Translate
Search:

Home
Download
Cheat Sheet

Documentation
Quickstart
Installation
Tutorial
Features
Performance
Advanced

Reference
Commands
Functions
• Aggregate • Window

Data Types
SQL Grammar
System Tables
Javadoc
PDF (1.5 MB)

Support
FAQ
Error Analyzer
Google Group (English)
Google Group (Japanese)
Google Group (Chinese)

Appendix
History & Roadmap
License
Build
Links
MVStore
Architecture

Performance

Performance Comparison

In many cases H2 is faster than other (open source and not open source) database engines. Please note this is n
a single connection benchmark run on one computer, with many very simple operations running against the datab
This benchmark does not include very complex queries. The embedded mode of H2 is faster than the client-serve
mode because the per-statement overhead is greatly reduced.

Embedded

Test Case	Unit	H2	HSQLDB	Derby
Simple: Init	ms	1019	1907	8280
Simple: Query (random)	ms	1304	873	1912
Simple: Query (sequential)	ms	835	1839	5415
Simple: Update (sequential)	ms	961	2333	21759
Simple: Delete (sequential)	ms	950	1922	32016
Simple: Memory Usage	MB	21	10	8
BenchA: Init	ms	919	2133	7528
BenchA: Transactions	ms	1219	2297	8541
BenchA: Memory Usage	MB	12	15	7
BenchB: Init	ms	905	1993	8049
BenchB: Transactions	ms	1091	583	1165
BenchB: Memory Usage	MB	17	11	8
BenchC: Init	ms	2491	4003	8064
BenchC: Transactions	ms	1979	803	2840
BenchC: Memory Usage	MB	19	22	9
Executed statements	#	1930995	1930995	1930995
Total time	ms	13673	20686	105569
Statements per second	#	141226	93347	18291

Client-Server

Test Case	Unit	H2 (Server)	HSQLDB	Derby	PostgreSQL	MySQL
Simple: Init	ms	16338	17198	27860	30156	29409
Simple: Query (random)	ms	3399	2582	6190	3315	3342
Simple: Query (sequential)	ms	21841	18699	42347	30774	32611
Simple: Update (sequential)	ms	6913	7745	28576	32698	11350
Simple: Delete (sequential)	ms	8051	9751	42202	44480	16555
Simple: Memory Usage	MB	22	11	9	0	1
BenchA: Init	ms	12996	14720	24722	26375	26060
BenchA: Transactions	ms	10134	10250	18452	21453	15877
BenchA: Memory Usage	MB	13	15	9	0	1
BenchB: Init	ms	15264	16889	28546	31610	29747

BenchB: Transactions	ms	3017	3376	1842	2771	1433
BenchB: Memory Usage	MB	17	12	11	1	1
BenchC: Init	ms	14020	10407	17655	19520	17532
BenchC: Transactions	ms	5076	3160	6411	6063	4530
BenchC: Memory Usage	MB	19	21	11	1	1
Executed statements	#	1930995	1930995	1930995	1930995	1930995
Total time	ms	117049	114777	244803	249215	188446
Statements per second	#	16497	16823	7887	7748	10246

Benchmark Results and Comments

H2

Version 1.4.177 (2014-04-12) was used for the test. For most operations, the performance of H2 is about the sam 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 2.3.2 was used for the test. Cached tables are used in this test (hsqldb.default_table_type=cached), and write delay is 1 second (SET WRITE_DELAY 1).

Derby

Version 10.10.1.1 was used for the test. Derby is clearly the slowest embedded database in this test. This seems a structural problem, because all operations are really slow. It will be hard for the developers of Derby to improve 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 performan 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. Enabli this setting improves performance by a factor of 2.6 (embedded mode) or 1.4 (server mode). Even if enabled, De still less than half as fast as H2 in default mode.

PostgreSQL

Version 9.1.5 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 memory usage of the JDBC driver is measured.

MySQL

Version 5.1.65-log 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), My is slow (around 140 statements per second in this test) because it tries to flush the data to disk for each commit. F 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 mer 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 performa 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 teste 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 connec

Real-World Tests

Good benchmarks emulate real-world use cases. This benchmark includes 4 test cases: BenchSimple uses one t and many small updates / deletes. BenchA is similar to the TPC-A test, but single connection / single threaded (s 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 t database engine). However MySQL and PostgreSQL are not Java databases and cannot be embedded into a Ja 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 JVM used is 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 bench 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 improvec using a bigger cache, but the amount of memory is limited. HSQLDB tables are kept fully in memory by default; th benchmark uses 'disk based' tables for all databases. Unfortunately, it is not so easy to calculate the memory usa PostgreSQL and MySQL, because they run in a different process than the test. This benchmark currently does nc 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) tr enforce this by default by calling fsync() to flush the buffers, but most hard drives don't actually flush all data. Ca the method slows down transaction commit a lot, but doesn't always make data durable. When comparing the res it is important to think about the effect. Many database suggest to 'batch' operations when possible. This benchm switches off autocommit when loading the data, and calls commit after each 1000 inserts. However many applicat need 'short' transactions at runtime (a commit after each update). This benchmark commits after each update / de in the simple benchmark, and after each business transaction in the other benchmarks. For databases that suppo 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) conne 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 db4o. This test was not run for a longer time, so please be aware that the results below are for older database ver (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 n 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` in the file `Jdbc.properties` .
- HSQLDB keeps the database open between tests, while H2 closes the database (losing all the cache). To cha 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 PolePo 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 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 databa

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 f are not scanned for viruses. The database engine never interprets the data stored in the files as programs, that m even if somebody would store a virus in a database file, this would be harmless (when the virus does not run, it c 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 cre 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 ca 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 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.

Index Hints

If you have determined that H2 is not using the optimal index for your query, you can use index hints to force H2 to use specific indexes.

```
SELECT * FROM TEST USE INDEX (index_name_1, index_name_2) WHERE X=1
```

Only indexes in the list will be used when choosing an index to use on the given table. There is no significance to the order in this list.

It is possible that no index in the list is chosen, in which case a full table scan will be used.

An empty list of index names forces a full table scan to be performed.

Each index in the list must exist.

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 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 row 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 indexes 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 NAME, FIRSTNAME) and an index on each column, then two indexes could be used for the query SELECT * FR TEST WHERE NAME='A' AND FIRSTNAME='B' , the index on NAME or the index on FIRSTNAME. It is not poss to use both indexes at the same time. Which index is used depends on the selectivity of the column. The selectivi 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 sh 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. 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 datab using CREATE MEMORY TABLE . In many cases, the rows itself will also be kept in-memory. Please note this m 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 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 the following example:

```
PreparedStatement prep = conn.prepareStatement(
    "SELECT * FROM TEST WHERE ID = ANY(?)");
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 optimizatio 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 use tree pages are split at the insertion point. To use this optimization, add SORTED before the SELECT statemen

```
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 applica is slow). Blind optimization or optimization based on guesses should be avoided, because usually it is not an effic strategy. There are various ways to analyze an application. Sometimes two implementations can be compared us 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 thread dumps. To do that, first run jps -l to get the process id, and then run jstack <pid> or kill -QUIT <pid> (Li 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 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 to 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 similar to the profiling data generated when using java -Xrunhprof . For this to work, the trace level needs to be 2 higher (TRACE_LEVEL_FILE=2). The easiest way to set the trace level is to append the setting to the database for example: jdbc:h2:~/test;TRACE_LEVEL_FILE=2 or jdbc:h2:tcp://localhost/~/test;TRACE_LEVEL_FILE=2 . A example, execute 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 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...
--   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 s 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 t database first processes the table INVOICE (using the primary key). For each row, it will additionally check that 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 will actually execute the query, unlike EXPLAIN which only prepares it. The following query scanned 1000 rows, to do that had to read 85 pages from the data area of the table. Running the query twice will not list the pages rea 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 i 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 is used.

For queries of the form SELECT COUNT(*), MIN(ID), MAX(ID) FROM TEST , the query plan includes the line /* 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 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 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 BIG then the value of this column is the row id, otherwise the database generates the row id automatically. There is a 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. Please note the row id is not available until after th row was added (that means, it can not be used in computed columns or constraints). If the query condition does r contain the row id (and if no other index can be used), then all rows of the table are scanned. A table scan iterates all rows in the table, in the order of the row id. To find out what strategy the database uses to retrieve the data, us 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) rea the remaining columns in the main data table (retrieved using the row id). An index on city, name, and first name (column index) allows to quickly search for rows when the city, name, and first name are known. If only the city an 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 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 o

```
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 cit

makes sense to write two queries and combine then using UNION . In this case, each individual query uses a dif
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 TABL
... AS SELECT ... is faster than CREATE TABLE(...); INSERT INTO ... SELECT ... .