```
/*
* Copyright 2004-2013 H2 Group. Multiple-Licensed under the H2 License,
* Version 1.0, and under the Eclipse Public License, Version 1.0
* (http://h2database.com/html/license.html).
* Initial Developer: H2 Group
*/
package org.h2.expression;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Reader;
import java.sql.Connection;
import java.sql.Date;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Time;
import java.sql.Timestamp;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.HashMap;
import java.util.Locale;
import java.util.TimeZone;
import java.util.regex.PatternSyntaxException;
import org.h2.command.Command;
import org.h2.command.Parser;
import org.h2.constant.ErrorCode;
import org.h2.engine.Constants;
import org.h2.engine.Database;
import org.h2.engine.Mode;
import org.h2.engine.Session;
import org.h2.message.DbException;
import org.h2.mvstore.DataUtils;
import org.h2.schema.Schema;
import org.h2.schema.Sequence;
import org.h2.security.BlockCipher;
import org.h2.security.CipherFactory;
import org.h2.security.SHA256;
import org.h2.store.fs.FileUtils;
import org.h2.table.Column;
import org.h2.table.ColumnResolver;
import org.h2.table.LinkSchema;
import org.h2.table.Table;
import org.h2.table.TableFilter;
import org.h2.tools.CompressTool;
import org.h2.tools.Csv;
import org.h2.util.AutoCloseInputStream;
import org.h2.util.DateTimeUtils;
import org.h2.util.JdbcUtils;
```

```
import org.h2.util.MathUtils;
import org.h2.util.New;
import org.h2.util.StatementBuilder;
import org.h2.util.StringUtils;
import org.h2.util.Utils;
import org.h2.value.DataType;
import org.h2.value.Value;
import org.h2.value.ValueArray;
import org.h2.value.ValueBoolean;
import org.h2.value.ValueBytes;
import org.h2.value.ValueDate;
import org.h2.value.ValueDouble;
import org.h2.value.ValueInt;
import org.h2.value.ValueLong;
import org.h2.value.ValueNull;
import org.h2.value.ValueResultSet;
import org.h2.value.ValueString;
import org.h2.value.ValueTime;
import org.h2.value.ValueTimestamp;
import org.h2.value.ValueUuid;
/**
* This class implements most built-in functions of this database.
*/
public class Function extends Expression implements FunctionCall {
public static final int ABS = 0, ACOS = 1, ASIN = 2, ATAN = 3, ATAN2 =
4, BITAND = 5, BITOR = 6, BITXOR = 7,
CEILING = 8, COS = 9, COT = 10, DEGREES = 11, EXP = 12, FLOOR =
13, LOG = 14, LOG10 = 15, MOD = 16,
PI = 17, POWER = 18, RADIANS = 19, RAND = 20, ROUND = 21,
ROUNDMAGIC = 22, SIGN = 23, SIN = 24, SQRT = 25,
TAN = 26, TRUNCATE = 27, SECURE_RAND = 28, HASH = 29, ENCRYPT =
30, DECRYPT = 31, COMPRESS = 32,
EXPAND = 33, ZERO = 34, RANDOM_UUID = 35, COSH = 36, SINH = 37,
TANH = 38, LN = 39;
public static final int ASCII = 50, BIT_LENGTH = 51, CHAR = 52,
CHAR_LENGTH = 53, CONCAT = 54, DIFFERENCE = 55,
HEXTORAW = 56, INSERT = 57, INSTR = 58, LCASE = 59, LEFT = 60,
LENGTH = 61, LOCATE = 62, LTRIM = 63,
OCTET_LENGTH = 64, RAWTOHEX = 65, REPEAT = 66, REPLACE = 67,
RIGHT = 68, RTRIM = 69, SOUNDEX = 70,
SPACE = 71, SUBSTR = 72, SUBSTRING = 73, UCASE = 74, LOWER =
75, UPPER = 76, POSITION = 77, TRIM = 78,
STRINGENCODE = 79, STRINGDECODE = 80, STRINGTOUTF8 = 81,
UTF8TOSTRING = 82, XMLATTR = 83, XMLNODE = 84,
XMLCOMMENT = 85, XMLCDATA = 86, XMLSTARTDOC = 87, XMLTEXT = 88,
REGEXP_REPLACE = 89, RPAD = 90, LPAD = 91,
CONCAT_WS = 92;
public static final int CURDATE = 100, CURTIME = 101, DATE_ADD = 102,
DATE_DIFF = 103, DAY_NAME = 104,
```

```
DAY_OF_MONTH = 105, DAY_OF_WEEK = 106, DAY_OF_YEAR = 107, HOUR
= 108, MINUTE = 109, MONTH = 110, MONTH_NAME = 111,
NOW = 112, QUARTER = 113, SECOND = 114, WEEK = 115, YEAR = 116,
CURRENT_DATE = 117, CURRENT_TIME = 118,
CURRENT_TIMESTAMP = 119, EXTRACT = 120, FORMATDATETIME = 121,
PARSEDATETIME = 122,
ISO_YEAR = 123, ISO_WEEK = 124, ISO_DAY_OF_WEEK = 125;
public static final int DATABASE = 150, USER = 151, CURRENT_USER = 152,
IDENTITY = 153, SCOPE_IDENTITY = 154,
AUTOCOMMIT = 155, READONLY = 156, DATABASE_PATH = 157,
LOCK_TIMEOUT = 158, DISK_SPACE_USED = 159;
public static final int IFNULL = 200, CASEWHEN = 201, CONVERT = 202,
CAST = 203, COALESCE = 204, NULLIF = 205,
CASE = 206, NEXTVAL = 207, CURRVAL = 208, ARRAY_GET = 209,
CSVREAD = 210, CSVWRITE = 211,
MEMORY_FREE = 212, MEMORY_USED = 213, LOCK_MODE = 214, SCHEMA =
215, SESSION_ID = 216, ARRAY_LENGTH = 217,
LINK_SCHEMA = 218, GREATEST = 219, LEAST = 220, CANCEL_SESSION
= 221, SET = 222, TABLE = 223, TABLE_DISTINCT = 224,
FILE_READ = 225, TRANSACTION_ID = 226, TRUNCATE_VALUE = 227,
NVL2 = 228, DECODE = 229, ARRAY_CONTAINS = 230;
/**
* This is called H2VERSION() and not VERSION(), because we return a
fake value
* for VERSION() when running under the PostgreSQL ODBC driver.
*/
public static final int H2VERSION = 231;
public static final int ROW_NUMBER = 300;
private static final int VAR_ARGS = -1;
private static final long PRECISION_UNKNOWN = -1;
private static final HashMap<String, FunctionInfo> FUNCTIONS =
New.hashMap();
private static final HashMap<String, Integer> DATE_PART =
New.hashMap();
private static final char[] SOUNDEX_INDEX = new char[128];
protected Expression[] args;
private final FunctionInfo info;
private ArrayList<Expression> varArgs;
private int dataType, scale;
private long precision = PRECISION_UNKNOWN;
private int displaySize;
private final Database database;
static {
// DATE_PART
DATE_PART.put("SQL_TSI_YEAR", Calendar.YEAR);
DATE_PART.put("YEAR", Calendar.YEAR);
DATE_PART.put("YYYY", Calendar.YEAR);
DATE_PART.put("YY", Calendar.YEAR);
DATE_PART.put("SQL_TSI_MONTH", Calendar.MONTH);
```

```
DATE_PART.put("MONTH", Calendar.MONTH);
DATE_PART.put("MM", Calendar.MONTH);
DATE_PART.put("M", Calendar.MONTH);
DATE_PART.put("SQL_TSI_WEEK", Calendar.WEEK_OF_YEAR);
DATE_PART.put("WW", Calendar.WEEK_OF_YEAR);
DATE_PART.put("WK", Calendar.WEEK_OF_YEAR);
DATE_PART.put("WEEK", Calendar.WEEK_OF_YEAR);
DATE_PART.put("DAY", Calendar.DAY_OF_MONTH);
DATE_PART.put("DD", Calendar.DAY_OF_MONTH);
DATE_PART.put("D", Calendar.DAY_OF_MONTH);
DATE_PART.put("SQL_TSI_DAY", Calendar.DAY_OF_MONTH);
DATE_PART.put("DAYOFYEAR", Calendar.DAY_OF_YEAR);
DATE_PART.put("DAY_OF_YEAR", Calendar.DAY_OF_YEAR);
DATE_PART.put("DY", Calendar.DAY_OF_YEAR);
DATE_PART.put("DOY", Calendar.DAY_OF_YEAR);
DATE_PART.put("SQL_TSI_HOUR", Calendar.HOUR_OF_DAY);
DATE_PART.put("HOUR", Calendar.HOUR_OF_DAY);
DATE_PART.put("HH", Calendar.HOUR_OF_DAY);
DATE_PART.put("SQL_TSI_MINUTE", Calendar.MINUTE);
DATE_PART.put("MINUTE", Calendar.MINUTE);
DATE_PART.put("MI", Calendar.MINUTE);
DATE_PART.put("N", Calendar.MINUTE);
DATE_PART.put("SQL_TSI_SECOND", Calendar.SECOND);
DATE_PART.put("SECOND", Calendar.SECOND);
DATE_PART.put("SS", Calendar.SECOND);
DATE_PART.put("S", Calendar.SECOND);
DATE_PART.put("MILLISECOND", Calendar.MILLISECOND);
DATE_PART.put("MS", Calendar.MILLISECOND);
// SOUNDEX_INDEX
String index = "7AEIOUY8HW1BFPV2CGJKQSXZ3DT4L5MN6R";
char number = 0;
for (int i = 0, length = index.length(); i < length; i++) {
char c = index.charAt(i);
if (c < '9') {
number = c;
} else {
SOUNDEX_INDEX[c] = number;
SOUNDEX_INDEX[Character.toLowerCase(c)] = number;
}
}
// FUNCTIONS
addFunction("ABS", ABS, 1, Value.NULL);
addFunction("ACOS", ACOS, 1, Value.DOUBLE);
addFunction("ASIN", ASIN, 1, Value.DOUBLE);
addFunction("ATAN", ATAN, 1, Value.DOUBLE);
addFunction("ATAN2", ATAN2, 2, Value.DOUBLE);
addFunction("BITAND", BITAND, 2, Value.LONG);
addFunction("BITOR", BITOR, 2, Value.LONG);
addFunction("BITXOR", BITXOR, 2, Value.LONG);
```

```
addFunction("CEILING", CEILING, 1, Value.DOUBLE);
addFunction("CEIL", CEILING, 1, Value.DOUBLE);
addFunction("COS", COS, 1, Value.DOUBLE);
addFunction("COSH", COSH, 1, Value.DOUBLE);
addFunction("COT", COT, 1, Value.DOUBLE);
addFunction("DEGREES", DEGREES, 1, Value.DOUBLE);
addFunction("EXP", EXP, 1, Value.DOUBLE);
addFunction("FLOOR", FLOOR, 1, Value.DOUBLE);
addFunction("LOG", LOG, 1, Value.DOUBLE);
addFunction("LN", LN, 1, Value.DOUBLE);
addFunction("LOG10", LOG10, 1, Value.DOUBLE);
addFunction("MOD", MOD, 2, Value.LONG);
addFunction("PI", PI, 0, Value.DOUBLE);
addFunction("POWER", POWER, 2, Value.DOUBLE);
addFunction("RADIANS", RADIANS, 1, Value.DOUBLE);
// RAND without argument: get the next value
// RAND with one argument: seed the random generator
addFunctionNotDeterministic("RAND", RAND, VAR_ARGS, Value.DOUBLE);
addFunctionNotDeterministic("RANDOM", RAND, VAR_ARGS,
Value.DOUBLE);
addFunction("ROUND", ROUND, VAR_ARGS, Value.DOUBLE);
addFunction("ROUNDMAGIC", ROUNDMAGIC, 1, Value.DOUBLE);
addFunction("SIGN", SIGN, 1, Value.INT);
addFunction("SIN", SIN, 1, Value.DOUBLE);
addFunction("SINH", SINH, 1, Value.DOUBLE);
addFunction("SQRT", SQRT, 1, Value.DOUBLE);
addFunction("TAN", TAN, 1, Value.DOUBLE);
addFunction("TANH", TANH, 1, Value.DOUBLE);
addFunction("TRUNCATE", TRUNCATE, VAR_ARGS, Value.NULL);
// same as TRUNCATE
addFunction("TRUNC", TRUNCATE, VAR_ARGS, Value.NULL);
addFunction("HASH", HASH, 3, Value.BYTES);
addFunction("ENCRYPT", ENCRYPT, 3, Value.BYTES);
addFunction("DECRYPT", DECRYPT, 3, Value.BYTES);
addFunctionNotDeterministic("SECURE_RAND", SECURE_RAND, 1,
Value.BYTES);
addFunction("COMPRESS", COMPRESS, VAR_ARGS, Value.BYTES);
addFunction("EXPAND", EXPAND, 1, Value.BYTES);
addFunction("ZERO", ZERO, 0, Value.INT);
addFunctionNotDeterministic("RANDOM_UUID", RANDOM_UUID, 0,
Value.UUID);
addFunctionNotDeterministic("SYS_GUID", RANDOM_UUID, 0,
Value.UUID);
// string
addFunction("ASCII", ASCII, 1, Value.INT);
addFunction("BIT_LENGTH", BIT_LENGTH, 1, Value.LONG);
addFunction("CHAR", CHAR, 1, Value.STRING);
addFunction("CHR", CHAR, 1, Value.STRING);
addFunction("CHAR_LENGTH", CHAR_LENGTH, 1, Value.INT);
```

```
// same as CHAR_LENGTH
addFunction("CHARACTER_LENGTH", CHAR_LENGTH, 1, Value.INT);
addFunctionWithNull("CONCAT", CONCAT, VAR_ARGS, Value.STRING);
addFunctionWithNull("CONCAT_WS", CONCAT_WS, VAR_ARGS,
Value.STRING);
addFunction("DIFFERENCE", DIFFERENCE, 2, Value.INT);
addFunction("HEXTORAW", HEXTORAW, 1, Value.STRING);
addFunctionWithNull("INSERT", INSERT, 4, Value.STRING);
addFunction("LCASE", LCASE, 1, Value.STRING);
addFunction("LEFT", LEFT, 2, Value.STRING);
addFunction("LENGTH", LENGTH, 1, Value.LONG);
// 2 or 3 arguments
addFunction("LOCATE", LOCATE, VAR_ARGS, Value.INT);
// alias for MSSQLServer
addFunction("CHARINDEX", LOCATE, VAR_ARGS, Value.INT);
// same as LOCATE with 2 arguments
addFunction("POSITION", LOCATE, 2, Value.INT);
addFunction("INSTR", INSTR, VAR_ARGS, Value.INT);
addFunction("LTRIM", LTRIM, VAR_ARGS, Value.STRING);
addFunction("OCTET_LENGTH", OCTET_LENGTH, 1, Value.LONG);
addFunction("RAWTOHEX", RAWTOHEX, 1, Value.STRING);
addFunction("REPEAT", REPEAT, 2, Value.STRING);
addFunction("REPLACE", REPLACE, VAR_ARGS, Value.STRING);
addFunction("RIGHT", RIGHT, 2, Value.STRING);
addFunction("RTRIM", RTRIM, VAR_ARGS, Value.STRING);
addFunction("SOUNDEX", SOUNDEX, 1, Value.STRING);
addFunction("SPACE", SPACE, 1, Value.STRING);
addFunction("SUBSTR", SUBSTR, VAR_ARGS, Value.STRING);
addFunction("SUBSTRING", SUBSTRING, VAR_ARGS, Value.STRING);
addFunction("UCASE", UCASE, 1, Value.STRING);
addFunction("LOWER", LOWER, 1, Value.STRING);
addFunction("UPPER", UPPER, 1, Value.STRING);
addFunction("POSITION", POSITION, 2, Value.INT);
addFunction("TRIM", TRIM, VAR_ARGS, Value.STRING);
addFunction("STRINGENCODE", STRINGENCODE, 1, Value.STRING);
addFunction("STRINGDECODE", STRINGDECODE, 1, Value.STRING);
addFunction("STRINGTOUTF8", STRINGTOUTF8, 1, Value.BYTES);
addFunction("UTF8TOSTRING", UTF8TOSTRING, 1, Value.STRING);
addFunction("XMLATTR", XMLATTR, 2, Value.STRING);
addFunctionWithNull("XMLNODE", XMLNODE, VAR_ARGS, Value.STRING);
addFunction("XMLCOMMENT", XMLCOMMENT, 1, Value.STRING);
addFunction("XMLCDATA", XMLCDATA, 1, Value.STRING);
addFunction("XMLSTARTDOC", XMLSTARTDOC, 0, Value.STRING);
addFunction("XMLTEXT", XMLTEXT, VAR_ARGS, Value.STRING);
addFunction("REGEXP_REPLACE", REGEXP_REPLACE, 3, Value.STRING);
addFunction("RPAD", RPAD, VAR_ARGS, Value.STRING);
addFunction("LPAD", LPAD, VAR_ARGS, Value.STRING);
// date
addFunctionNotDeterministic("CURRENT_DATE", CURRENT_DATE, 0,
```

```
Value.DATE);
addFunctionNotDeterministic("CURDATE", CURDATE, 0, Value.DATE);
// alias for MSSQLServer
addFunctionNotDeterministic("GETDATE", CURDATE, 0, Value.DATE);
addFunctionNotDeterministic("CURRENT_TIME", CURRENT_TIME, 0,
Value.TIME);
addFunctionNotDeterministic("CURTIME", CURTIME, 0, Value.TIME);
addFunctionNotDeterministic("CURRENT_TIMESTAMP", CURRENT_TIMESTAMP,
VAR_ARGS, Value.TIMESTAMP);
addFunctionNotDeterministic("NOW", NOW, VAR_ARGS, Value.TIMESTAMP);
addFunction("DATEADD", DATE_ADD, 3, Value.TIMESTAMP);
addFunction("TIMESTAMPADD", DATE_ADD, 3, Value.LONG);
addFunction("DATEDIFF", DATE_DIFF, 3, Value.LONG);
addFunction("TIMESTAMPDIFF", DATE_DIFF, 3, Value.LONG);
addFunction("DAYNAME", DAY_NAME, 1, Value.STRING);
addFunction("DAYNAME", DAY_NAME, 1, Value.STRING);
addFunction("DAY", DAY_OF_MONTH, 1, Value.INT);
addFunction("DAY_OF_MONTH", DAY_OF_MONTH, 1, Value.INT);
addFunction("DAY_OF_WEEK", DAY_OF_WEEK, 1, Value.INT);
addFunction("DAY_OF_YEAR", DAY_OF_YEAR, 1, Value.INT);
addFunction("DAYOFMONTH", DAY_OF_MONTH, 1, Value.INT);
addFunction("DAYOFWEEK", DAY_OF_WEEK, 1, Value.INT);
addFunction("DAYOFYEAR", DAY_OF_YEAR, 1, Value.INT);
addFunction("HOUR", HOUR, 1, Value.INT);
addFunction("MINUTE", MINUTE, 1, Value.INT);
addFunction("MONTH", MONTH, 1, Value.INT);
addFunction("MONTHNAME", MONTH_NAME, 1, Value.STRING);
addFunction("QUARTER", QUARTER, 1, Value.INT);
addFunction("SECOND", SECOND, 1, Value.INT);
addFunction("WEEK", WEEK, 1, Value.INT);
addFunction("YEAR", YEAR, 1, Value.INT);
addFunction("EXTRACT", EXTRACT, 2, Value.INT);
addFunctionWithNull("FORMATDATETIME", FORMATDATETIME, VAR_ARGS,
Value.STRING);
addFunctionWithNull("PARSEDATETIME", PARSEDATETIME, VAR_ARGS,
Value.TIMESTAMP);
addFunction("ISO_YEAR", ISO_YEAR, 1, Value.INT);
addFunction("ISO_WEEK", ISO_WEEK, 1, Value.INT);
addFunction("ISO_DAY_OF_WEEK", ISO_DAY_OF_WEEK, 1, Value.INT);
// system
addFunctionNotDeterministic("DATABASE", DATABASE, 0, Value.STRING);
addFunctionNotDeterministic("USER", USER, 0, Value.STRING);
addFunctionNotDeterministic("CURRENT_USER", CURRENT_USER, 0,
Value.STRING);
addFunctionNotDeterministic("IDENTITY", IDENTITY, 0, Value.LONG);
addFunctionNotDeterministic("SCOPE_IDENTITY", SCOPE_IDENTITY, 0,
Value.LONG);
addFunctionNotDeterministic("IDENTITY_VAL_LOCAL", IDENTITY, 0,
Value.LONG);
```

```
addFunctionNotDeterministic("LAST_INSERT_ID", IDENTITY, 0,
Value.LONG);
addFunctionNotDeterministic("LASTVAL", IDENTITY, 0, Value.LONG);
addFunctionNotDeterministic("AUTOCOMMIT", AUTOCOMMIT, 0,
Value.BOOLEAN);
addFunctionNotDeterministic("READONLY", READONLY, 0,
Value.BOOLEAN);
addFunction("DATABASE_PATH", DATABASE_PATH, 0, Value.STRING);
addFunctionNotDeterministic("LOCK_TIMEOUT", LOCK_TIMEOUT, 0,
Value.INT);
addFunctionWithNull("IFNULL", IFNULL, 2, Value.NULL);
addFunctionWithNull("ISNULL", IFNULL, 2, Value.NULL);
addFunctionWithNull("CASEWHEN", CASEWHEN, 3, Value.NULL);
addFunctionWithNull("CONVERT", CONVERT, 1, Value.NULL);
addFunctionWithNull("CAST", CAST, 1, Value.NULL);
addFunctionWithNull("TRUNCATE_VALUE", TRUNCATE_VALUE, 3,
Value.NULL);
addFunctionWithNull("COALESCE", COALESCE, VAR_ARGS, Value.NULL);
addFunctionWithNull("NVL", COALESCE, VAR_ARGS, Value.NULL);
addFunctionWithNull("NVL2", NVL2, 3, Value.NULL);
addFunctionWithNull("NULLIF", NULLIF, 2, Value.NULL);
addFunctionWithNull("CASE", CASE, VAR_ARGS, Value.NULL);
addFunctionNotDeterministic("NEXTVAL", NEXTVAL, VAR_ARGS,
Value.LONG);
addFunctionNotDeterministic("CURRVAL", CURRVAL, VAR_ARGS,
Value.LONG);
addFunction("ARRAY_GET", ARRAY_GET, 2, Value.STRING);
addFunction("ARRAY_CONTAINS", ARRAY_CONTAINS, 2, Value.BOOLEAN,
false, true, false);
addFunction("CSVREAD", CSVREAD, VAR_ARGS, Value.RESULT_SET, false,
false, true);
addFunction("CSVWRITE", CSVWRITE, VAR_ARGS, Value.INT, false,
false, false);
addFunctionNotDeterministic("MEMORY_FREE", MEMORY_FREE, 0,
Value.INT);
addFunctionNotDeterministic("MEMORY_USED", MEMORY_USED, 0,
Value.INT);
addFunctionNotDeterministic("LOCK_MODE", LOCK_MODE, 0, Value.INT);
addFunctionNotDeterministic("SCHEMA", SCHEMA, 0, Value.STRING);
addFunctionNotDeterministic("SESSION_ID", SESSION_ID, 0,
Value.INT);
addFunction("ARRAY_LENGTH", ARRAY_LENGTH, 1, Value.INT);
addFunctionNotDeterministic("LINK_SCHEMA", LINK_SCHEMA, 6,
Value.RESULT_SET);
addFunctionWithNull("LEAST", LEAST, VAR_ARGS, Value.NULL);
addFunctionWithNull("GREATEST", GREATEST, VAR_ARGS, Value.NULL);
addFunctionNotDeterministic("CANCEL_SESSION", CANCEL_SESSION, 1,
Value.BOOLEAN);
addFunction("SET", SET, 2, Value.NULL, false, false, false);
```

```
addFunction("FILE_READ", FILE_READ, VAR_ARGS, Value.NULL, false,
false, false);
addFunctionNotDeterministic("TRANSACTION_ID", TRANSACTION_ID, 0,
Value.STRING);
addFunctionWithNull("DECODE", DECODE, VAR_ARGS, Value.NULL);
addFunctionNotDeterministic("DISK_SPACE_USED", DISK_SPACE_USED, 1,
Value.LONG);
addFunction("H2VERSION", H2VERSION, 0, Value.STRING);
// TableFunction
addFunctionWithNull("TABLE", TABLE, VAR_ARGS, Value.RESULT_SET);
addFunctionWithNull("TABLE_DISTINCT", TABLE_DISTINCT, VAR_ARGS,
Value.RESULT_SET);
// pseudo function
addFunctionWithNull("ROW_NUMBER", ROW_NUMBER, 0, Value.LONG);
}
protected Function(Database database, FunctionInfo info) {
this.database = database;
this.info = info;
if (info.parameterCount == VAR_ARGS) {
varArgs = New.arrayList();
} else {
args = new Expression[info.parameterCount];
}
}
private static void addFunction(String name, int type, int
parameterCount, int dataType,
boolean nullIfParameterIsNull, boolean deterministic, boolean
fast) {
FunctionInfo info = new FunctionInfo();
info.name = name;
info.type = type;
info.parameterCount = parameterCount;
info.dataType = dataType;
info.nullIfParameterIsNull = nullIfParameterIsNull;
info.deterministic = deterministic;
info.fast = fast;
FUNCTIONS.put(name, info);
}
private static void addFunctionNotDeterministic(String name, int type,
int parameterCount, int dataType) {
addFunction(name, type, parameterCount, dataType, true, false,
false);
}
private static void addFunction(String name, int type, int
parameterCount, int dataType) {
addFunction(name, type, parameterCount, dataType, true, true,
false);
}
private static void addFunctionWithNull(String name, int type, int
```

```
parameterCount, int dataType) {
addFunction(name, type, parameterCount, dataType, false, true,
false);
}
/**
* Get the function info object for this function, or null if there is
no
* such function.
*
* @param name the function name
* @return the function info
*/
private static FunctionInfo getFunctionInfo(String name) {
return FUNCTIONS.get(name);
}
/**
* Get an instance of the given function for this database.
* If no function with this name is found, null is returned.
*
* @param database the database
* @param name the function name
* @return the function object or null
*/
public static Function getFunction(Database database, String name) {
if (!database.getSettings().databaseToUpper) {
// if not yet converted to uppercase, do it now
name = StringUtils.toUpperEnglish(name);
}
FunctionInfo info = getFunctionInfo(name);
if (info == null) {
return null;
}
switch(info.type) {
case TABLE:
case TABLE_DISTINCT:
return new TableFunction(database, info, Long.MAX_VALUE);
default:
return new Function(database, info);
}
}
/**
* Set the parameter expression at the given index.
*
* @param index the index (0, 1,...)
* @param param the expression
*/
public void setParameter(int index, Expression param) {
if (varArgs != null) {
varArgs.add(param);
```

```
} else {
if (index >= args.length) {
throw DbException.get(ErrorCode.INVALID_PARAMETER_COUNT_2,
info.name,
"" + args.length);
}
args[index] = param;
}
}
private static strictfp double log10(double value) {
return roundmagic(StrictMath.log(value) / StrictMath.log(10));
}
@Override
public Value getValue(Session session) {
return getValueWithArgs(session, args);
}
private Value getSimpleValue(Session session, Value v0, Expression[]
args, Value[] values) {
Value result;
switch (info.type) {
case ABS:
result = v0.getSignum() > 0 ? v0 : v0.negate();
break;
case ACOS:
result = ValueDouble.get(Math.acos(v0.getDouble()));
break;
case ASIN:
result = ValueDouble.get(Math.asin(v0.getDouble()));
break;
case ATAN:
result = ValueDouble.get(Math.atan(v0.getDouble()));
break;
case CEILING:
result = ValueDouble.get(Math.ceil(v0.getDouble()));
break;
case COS:
result = ValueDouble.get(Math.cos(v0.getDouble()));
break;
case COSH:
result = ValueDouble.get(Math.cosh(v0.getDouble()));
break;
case COT: {
double d = Math.tan(v0.getDouble());
if (d == 0.0) {
throw DbException.get(ErrorCode.DIVISION_BY_ZERO_1,
getSQL());
}
result = ValueDouble.get(1. / d);
break;
```

```
}
case DEGREES:
result = ValueDouble.get(Math.toDegrees(v0.getDouble()));
break;
case EXP:
result = ValueDouble.get(Math.exp(v0.getDouble()));
break;
case FLOOR:
result = ValueDouble.get(Math.floor(v0.getDouble()));
break;
case LN:
result = ValueDouble.get(Math.log(v0.getDouble()));
break;
case LOG:
if (database.getMode().logIsLogBase10) {
result = ValueDouble.get(Math.log10(v0.getDouble()));
} else {
result = ValueDouble.get(Math.log(v0.getDouble()));
}
break;
case LOG10:
result = ValueDouble.get(log10(v0.getDouble()));
break;
case PI:
result = ValueDouble.get(Math.PI);
break;
case RADIANS:
result = ValueDouble.get(Math.toRadians(v0.getDouble()));
break;
case RAND: {
if (v0 != null) {
session.getRandom().setSeed(v0.getInt());
}
result = ValueDouble.get(session.getRandom().nextDouble());
break;
}
case ROUNDMAGIC:
result = ValueDouble.get(roundmagic(v0.getDouble()));
break;
case SIGN:
result = ValueInt.get(v0.getSignum());
break;
case SIN:
result = ValueDouble.get(Math.sin(v0.getDouble()));
break;
case SINH:
result = ValueDouble.get(Math.sinh(v0.getDouble()));
break;
case SQRT:
```

```
result = ValueDouble.get(Math.sqrt(v0.getDouble()));
break;
case TAN:
result = ValueDouble.get(Math.tan(v0.getDouble()));
break;
case TANH:
result = ValueDouble.get(Math.tanh(v0.getDouble()));
break;
case SECURE_RAND:
result =
ValueBytes.getNoCopy(MathUtils.secureRandomBytes(v0.getInt()));
break;
case EXPAND:
result =
ValueBytes.getNoCopy(CompressTool.getInstance().expand(v0.getBytesNoCopy())
);
break;
case ZERO:
result = ValueInt.get(0);
break;
case RANDOM_UUID:
result = ValueUuid.getNewRandom();
break;
// string
case ASCII: {
String s = v0.getString();
if (s.length() == 0) {
result = ValueNull.INSTANCE;
} else {
result = ValueInt.get(s.charAt(0));
}
break;
}
case BIT_LENGTH:
result = ValueLong.get(16 * length(v0));
break;
case CHAR:
result = ValueString.get(String.valueOf((char) v0.getInt()));
break;
case CHAR_LENGTH:
case LENGTH:
result = ValueLong.get(length(v0));
break;
case OCTET_LENGTH:
result = ValueLong.get(2 * length(v0));
break;
case CONCAT_WS:
case CONCAT: {
result = ValueNull.INSTANCE;
```

```
int start = 0;
String separator = "";
if (info.type == CONCAT_WS) {
start = 1;
separator = getNullOrValue(session, args, values,
0).getString();
}
for (int i = start; i < args.length; i++) {
Value v = getNullOrValue(session, args, values, i);
if (v == ValueNull.INSTANCE) {
continue;
}
if (result == ValueNull.INSTANCE) {
result = v;
} else {
String tmp = v.getString();
if (!StringUtils.isNullOrEmpty(separator)
&& !StringUtils.isNullOrEmpty(tmp)) {
tmp = separator.concat(tmp);
}
result =
ValueString.get(result.getString().concat(tmp));
}
}
if (info.type == CONCAT_WS) {
if (separator != null && result == ValueNull.INSTANCE) {
result = ValueString.get("");
}
}
break;
}
case HEXTORAW:
result = ValueString.get(hexToRaw(v0.getString()));
break;
case LOWER:
case LCASE:
// TODO this is locale specific, need to document or provide a
way
// to set the locale
result = ValueString.get(v0.getString().toLowerCase());
break;
case RAWTOHEX:
result = ValueString.get(rawToHex(v0.getString()));
break;
case SOUNDEX:
result = ValueString.get(getSoundex(v0.getString()));
break;
case SPACE: {
int len = Math.max(0, v0.getInt());
```

```
char[] chars = new char[len];
for (int i = len - 1; i >= 0; i--) {
chars[i] = ' ';
}
result = ValueString.get(new String(chars));
break;
}
case UPPER:
case UCASE:
// TODO this is locale specific, need to document or provide a
way
// to set the locale
result = ValueString.get(v0.getString().toUpperCase());
break;
case STRINGENCODE:
result =
ValueString.get(StringUtils.javaEncode(v0.getString()));
break;
case STRINGDECODE:
result =
ValueString.get(StringUtils.javaDecode(v0.getString()));
break;
case STRINGTOUTF8:
result =
ValueBytes.getNoCopy(v0.getString().getBytes(Constants.UTF8));
break;
case UTF8TOSTRING:
result = ValueString.get(new String(v0.getBytesNoCopy(),
Constants.UTF8));
break;
case XMLCOMMENT:
result =
ValueString.get(StringUtils.xmlComment(v0.getString()));
break;
case XMLCDATA:
result = ValueString.get(StringUtils.xmlCData(v0.getString()));
break;
case XMLSTARTDOC:
result = ValueString.get(StringUtils.xmlStartDoc());
break;
case DAY_NAME: {
SimpleDateFormat dayName = new SimpleDateFormat("EEEE",
Locale.ENGLISH);
result = ValueString.get(dayName.format(v0.getDate()));
break;
}
case DAY_OF_MONTH:
result = ValueInt.get(DateTimeUtils.getDatePart(v0.getDate(),
Calendar.DAY_OF_MONTH));
```

```
break;
case DAY_OF_WEEK:
result = ValueInt.get(DateTimeUtils.getDatePart(v0.getDate(),
Calendar.DAY_OF_WEEK));
break;
case DAY_OF_YEAR:
result = ValueInt.get(DateTimeUtils.getDatePart(v0.getDate(),
Calendar.DAY_OF_YEAR));
break;
case HOUR:
result =
ValueInt.get(DateTimeUtils.getDatePart(v0.getTimestamp(),
Calendar.HOUR_OF_DAY));
break;
case MINUTE:
result =
ValueInt.get(DateTimeUtils.getDatePart(v0.getTimestamp(),
Calendar.MINUTE));
break;
case MONTH:
result = ValueInt.get(DateTimeUtils.getDatePart(v0.getDate(),
Calendar.MONTH));
break;
case MONTH_NAME: {
SimpleDateFormat monthName = new SimpleDateFormat("MMMM",
Locale.ENGLISH);
result = ValueString.get(monthName.format(v0.getDate()));
break;
}
case QUARTER:
result = ValueInt.get((DateTimeUtils.getDatePart(v0.getDate(),
Calendar.MONTH) - 1) / 3 + 1);
break;
case SECOND:
result =
ValueInt.get(DateTimeUtils.getDatePart(v0.getTimestamp(),
Calendar.SECOND));
break;
case WEEK:
result = ValueInt.get(DateTimeUtils.getDatePart(v0.getDate(),
Calendar.WEEK_OF_YEAR));
break;
case YEAR:
result = ValueInt.get(DateTimeUtils.getDatePart(v0.getDate(),
Calendar.YEAR));
break;
case ISO_YEAR:
result = ValueInt.get(DateTimeUtils.getIsoYear(v0.getDate()));
break;
```

```
case ISO_WEEK:
result = ValueInt.get(DateTimeUtils.getIsoWeek(v0.getDate()));
break;
case ISO_DAY_OF_WEEK:
result =
ValueInt.get(DateTimeUtils.getIsoDayOfWeek(v0.getDate()));
break;
case CURDATE:
case CURRENT_DATE: {
long now = session.getTransactionStart();
// need to normalize
result = ValueDate.get(new Date(now));
break;
}
case CURTIME:
case CURRENT_TIME: {
long now = session.getTransactionStart();
// need to normalize
result = ValueTime.get(new Time(now));
break;
}
case NOW:
case CURRENT_TIMESTAMP: {
long now = session.getTransactionStart();
ValueTimestamp vt = ValueTimestamp.get(new Timestamp(now));
if (v0 != null) {
Mode mode = database.getMode();
vt = (ValueTimestamp)
vt.convertScale(mode.convertOnlyToSmallerScale, v0.getInt());
}
result = vt;
break;
}
case DATABASE:
result = ValueString.get(database.getShortName());
break;
case USER:
case CURRENT_USER:
result = ValueString.get(session.getUser().getName());
break;
case IDENTITY:
result = session.getLastIdentity();
break;
case SCOPE_IDENTITY:
result = session.getLastScopeIdentity();
break;
case AUTOCOMMIT:
result = ValueBoolean.get(session.getAutoCommit());
break;
```

```
case READONLY:
result = ValueBoolean.get(database.isReadOnly());
break;
case DATABASE_PATH: {
String path = database.getDatabasePath();
result = path == null ? (Value) ValueNull.INSTANCE :
ValueString.get(path);
break;
}
case LOCK_TIMEOUT:
result = ValueInt.get(session.getLockTimeout());
break;
case DISK_SPACE_USED:
result = ValueLong.get(getDiskSpaceUsed(session, v0));
break;
case CAST:
case CONVERT: {
v0 = v0.convertTo(dataType);
Mode mode = database.getMode();
v0 = v0.convertScale(mode.convertOnlyToSmallerScale, scale);
v0 = v0.convertPrecision(getPrecision(), false);
result = v0;
break;
}
case MEMORY_FREE:
session.getUser().checkAdmin();
result = ValueInt.get(Utils.getMemoryFree());
break;
case MEMORY_USED:
session.getUser().checkAdmin();
result = ValueInt.get(Utils.getMemoryUsed());
break;
case LOCK_MODE:
result = ValueInt.get(database.getLockMode());
break;
case SCHEMA:
result = ValueString.get(session.getCurrentSchemaName());
break;
case SESSION_ID:
result = ValueInt.get(session.getId());
break;
case IFNULL: {
result = v0;
if (v0 == ValueNull.INSTANCE) {
result = getNullOrValue(session, args, values, 1);
}
break;
}
case CASEWHEN: {
```

```
Value v;
if (v0 == ValueNull.INSTANCE || !
v0.getBoolean().booleanValue()) {
v = getNullOrValue(session, args, values, 2);
} else {
v = getNullOrValue(session, args, values, 1);
}
result = v.convertTo(dataType);
break;
}
case DECODE: {
int index = -1;
for (int i = 1; i < args.length - 1; i += 2) {
if (database.areEqual(v0, getNullOrValue(session, args,
values, i))) {
index = i + 1;
}
}
if (index < 0 && args.length % 2 == 0) {
index = args.length - 1;
}
Value v = index < 0 ? ValueNull.INSTANCE :
getNullOrValue(session, args, values, index);
result = v.convertTo(dataType);
break;
}
case NVL2: {
Value v;
if (v0 == ValueNull.INSTANCE) {
v = getNullOrValue(session, args, values, 2);
} else {
v = getNullOrValue(session, args, values, 1);
}
result = v.convertTo(dataType);
break;
}
case COALESCE: {
result = v0;
for (int i = 0; i < args.length; i++) {
Value v = getNullOrValue(session, args, values, i);
if (!(v == ValueNull.INSTANCE)) {
result = v.convertTo(dataType);
break;
}
}
break;
}
case GREATEST:
case LEAST: {
```

```
result = ValueNull.INSTANCE;
for (int i = 0; i < args.length; i++) {
Value v = getNullOrValue(session, args, values, i);
if (!(v == ValueNull.INSTANCE)) {
v = v.convertTo(dataType);
if (result == ValueNull.INSTANCE) {
result = v;
} else {
int comp = database.compareTypeSave(result, v);
if (info.type == GREATEST && comp < 0) {
result = v;
} else if (info.type == LEAST && comp > 0) {
result = v;
}
}
}
}
break;
}
case CASE: {
result = null;
int i = 0;
for (; i < args.length; i++) {
Value when = getNullOrValue(session, args, values, i++);
if (Boolean.TRUE.equals(when)) {
result = getNullOrValue(session, args, values, i);
break;
}
}
if (result == null) {
result = i < args.length ? getNullOrValue(session, args,
values, i) : ValueNull.INSTANCE;
}
break;
}
case ARRAY_GET: {
if (v0.getType() == Value.ARRAY) {
Value v1 = getNullOrValue(session, args, values, 1);
int element = v1.getInt();
Value[] list = ((ValueArray) v0).getList();
if (element < 1 || element > list.length) {
result = ValueNull.INSTANCE;
} else {
result = list[element - 1];
}
} else {
result = ValueNull.INSTANCE;
}
break;
```

```
}
case ARRAY_LENGTH: {
if (v0.getType() == Value.ARRAY) {
Value[] list = ((ValueArray) v0).getList();
result = ValueInt.get(list.length);
} else {
result = ValueNull.INSTANCE;
}
break;
}
case ARRAY_CONTAINS: {
result = ValueBoolean.get(false);
if (v0.getType() == Value.ARRAY) {
Value v1 = getNullOrValue(session, args, values, 1);
Value[] list = ((ValueArray) v0).getList();
for (Value v : list) {
if (v.equals(v1)) {
result = ValueBoolean.get(true);
break;
}
}
}
break;
}
case CANCEL_SESSION: {
result = ValueBoolean.get(cancelStatement(session,
v0.getInt()));
break;
}
case TRANSACTION_ID: {
result = session.getTransactionId();
break;
}
default:
result = null;
}
return result;
}
private static boolean cancelStatement(Session session, int
targetSessionId) {
session.getUser().checkAdmin();
Session[] sessions = session.getDatabase().getSessions(false);
for (Session s : sessions) {
if (s.getId() == targetSessionId) {
Command c = s.getCurrentCommand();
if (c == null) {
return false;
}
c.cancel();
```

```
return true;
}
}
return false;
}
private static long getDiskSpaceUsed(Session session, Value v0) {
Parser p = new Parser(session);
String sql = v0.getString();
Table table = p.parseTableName(sql);
return table.getDiskSpaceUsed();
}
private static Value getNullOrValue(Session session, Expression[] args,
Value[] values, int i) {
if (i >= args.length) {
return null;
}
Value v = values[i];
if (v == null) {
v = values[i] = args[i].getValue(session);
}
return v;
}
private Value getValueWithArgs(Session session, Expression[] args) {
Value[] values = new Value[args.length];
if (info.nullIfParameterIsNull) {
for (int i = 0; i < args.length; i++) {
Expression e = args[i];
Value v = e.getValue(session);
if (v == ValueNull.INSTANCE) {
return ValueNull.INSTANCE;
}
values[i] = v;
}
}
Value v0 = getNullOrValue(session, args, values, 0);
Value resultSimple = getSimpleValue(session, v0, args, values);
if (resultSimple != null) {
return resultSimple;
}
Value v1 = getNullOrValue(session, args, values, 1);
Value v2 = getNullOrValue(session, args, values, 2);
Value v3 = getNullOrValue(session, args, values, 3);
Value v4 = getNullOrValue(session, args, values, 4);
Value v5 = getNullOrValue(session, args, values, 5);
Value result;
switch (info.type) {
case ATAN2:
result = ValueDouble.get(Math.atan2(v0.getDouble(),
v1.getDouble()));
```

```
break;
case BITAND:
result = ValueLong.get(v0.getLong() & v1.getLong());
break;
case BITOR:
result = ValueLong.get(v0.getLong() | v1.getLong());
break;
case BITXOR:
result = ValueLong.get(v0.getLong() ^ v1.getLong());
break;
case MOD: {
long x = v1.getLong();
if (x == 0) {
throw DbException.get(ErrorCode.DIVISION_BY_ZERO_1,
getSQL());
}
result = ValueLong.get(v0.getLong() % x);
break;
}
case POWER:
result = ValueDouble.get(Math.pow(v0.getDouble(),
v1.getDouble()));
break;
case ROUND: {
double f = v1 == null ? 1. : Math.pow(10., v1.getDouble());
result = ValueDouble.get(Math.round(v0.getDouble() * f) / f);
break;
}
case TRUNCATE: {
if (v0.getType() == Value.TIMESTAMP) {
java.sql.Timestamp d = v0.getTimestamp();
Calendar c = Calendar.getInstance();
c.setTime(d);
c.set(Calendar.HOUR_OF_DAY, 0);
c.set(Calendar.MINUTE, 0);
c.set(Calendar.SECOND, 0);
c.set(Calendar.MILLISECOND, 0);
result = ValueTimestamp.get(new
java.sql.Timestamp(c.getTimeInMillis()));
} else {
double d = v0.getDouble();
int p = v1 == null ? 0 : v1.getInt();
double f = Math.pow(10., p);
double g = d * f;
result = ValueDouble.get(((d < 0) ? Math.ceil(g) :
Math.floor(g)) / f);
}
break;
}
```

```
case HASH:
result = ValueBytes.getNoCopy(getHash(v0.getString(),
v1.getBytesNoCopy(), v2.getInt()));
break;
case ENCRYPT:
result = ValueBytes.getNoCopy(encrypt(v0.getString(),
v1.getBytesNoCopy(), v2.getBytesNoCopy()));
break;
case DECRYPT:
result = ValueBytes.getNoCopy(decrypt(v0.getString(),
v1.getBytesNoCopy(), v2.getBytesNoCopy()));
break;
case COMPRESS: {
String algorithm = null;
if (v1 != null) {
algorithm = v1.getString();
}
result =
ValueBytes.getNoCopy(CompressTool.getInstance().compress(v0.getBytesNoCopy(
), algorithm));
break;
}
case DIFFERENCE:
result = ValueInt.get(getDifference(v0.getString(),
v1.getString()));
break;
case INSERT: {
if (v1 == ValueNull.INSTANCE || v2 == ValueNull.INSTANCE) {
result = v1;
} else {
result = ValueString.get(insert(v0.getString(),
v1.getInt(), v2.getInt(), v3.getString()));
}
break;
}
case LEFT:
result = ValueString.get(left(v0.getString(), v1.getInt()));
break;
case LOCATE: {
int start = v2 == null ? 0 : v2.getInt();
result = ValueInt.get(locate(v0.getString(), v1.getString(),
start));
break;
}
case INSTR: {
int start = v2 == null ? 0 : v2.getInt();
result = ValueInt.get(locate(v1.getString(), v0.getString(),
start));
break;
```

```
}
case REPEAT: {
int count = Math.max(0, v1.getInt());
result = ValueString.get(repeat(v0.getString(), count));
break;
}
case REPLACE: {
String s0 = v0.getString();
String s1 = v1.getString();
String s2 = (v2 == null) ? "" : v2.getString();
result = ValueString.get(replace(s0, s1, s2));
break;
}
case RIGHT:
result = ValueString.get(right(v0.getString(), v1.getInt()));
break;
case LTRIM:
result = ValueString.get(StringUtils.trim(v0.getString(), true,
false, v1 == null ? " " : v1.getString()));
break;
case TRIM:
result = ValueString.get(StringUtils.trim(v0.getString(), true,
true, v1 == null ? " " : v1.getString()));
break;
case RTRIM:
result = ValueString.get(StringUtils.trim(v0.getString(),
false, true, v1 == null ? " " : v1.getString()));
break;
case SUBSTR:
case SUBSTRING: {
String s = v0.getString();
int offset = v1.getInt();
if (offset < 0) {
offset = s.length() + offset + 1;
}
int length = v2 == null ? s.length() : v2.getInt();
result = ValueString.get(substring(s, offset, length));
break;
}
case POSITION:
result = ValueInt.get(locate(v0.getString(), v1.getString(),
0));
break;
case XMLATTR:
result = ValueString.get(StringUtils.xmlAttr(v0.getString(),
v1.getString()));
break;
case XMLNODE: {
String attr = v1 == null ? null : v1 == ValueNull.INSTANCE ?
```

```
null : v1.getString();
String content = v2 == null ? null : v2 == ValueNull.INSTANCE ?
null : v2.getString();
boolean indent = v3 == null ? true : v3.getBoolean();
result = ValueString.get(StringUtils.xmlNode(v0.getString(),
attr, content, indent));
break;
}
case REGEXP_REPLACE: {
String regexp = v1.getString();
try {
result = ValueString.get(v0.getString().replaceAll(regexp,
v2.getString()));
} catch (PatternSyntaxException e) {
throw DbException.get(ErrorCode.LIKE_ESCAPE_ERROR_1, e,
regexp);
}
break;
}
case RPAD:
result = ValueString.get(StringUtils.pad(v0.getString(),
v1.getInt(), v2 == null ? null : v2.getString(), true));
break;
case LPAD:
result = ValueString.get(StringUtils.pad(v0.getString(),
v1.getInt(), v2 == null ? null : v2.getString(), false));
break;
case H2VERSION:
result = ValueString.get(Constants.getVersion());
break;
case DATE_ADD:
result = ValueTimestamp.get(dateadd(v0.getString(),
v1.getInt(), v2.getTimestamp()));
break;
case DATE_DIFF:
result = ValueLong.get(datediff(v0.getString(),
v1.getTimestamp(), v2.getTimestamp()));
break;
case EXTRACT: {
int field = getDatePart(v0.getString());
result =
ValueInt.get(DateTimeUtils.getDatePart(v1.getTimestamp(), field));
break;
}
case FORMATDATETIME: {
if (v0 == ValueNull.INSTANCE || v1 == ValueNull.INSTANCE) {
result = ValueNull.INSTANCE;
} else {
String locale = v2 == null ? null : v2 ==
```

```
ValueNull.INSTANCE ? null : v2.getString();
String tz = v3 == null ? null : v3 == ValueNull.INSTANCE ?
null : v3.getString();
result =
ValueString.get(DateTimeUtils.formatDateTime(v0.getTimestamp(),
v1.getString(), locale, tz));
}
break;
}
case PARSEDATETIME: {
if (v0 == ValueNull.INSTANCE || v1 == ValueNull.INSTANCE) {
result = ValueNull.INSTANCE;
} else {
String locale = v2 == null ? null : v2 ==
ValueNull.INSTANCE ? null : v2.getString();
String tz = v3 == null ? null : v3 == ValueNull.INSTANCE ?
null : v3.getString();
java.util.Date d =
DateTimeUtils.parseDateTime(v0.getString(), v1.getString(), locale, tz);
result = ValueTimestamp.get(new Timestamp(d.getTime()));
}
break;
}
case NULLIF:
result = database.areEqual(v0, v1) ? ValueNull.INSTANCE : v0;
break;
// system
case NEXTVAL: {
Sequence sequence = getSequence(session, v0, v1);
SequenceValue value = new SequenceValue(sequence);
result = value.getValue(session);
break;
}
case CURRVAL: {
Sequence sequence = getSequence(session, v0, v1);
result = ValueLong.get(sequence.getCurrentValue());
break;
}
case CSVREAD: {
String fileName = v0.getString();
String columnList = v1 == null ? null : v1.getString();
Csv csv = new Csv();
String options = v2 == null ? null : v2.getString();
String charset = null;
if (options != null && options.indexOf('=') >= 0) {
charset = csv.setOptions(options);
} else {
charset = options;
String fieldSeparatorRead = v3 == null ? null :
```

```
v3.getString();
String fieldDelimiter = v4 == null ? null : v4.getString();
String escapeCharacter = v5 == null ? null :
v5.getString();
Value v6 = getNullOrValue(session, args, values, 6);
String nullString = v6 == null ? null : v6.getString();
setCsvDelimiterEscape(csv, fieldSeparatorRead,
fieldDelimiter, escapeCharacter);
csv.setNullString(nullString);
}
char fieldSeparator = csv.getFieldSeparatorRead();
String[] columns = StringUtils.arraySplit(columnList,
fieldSeparator, true);
try {
ValueResultSet vr = ValueResultSet.get(csv.read(fileName,
columns, charset));
result = vr;
} catch (SQLException e) {
throw DbException.convert(e);
}
break;
}
case LINK_SCHEMA: {
session.getUser().checkAdmin();
Connection conn = session.createConnection(false);
ResultSet rs = LinkSchema.linkSchema(conn, v0.getString(),
v1.getString(), v2.getString(), v3.getString(),
v4.getString(), v5.getString());
result = ValueResultSet.get(rs);
break;
}
case CSVWRITE: {
session.getUser().checkAdmin();
Connection conn = session.createConnection(false);
Csv csv = new Csv();
String options = v2 == null ? null : v2.getString();
String charset = null;
if (options != null && options.indexOf('=') >= 0) {
charset = csv.setOptions(options);
} else {
charset = options;
String fieldSeparatorWrite = v3 == null ? null :
v3.getString();
String fieldDelimiter = v4 == null ? null : v4.getString();
String escapeCharacter = v5 == null ? null :
v5.getString();
Value v6 = getNullOrValue(session, args, values, 6);
String nullString = v6 == null ? null : v6.getString();
Value v7 = getNullOrValue(session, args, values, 7);
```

```
String lineSeparator = v7 == null ? null : v7.getString();
setCsvDelimiterEscape(csv, fieldSeparatorWrite,
fieldDelimiter, escapeCharacter);
csv.setNullString(nullString);
if (lineSeparator != null) {
csv.setLineSeparator(lineSeparator);
}
}
try {
int rows = csv.write(conn, v0.getString(), v1.getString(),
charset);
result = ValueInt.get(rows);
} catch (SQLException e) {
throw DbException.convert(e);
}
break;
}
case SET: {
Variable var = (Variable) args[0];
session.setVariable(var.getName(), v1);
result = v1;
break;
}
case FILE_READ: {
session.getUser().checkAdmin();
String fileName = v0.getString();
boolean blob = args.length == 1;
try {
InputStream in = new
AutoCloseInputStream(FileUtils.newInputStream(fileName));
if (blob) {
result = database.getLobStorage().createBlob(in, -1);
} else {
Reader reader;
if (v1 == ValueNull.INSTANCE) {
reader = new InputStreamReader(in);
} else {
reader = new InputStreamReader(in, v1.getString());
}
result = database.getLobStorage().createClob(reader,
-1);
}
} catch (IOException e) {
throw DbException.convertIOException(e, fileName);
}
break;
}
case TRUNCATE_VALUE: {
result = v0.convertPrecision(v1.getLong(), v2.getBoolean());
```

```
break;
}
case XMLTEXT:
if (v1 == null) {
result =
ValueString.get(StringUtils.xmlText(v0.getString()));
} else {
result =
ValueString.get(StringUtils.xmlText(v0.getString(), v1.getBoolean()));
}
break;
default:
throw DbException.throwInternalError("type=" + info.type);
}
return result;
}
private Sequence getSequence(Session session, Value v0, Value v1) {
String schemaName, sequenceName;
if (v1 == null) {
Parser p = new Parser(session);
String sql = v0.getString();
Expression expr = p.parseExpression(sql);
if (expr instanceof ExpressionColumn) {
ExpressionColumn seq = (ExpressionColumn) expr;
schemaName = seq.getOriginalTableAliasName();
if (schemaName == null) {
schemaName = session.getCurrentSchemaName();
sequenceName = sql;
} else {
sequenceName = seq.getColumnName();
}
} else {
throw DbException.getSyntaxError(sql, 1);
}
} else {
schemaName = v0.getString();
sequenceName = v1.getString();
}
Schema s = database.findSchema(schemaName);
if (s == null) {
schemaName = StringUtils.toUpperEnglish(schemaName);
s = database.getSchema(schemaName);
}
Sequence seq = s.findSequence(sequenceName);
if (seq == null) {
sequenceName = StringUtils.toUpperEnglish(sequenceName);
seq = s.getSequence(sequenceName);
}
return seq;
```

```
}
private static long length(Value v) {
switch (v.getType()) {
case Value.BLOB:
case Value.CLOB:
case Value.BYTES:
case Value.JAVA_OBJECT:
return v.getPrecision();
default:
return v.getString().length();
}
}
private static byte[] getPaddedArrayCopy(byte[] data, int blockSize) {
int size = MathUtils.roundUpInt(data.length, blockSize);
byte[] newData = DataUtils.newBytes(size);
System.arraycopy(data, 0, newData, 0, data.length);
return newData;
}
private static byte[] decrypt(String algorithm, byte[] key, byte[]
data) {
BlockCipher cipher = CipherFactory.getBlockCipher(algorithm);
byte[] newKey = getPaddedArrayCopy(key, cipher.getKeyLength());
cipher.setKey(newKey);
byte[] newData = getPaddedArrayCopy(data, BlockCipher.ALIGN);
cipher.decrypt(newData, 0, newData.length);
return newData;
}
private static byte[] encrypt(String algorithm, byte[] key, byte[]
data) {
BlockCipher cipher = CipherFactory.getBlockCipher(algorithm);
byte[] newKey = getPaddedArrayCopy(key, cipher.getKeyLength());
cipher.setKey(newKey);
byte[] newData = getPaddedArrayCopy(data, BlockCipher.ALIGN);
cipher.encrypt(newData, 0, newData.length);
return newData;
}
private static byte[] getHash(String algorithm, byte[] bytes, int
iterations) {
if (!"SHA256".equalsIgnoreCase(algorithm)) {
throw DbException.getInvalidValueException("algorithm",
algorithm);
}
for (int i = 0; i < iterations; i++) {
bytes = SHA256.getHash(bytes, false);
}
return bytes;
}
/**
* Check if a given string is a valid date part string.
```

```
*
* @param part the string
* @return true if it is
*/
public static boolean isDatePart(String part) {
Integer p = DATE_PART.get(StringUtils.toUpperEnglish(part));
return p != null;
}
private static int getDatePart(String part) {
Integer p = DATE_PART.get(StringUtils.toUpperEnglish(part));
if (p == null) {
throw DbException.getInvalidValueException("date part", part);
}
return p.intValue();
}
private static Timestamp dateadd(String part, int count, Timestamp d) {
int field = getDatePart(part);
Calendar calendar = Calendar.getInstance();
int nanos = d.getNanos() % 1000000;
calendar.setTime(d);
calendar.add(field, count);
long t = calendar.getTime().getTime();
Timestamp ts = new Timestamp(t);
ts.setNanos(ts.getNanos() + nanos);
return ts;
}
/**
* Calculate the number of crossed unit boundaries between two
timestamps.
* This method is supported for MS SQL Server compatibility.
* <pre>
* DATEDIFF(YEAR, '2004-12-31', '2005-01-01') = 1
* </pre>
*
* @param part the part
* @param d1 the first date
* @param d2 the second date
* @return the number of crossed boundaries
*/
private static long datediff(String part, Timestamp d1, Timestamp d2) {
int field = getDatePart(part);
Calendar calendar = Calendar.getInstance();
long t1 = d1.getTime(), t2 = d2.getTime();
// need to convert to UTC, otherwise we get inconsistent results
with
// certain time zones (those that are 30 minutes off)
TimeZone zone = calendar.getTimeZone();
calendar.setTime(d1);
t1 += zone.getOffset(calendar.get(Calendar.ERA),
```

```
calendar.get(Calendar.YEAR), calendar.get(Calendar.MONTH),
calendar.get(Calendar.DAY_OF_MONTH),
calendar.get(Calendar.DAY_OF_WEEK), calendar
.get(Calendar.MILLISECOND));
calendar.setTime(d2);
t2 += zone.getOffset(calendar.get(Calendar.ERA),
calendar.get(Calendar.YEAR), calendar.get(Calendar.MONTH),
calendar.get(Calendar.DAY_OF_MONTH),
calendar.get(Calendar.DAY_OF_WEEK), calendar
.get(Calendar.MILLISECOND));
switch (field) {
case Calendar.MILLISECOND:
return t2 - t1;
case Calendar.SECOND:
case Calendar.MINUTE:
case Calendar.HOUR_OF_DAY: {
// first 'normalize' the numbers so both are not negative
long hour = 60 * 60 * 1000;
long add = Math.min(t1 / hour * hour, t2 / hour * hour);
t1 -= add;
t2 -= add;
switch (field) {
case Calendar.SECOND:
return t2 / 1000 - t1 / 1000;
case Calendar.MINUTE:
return t2 / (60 * 1000) - t1 / (60 * 1000);
case Calendar.HOUR_OF_DAY:
return t2 / hour - t1 / hour;
default:
throw DbException.throwInternalError("field:" + field);
}
}
case Calendar.DATE:
return t2 / (24 * 60 * 60 * 1000) - t1 / (24 * 60 * 60 * 1000);
default:
break;
}
calendar.setTime(new Timestamp(t1));
int year1 = calendar.get(Calendar.YEAR);
int month1 = calendar.get(Calendar.MONTH);
calendar.setTime(new Timestamp(t2));
int year2 = calendar.get(Calendar.YEAR);
int month2 = calendar.get(Calendar.MONTH);
int result = year2 - year1;
if (field == Calendar.MONTH) {
result = 12 * result + (month2 - month1);
}
return result;
}
```

```
private static String substring(String s, int start, int length) {
int len = s.length();
start--;
if (start < 0) {
start = 0;
}
if (length < 0) {
length = 0;
}
start = (start > len) ? len : start;
if (start + length > len) {
length = len - start;
}
return s.substring(start, start + length);
}
private static String replace(String s, String replace, String with) {
if (s == null || replace == null || with == null) {
return null;
}
if (replace.length() == 0) {
// avoid out of memory
return s;
}
StringBuilder buff = new StringBuilder(s.length());
int start = 0;
int len = replace.length();
while (true) {
int i = s.indexOf(replace, start);
if (i == -1) {
break;
}
buff.append(s.substring(start, i)).append(with);
start = i + len;
}
buff.append(s.substring(start));
return buff.toString();
}
private static String repeat(String s, int count) {
StringBuilder buff = new StringBuilder(s.length() * count);
while (count-- > 0) {
buff.append(s);
}
return buff.toString();
}
private static String rawToHex(String s) {
int length = s.length();
StringBuilder buff = new StringBuilder(4 * length);
for (int i = 0; i < length; i++) {
String hex = Integer.toHexString(s.charAt(i) & 0xffff);
```

```
for (int j = hex.length(); j < 4; j++) {
buff.append('0');
}
buff.append(hex);
}
return buff.toString();
}
private static int locate(String search, String s, int start) {
if (start < 0) {
int i = s.length() + start;
return s.lastIndexOf(search, i) + 1;
}
int i = (start == 0) ? 0 : start - 1;
return s.indexOf(search, i) + 1;
}
private static String right(String s, int count) {
if (count < 0) {
count = 0;
} else if (count > s.length()) {
count = s.length();
}
return s.substring(s.length() - count);
}
private static String left(String s, int count) {
if (count < 0) {
count = 0;
} else if (count > s.length()) {
count = s.length();
}
return s.substring(0, count);
}
private static String insert(String s1, int start, int length, String
s2) {
if (s1 == null) {
return s2;
}
if (s2 == null) {
return s1;
}
int len1 = s1.length();
int len2 = s2.length();
start--;
if (start < 0 || length <= 0 || len2 == 0 || start > len1) {
return s1;
}
if (start + length > len1) {
length = len1 - start;
}
return s1.substring(0, start) + s2 + s1.substring(start + length);
```

```
}
private static String hexToRaw(String s) {
// TODO function hextoraw compatibility with oracle
int len = s.length();
if (len % 4 != 0) {
throw DbException.get(ErrorCode.DATA_CONVERSION_ERROR_1, s);
}
StringBuilder buff = new StringBuilder(len / 4);
for (int i = 0; i < len; i += 4) {
try {
char raw = (char) Integer.parseInt(s.substring(i, i + 4),
16);
buff.append(raw);
} catch (NumberFormatException e) {
throw DbException.get(ErrorCode.DATA_CONVERSION_ERROR_1,
s);
}
}
return buff.toString();
}
private static int getDifference(String s1, String s2) {
// TODO function difference: compatibility with SQL Server and
HSQLDB
s1 = getSoundex(s1);
s2 = getSoundex(s2);
int e = 0;
for (int i = 0; i < 4; i++) {
if (s1.charAt(i) == s2.charAt(i)) {
e++;
}
}
return e;
}
private static double roundmagic(double d) {
if ((d < 0.0000000000001) && (d > -0.0000000000001)) {
return 0.0;
}
if ((d > 1000000000000.) || (d < -1000000000000.)) {
return d;
}
StringBuilder s = new StringBuilder();
s.append(d);
if (s.toString().indexOf("E") >= 0) {
return d;
}
int len = s.length();
if (len < 16) {
return d;
}
```

```
if (s.toString().indexOf(".") > len - 3) {
return d;
}
s.delete(len - 2, len);
len -= 2;
char c1 = s.charAt(len - 2);
char c2 = s.charAt(len - 3);
char c3 = s.charAt(len - 4);
if ((c1 == '0') && (c2 == '0') && (c3 == '0')) {
s.setCharAt(len - 1, '0');
} else if ((c1 == '9') && (c2 == '9') && (c3 == '9')) {
s.setCharAt(len - 1, '9');
s.append('9');
s.append('9');
s.append('9');
}
return Double.parseDouble(s.toString());
}
private static String getSoundex(String s) {
int len = s.length();
char[] chars = { '0', '0', '0', '0' };
char lastDigit = '0';
for (int i = 0, j = 0; i < len && j < 4; i++) {
char c = s.charAt(i);
char newDigit = c > SOUNDEX_INDEX.length ? 0 :
SOUNDEX_INDEX[c];
if (newDigit != 0) {
if (j == 0) {
chars[j++] = c;
lastDigit = newDigit;
} else if (newDigit <= '6') {
if (newDigit != lastDigit) {
chars[j++] = newDigit;
lastDigit = newDigit;
}
} else if (newDigit == '7') {
lastDigit = newDigit;
}
}
}
return new String(chars);
}
@Override
public int getType() {
return dataType;
}
@Override
public void mapColumns(ColumnResolver resolver, int level) {
for (Expression e : args) {
```

```
e.mapColumns(resolver, level);
}
}
/**
* Check if the parameter count is correct.
*
* @param len the number of parameters set
* @throws DbException if the parameter count is incorrect
*/
protected void checkParameterCount(int len) {
int min = 0, max = Integer.MAX_VALUE;
switch (info.type) {
case COALESCE:
case CSVREAD:
case LEAST:
case GREATEST:
min = 1;
break;
case NOW:
case CURRENT_TIMESTAMP:
case RAND:
max = 1;
break;
case COMPRESS:
case LTRIM:
case RTRIM:
case TRIM:
case FILE_READ:
case ROUND:
case XMLTEXT:
case TRUNCATE:
min = 1;
max = 2;
break;
case REPLACE:
case LOCATE:
case INSTR:
case SUBSTR:
case SUBSTRING:
case LPAD:
case RPAD:
min = 2;
max = 3;
break;
case CASE:
case CONCAT:
case CONCAT_WS:
case CSVWRITE:
min = 2;
```

```
break;
case XMLNODE:
min = 1;
max = 4;
break;
case FORMATDATETIME:
case PARSEDATETIME:
min = 2;
max = 4;
break;
case CURRVAL:
case NEXTVAL:
min = 1;
max = 2;
break;
case DECODE:
min = 3;
break;
default:
DbException.throwInternalError("type=" + info.type);
}
boolean ok = (len >= min) && (len <= max);
if (!ok) {
throw DbException.get(ErrorCode.INVALID_PARAMETER_COUNT_2,
info.name, min + ".." + max);
}
}
/**
* This method is called after all the parameters have been set.
* It checks if the parameter count is correct.
*
* @throws DbException if the parameter count is incorrect.
*/
public void doneWithParameters() {
if (info.parameterCount == VAR_ARGS) {
int len = varArgs.size();
checkParameterCount(len);
args = new Expression[len];
varArgs.toArray(args);
varArgs = null;
} else {
int len = args.length;
if (len > 0 && args[len - 1] == null) {
throw DbException
.get(ErrorCode.INVALID_PARAMETER_COUNT_2,
info.name, "" + len);
}
}
}
```

```
public void setDataType(Column col) {
dataType = col.getType();
precision = col.getPrecision();
displaySize = col.getDisplaySize();
scale = col.getScale();
}
@Override
public Expression optimize(Session session) {
boolean allConst = info.deterministic;
for (int i = 0; i < args.length; i++) {
Expression e = args[i].optimize(session);
args[i] = e;
if (!e.isConstant()) {
allConst = false;
}
}
int t, s, d;
long p;
Expression p0 = args.length < 1 ? null : args[0];
switch (info.type) {
case IFNULL:
case NULLIF:
case COALESCE:
case LEAST:
case GREATEST:
case DECODE: {
t = Value.UNKNOWN;
s = 0;
p = 0;
d = 0;
int i = 0;
for (Expression e : args) {
if (info.type == DECODE) {
if (i < 2 || ((i % 2 == 1) && (i != args.length - 1)))
{
// decode(a, b, whenB)
// decode(a, b, whenB, else)
// decode(a, b, whenB, c, whenC)
// decode(a, b, whenB, c, whenC, end)
i++;
continue;
}
}
if (e != ValueExpression.getNull()) {
int type = e.getType();
if (type != Value.UNKNOWN && type != Value.NULL) {
t = Value.getHigherOrder(t, type);
s = Math.max(s, e.getScale());
p = Math.max(p, e.getPrecision());
```

```
d = Math.max(d, e.getDisplaySize());
}
}
i++;
}
if (t == Value.UNKNOWN) {
t = Value.STRING;
s = 0;
p = Integer.MAX_VALUE;
d = Integer.MAX_VALUE;
}
break;
}
case CASEWHEN:
t = Value.getHigherOrder(args[1].getType(), args[2].getType());
p = Math.max(args[1].getPrecision(), args[2].getPrecision());
d = Math.max(args[1].getDisplaySize(),
args[2].getDisplaySize());
s = Math.max(args[1].getScale(), args[2].getScale());
break;
case NVL2:
switch (args[1].getType()) {
case Value.STRING:
case Value.CLOB:
case Value.STRING_FIXED:
case Value.STRING_IGNORECASE:
t = args[1].getType();
break;
default:
t = Value.getHigherOrder(args[1].getType(),
args[2].getType());
break;
}
p = Math.max(args[1].getPrecision(), args[2].getPrecision());
d = Math.max(args[1].getDisplaySize(),
args[2].getDisplaySize());
s = Math.max(args[1].getScale(), args[2].getScale());
break;
case CAST:
case CONVERT:
case TRUNCATE_VALUE:
// data type, precision and scale is already set
t = dataType;
p = precision;
s = scale;
d = displaySize;
break;
case TRUNCATE:
t = p0.getType();
```

```
s = p0.getScale();
p = p0.getPrecision();
d = p0.getDisplaySize();
if (t == Value.NULL) {
t = Value.INT;
p = ValueInt.PRECISION;
d = ValueInt.DISPLAY_SIZE;
s = 0;
} else if (t == Value.TIMESTAMP) {
t = Value.DATE;
p = ValueDate.PRECISION;
s = 0;
d = ValueDate.DISPLAY_SIZE;
}
break;
case ABS:
case FLOOR:
case RADIANS:
case ROUND:
case POWER:
t = p0.getType();
s = p0.getScale();
p = p0.getPrecision();
d = p0.getDisplaySize();
if (t == Value.NULL) {
t = Value.INT;
p = ValueInt.PRECISION;
d = ValueInt.DISPLAY_SIZE;
s = 0;
}
break;
case SET: {
Expression p1 = args[1];
t = p1.getType();
p = p1.getPrecision();
s = p1.getScale();
d = p1.getDisplaySize();
if (!(p0 instanceof Variable)) {
throw
DbException.get(ErrorCode.CAN_ONLY_ASSIGN_TO_VARIABLE_1, p0.getSQL());
}
break;
}
case FILE_READ: {
if (args.length == 1) {
t = Value.BLOB;
} else {
t = Value.CLOB;
}
```

```
p = Integer.MAX_VALUE;
s = 0;
d = Integer.MAX_VALUE;
break;
}
case SUBSTRING:
case SUBSTR: {
t = info.dataType;
p = args[0].getPrecision();
s = 0;
if (args[1].isConstant()) {
// if only two arguments are used,
// subtract offset from first argument length
p -= args[1].getValue(session).getLong() - 1;
}
if (args.length == 3 && args[2].isConstant()) {
// if the third argument is constant it is at most this
value
p = Math.min(p, args[2].getValue(session).getLong());
}
p = Math.max(0, p);
d = MathUtils.convertLongToInt(p);
break;
}
default:
t = info.dataType;
DataType type = DataType.getDataType(t);
p = PRECISION_UNKNOWN;
d = 0;
s = type.defaultScale;
}
dataType = t;
precision = p;
scale = s;
displaySize = d;
if (allConst) {
Value v = getValue(session);
if (v == ValueNull.INSTANCE) {
if (info.type == CAST || info.type == CONVERT) {
return this;
}
}
return ValueExpression.get(v);
}
return this;
}
@Override
public void setEvaluatable(TableFilter tableFilter, boolean b) {
for (Expression e : args) {
```

```
if (e != null) {
e.setEvaluatable(tableFilter, b);
}
}
}
@Override
public int getScale() {
return scale;
}
@Override
public long getPrecision() {
if (precision == PRECISION_UNKNOWN) {
calculatePrecisionAndDisplaySize();
}
return precision;
}
@Override
public int getDisplaySize() {
if (precision == PRECISION_UNKNOWN) {
calculatePrecisionAndDisplaySize();
}
return displaySize;
}
private void calculatePrecisionAndDisplaySize() {
switch (info.type) {
case ENCRYPT:
case DECRYPT:
precision = args[2].getPrecision();
displaySize = args[2].getDisplaySize();
break;
case COMPRESS:
precision = args[0].getPrecision();
displaySize = args[0].getDisplaySize();
break;
case CHAR:
precision = 1;
displaySize = 1;
break;
case CONCAT:
precision = 0;
displaySize = 0;
for (Expression e : args) {
precision += e.getPrecision();
displaySize = MathUtils.convertLongToInt((long) displaySize
+ e.getDisplaySize());
if (precision < 0) {
precision = Long.MAX_VALUE;
}
}
```

```
break;
case HEXTORAW:
precision = (args[0].getPrecision() + 3) / 4;
displaySize = MathUtils.convertLongToInt(precision);
break;
case LCASE:
case LTRIM:
case RIGHT:
case RTRIM:
case UCASE:
case LOWER:
case UPPER:
case TRIM:
case STRINGDECODE:
case UTF8TOSTRING:
case TRUNCATE:
precision = args[0].getPrecision();
displaySize = args[0].getDisplaySize();
break;
case RAWTOHEX:
precision = args[0].getPrecision() * 4;
displaySize = MathUtils.convertLongToInt(precision);
break;
case SOUNDEX:
precision = 4;
displaySize = (int) precision;
break;
case DAY_NAME:
case MONTH_NAME:
// day and month names may be long in some languages
precision = 20;
displaySize = (int) precision;
break;
default:
DataType type = DataType.getDataType(dataType);
precision = type.defaultPrecision;
displaySize = type.defaultDisplaySize;
}
}
@Override
public String getSQL() {
StatementBuilder buff = new StatementBuilder(info.name);
buff.append('(');
switch (info.type) {
case CAST: {
buff.append(args[0].getSQL()).append(" AS ").
append(new Column(null, dataType, precision, scale,
displaySize).getCreateSQL());
break;
```

```
}
case CONVERT: {
buff.append(args[0].getSQL()).append(',').
append(new Column(null, dataType, precision, scale,
displaySize).getCreateSQL());
break;
}
case EXTRACT: {
ValueString v = (ValueString) ((ValueExpression)
args[0]).getValue(null);
buff.append(v.getString()).append(" FROM
").append(args[1].getSQL());
break;
}
default: {
for (Expression e : args) {
buff.appendExceptFirst(", ");
buff.append(e.getSQL());
}
}
}
return buff.append(')').toString();
}
@Override
public void updateAggregate(Session session) {
for (Expression e : args) {
if (e != null) {
e.updateAggregate(session);
}
}
}
public int getFunctionType() {
return info.type;
}
@Override
public String getName() {
return info.name;
}
@Override
public int getParameterCount() {
return args.length;
}
@Override
public ValueResultSet getValueForColumnList(Session session,
Expression[] argList) {
switch (info.type) {
case CSVREAD: {
String fileName = argList[0].getValue(session).getString();
if (fileName == null) {
```

```
throw DbException.get(ErrorCode.PARAMETER_NOT_SET_1,
"fileName");
}
String columnList = argList.length < 2 ? null :
argList[1].getValue(session).getString();
Csv csv = new Csv();
String options = argList.length < 3 ? null :
argList[2].getValue(session).getString();
String charset = null;
if (options != null && options.indexOf('=') >= 0) {
charset = csv.setOptions(options);
} else {
charset = options;
String fieldSeparatorRead = argList.length < 4 ? null :
argList[3].getValue(session).getString();
String fieldDelimiter = argList.length < 5 ? null :
argList[4].getValue(session).getString();
String escapeCharacter = argList.length < 6 ? null :
argList[5].getValue(session).getString();
setCsvDelimiterEscape(csv, fieldSeparatorRead,
fieldDelimiter, escapeCharacter);
}
char fieldSeparator = csv.getFieldSeparatorRead();
String[] columns = StringUtils.arraySplit(columnList,
fieldSeparator, true);
ResultSet rs = null;
ValueResultSet x;
try {
rs = csv.read(fileName, columns, charset);
x = ValueResultSet.getCopy(rs, 0);
} catch (SQLException e) {
throw DbException.convert(e);
} finally {
JdbcUtils.closeSilently(rs);
}
return x;
}
default:
break;
}
return (ValueResultSet) getValueWithArgs(session, argList);
}
private static void setCsvDelimiterEscape(Csv csv, String
fieldSeparator,
String fieldDelimiter, String escapeCharacter) {
if (fieldSeparator != null) {
csv.setFieldSeparatorWrite(fieldSeparator);
if (fieldSeparator.length() > 0) {
char fs = fieldSeparator.charAt(0);
```

```
csv.setFieldSeparatorRead(fs);
}
}
if (fieldDelimiter != null) {
char fd = fieldDelimiter.length() == 0 ? 0 :
fieldDelimiter.charAt(0);
csv.setFieldDelimiter(fd);
}
if (escapeCharacter != null) {
char ec = escapeCharacter.length() == 0 ? 0 :
escapeCharacter.charAt(0);
csv.setEscapeCharacter(ec);
}
}
@Override
public Expression[] getArgs() {
return args;
}
@Override
public boolean isEverything(ExpressionVisitor visitor) {
for (Expression e : args) {
if (e != null && !e.isEverything(visitor)) {
return false;
}
}
switch (visitor.getType()) {
case ExpressionVisitor.DETERMINISTIC:
case ExpressionVisitor.QUERY_COMPARABLE:
case ExpressionVisitor.READONLY:
return info.deterministic;
case ExpressionVisitor.EVALUATABLE:
case ExpressionVisitor.GET_DEPENDENCIES:
case ExpressionVisitor.INDEPENDENT:
case ExpressionVisitor.NOT_FROM_RESOLVER:
case ExpressionVisitor.OPTIMIZABLE_MIN_MAX_COUNT_ALL:
case ExpressionVisitor.SET_MAX_DATA_MODIFICATION_ID:
case ExpressionVisitor.GET_COLUMNS:
return true;
default:
throw DbException.throwInternalError("type=" +
visitor.getType());
}
}
@Override
public int getCost() {
int cost = 3;
for (Expression e : args) {
cost += e.getCost();
}
```

```
return cost;
}
@Override
public boolean isDeterministic() {
return info.deterministic;
}
@Override
public boolean isFast() {
return info.fast;
}
}
/*
* Copyright 2004-2013 H2 Group. Multiple-Licensed under the H2 License,
* Version 1.0, and under the Eclipse Public License, Version 1.0
* (http://h2database.com/html/license.html).
* Initial Developer: H2 Group
*/
package org.h2.expression;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import org.h2.command.dml.Select;
import org.h2.command.dml.SelectOrderBy;
import org.h2.constant.ErrorCode;
import org.h2.engine.Session;
import org.h2.index.Cursor;
import org.h2.index.Index;
import org.h2.message.DbException;
import org.h2.result.SearchRow;
import org.h2.result.SortOrder;
import org.h2.table.Column;
import org.h2.table.ColumnResolver;
import org.h2.table.Table;
import org.h2.table.TableFilter;
import org.h2.util.New;
import org.h2.util.StatementBuilder;
import org.h2.util.StringUtils;
import org.h2.value.DataType;
import org.h2.value.Value;
import org.h2.value.ValueArray;
import org.h2.value.ValueBoolean;
import org.h2.value.ValueDouble;
import org.h2.value.ValueInt;
import org.h2.value.ValueLong;
import org.h2.value.ValueNull;
import org.h2.value.ValueString;
/**
* Implements the integrated aggregate functions, such as COUNT, MAX, SUM.
```

```
*/
public class Aggregate extends Expression {
/**
* The aggregate type for COUNT(*).
*/
public static final int COUNT_ALL = 0;
/**
* The aggregate type for COUNT(expression).
*/
public static final int COUNT = 1;
/**
* The aggregate type for GROUP_CONCAT(...).
*/
public static final int GROUP_CONCAT = 2;
/**
* The aggregate type for SUM(expression).
*/
static final int SUM = 3;
/**
* The aggregate type for MIN(expression).
*/
static final int MIN = 4;
/**
* The aggregate type for MAX(expression).
*/
static final int MAX = 5;
/**
* The aggregate type for AVG(expression).
*/
static final int AVG = 6;
/**
* The aggregate type for STDDEV_POP(expression).
*/
static final int STDDEV_POP = 7;
/**
* The aggregate type for STDDEV_SAMP(expression).
*/
static final int STDDEV_SAMP = 8;
/**
* The aggregate type for VAR_POP(expression).
*/
static final int VAR_POP = 9;
/**
* The aggregate type for VAR_SAMP(expression).
*/
static final int VAR_SAMP = 10;
/**
* The aggregate type for BOOL_OR(expression).
*/
```

```
static final int BOOL_OR = 11;
/**
* The aggregate type for BOOL_AND(expression).
*/
static final int BOOL_AND = 12;
/**
* The aggregate type for SELECTIVITY(expression).
*/
static final int SELECTIVITY = 13;
/**
* The aggregate type for HISTOGRAM(expression).
*/
static final int HISTOGRAM = 14;
/**
* The aggregate type for MEDIAN(expression).
*/
static final int MEDIAN =15;
/**
* The aggregate type for MODE(expression).
*/
static final int MODE =16;
/**
* The aggregate type for FIRST(expression).
*/
static final int FIRST =17;
/**
* The aggregate type for MODE(expression).
*/
static final int LAST =18;
private static final HashMap<String, Integer> AGGREGATES =
New.hashMap();
private final int type;
private final Select select;
private final boolean distinct;
private Expression on;
private Expression groupConcatSeparator;
private ArrayList<SelectOrderBy> groupConcatOrderList;
private SortOrder groupConcatSort;
private int dataType, scale;
private long precision;
private int displaySize;
private int lastGroupRowId;
/**
* Create a new aggregate object.
*
* @param type the aggregate type
* @param on the aggregated expression
* @param select the select statement
* @param distinct if distinct is used
```

```
*/
public Aggregate(int type, Expression on, Select select, boolean
distinct) {
this.type = type;
this.on = on;
this.select = select;
this.distinct = distinct;
}
static {
addAggregate("COUNT", COUNT);
addAggregate("SUM", SUM);
addAggregate("MIN", MIN);
addAggregate("MAX", MAX);
addAggregate("AVG", AVG);
addAggregate("MEDIAN", MEDIAN);
addAggregate("MODE", MODE);
addAggregate("FIRST", FIRST);
addAggregate("LAST", LAST);
addAggregate("GROUP_CONCAT", GROUP_CONCAT);
addAggregate("STDDEV_SAMP", STDDEV_SAMP);
addAggregate("STDDEV", STDDEV_SAMP);
addAggregate("STDDEV_POP", STDDEV_POP);
addAggregate("STDDEVP", STDDEV_POP);
addAggregate("VAR_POP", VAR_POP);
addAggregate("VARP", VAR_POP);
addAggregate("VAR_SAMP", VAR_SAMP);
addAggregate("VAR", VAR_SAMP);
addAggregate("VARIANCE", VAR_SAMP);
addAggregate("BOOL_OR", BOOL_OR);
// HSQLDB compatibility, but conflicts with x > EVERY(...)
addAggregate("SOME", BOOL_OR);
addAggregate("BOOL_AND", BOOL_AND);
// HSQLDB compatibility, but conflicts with x > SOME(...)
addAggregate("EVERY", BOOL_AND);
addAggregate("SELECTIVITY", SELECTIVITY);
addAggregate("HISTOGRAM", HISTOGRAM);
}
private static void addAggregate(String name, int type) {
AGGREGATES.put(name, type);
}
/**
* Get the aggregate type for this name, or -1 if no aggregate has been
* found.
*
* @param name the aggregate function name
* @return -1 if no aggregate function has been found, or the aggregate
type
*/
public static int getAggregateType(String name) {
```

```
Integer type = AGGREGATES.get(name);
return type == null ? -1 : type.intValue();
}
/**
* Set the order for GROUP_CONCAT() aggregate.
*
* @param orderBy the order by list
*/
public void setGroupConcatOrder(ArrayList<SelectOrderBy> orderBy) {
this.groupConcatOrderList = orderBy;
}
/**
* Set the separator for the GROUP_CONCAT() aggregate.
*
* @param separator the separator expression
*/
public void setGroupConcatSeparator(Expression separator) {
this.groupConcatSeparator = separator;
}
private SortOrder initOrder(Session session) {
int size = groupConcatOrderList.size();
int[] index = new int[size];
int[] sortType = new int[size];
for (int i = 0; i < size; i++) {
SelectOrderBy o = groupConcatOrderList.get(i);
index[i] = i + 1;
int order = o.descending ? SortOrder.DESCENDING :
SortOrder.ASCENDING;
sortType[i] = order;
}
return new SortOrder(session.getDatabase(), index, sortType);
}
@Override
public void updateAggregate(Session session) {
// TODO aggregates: check nested MIN(MAX(ID)) and so on
// if(on != null) {
// on.updateAggregate();
// }
HashMap<Expression, Object> group = select.getCurrentGroup();
if (group == null) {
// this is a different level (the enclosing query)
return;
}
int groupRowId = select.getCurrentGroupRowId();
if (lastGroupRowId == groupRowId) {
// already visited
return;
}
lastGroupRowId = groupRowId;
```

```
AggregateData data = (AggregateData) group.get(this);
if (data == null) {
data = AggregateData.create(type);
group.put(this, data);
}
Value v = on == null ? null : on.getValue(session);
if (type == GROUP_CONCAT) {
if (v != ValueNull.INSTANCE) {
v = v.convertTo(Value.STRING);
if (groupConcatOrderList != null) {
int size = groupConcatOrderList.size();
Value[] array = new Value[1 + size];
array[0] = v;
for (int i = 0; i < size; i++) {
SelectOrderBy o = groupConcatOrderList.get(i);
array[i + 1] = o.expression.getValue(session);
}
v = ValueArray.get(array);
}
}
}
data.add(session.getDatabase(), dataType, distinct, v);
}
@Override
public Value getValue(Session session) {
if (select.isQuickAggregateQuery()) {
switch (type) {
case COUNT:
case COUNT_ALL:
Table table = select.getTopTableFilter().getTable();
return ValueLong.get(table.getRowCount(session));
case MIN:
case MAX:
boolean first = type == MIN;
Index index = getColumnIndex();
int sortType = index.getIndexColumns()[0].sortType;
if ((sortType & SortOrder.DESCENDING) != 0) {
first = !first;
}
Cursor cursor = index.findFirstOrLast(session, first);
SearchRow row = cursor.getSearchRow();
Value v;
if (row == null) {
v = ValueNull.INSTANCE;
} else {
v = row.getValue(index.getColumns()[0].getColumnId());
}
return v;
default:
```

```
DbException.throwInternalError("type=" + type);
}
}
HashMap<Expression, Object> group = select.getCurrentGroup();
if (group == null) {
throw
DbException.get(ErrorCode.INVALID_USE_OF_AGGREGATE_FUNCTION_1, getSQL());
}
AggregateData data = (AggregateData) group.get(this);
if (data == null) {
data = AggregateData.create(type);
}
Value v = data.getValue(session.getDatabase(), dataType, distinct);
if (type == GROUP_CONCAT) {
ArrayList<Value> list = ((AggregateDataGroupConcat)
data).getList();
if (list == null || list.size() == 0) {
return ValueNull.INSTANCE;
}
if (groupConcatOrderList != null) {
final SortOrder sortOrder = groupConcatSort;
Collections.sort(list, new Comparator<Value>() {
@Override
public int compare(Value v1, Value v2) {
Value[] a1 = ((ValueArray) v1).getList();
Value[] a2 = ((ValueArray) v2).getList();
return sortOrder.compare(a1, a2);
}
});
}
StatementBuilder buff = new StatementBuilder();
String sep = groupConcatSeparator == null ? "," :
groupConcatSeparator.getValue(session).getString();
for (Value val : list) {
String s;
if (val.getType() == Value.ARRAY) {
s = ((ValueArray) val).getList()[0].getString();
} else {
s = val.getString();
}
if (s == null) {
continue;
}
if (sep != null) {
buff.appendExceptFirst(sep);
}
buff.append(s);
}
v = ValueString.get(buff.toString());
```

```
}
return v;
}
@Override
public int getType() {
return dataType;
}
@Override
public void mapColumns(ColumnResolver resolver, int level) {
if (on != null) {
on.mapColumns(resolver, level);
}
if (groupConcatOrderList != null) {
for (SelectOrderBy o : groupConcatOrderList) {
o.expression.mapColumns(resolver, level);
}
}
if (groupConcatSeparator != null) {
groupConcatSeparator.mapColumns(resolver, level);
}
}
@Override
public Expression optimize(Session session) {
if (on != null) {
on = on.optimize(session);
dataType = on.getType();
scale = on.getScale();
precision = on.getPrecision();
displaySize = on.getDisplaySize();
}
if (groupConcatOrderList != null) {
for (SelectOrderBy o : groupConcatOrderList) {
o.expression = o.expression.optimize(session);
}
groupConcatSort = initOrder(session);
}
if (groupConcatSeparator != null) {
groupConcatSeparator = groupConcatSeparator.optimize(session);
}
switch (type) {
case GROUP_CONCAT:
dataType = Value.STRING;
scale = 0;
precision = displaySize = Integer.MAX_VALUE;
break;
case COUNT_ALL:
case COUNT:
dataType = Value.LONG;
scale = 0;
```

```
precision = ValueLong.PRECISION;
displaySize = ValueLong.DISPLAY_SIZE;
break;
case SELECTIVITY:
dataType = Value.INT;
scale = 0;
precision = ValueInt.PRECISION;
displaySize = ValueInt.DISPLAY_SIZE;
break;
case HISTOGRAM:
dataType = Value.ARRAY;
scale = 0;
precision = displaySize = Integer.MAX_VALUE;
break;
case SUM:
if (dataType == Value.BOOLEAN) {
// example: sum(id > 3) (count the rows)
dataType = Value.LONG;
} else if (!DataType.supportsAdd(dataType)) {
throw
DbException.get(ErrorCode.SUM_OR_AVG_ON_WRONG_DATATYPE_1, getSQL());
} else {
dataType = DataType.getAddProofType(dataType);
}
break;
case AVG:
if (!DataType.supportsAdd(dataType)) {
throw
DbException.get(ErrorCode.SUM_OR_AVG_ON_WRONG_DATATYPE_1, getSQL());
}
break;
case MEDIAN:
if (!DataType.supportsAdd(dataType)) {
throw
DbException.get(ErrorCode.SUM_OR_AVG_ON_WRONG_DATATYPE_1, getSQL());
}
break;
case MODE:
if (!DataType.supportsAdd(dataType)) {
throw
DbException.get(ErrorCode.SUM_OR_AVG_ON_WRONG_DATATYPE_1, getSQL());
}
break;
case MIN:
case MAX:
case FIRST:
case LAST:
break;
case STDDEV_POP:
```

```
case STDDEV_SAMP:
case VAR_POP:
case VAR_SAMP:
dataType = Value.DOUBLE;
precision = ValueDouble.PRECISION;
displaySize = ValueDouble.DISPLAY_SIZE;
scale = 0;
break;
case BOOL_AND:
case BOOL_OR:
dataType = Value.BOOLEAN;
precision = ValueBoolean.PRECISION;
displaySize = ValueBoolean.DISPLAY_SIZE;
scale = 0;
break;
default:
DbException.throwInternalError("type=" + type);
}
return this;
}
@Override
public void setEvaluatable(TableFilter tableFilter, boolean b) {
if (on != null) {
on.setEvaluatable(tableFilter, b);
}
if (groupConcatOrderList != null) {
for (SelectOrderBy o : groupConcatOrderList) {
o.expression.setEvaluatable(tableFilter, b);
}
}
if (groupConcatSeparator != null) {
groupConcatSeparator.setEvaluatable(tableFilter, b);
}
}
@Override
public int getScale() {
return scale;
}
@Override
public long getPrecision() {
return precision;
}
@Override
public int getDisplaySize() {
return displaySize;
}
private String getSQLGroupConcat() {
StatementBuilder buff = new StatementBuilder("GROUP_CONCAT(");
if (distinct) {
```

```
buff.append("DISTINCT ");
}
buff.append(on.getSQL());
if (groupConcatOrderList != null) {
buff.append(" ORDER BY ");
for (SelectOrderBy o : groupConcatOrderList) {
buff.appendExceptFirst(", ");
buff.append(o.expression.getSQL());
if (o.descending) {
buff.append(" DESC");
}
}
}
if (groupConcatSeparator != null) {
buff.append(" SEPARATOR
").append(groupConcatSeparator.getSQL());
}
return buff.append(')').toString();
}
@Override
public String getSQL() {
String text;
switch (type) {
case GROUP_CONCAT:
return getSQLGroupConcat();
case COUNT_ALL:
return "COUNT(*)";
case COUNT:
text = "COUNT";
break;
case SELECTIVITY:
text = "SELECTIVITY";
break;
case HISTOGRAM:
text = "HISTOGRAM";
break;
case SUM:
text = "SUM";
break;
case MIN:
text = "MIN";
break;
case MAX:
text = "MAX";
break;
case AVG:
text = "AVG";
break;
case MEDIAN:
```

```
text = "MEDIAN";
break;
case MODE:
text = "MODE";
break;
case FIRST:
text = "FIRST";
break;
case LAST:
text = "LAST";
break;
case STDDEV_POP:
text = "STDDEV_POP";
break;
case STDDEV_SAMP:
text = "STDDEV_SAMP";
break;
case VAR_POP:
text = "VAR_POP";
break;
case VAR_SAMP:
text = "VAR_SAMP";
break;
case BOOL_AND:
text = "BOOL_AND";
break;
case BOOL_OR:
text = "BOOL_OR";
break;
default:
throw DbException.throwInternalError("type=" + type);
}
if (distinct) {
return text + "(DISTINCT " + on.getSQL() + ")";
}
return text + StringUtils.enclose(on.getSQL());
}
private Index getColumnIndex() {
if (on instanceof ExpressionColumn) {
ExpressionColumn col = (ExpressionColumn) on;
Column column = col.getColumn();
TableFilter filter = col.getTableFilter();
if (filter != null) {
Table table = filter.getTable();
Index index = table.getIndexForColumn(column);
return index;
}
}
return null;
```

```
}
@Override
public boolean isEverything(ExpressionVisitor visitor) {
if (visitor.getType() ==
ExpressionVisitor.OPTIMIZABLE_MIN_MAX_COUNT_ALL) {
switch (type) {
case COUNT:
if (!distinct && on.getNullable() == Column.NOT_NULLABLE) {
return visitor.getTable().canGetRowCount();
}
return false;
case COUNT_ALL:
return visitor.getTable().canGetRowCount();
case MIN:
case MAX:
Index index = getColumnIndex();
return index != null;
default:
return false;
}
}
if (on != null && !on.isEverything(visitor)) {
return false;
}
if (groupConcatSeparator != null && !
groupConcatSeparator.isEverything(visitor)) {
return false;
}
if (groupConcatOrderList != null) {
for (int i = 0, size = groupConcatOrderList.size(); i < size;
i++) {
SelectOrderBy o = groupConcatOrderList.get(i);
if (!o.expression.isEverything(visitor)) {
return false;
}
}
}
return true;
}
@Override
public int getCost() {
return (on == null) ? 1 : on.getCost() + 1;
}
}
```

```
/*
 * Copyright 2004-2013 H2 Group. Multiple-Licensed under the H2 License,
 * Version 1.0, and under the Eclipse Public License, Version 1.0
 * (http://h2database.com/html/license.html).
 * Initial Developer: H2 Group
 */
package org.h2.expression;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import org.h2.engine.Database;
import org.h2.message.DbException;
import org.h2.util.ValueHashMap;
import org.h2.value.DataType;
import org.h2.value.Value;
import org.h2.value.ValueBoolean;
import org.h2.value.ValueDouble;
import org.h2.value.ValueLong;
import org.h2.value.ValueNull;

/**
 * Data stored while calculating an aggregate.
 */
class AggregateDataDefault extends AggregateData {

    private final int aggregateType;
    private long count;
    private ValueHashMap<AggregateDataDefault> distinctValues;
    private Value value;
    private double m2, mean;
    private int median;
    ArrayList a = new ArrayList();
    ArrayList arrayList = new ArrayList();
    ArrayList arrayListForFirst = new ArrayList();
    ArrayList arrayListForLast = new ArrayList();
    //HashMap hashmap = new HashMap();

    /**
     * @param aggregateType the type of the aggregate operation
     */
    AggregateDataDefault(int aggregateType) {
        this.aggregateType = aggregateType;
    }

    @Override
    void add(Database database, int dataType, boolean distinct, Value v) {
        if (v == ValueNull.INSTANCE) {
            return;
        }
        count++;
```

```
        if (distinct) {
            if (distinctValues == null) {
                distinctValues = ValueHashMap.newInstance();
            }
            distinctValues.put(v, this);
            return;
        }
        switch (aggregateType) {
            case Aggregate.SUM:
                if (value == null) {
                    value = v.convertTo(dataType);
                } else {
                    v = v.convertTo(value.getType());
                    value = value.add(v);
                }
                break;
            case Aggregate.MEDIAN:
                if (value == null) {
                    value =
v.convertTo(DataType.getAddProofType(dataType));
                } else {
                    v = v.convertTo(value.getType());
                    //value = value.add(v);
                    a.add(v);

                }
                break;
            case Aggregate.MODE:
                if (value == null) {
                    value =
v.convertTo(DataType.getAddProofType(dataType));
                } else {
                    v = v.convertTo(value.getType());
                    //value = value.add(v);
                    //if (hashmap.containsKey(v)) {
                    //increment count;
                    //  int val = (int)hashmap.get(v);
                    //  hashmap.put(v, val+1);
                    //} else {
                    //add in the hashmap
                    //  hashmap.put(v, 1);
                    arrayList.add(v);
                }
                //}
                break;
            case Aggregate.AVG:
                if (value == null) {
                    value =
v.convertTo(DataType.getAddProofType(dataType));
                } else {
                    v = v.convertTo(value.getType());
                    value = value.add(v);
                }
                break;
```

```
            case Aggregate.MIN:
                if (value == null || database.compare(v, value) < 0) {
                    value = v;
                }
                break;
            case Aggregate.MAX:
                if (value == null || database.compare(v, value) > 0) {
                    value = v;
                }
                break;
            case Aggregate.FIRST:
                if (value == null) {
                    value = v;
                    arrayListForFirst.add(v);
                }
                break;
            case Aggregate.LAST:
                if (value == null) {
                    value =
v.convertTo(DataType.getAddProofType(dataType));
                } else {
                    v = v.convertTo(value.getType());
                    //value = value.add(v);
                    arrayListForLast.add(v);
                }
                break;
            case Aggregate.STDDEV_POP:
            case Aggregate.STDDEV_SAMP:
            case Aggregate.VAR_POP:
            case Aggregate.VAR_SAMP: {
                // Using Welford's method, see also
                //
http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance
                // http://www.johndcook.com/standard_deviation.html
                double x = v.getDouble();
                if (count == 1) {
                    mean = x;
                    m2 = 0;
                } else {
                    double delta = x - mean;
                    mean += delta / count;
                    m2 += delta * (x - mean);
                }
                break;
            }
            case Aggregate.BOOL_AND:
                v = v.convertTo(Value.BOOLEAN);
                if (value == null) {
                    value = v;
                } else {
                    value =
ValueBoolean.get(value.getBoolean().booleanValue() &&
v.getBoolean().booleanValue());
                }
```

```
                break;
            case Aggregate.BOOL_OR:
                v = v.convertTo(Value.BOOLEAN);
                if (value == null) {
                    value = v;
                } else {
                    value =
ValueBoolean.get(value.getBoolean().booleanValue() ||
v.getBoolean().booleanValue());
                }
                break;
            default:
                DbException.throwInternalError("type=" + aggregateType);
        }
    }

    @Override
    Value getValue(Database database, int dataType, boolean distinct) {
        if (distinct) {
            count = 0;
            groupDistinct(database, dataType);
        }
        Value v = null;
        switch (aggregateType) {
            case Aggregate.SUM:
            case Aggregate.MIN:
            case Aggregate.MAX:
            case Aggregate.BOOL_OR:
            case Aggregate.BOOL_AND:
                v = value;
                break;
            case Aggregate.AVG:
                if (value != null) {
                    v = divide(value, count);
                }
                break;
            case Aggregate.MEDIAN:
                int i = 0;
                Value med = null;
                Iterator itr = a.iterator();
                //Collections.sort(a);
                median = (int) (count / 2);
                while (itr.hasNext()) {
                    med = (Value) itr.next();
                    i++;
                    if (i == median) {
                        v = med;
                        break;
                    }
                }
                v = med;
                break;
            case Aggregate.MODE:
                Value mode = null;
```

```
                Value[] listArray;
                int maxCount = 0;
                listArray = (Value[]) arrayList.toArray(new Value[(int)
count]);

                for (int k = 0; k < count; k++) {
                    int c = 0;
                    for (int j = 0; j < count; j++) {
                        if (listArray[k] == listArray[j]) {
                            ++c;
                        }
                        if (c > maxCount) {
                            maxCount = c;
                            mode = listArray[k];
                        }
                    }
                }
                v = mode;
                break;
            case Aggregate.FIRST:

                Iterator itrFirst = arrayListForFirst.iterator();
                Value first = (Value) itrFirst.next();
                v = first;
                break;

            case Aggregate.LAST:
                Value last = null;
                int cnt = 0;
                Iterator itrLast = arrayListForLast.iterator();
                while (itrLast.hasNext()) {
                    cnt++;
                    if (cnt == count) {
                        last = (Value) itrLast.next();

                    }
                }
                v = last;
                break;
            case Aggregate.STDDEV_POP: {
                if (count < 1) {
                    return ValueNull.INSTANCE;
                }
                v = ValueDouble.get(Math.sqrt(m2 / count));
                break;
            }
            case Aggregate.STDDEV_SAMP: {
                if (count < 2) {
                    return ValueNull.INSTANCE;
                }
                v = ValueDouble.get(Math.sqrt(m2 / (count - 1)));
                break;
            }
            case Aggregate.VAR_POP: {
```

```
                if (count < 1) {
                    return ValueNull.INSTANCE;
                }
                v = ValueDouble.get(m2 / count);
                break;
            }
            case Aggregate.VAR_SAMP: {
                if (count < 2) {
                    return ValueNull.INSTANCE;
                }
                v = ValueDouble.get(m2 / (count - 1));
                break;
            }
            default:
                DbException.throwInternalError("type=" + aggregateType);
        }
        return v == null ? ValueNull.INSTANCE : v.convertTo(dataType);
    }

    private static Value divide(Value a, long by) {
        if (by == 0) {
            return ValueNull.INSTANCE;
        }
        int type = Value.getHigherOrder(a.getType(), Value.LONG);
        Value b = ValueLong.get(by).convertTo(type);
        a = a.convertTo(type).divide(b);
        return a;
    }

    private void groupDistinct(Database database, int dataType) {
        if (distinctValues == null) {
            return;
        }
        count = 0;
        for (Value v : distinctValues.keys()) {
            add(database, dataType, false, v);
        }
    }
}
```