lang.Double. See also Mapping SQL and Java Types - 8.3.10 FLOAT.

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.

How to Contribute to this Project?

There are various way to help develop an open source project like H2. The first step could be to translate the error messages and the GUI to your native language. Then, you could provide patches. Please start with small patches. That could be adding a test case to improve the code coverage (the target code coverage for this project is 90%, higher is better). You will have to develop, build and run the tests. Once you are familiar with the code, you could implement missing features from the feature request list. I suggest to start with very small features that are easy to implement. Keep in mind to provide test cases as well.